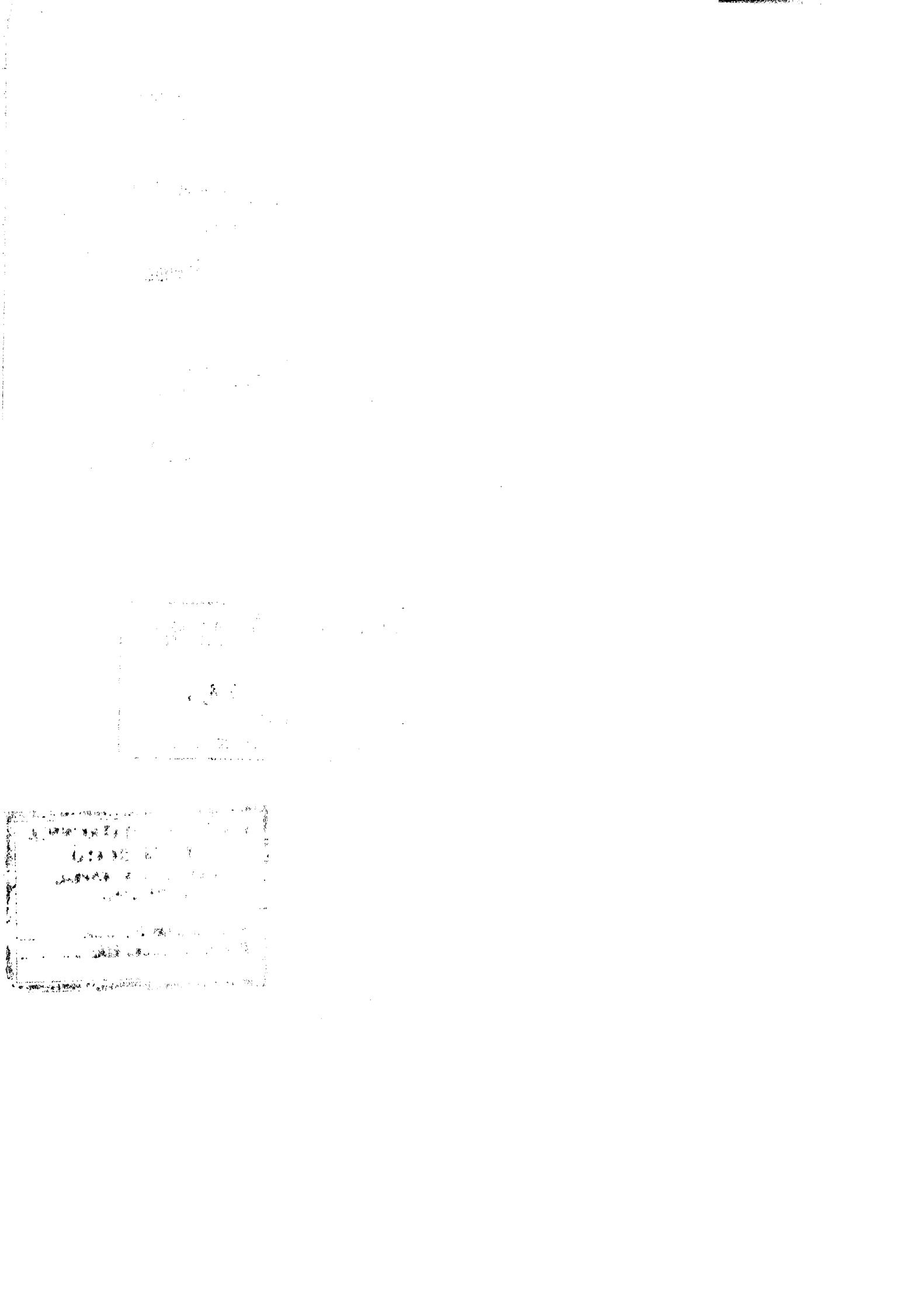


Análisis y diseño orientado a objetos con aplicaciones

INVENTARIO

FECHA INGRESO 14/05/96
COMPRO EN LIBRERIA RJAC S/.92.000 =
DEPENDENCIA BIBLIOTECA
Jorge Portillo
INVENTARIO





Análisis y diseño orientado a objetos con aplicaciones

Grady Booch

2.^a edición

Versión en español de

Juan Manuel Cueva Lovelle
Agustín Cernuda del Río
Universidad de Oviedo, España

Con la colaboración de

Luis Joyanes Aguilar
*Universidad Pontificia de Salamanca
Campus de Madrid, España*

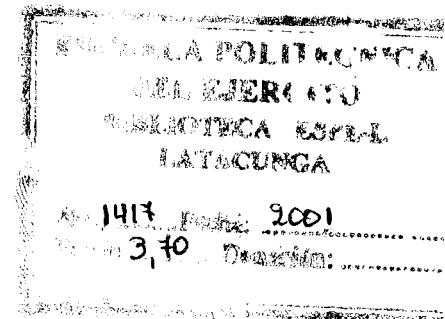
Gabriel Guerrero
saxsa, S.C.
México



ADDISON-WESLEY / DIAZ DE SANTOS

Argentina · Chile · Colombia · Ecuador · España
Estados Unidos · México · Perú · Puerto Rico · Venezuela

BIBLIOTECA DEL INSTITUTO TECNOLÓGICO SUPERIOR DEL EJERCITO	
LATACUNGA	
FECHA DE INGRESO	5/01/96
NO.	DONACIÓN
COMPRA	DANJE



Versión en español de la obra titulada *Object-Oriented Analysis and Design with Applications. Second Edition* de Grady Booch, publicada originalmente en inglés por Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, E.U.A. ©1994

Esta edición en español es la única autorizada.

Copublicación de Addison-Wesley Iberoamericana, S. A.
y Ediciones Díaz de Santos, S. A.

© 1996 por Addison-Wesley Iberoamericana, S. A.
Wilmington, Delaware, E.U.A.

Reservados todos los derechos.

«No está permitida la reproducción total o parcial de este libro,
ni su tratamiento informático, ni la transmisión de ninguna
forma o por cualquier medio, ya sea electrónico, mecánico,
por fotocopia, por registro u otros métodos, sin el permiso
previo y por escrito de los titulares del Copyright»

Impreso en Estados Unidos de América. *Printed in U.S.A.*

ISBN: 0-201-60122-2

1 2 3 4 5 6 7 8 9 -MA- 99 98 97 96 95

Conceptos

Primera sección: Conceptos 1

- Capítulo 1: Complejidad 3
- Capítulo 2: El modelo de objetos 31
- Capítulo 3: Clases y objetos 95
- Capítulo 4: Clasificación 167

Segunda sección: El método 197

- Capítulo 5: La notación 199
- Capítulo 6: El proceso 263
- Capítulo 7: Aspectos pragmáticos 305

Tercera sección: Aplicaciones 335

- Capítulo 8: Adquisición de datos: Estación de monitorización del clima 337
- Capítulo 9: Marcos de referencia: Biblioteca básica de clases 375
- Capítulo 10: Computación cliente/servidor: Mantenimiento de catálogos 431
- Capítulo 11: Inteligencia artificial: Criptoanálisis 471
- Capítulo 12: Dirección y control: Gestión de tráfico 507

Apéndice: Lenguajes de programación orientados a objetos 537

Glosario 559

Bibliografía clasificada 569

Índice analítico 621

Índice general

Prefacio	XIII
Primera sección: Conceptos	1
1. Complejidad	3
1.1. La complejidad inherente al software	3
1.2. La estructura de los sistemas complejos	9
1.3. Imponiendo orden al caos	17
1.4. Del diseño de sistemas complejos	23
Recuadro: Categorías de métodos de diseño	19
2. El modelo de objetos	31
2.1. La evolución del modelo de objetos	32
2.2. Elementos del modelo de objetos	44
2.3. Aplicación del modelo de objetos	86
Recuadro: Fundamentos del modelo de objetos	40
3. Clases y objetos	95
3.1. La naturaleza de los objetos	95
3.2. Relaciones entre objetos	113
3.3. La naturaleza de una clase	120
3.4. Relaciones entre clases	123
3.5. La interacción entre clases y objetos	154
3.6. De la construcción de clases y objetos de calidad	155
Recuadro: Invocando un método	135
4. Clasificación	167
4.1. La importancia de una clasificación correcta	167
4.2. Identificando clases y objetos	172
4.3. Abstracciones y mecanismos clave	185
Recuadro: Un problema de clasificación	173

Segunda sección: El método	197
5. La notación	199
5.1. Elementos de la notación	200
5.2. Diagrama de clases	205
5.3. Diagramas de transición de estados	229
5.4. Diagramas de objetos	239
5.5. Diagramas de interacción	248
5.6. Diagramas de módulos	250
5.7. Diagramas de procesos	255
5.8. Aplicación de la notación	258
6. El proceso	263
6.1. Principios iniciales	263
6.2. El microproceso del desarrollo	269
6.3. El macroproceso del desarrollo	283
7. Aspectos pragmáticos	305
7.1. Gestión y planificación	306
7.2. Administración de personal	309
7.3. Gestión de versiones	314
7.4. Reutilización	316
7.5. Control de calidad y métricas	318
7.6. Documentación	321
7.7. Herramientas	322
7.8. Temas especiales	326
7.9. Las ventajas y los riesgos del desarrollo orientado a objetos ..	328
Tercera sección: Aplicaciones	335
8. Adquisición de datos: Estación de monitorización del clima	337
8.1. Análisis	337
8.2. Diseño	358
8.3. Evolución	364
8.4. Mantenimiento	373
Recuadro: Requisitos de la estación de monitorización del clima ..	338
9. Marcos de referencia: Biblioteca básica de clases	375
9.1. Análisis	376
9.2. Diseño	382
9.3. Evolución	418
9.4. Mantenimiento	426
Recuadro: Requisitos de biblioteca de clases básicas	377

10. Computación cliente/servidor: Mantenimiento de catálogos	433
10.1. Análisis	434
10.2. Diseño	457
10.3. Evolución	468
10.4. Mantenimiento	470
Recuadro: Requisitos del sistema de mantenimiento de catálogos	435
11. Inteligencia artificial: Criptoanálisis	473
11.1. Análisis	474
11.2. Diseño	482
11.3. Evolución	500
11.4. Mantenimiento	504
Recuadro: Requisito del criptoanálisis	475
12. Dirección y control: Gestión de tráfico	509
12.1. Análisis	510
12.2. Diseño	519
12.3. Evolución	529
12.4. Mantenimiento	533
Recuadro: Requisitos del sistema de gestión de tráfico	511
Epílogo	537
Apéndice	539
A.1. Concepto	539
A.2. Smalltalk	540
A.3. Object Pascal	546
A.4. C++	547
A.5. Common Lisp Object System	552
A.6. Ada	553
A.7. Eiffel	555
A.8. Otros lenguajes de programación orientados a objetos	556
Glosario	561
Bibliografía clasificada	571
A. Clasificación	571
B. Análisis orientado a objetos	573
C. Aplicaciones orientadas a objetos	575
D. Arquitecturas orientadas a objetos	582
E. Bases de datos orientadas a objetos	584
F. Diseño orientado a objetos	588
G. Programación orientada a objetos	594
H. Ingeniería de software	606

I.	Referencias especiales	612
J.	Teoría	613
K.	Herramientas y entornos	620
	Vocabulario técnico bilingüe	623
	Índice analítico	629

Prefacio

El hombre, por la gracia de Dios, ansía paz espiritual, logros estéticos, seguridad familiar, justicia y libertad, ninguna de las cuales se satisface directamente por la productividad industrial. Pero la productividad permite compartir lo abundante en lugar de luchar por lo escaso; proporciona tiempo para asuntos espirituales, estéticos y familiares. Permite a la sociedad delegar prácticas especiales en instituciones de religión y justicia, y la defensa de la libertad.

HARLAN MILLS*
DPMA and Human Productivity

Como profesionales de la informática hacemos lo posible por construir sistemas que sean útiles y que funcionen; como ingenieros del software, nos enfrentamos a la tarea de crear sistemas complejos en presencia de recursos humanos y de computación escasos. A lo largo de los últimos años, la tecnología orientada a objetos se ha desarrollado en diferentes segmentos de las ciencias de la computación como un medio para manejar la complejidad inherente a sistemas de muy diversos tipos. El modelo de objetos ha demostrado ser un concepto muy potente y unificador.

Cambios a la primera edición

Desde la publicación de la primera edición de *Diseño orientado a objetos con aplicaciones*, la tecnología orientada a objetos ha progresado realmente hacia la tendencia principal del desarrollo de software con robustez industrial. Hemos encontrado el uso del paradigma orientado a objetos a lo largo del mundo, al servicio de dominios tan diversos como la administración de transacciones bancarias, la automatización de boleras, la gestión de bienes públicos o la construcción del mapa del genoma humano. Muchos de los sistemas operativos, bases de datos, sistemas de telefonía, sistemas de aeronáutica o aplicaciones multimedia de la próxima generación se están escribiendo mediante el uso de técnicas orientadas a objetos. Muchos de tales proyectos han elegido el uso de la tec-

* Mills, H., 1985. *DPMA and Human Productivity*. Houston, TX: Data Processing Management Association.

nología orientada a objetos por la sencilla razón de que parece no haber otra forma de producir económicamente un sistema de programación duradero y flexible.

Desde hace varios años, se ha aplicado la notación y el proceso descrito en *Diseño orientado a objetos con aplicaciones*¹ en cientos de proyectos. A través de nuestro propio trabajo con varios de esos proyectos, así como de la generosa contribución de muchas personas que se tomaron la molestia de comunicarse con nosotros, hemos hallado vías para mejorar nuestro método, en términos de una mejor articulación del proceso, añadiendo y clarificando alguna semántica que de otra forma no aparecería o resultaría difícil de expresar con la notación original, y simplificando esta última donde fuese posible.

Durante este tiempo, han aparecido muchos otros métodos, incluyendo los trabajos de Jacobson, Rumbaugh, Coad y Yourdon, Constantine, Shlaer y Mellor, Martin y Odell, Wasserman, Goldberg y Rubin, Embley, Wirfs-Brock, Goldstein y Alger, Henderson-Sellers, Firesmith, y otros. El trabajo de Rumbaugh es particularmente interesante, porque como él mismo indica, nuestros métodos son más similares que diferentes. Hemos inspeccionado muchos de esos métodos, hemos entrevistado a desarrolladores y directores que los aplicaron y, cuando ha sido posible, hemos intentado utilizarlos nosotros mismos. Puesto que estamos interesados en ayudar a que los proyectos triunfen con la tecnología orientada a objetos, y no tanto en promover dogmáticamente su práctica por razones puramente emocionales o históricas, hemos intentado incorporar lo mejor de cada uno de estos métodos en nuestro propio trabajo. Reconocemos de buena voluntad las contribuciones únicas y fundamentales que cada una de estas personas ha aportado a este campo.

Está en los mejores intereses de la industria de construcción del software, y en particular de la tecnología orientada a objetos, que existan notaciones estandarizadas para el desarrollo. Por eso esta edición presenta una notación unificada que, en la medida de lo posible, elimina las diferencias puramente formales entre nuestra notación y la de otros, particularmente las de Jacobson y Rumbaugh. Una vez más, y para promover el uso del método sin restricciones, la notación es de dominio público.

Los objetivos, la audiencia y la estructura de esta edición son los mismos que en la primera. Sin embargo, hay cinco diferencias principales entre ambas.

Primero: el Capítulo 5 se ha expandido para proporcionar un detalle mucho más específico acerca de la notación unificada. Para mejorar la comprensión del lector en este sentido, se distingue explícitamente entre sus elementos fundamentales y avanzados. Además, se ha prestado una atención especial a cómo las diferentes vistas de la notación se integran entre sí.

Segundo: los Capítulos 6 y 7, que se refieren al proceso y pragmática del análisis y el diseño orientados a objetos, se han alargado en gran medida. Se ha

¹ Incluyendo mis propios proyectos. Al fin y al cabo, soy un desarrollador, no sólo un metodólogo (diseñador de metodologías). La primera pregunta que se debería plantear a cualquier metodólogo es si utiliza sus propios métodos para desarrollar software.

cambiado también el título de esta segunda edición para reflejar el hecho de que nuestro proceso verdaderamente engloba tanto el análisis como el diseño.

Tercero: se ha decidido expresar todos los ejemplos de programación que aparecen en el texto principal en C++. Este lenguaje se está convirtiendo rápidamente en el estándar «de facto» en muchos dominios de aplicaciones; además, la mayoría de los desarrolladores profesionales versados en otros lenguajes de programación orientados a objetos pueden leer C++. Esto no significa que consideremos otros lenguajes —como Smalltalk, CLOS, Ada o Eiffel— menos importantes. El centro de atención de este libro es el análisis y el diseño, y puesto que necesitamos expresar ejemplos concretos, decidimos hacerlo en un lenguaje razonablemente común. Donde sea aplicable, describimos la semántica propia de esos otros lenguajes y su impacto sobre el método.

Cuarto: esta edición introduce varios ejemplos nuevos de aplicación. En varios dominios de aplicaciones han surgido ciertos modismos y marcos de referencia específicos, y los ejemplos aprovechan estas prácticas. Por ejemplo, la computación cliente/servidor proporciona las bases de un ejemplo de aplicación revisado.

Por último, casi cada capítulo proporciona referencias y discusiones acerca de la tecnología orientada a objetos de relevancia que ha aparecido desde la primera edición.

Objetivos

Este libro proporciona una guía práctica en la construcción de sistemas orientados a objetos. Sus objetivos específicos son:

- Proporcionar una comprensión sólida de los conceptos fundamentales del modelo de objetos.
- Facilitar el dominio de la notación y el proceso del análisis y diseño orientados a objetos.
- Mostrar la aplicación realista del desarrollo orientado a objetos en una variedad de dominios de problemas.

Todos los conceptos presentados aquí se apoyan en fundamentos teóricos sólidos; pero éste es, antes que nada, un libro pragmático que se dirige a los intereses y necesidades prácticas de la comunidad de la Ingeniería del Software.

Audiencia

Este libro está escrito tanto para el profesional de la informática como para el estudiante.

- Para el ingeniero del software en ejercicio se muestra cómo utilizar tecnología orientada a objetos para resolver problemas reales.
- En su faceta de analista o arquitecto, se ofrece al lector un camino desde los requerimientos hasta la implantación, utilizando análisis y diseño orientados a objetos. Se desarrolla su habilidad para distinguir las «buenas» arquitecturas orientadas a objetos de las «malas», y a estudiar diseños alternativos cuando la terquedad del mundo real obliga a ello. O quizás sea más importante aún el hecho de que se le ofrecen nuevos enfoques para razonar sobre sistemas complejos.
- Para el director de un equipo de desarrollo, se proporciona cierta perspicacia a la hora de asignar recursos y de manejar los riesgos asociados con sistemas de software complejos.
- Para el constructor de herramientas y el usuario correspondiente, se ofrece un tratamiento riguroso de la notación y el proceso del desarrollo orientado a objetos como base de herramientas de ingeniería del software asistida por computador (herramientas CASE).
- Para el estudiante, se proporciona la instrucción necesaria para adquirir varias habilidades que resultan importantes en la ciencia y arte de desarrollar sistemas complejos.

Este libro es también conveniente para su utilización en cursos de graduados y estudiantes universitarios, así como en seminarios profesionales y estudio individual. Al centrarse primordialmente en un método de desarrollo de software, resulta sumamente apropiado para cursos sobre ingeniería del software y programación avanzada, y como suplemento para cursos que involucren lenguajes específicos de programación orientada a objetos.

Estructura

El libro se divide en tres secciones principales —Conceptos, El método y Aplicaciones— con considerable material suplementario disperso a lo largo del mismo.

Conceptos

La primera sección examina la complejidad inherente al software y las formas en que esta complejidad se manifiesta. Se presenta el modelo de objetos como una forma de ayudar a manejar esta complejidad. En detalle, se examinan los elementos fundamentales del modelo orientado a objetos: abstracción, encapsulamiento, modularidad, jerarquía, tipos, concurrencia y permanencia. Se plantean preguntas básicas como ¿qué es una clase? o ¿qué es un objeto? Ya que la identificación de clases y objetos significativos es la tarea clave en el desarro-

llo orientado a objetos, se emplea un tiempo considerable en estudiar la naturaleza de la clasificación. En particular, se examinan aproximaciones a la clasificación en otras disciplinas, como biología, lengua o psicología, para aplicar entonces estas enseñanzas al problema de descubrir clases y objetos en sistemas de software.

El método

La segunda sección presenta un método para el desarrollo de sistemas complejos basados en el modelo de objetos. Primero se introduce una notación gráfica para el análisis y diseño orientados a objetos, y a continuación su proceso. Se examinan también los aspectos prácticos del desarrollo orientado a objetos —en particular, su lugar en el ciclo de vida del desarrollo del software y sus implicaciones en la gestión de proyectos.

Aplicaciones

La última sección ofrece una colección de cinco ejemplos, completos y significativos, que engloban una selección de diversos dominios de problemas: adquisición de datos, entornos de aplicaciones, gestión de información cliente/servidor, inteligencia artificial, y dirección y control. Se han elegido estos dominios concretos porque son representativos de los diversos tipos de problemas complejos a los que se enfrenta el ingeniero del software en el ejercicio de su labor. Es fácil mostrar cómo se aplican ciertos principios a problemas sencillos, pero, ya que el centro de atención de este libro está en construir sistemas útiles para el mundo real, hay mayor interés en mostrar cómo el modelo de objetos aumenta de escala hacia aplicaciones complejas. Algunos lectores pueden encontrar poco familiares los dominios de problema elegidos, así que se comienza cada aplicación con una breve discusión de la tecnología básica involucrada (tal como diseño de bases de datos y arquitectura de sistemas de pizarra). El desarrollo de sistemas de software rara vez se deja manejar por aproximaciones «de libro de recetas»; por tanto, se enfatiza el desarrollo incremental de las aplicaciones, guiado por una serie de principios adecuados y modelos bien formados.

Material suplementario

Se ofrece una considerable cantidad de material suplementario a lo largo del libro. Casi todos los capítulos tienen cuadros que proporcionan información sobre conceptos importantes, como la mecánica de selección de método en diferentes lenguajes de programación orientados a objetos. También se incluye un apéndice sobre lenguajes de programación orientados a objetos, en el que se considera la distinción entre lenguajes de programación basados en objetos y orientados a objetos, y la evolución y propiedades esenciales de ambas catego-

rías de lenguajes. Para los lectores poco familiarizados con ciertos lenguajes orientados a objetos se ofrece un sumario de las características de algunos bastante extendidos, con ejemplos. También se incluye un glosario de términos de uso corriente y una extensa bibliografía clasificada que suministra referencias al material original sobre el modelo de objetos. Por último, las páginas finales contienen un sumario de la notación y el proceso del método de desarrollo orientado a objetos.

Aparte del texto, en esta segunda edición está disponible una Guía del Profesor con ejercicios propuestos, cuestiones a discutir y proyectos, que podría resultar muy útil en las clases así como servir de estímulo a lectores individuales. La *Guía del profesor con ejercicios* (la edición original es *Instructor's Guide with Exercises*, ISBN 0-8053-5341-0) ha sido realizada por Mary Beth Rosson, del laboratorio Thomas J. Watson de IBM. Puede conseguirse una copia gratuita, contactando con Benjamin/Cummings en Redwood City, California, o por Internet (bookinfo@bc.aw.com).

Existen herramientas que soportan el método de Booch, y pueden obtenerse de diversas fuentes. Para más información, puede contactarse con Rational en cualquiera de los números que aparecen en la última página de este libro. Además, por medio de Benjamin/Cummings, los usuarios docentes pueden obtener software que soporta esta notación.

Cómo utilizar este libro

Este libro puede leerse de principio a fin o puede utilizarse de formas menos estructuradas. Si el lector busca una comprensión profunda de los conceptos fundamentales del modelo de objetos o el porqué de los principios del desarrollo orientado a objetos, debería comenzar con el primer capítulo y continuar en orden. Si está interesado principalmente en el aprendizaje de los detalles de la notación y el proceso de análisis y diseño orientados a objetos, puede comenzar por los Capítulos 5 y 6; el Capítulo 7 es especialmente útil para directores de proyectos que utilizan este método. Si está más interesado en la aplicación práctica de la tecnología orientada a objetos a un dominio específico, puede seleccionar cualquiera de los capítulos del 8 al 12 o todos ellos.

Reconocimientos

Este libro está dedicado a mi esposa, Jan, por su cariñoso apoyo.

A lo largo de la primera y segunda ediciones, varias personas han perfilado mis ideas acerca del desarrollo orientado a objetos. Por sus contribuciones, quiero dar las gracias especialmente a Sam Adams, Mike Akroid, Glenn Andert, Sid

Bailin, Kent Beck, Daniel Bobrow, Dick Bolz, Dave Bulman, Dave Bernstein, Kayvan Carun, Dave Collins, Steve Cook, Damian Conway, Jim Coplien, Brad Cox, Ward Cunningham, Tom DeMarco, Mike Devlin, Richard Gabriel, William Genemaras, Adele Goldberg, Ian Graham, Tony Hoare, Jon Hopkins, Michael Jackson, Ralph Johnson, James Kempf, Norm Kerth, Jordan Kreindler, Doug Lea, Phil Levy, Barbara Liskov, Cliff Longman, James MacFarlane, Masoud Milani, Harlan Mills, Robert Murray, Steve Neis, Gene Ouye, Dave Parnas, Bill Riddel, Mary Beth Rosson, Kenny Rubin, Jim Rumbaugh, Kurt Schmucker, Ed Seidewitz, Dan Shiffman, Dave Stevenson, Bjarne Stroustrup, Dave Thomas, Mike Vilot, Tony Wasserman, Peter Wegner, Iseult White, John Williams, Lloyd Williams, Mario Wolczko, Niklaus Wirth y Ed Yourdon.

Una buena parte de los aspectos prácticos de este libro se derivan de mi implicación en sistemas de software complejos que se desarrollaron en todo el mundo en compañías como Apple, Alcatel, Andersen Consulting, AT&T, Autotrol, Bell Northern Research, Boeing, Borland, Computer Sciences Corporation, Contel, Ericsson, Ferranti, General Electric, GTE, Holland Signaal, Hughes Aircraft Company, IBM, Lockheed, Martin Marietta, Motorola, NTT, Philips, Rockwell International, Shell Oil, Symantec, Telligent y TRW. Tuve la oportunidad de colaborar con literalmente cientos de ingenieros del software profesionales y sus directores, y agradezco a todos ellos su ayuda a la hora de hacer que este libro resulte relevante de cara a problemas del mundo real.

Deseo dirigir un reconocimiento especial a Rational por su apoyo a mi trabajo. Gracias también a mi editor, Dan Joraanstad, por sus ánimos durante este proyecto, y a Tony Hall, cuyos dibujos ilustran lo que de otro modo sería solamente otro pesado libro técnico. Por último, gracias a mis tres gatos, Camy, Annie y Shadow, que me acompañaron en muchas noches de trabajo.

Notas bibliográficas

Mills, H. 1985. *DPMA and Human Productivity*. Houston, TX: Data Processing Management Association.



Conceptos

Sir Isaac Newton lo admitió secretamente a algunos amigos: comprendía cómo se comportaba la gravedad, pero no cómo funcionaba.

LILY TOMLIN*
The Search for Signs of Intelligent Life in the Universe

* Wagner, J. 1986. *The Search for Signs of Intelligent Life in the Universe*. New York, NY: Harper and Row, p. 202. Autorización de ICM, Inc.

Complejidad

Un médico, un ingeniero civil y una informática estaban discutiendo acerca de cuál era la profesión más antigua del mundo. El médico señaló: «Bueno, en la Biblia dice que Dios creó a Eva de una costilla que le quitó a Adán. Evidentemente, esto requirió cirugía, y por eso bien puedo afirmar que la mía es la profesión más antigua del mundo.» El ingeniero interrumpió y dijo: «Pero incluso antes, en el Génesis, se dice que Dios creó el orden de los cielos y la tierra a partir del caos. Esta fue la primera y desde luego la más espectacular aplicación de la ingeniería civil. Por tanto, querido doctor, está usted equivocado: la mía es la más antigua profesión del mundo.» La informática se reclinó en su silla, sonrió, y dijo tranquilamente: «Pero bueno, ¿quién pensáis que creó el caos?»

1.1. La complejidad inherente al software

Las propiedades de los sistemas de software simples y complejos

Una estrella moribunda al borde del colapso, un niño aprendiendo a leer, glóbulos blancos que se apresuran a atacar a un virus: no son más que unos pocos de los objetos del mundo físico que conllevan una complejidad verdaderamente aterradora. El software puede también involucrar elementos de gran complejidad; sin embargo, la complejidad que se encuentra aquí es de un tipo fundamentalmente diferente. Como apunta Brooks, «Einstein arguyó que debe haber explicaciones simplificadas de la naturaleza, porque Dios no es caprichoso ni arbitrario. No hay fe semejante que conforte al ingeniero del software. Mucha de la complejidad que debe dominar es complejidad arbitraria»[1].

Ya se sabe que algunos sistemas de software no son complejos. Son las aplicaciones altamente intrascendentes que son especificadas, construidas, mantenidas y utilizadas por la misma persona, habitualmente el programador aficionado o el desarrollador profesional que trabaja en solitario. Esto no significa que

todos estos sistemas sean toscos o poco elegantes, ni se pretende quitar mérito a sus creadores. Tales sistemas tienden a tener un propósito muy limitado y un ciclo de vida muy corto. Uno puede permitirse tirarlos a la basura y reemplazarlos con software completamente nuevo en lugar de intentar reutilizarlos, repararlos o extender su funcionalidad. El desarrollo de estas aplicaciones es generalmente más tedioso que difícil; por consiguiente, el aprender a diseñarlas es algo que no nos interesa.

En lugar de eso, interesan mucho más los desafíos que plantea el desarrollo de lo que llamaremos *software de dimensión industrial*. Aquí se encuentran aplicaciones que exhiben un conjunto muy rico de comportamientos, como ocurre, por ejemplo, en sistemas reactivos que dirigen o son dirigidos por eventos del mundo físico, y para los cuales el tiempo y el espacio son recursos escasos; aplicaciones que mantienen la integridad de cientos de miles de registros de información mientras permiten actualizaciones y consultas concurrentes; y sistemas para la gestión y control de entidades del mundo real, tales como los controladores de tráfico aéreo o ferroviario. Los sistemas de software de esta clase tienden a tener un ciclo de vida largo, y a lo largo del tiempo muchos usuarios llegan a depender de su funcionamiento correcto. En el mundo del software de dimensión industrial se encuentran también marcos estructurales que simplifican la creación de aplicaciones orientadas a un dominio específico, y programas que mimetizan algunos aspectos de la inteligencia humana. Aunque tales aplicaciones son generalmente productos para investigación y desarrollo, no son menos complejas, porque sirven como medio y artefacto para un desarrollo incremental y exploratorio.

La característica distintiva del software de dimensión industrial es que resulta sumamente difícil, si no imposible, para el desarrollador individual comprender todas las sutilidades de su diseño. Para hablar claro, la complejidad de tales sistemas excede la capacidad intelectual humana. Lamentablemente, la complejidad de la que se habla parece ser una propiedad esencial de todos los sistemas de software de gran tamaño. Con *esencial* quiere decirse que puede dominarse esa complejidad, pero nunca eliminarla.

Ciertamente, siempre habrá genios, personas de habilidad extraordinaria que pueden hacer el trabajo de varios simples desarrolladores mortales, los equivalentes en ingeniería del software a Frank Lloyd Wright o Leonardo da Vinci. Estas son las personas a quienes se desearía como arquitectos de un sistema: las que idean lenguajes, mecanismos y marcos estructurales que otros pueden utilizar como fundamentos arquitectónicos de otras aplicaciones o sistemas. Sin embargo, como Peters observa, «El mundo está poblado de genios solamente de forma dispersa. No hay razón para creer que la comunidad de la ingeniería del software posee una proporción extraordinariamente grande de ellos» [2]. Aunque hay un toque de genialidad en todos nosotros, en el dominio del software de dimensión industrial no se puede confiar siempre en la inspiración divina. Por tanto, hay que considerar vías de mayor disciplina para dominar la complejidad. Para una mejor comprensión de lo que se pretende controlar, se va a

examinar por qué la complejidad es una propiedad esencial de todos los sistemas de software.

Por qué el software es complejo de forma innata

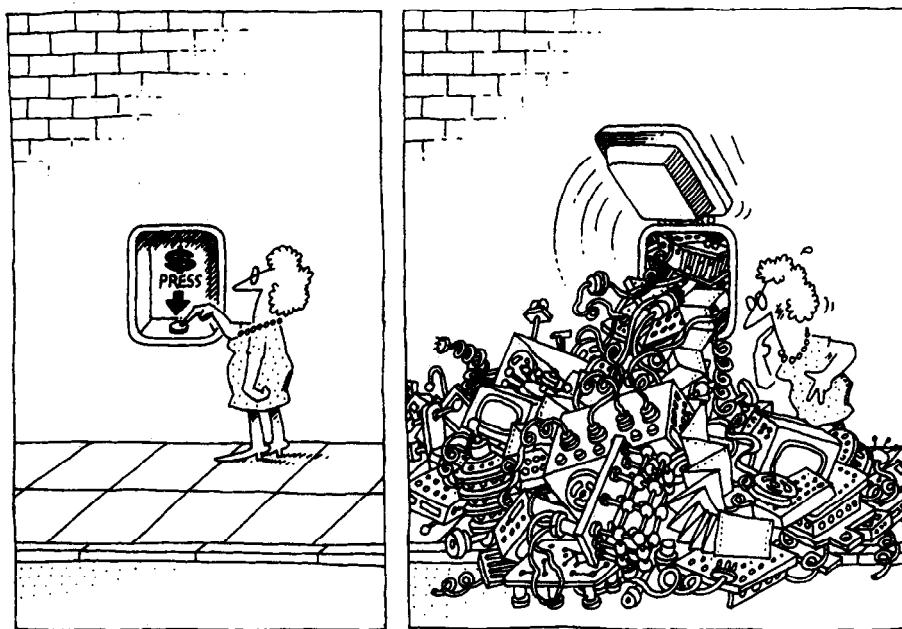
Como sugiere Brooks, «la complejidad del software es una propiedad esencial, no accidental» [3]. Se observa que esta complejidad inherente se deriva de cuatro elementos: la complejidad del dominio del problema, la dificultad de gestionar el proceso de desarrollo, la flexibilidad que se puede alcanzar a través del software y los problemas que plantea la caracterización del comportamiento de sistemas discretos.

La complejidad del dominio del problema. Los problemas que se intentan resolver con el software llevan a menudo elementos de complejidad ineludible, en los que se encuentra una miríada* de requisitos que compiten entre sí, que quizás incluso se contradicen. Considerense los requisitos para el sistema electrónico de un avión multimotor, un sistema de conmutación para teléfonos celulares o un robot autónomo. La funcionalidad pura de tales sistemas es difícil incluso de comprender, pero añádanse además todos los requerimientos no funcionales, tales como facilidad de uso, rendimiento, coste, capacidad de supervivencia y fiabilidad, que a menudo están implícitos. Esta ilimitada complejidad externa es lo que causa la complejidad arbitraria a la que Brooks se refiere.

Esta complejidad externa surge habitualmente de «desacoplamientos de impedancia» que existen entre los usuarios de un sistema y sus desarrolladores; los usuarios suelen encontrar grandes dificultades al intentar expresar con precisión sus necesidades en una forma que los desarrolladores puedan comprender. En casos extremos, los usuarios pueden no tener más que ideas vagas de lo que desean de un sistema de software. Esto no es en realidad achacable a los usuarios ni a los desarrolladores del sistema; más bien ocurre porque cada uno de los grupos no suele conocer suficientemente el dominio del otro. Los usuarios y los desarrolladores tienen perspectivas diferentes sobre la naturaleza del problema y realizan distintas suposiciones sobre la naturaleza de la solución. En la actualidad, aun cuando los usuarios tuvieran un conocimiento perfecto de sus necesidades, se dispone de pocos instrumentos para plasmar estos requisitos con exactitud. La forma habitual de expresar requisitos hoy en día es mediante grandes cantidades de texto, ocasionalmente acompañadas de unos pocos dibujos. Estos documentos son difíciles de comprender, están abiertos a diversas interpretaciones, y demasiado frecuentemente contienen elementos que invaden el diseño en lugar de limitarse a ser requisitos esenciales.

Una complicación adicional es que los requisitos de un sistema de software cambian frecuentemente durante su desarrollo, especialmente porque la mera existencia de un proyecto de desarrollo de software altera las reglas del pro-

* Mirada: cantidad muy grande, pero indefinida. (*N. del T.*)



La tarea del equipo de desarrollo de software es ofrecer ilusión de simplicidad.

blema. La observación de productos de las primeras fases, como documentos de diseño y prototipos, y la posterior utilización de un sistema cuando ya está instalado y operativo, son factores que llevan a los usuarios a comprender y articular mejor sus necesidades reales. Al mismo tiempo, este proceso ayuda a los desarrolladores a comprender el dominio del problema, capacitándoles para responder mejor a preguntas que iluminan los rincones oscuros del comportamiento deseado de un sistema.

Ya que un sistema grande de software es una inversión considerable, no es admisible el desechar un sistema existente cada vez que los requerimientos cambian. Esté o no previsto, los sistemas grandes tienden a evolucionar en el tiempo, situación que con frecuencia se etiqueta de forma incorrecta con el término *mantenimiento del software*. Siendo precisos, es mantenimiento cuando se corrigen errores; es *evolución* cuando se responde a requerimientos que cambian; es *conservación* cuando se siguen empleando medios extraordinarios para mantener en operación un elemento de software anticuado y decadente. Desafortunadamente, la realidad sugiere que un porcentaje exagerado de los recursos de desarrollo del software se emplean en conservación del mismo.

La dificultad de gestionar el proceso de desarrollo. La tarea fundamental del equipo de desarrollo de software es dar vida a una ilusión de simplicidad —para defender a los usuarios de esta vasta y a menudo arbitraria complejidad

externa. Ciertamente, el tamaño no es una gran virtud para un sistema de software. Se hace lo posible por escribir menos código mediante la invención de mecanismos ingeniosos y potentes que dan esta ilusión de simplicidad, así como mediante la reutilización de marcos estructurales de diseños y código ya existentes. Sin embargo, a veces es imposible eludir el mero volumen de los requerimientos de un sistema y se plantea la obligación de o bien escribir una enorme cantidad de nuevo software o bien reusar software existente de nuevas formas. No hace más de dos décadas que los programas en lenguaje ensamblador de tan sólo unos pocos miles de líneas de código ponían a prueba los límites de la capacidad de la ingeniería del software. Hoy en día no es extraño encontrar sistemas ya terminados cuyo tamaño se mide en cientos de miles, o incluso millones de líneas de código (y todo esto en un lenguaje de programación de alto nivel, además). Nadie puede comprender completamente tal sistema a título individual. Incluso si se descompone la implantación de formas significativas, aún hay que enfrentarse a cientos y a veces miles de módulos separados. Esta cantidad de trabajo exige la utilización de un equipo de desarrolladores, y de forma ideal se utiliza un equipo tan pequeño como sea posible. Sin embargo, da igual el tamaño, siempre hay retos considerables asociados con el desarrollo en equipo. Un mayor número de miembros implica una comunicación más compleja y por tanto una coordinación más difícil, particularmente si el equipo está disperso geográficamente, y esta situación no es nada excepcional en proyectos muy grandes. Con un equipo de desarrolladores, el reto clave de la dirección es siempre mantener una unidad e integridad en el diseño.

La flexibilidad que se puede alcanzar a través del software. Una compañía de construcción de edificios normalmente no gestiona su propia explotación forestal para cosechar árboles de los que obtener madera; es extremadamente inusual construir una acería en la obra con el fin de hacer vigas a medida para el nuevo edificio. Sin embargo, en la industria del software este comportamiento es frecuente. El software ofrece la flexibilidad máxima por lo que un desarrollador puede expresar casi cualquier clase de abstracción. Esta flexibilidad resulta ser una propiedad que seduce increíblemente, sin embargo, porque también empuja al desarrollador a construir por sí mismo prácticamente todos los bloques fundamentales sobre los que se apoyan estas abstracciones de más alto nivel. Mientras la industria de la construcción tiene normativas uniformes de construcción y estándares para la calidad de los materiales, existen muy pocos estándares similares en la industria del software. Como consecuencia, el desarrollo del software sigue siendo un negocio enormemente laborioso.

Los problemas de caracterizar el comportamiento de sistemas discretos. Si se lanza una pelota al aire, se puede predecir de manera fiable su trayectoria porque se sabe que, en condiciones normales, hay ciertas leyes físicas aplicables. Sería muy sorprendente si, simplemente por haber arrojado la pelota un poco más fuerte, se detuviera de repente a mitad del vuelo y saliese disparada hacia

arriba¹. En una simulación por software poco depurada del movimiento del balón, es bastante fácil que se produzca exactamente ese tipo de comportamiento.

En una aplicación de gran tamaño puede haber cientos o hasta miles de variables, así como más de un posible flujo del control. El conjunto de todas estas variables, sus valores actuales, y la dirección de ejecución y pila actual de cada uno de los procesos del sistema constituyen el estado actual de la aplicación. Al ejecutarse el software en computadores digitales, se tiene un sistema con estados discretos. En contraste, los sistemas analógicos como el movimiento de la pelota lanzada son sistemas continuos. Parnas sugiere que «cuando se afirma que un sistema se describe con una función continua, quiere decirse que no puede contener sorpresas ocultas. Pequeños cambios en las entradas siempre producirán cambios consecuentemente pequeños en las salidas» [4]. Por el contrario, los sistemas discretos por su propia naturaleza tienen un número finito de estados posibles; en sistemas grandes hay una explosión combinatoria que hace este número enorme. Se intenta diseñar los sistemas con una separación de intereses, de forma que el comportamiento de una parte del sistema tenga mínimo impacto en el comportamiento de otra parte del mismo. Sin embargo, sigue dándose el hecho de que las transiciones de fase entre estados discretos no pueden modelarse con funciones continuas. Todos los eventos externos a un sistema de software tienen la posibilidad de llevar a ese sistema a un nuevo estado, y más aún, la transición de estado a estado no siempre es determinista. En las peores circunstancias, un evento externo puede corromper el estado del sistema, porque sus diseñadores olvidaron tener en cuenta ciertas interacciones entre eventos. Por ejemplo, imagínese un avión comercial cuyos alerones y ambientación de cabina son manejados por un solo computador. No sería muy deseable que, como consecuencia de que un pasajero en el asiento 38J encendiera su luz, el avión hiciera inmediatamente un picado. En sistemas continuos este tipo de comportamiento sería improbable, pero en sistemas discretos todos los eventos externos pueden afectar a cualquier parte del estado interno del sistema. Ciertamente, esta es la razón primaria para probar a fondo los sistemas, pero para cualquier sistema que no sea trivial, es imposible hacer una prueba exhaustiva. Ya que no se dispone ni de las herramientas matemáticas ni de la capacidad intelectual necesarias para modelar el comportamiento completo de grandes sistemas discretos, hay que contentarse con un grado de confianza aceptable por lo que se refiere a su corrección.

¹ Realmente, incluso los sistemas continuos pueden mostrar un comportamiento muy complejo, por la presencia del caos. El caos introduce una aleatoriedad que hace imposible predecir con precisión el estado futuro de un sistema. Por ejemplo, dado el estado inicial de dos gotas de agua en la parte alta de un río, no puede predecirse exactamente dónde estará una con relación con la otra al llegar a la desembocadura. Se ha encontrado el caos en sistemas tan diversos como el clima, reacciones químicas, sistemas biológicos e incluso redes de computadores. Afortunadamente, parece haber un orden subyacente en todos los sistemas caóticos, en forma de patrones llamados *atractores*.

Las consecuencias de la complejidad ilimitada

«Cuanto más complejo sea el sistema, más abierto está al derrumbamiento total» [5]. Un constructor pensaría raramente en añadir un subsótano a un edificio ya construido de cien plantas; hacer tal cosa sería muy costoso e indudablemente sería una invitación al fracaso. Asombrosamente, los usuarios de sistemas de software casi nunca se lo piensan dos veces a la hora de solicitar cambios equivalentes. De todas formas, argumentan, es simplemente cosa de programar.

Nuestro fracaso en dominar la complejidad del software lleva a proyectos retrasados, que exceden el presupuesto, y que son deficientes respecto a los requerimientos fijados. A menudo se llama a esta situación la *crisis del software*, pero, francamente, una enfermedad que ha existido tanto tiempo debe considerarse normal. Tristemente, esta crisis se traduce en el desperdicio de recursos humanos —un bien de lo más precioso—, así como en una considerable pérdida de oportunidad. Simplemente, no hay suficientes buenos desarrolladores para crear todo el nuevo software que necesitan los usuarios. Peor aún, un porcentaje considerable del personal de desarrollo en cualquier organización debe muchas veces estar dedicado al mantenimiento o la conservación de software geriátrico. Dada la contribución tanto directa como indirecta del software a la base económica de la mayoría de los países industrializados, y considerando en qué medida el software puede amplificar la potencia del individuo, es inaceptable permitir que esta situación se mantenga.

¿Cómo puede cambiarse esta imagen desoladora? Ya que el problema subyacente surge de la complejidad inherente al software, la sugerencia que se plantea aquí es estudiar en primer lugar cómo se organizan los sistemas complejos en otras disciplinas. Realmente, si se echa una mirada al mundo que nos rodea, se observarán sistemas con éxito dentro de una complejidad que habrá que tener en cuenta. Algunos de esos sistemas son obras humanas, como el transbordador espacial, el túnel bajo el Canal de la Mancha, y grandes organizaciones como Microsoft o General Electric. De hecho aparecen en la naturaleza sistemas mucho más complejos aún, como el sistema circulatorio humano o la estructura de una planta.

1.2. La estructura de los sistemas complejos

Ejemplos de sistemas complejos

La estructura de un computador personal. Un computador personal es un dispositivo de complejidad moderada. Muchos de ellos se componen de los mismos elementos básicos: una unidad central de proceso (*Central Processing Unit*,

CPU)², un monitor, un teclado y alguna clase de dispositivo de almacenamiento secundario, habitualmente una unidad de disco flexible o disco duro. Puede tomarse cualquiera de estas partes y descomponerla aún más. Por ejemplo, una CPU suele incluir memoria primaria, una unidad aritmético-lógica (*Arithmetic/Logic Unit*, ALU), y un bus al que están conectados los dispositivos periféricos. Cada una de estas partes puede a su vez descomponerse más: una ALU puede dividirse en registros y lógica de control aleatorio, las cuales están constituidas por elementos aún más sencillos, como puertas NAND, inversores, etcétera.

Aquí se ve la naturaleza jerárquica de un sistema complejo. Un computador personal funciona correctamente sólo merced a la actividad colaboradora de cada una de sus partes principales. Juntas, esas partes antes separadas forman un todo lógico. Realmente, puede razonarse sobre cómo funciona un computador sólo gracias a que puede descomponerse en partes susceptibles de ser estudiadas por separado. Así, puede estudiarse el funcionamiento de un monitor independientemente de cómo funcione la unidad de disco duro. Del mismo modo, puede estudiarse la ALU sin tener en cuenta el subsistema de memoria principal.

Los sistemas complejos no sólo son jerárquicos, sino que los niveles de esta jerarquía representan diferentes niveles de abstracción, cada uno de los cuales se construye sobre el otro, y cada uno de los cuales es comprensible por sí mismo. A cada nivel de abstracción, se encuentra una serie de dispositivos que colaboran para proporcionar servicios a capas más altas. Se elige determinado nivel de abstracción para satisfacer necesidades particulares. Por ejemplo, si se intentase resolver un problema de temporización en la memoria primaria, sería correcto examinar la arquitectura del computador a nivel de puertas lógicas, pero este nivel de abstracción no sería apropiado si se tratase de encontrar dónde falla una aplicación de hoja de cálculo.

La estructura de plantas y animales. En botánica, los científicos buscan comprender las similitudes y diferencias entre plantas estudiando su morfología, es decir, su forma y estructura. Las plantas son complejos organismos multicelulares, y comportamientos tan complejos como la fotosíntesis o la transpiración tienen origen en la actividad cooperativa de varios sistemas orgánicos de la planta.

Las plantas están formadas por tres estructuras principales (raíces, tallos y hojas), y cada una de ellas tiene su propia estructura. Por ejemplo, las raíces constan de raíces principales, pelos radicales, el ápice y la cofia. De manera análoga, una sección transversal de una hoja revela su epidermis, mesofilo y tejido vascular. Cada una de estas estructuras se compone, a su vez, de una serie de células, y dentro de cada célula se encuentra otro nivel más de complejidad, que abarca elementos tales como cloroplastos, un núcleo, y así sucesivamente. Al

² Se mantienen las siglas en inglés (CPU y ALU) por estar su uso ampliamente difundido y arraigado en el mundo de la Informática. (N. del T.)

igual que en la estructura de un computador, las partes de una planta forman una jerarquía, y cada nivel de esta jerarquía conlleva su propia complejidad.

Todas las partes al mismo nivel de abstracción interactúan de formas perfectamente definidas. Por ejemplo, al más alto nivel de abstracción, las raíces son responsables de absorber agua y minerales del suelo. Las raíces interactúan con los tallos, que transportan estas materias primas hasta las hojas. Las hojas por su parte utilizan el agua y los minerales proporcionados por los tallos para producir alimentos mediante la fotosíntesis.

Siempre hay fronteras claras entre el exterior y el interior de determinado nivel. Por ejemplo, se puede afirmar que las partes de una hoja trabajan juntas para proporcionar la funcionalidad de la misma como un todo, y además tienen una interacción pequeña o indirecta con las partes elementales de las raíces. En palabras más simples, hay una clara separación de intereses entre las partes a diferentes niveles de abstracción.

En un computador se encuentra que las puertas NAND se usan tanto en el diseño de la CPU como en la unidad de disco duro. Igualmente, hay una notable cantidad de similitudes que abarcan a todas las partes de la jerarquía estructural de una planta. Esta es la forma que tiene Dios de conseguir una economía de expresión. Por ejemplo, las células sirven como bloques básicos de construcción en todas las estructuras de las plantas; en última instancia, las raíces, los tallos y las hojas de las plantas están compuestos, todos ellos, por células. Sin embargo, aunque cada uno de esos elementos primitivos es en realidad una célula, hay muchos tipos de célula diferentes. Por ejemplo, hay células con o sin cloroplastos, células con paredes impermeables al agua y células con paredes permeables, e incluso células vivas y células muertas.

En el estudio de la morfología de una planta no se encuentran partes individuales que sean responsables cada una de un único y pequeño paso en un solo proceso más grande, tal como la fotosíntesis. De hecho, no hay partes centralizadas que coordinen directamente las actividades de las partes de niveles inferiores. En lugar de esto, se encuentran partes separadas que actúan como agentes independientes, cada uno de los cuales exhibe algún comportamiento bastante complejo, y cada uno de los cuales contribuye a muchas funciones de nivel superior. Sólo a través de la cooperación mutua de colecciones significativas de estos agentes se ve la funcionalidad de nivel superior de una planta. La ciencia de la complejidad llama a esto *comportamiento emergente*: El comportamiento del todo es mayor que la suma de sus partes [6].

Contemplando brevemente el campo de la zoología, se advierte que los animales multicelulares muestran una estructura jerárquica similar a la de las plantas: colecciones de células forman tejidos, los tejidos trabajan juntos como órganos, grupos de órganos definen sistemas (tales como el digestivo), y así sucesivamente. No puede evitarse el notar de nuevo la tremenda economía de expresión de Dios: el bloque de construcción fundamental de toda la materia animal es la célula, de la misma forma que la célula es la estructura elemental de la vida de cualquier planta. Por supuesto, hay diferencias entre ambas. Por ejemplo, las células vegetales están encerradas entre paredes rígidas de celulosa,

pero las células animales no lo están. A pesar de estas diferencias, sin embargo, ambas estructuras son innegablemente células. Este es un ejemplo de mecanismos comunes entre distintos dominios.

La vida vegetal y la vida animal también comparten una serie de mecanismos por encima del nivel celular. Por ejemplo, ambos usan una forma de sistema vascular para transportar nutrientes dentro del organismo, y ambos muestran diferenciación sexual entre miembros de la misma especie.

La estructura de la materia. El estudio de campos tan diversos como la astronomía y la física nuclear proporciona muchos otros ejemplos de sistemas increíblemente complejos. Abarcando estas dos disciplinas, se encuentra otra jerarquía estructural más. Los astrónomos estudian las galaxias que se disponen en cúmulos, y las estrellas, planetas y distintos residuos son los elementos constitutivos de las galaxias. De la misma forma, los físicos nucleares están interesados en una jerarquía estructural, pero a una escala completamente diferente. Los átomos están hechos de electrones, protones y neutrones; los electrones parecen ser partículas elementales, pero los protones, neutrones y otras partículas están formados por componentes más básicos llamados *quarks*.

Una vez más se ve que esta vasta jerarquía está unificada por una gran similitud en forma de mecanismos compartidos. Específicamente, parece haber sólo cuatro tipos distintos de fuerzas actuando en el universo: gravedad, interacción electromagnética, interacción fuerte e interacción débil. Muchas leyes de la física que involucran a estas fuerzas elementales, tales como las leyes de conservación de la energía y del momento, se aplican tanto a las galaxias como a los quarks.

La estructura de las instituciones sociales. Como ejemplo final de sistemas complejos, obsérvese la estructura de las instituciones sociales. Los grupos de personas se reúnen para realizar tareas que no pueden ser hechas por individuos. Algunas organizaciones son transitorias, y algunas perduran durante generaciones. A medida que las organizaciones crecen, se ve emergir una jerarquía distinta. Las multinacionales contienen compañías, que a su vez están compuestas por divisiones, que a su vez contienen delegaciones, que a su vez contienen oficinas locales, y así sucesivamente. Si la organización perdura, las fronteras entre estas partes pueden cambiar, y a lo largo del tiempo, puede surgir una nueva y más estable jerarquía.

Las relaciones entre las distintas partes de una organización grande son las mismas que las que se han observado entre los componentes de un computador, o una planta, o incluso una galaxia. Específicamente, el grado de interacción entre empleados dentro de una sola oficina es mayor que entre empleados de oficinas diferentes. Un funcionario encargado del correo no suele interactuar con el director general de una compañía, pero lo hace a menudo con otras personas que trabajan en Recepción. También aquí estos diferentes niveles están unificados por mecanismos comunes. Tanto el funcionario como el director ge-

neral reciben su sueldo de la misma organización financiera, y ambos comparten una infraestructura común, como el sistema telefónico de la compañía, para llevar a cabo sus cometidos.

Los cinco atributos de un sistema complejo

Partiendo de lo visto hasta el momento, se concluye que hay cinco atributos comunes a todos los sistemas complejos. Basándose en el trabajo de Simon y Ando, Courtois sugiere lo siguiente:

1. «Frecuentemente, la complejidad toma la forma de una jerarquía, por lo cual un sistema complejo se compone de subsistemas relacionados que tienen a su vez sus propios subsistemas, y así sucesivamente, hasta que se alcanza algún nivel ínfimo de componentes elementales»[7].

Simon apunta que «el hecho de que muchos sistemas complejos tengan una estructura jerárquica y casi descomponible es un factor importante de ayuda que nos capacita para comprender, describir e incluso «ver» estos sistemas y sus partes» [8]. De hecho, prácticamente sólo se puede comprender aquellos sistemas que tienen una estructura jerárquica.

Es importante notar que la arquitectura de un sistema complejo es función de sus componentes tanto como de las relaciones jerárquicas entre esos componentes. Como observa Rechtin, «Todos los sistemas tienen subsistemas y todos los sistemas son parte de sistemas mayores... El valor añadido por un sistema debe proceder de las relaciones entre las partes, no de las partes por sí mismas»[9].

En lo que se refiere a la naturaleza de los componentes constitutivos de un sistema complejo, la experiencia sugiere que

2. La elección de qué componentes de un sistema son primitivos es relativamente arbitraria y queda en gran medida a decisión del observador.

Lo que es primitivo para un observador puede estar a un nivel de abstracción mucho más alto para otro.

Simon llama a los sistemas jerárquicos *descomponibles*, porque pueden dividirse en partes identificables; los llama *casi descomponibles*, porque sus partes no son completamente independientes. Esto conduce a otro atributo común a todos los sistemas complejos:

3. «Los enlaces internos de los componentes suelen ser más fuertes que los enlaces entre componentes. Este hecho tiene el efecto de separar la dinámica de alta frecuencia de los componentes —que involucra a la estructura interna de los mismos— de la dinámica de baja frecuencia —que involucra la interacción entre los componentes»[10].

Esta diferencia entre interacciones intracomponentes e intercomponentes proporciona una separación clara de intereses entre las diferentes partes de un sistema, posibilitando el estudio de cada parte de forma relativamente aislada.

Como se ha examinado ya, muchos sistemas complejos se han implantado con economía de expresión. Así, Simon apunta que

4. Los sistemas jerárquicos están compuestos usualmente de sólo unas pocas clases diferentes de subsistemas en varias combinaciones y disposiciones» [11].

En otras palabras, los sistemas complejos tienen patrones comunes. Estos patrones pueden conllevar la reutilización de componentes pequeños, tales como las células que se encuentran en plantas y animales, o de estructuras mayores, como los sistemas vasculares que también aparecen en ambas clases de seres vivos.

Anteriormente se señaló que los sistemas complejos tienden a evolucionar en el tiempo. Como sugiere Simon, «los sistemas complejos evolucionarán a partir de sistemas simples con mucha mayor rapidez si hay formas intermedias estables que si no las hay» [12]. En términos más dramáticos, Gall sostiene que

5. «Se encontrará invariablemente que un sistema complejo que funciona ha evolucionado de un sistema simple que funcionaba... Un sistema complejo diseñado desde cero nunca funciona y no puede parchearse para conseguir que lo haga. Hay que volver a empezar, partiendo de un sistema simple que funcione» [13].

A medida que los sistemas evolucionan, objetos que en su momento se consideraron complejos se convierten en objetos primitivos sobre los que se construyen sistemas más complejos aún. Es más, nunca se pueden crear estos objetos primitivos de forma correcta la primera vez: hay que utilizarlos antes en un contexto, y mejorarlos entonces con el tiempo cuando se aprende más sobre el comportamiento real del sistema.

Complejidad organizada y desorganizada

La forma canónica de un sistema complejo. El descubrimiento de abstracciones y mecanismos comunes facilita en gran medida la comprensión de los sistemas complejos. Por ejemplo, sólo con unos pocos minutos de instrucción, un piloto experimentado puede entrar en un avión multimotor de propulsión a chorro en el que nunca había volado, y llevar el vehículo con seguridad. Habiendo reconocido las propiedades comunes a todos los aviones de ese estilo, como el funcionamiento del timón, alerones y palanca de gas, el piloto necesita sobre todo aprender qué propiedades son únicas a ese modelo concreto. Si el piloto sabe ya pilotar un avión dado, es mucho más fácil saber pilotar uno parecido.

Este ejemplo sugiere que se ha venido utilizando el término *jerarquía* de forma un tanto imprecisa. La mayoría de los sistemas interesantes no contienen una sola jerarquía; en lugar de eso, se encuentra que en un solo sistema complejo suelen estar presentes muchas jerarquías diferentes. Por ejemplo, un avión puede estudiarse descomponiéndolo en su sistema de propulsión, sistema de

control de vuelo, etc. Esta descomposición representa una jerarquía estructural o «parte-de» (*part of*). Pero puede diseccionarse el sistema por una vía completamente distinta. Por ejemplo, un motor *turbofan* (turboventilador) es un tipo específico de motor de propulsión a chorro (*jet*), y un Pratt and Whitney TF30 es un tipo específico de motor turbofan. Dicho de otro modo, un motor de propulsión a chorro representa una generalización de las propiedades comunes a todos los tipos de motor de propulsión a chorro; un motor *turbofan* (turboventilador) no es más que un tipo especializado de motor de propulsión a chorro, con propiedades que lo distinguen, por ejemplo, de los motores *ramjet* (estatorreactor).

Esta segunda jerarquía representa una jerarquía de tipos «es-un» (*is a*). A través de la experiencia, se ha encontrado que es esencial ver un sistema desde ambas perspectivas, estudiando tanto su jerarquía «de tipos» como su jerarquía «parte-de» (*part of*). Por razones que se aclararán en el siguiente capítulo, se llama a esas jerarquías *estructura de clases* y *estructura de objetos*, respectivamente³.

Combinando el concepto de la estructura de clases y objetos con los cinco atributos de un sistema complejo, se encuentra que prácticamente todos los sistemas complejos adoptan una misma forma (canónica), como se muestra en la Figura 1.1. Aquí se ven las dos jerarquías «ortogonales» del sistema: su estructura de clases y su estructura de objetos. Cada jerarquía está dividida en capas, con las clases y objetos más abstractos construidos a partir de otros más primitivos. La elección de las clases y objetos primitivos depende del problema que se maneja. Sobre todo entre las partes de la estructura de objetos, existen colaboraciones estrechas entre objetos del mismo nivel de abstracción. Una mirada al interior de cualquier nivel dado revela un nuevo nivel de complejidad. Nótese también que la estructura de clases y la estructura de objetos no son del todo independientes; antes bien, cada objeto de la estructura de objetos representa una instancia específica de alguna clase. Como sugiere la figura, existen por lo general muchos más objetos que clases en un sistema complejo. Así, mostrando la jerarquía «parte de» y la jerarquía «es un», se expone de forma explícita la redundancia del sistema que se está considerando. Si no se manifiesta la estructura de clases de un sistema, habría que duplicar el conocimiento acerca de las propiedades de cada parte individual. Con la inclusión de la estructura de clases, se capturan esas propiedades comunes en un solo lugar.

La experiencia hace pensar que los sistemas complejos de software con más éxito son aquellos cuyos diseños incluyen de forma explícita una estructura de clases y objetos bien construida y que posee los cinco atributos de los sistemas complejos que se describieron en la sección anterior. A fin de que no se soslaye la importancia de esta observación, seamos incluso más directos: sólo a título muy excepcional se encuentran sistemas de software que se entreguen a tiempo,

³ Los sistemas complejos de software contienen otros tipos de jerarquía. Tienen particular importancia su estructura de módulos, que describe las relaciones entre los componentes físicos del sistema, y la jerarquía de procesos, que describe las relaciones entre los componentes dinámicos del sistema.

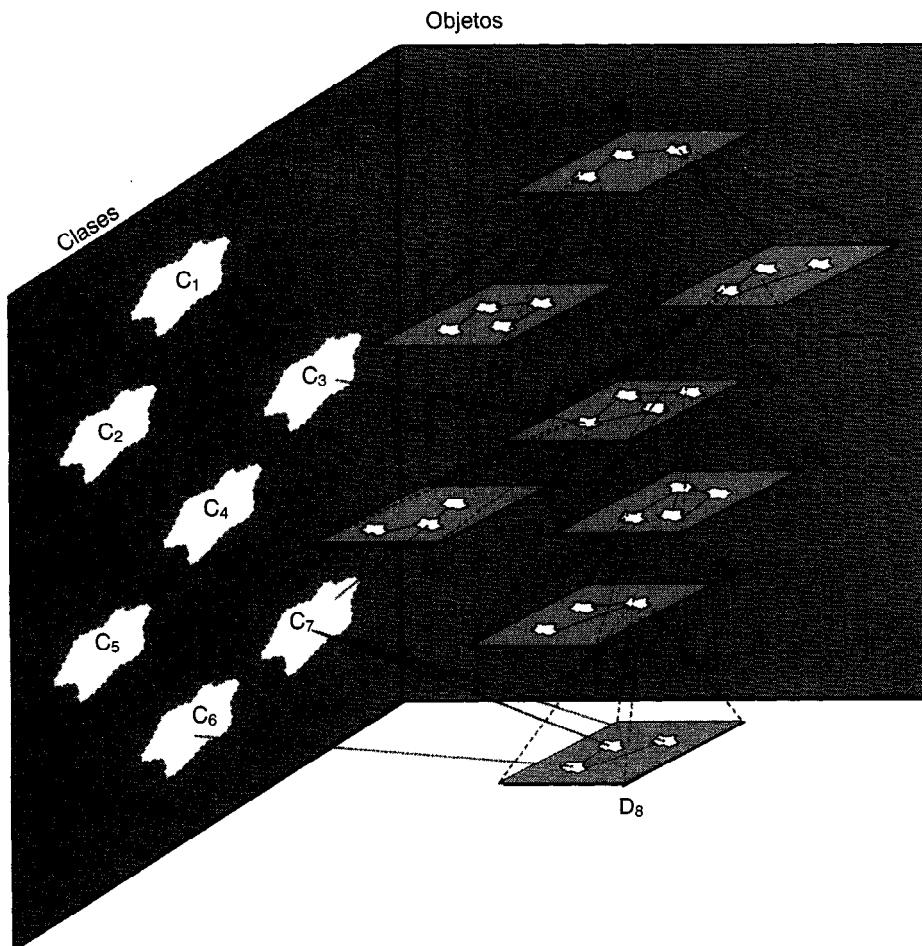


Figura 1.1. La forma canónica de un sistema complejo.

dentro del presupuesto, y que satisfagan sus requerimientos, a menos que se hayan diseñado teniendo en cuenta estos factores.

De forma conjunta, nos referimos a la estructura de clases y de objetos de un sistema como su *arquitectura*.

Las limitaciones de la capacidad humana para enfrentarse a la complejidad. Si se sabe que el diseño de sistemas de software complejos debería ser de esta forma, ¿por qué sigue habiendo entonces serios problemas para desarrollarlos con éxito? Tal como se discute en el próximo capítulo, este concepto de la complejidad organizada del software (cuyas directrices denominamos como *modelo de objetos*) es relativamente nuevo. Sin embargo, hay además otro fac-

tor dominante: las limitaciones fundamentales de la capacidad humana para tratar la complejidad.

Cuando se comienza a analizar por primera vez un sistema complejo de software, aparecen muchas partes que deben interactuar de múltiples e intrincadas formas, percibiéndose pocos elementos comunes entre las partes o sus interacciones: este es un ejemplo de complejidad desorganizada. Al trabajar para introducir organización en esta complejidad a través del proceso de diseño, hay que pensar en muchas cosas a la vez. Por ejemplo, en un sistema de control de tráfico aéreo, hay que tratar con el estado de muchos aviones diferentes al mismo tiempo, implicando a propiedades tales como su posición, velocidad y rumbo. Sobre todo en el caso de sistemas discretos, hay que enfrentarse con un *espacio de estados* claramente grande, intrincado y a veces no determinista. Por desgracia, es completamente imposible que una sola persona pueda seguir la pista a todos estos detalles simultáneamente. Experimentos psicológicos, como los de Miller, revelan que el máximo número de bloques de información que un individuo puede comprender de forma simultánea es del orden de siete más o menos dos [14]. Esta capacidad de canal parece estar relacionada con la capacidad de memoria a corto plazo. Simon añade que la velocidad de proceso es un factor limitador: la mente necesita alrededor de cinco segundos para aceptar un nuevo bloque de información [15].

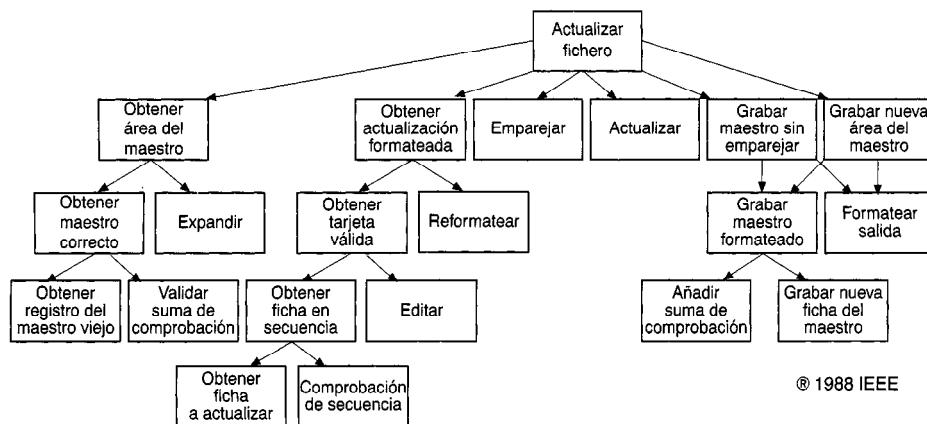
De esta forma se plantea un dilema fundamental. La complejidad de los sistemas de software que hay que desarrollar se va incrementando, pero existen limitaciones básicas sobre la habilidad humana para enfrentarse a esta complejidad. ¿Cómo resolver entonces este problema?

1.3. Imponiendo orden al caos

El papel (rol) de la descomposición

Como sugiere Dijkstra, «La técnica de dominar la complejidad se conoce desde tiempos remotos: *divide et impera* (divide y vencerás)» [16]. Cuando se diseña un sistema de software complejo, es esencial descomponerlo en partes más y más pequeñas, cada una de las cuales se puede refinar entonces de forma independiente. De este modo se satisface la restricción fundamental que existe sobre la capacidad de canal de la comprensión humana: para entender un nivel dado de un sistema, basta con comprender unas pocas partes (no necesariamente todas) a la vez. Realmente, como observa Parnas, la descomposición inteligente ataca directamente la complejidad inherente al software forzando una división del espacio de estados del sistema [17].

Descomposición algorítmica. Casi todos los informáticos han sido adiestrados en el dogma del diseño estructurado descendente, y por eso suelen afrontar



© 1988 IEEE

Figura 1.2. Descomposición algorítmica.

la descomposición como una simple cuestión de descomposición algorítmica, en la que cada módulo del sistema representa a un paso importante de algún proceso global. La Figura 1.2 es un ejemplo de uno de los productos del diseño estructurado, un diagrama estructural que muestra las relaciones entre varios elementos funcionales de la solución. Este gráfico en concreto ilustra parte del diseño de un programa que actualiza el contenido de un archivo maestro. Se generó automáticamente a partir de un diagrama de flujo de datos mediante un sistema experto que contiene las reglas del diseño estructurado [18].

Descomposición orientada a objetos. Se sugiere aquí que existe una posible descomposición alternativa para el mismo problema. En la Figura 1.3, se ha descompuesto el sistema de acuerdo con las abstracciones clave del dominio del problema. En vez de descomponer el problema en pasos como *Obtener actualización formateada* y *Añadir sumas de comprobación*, se han identificado objetos como *Archivo Maestro* y *Suma de Comprobación*, que se derivan directamente del vocabulario del dominio del problema.

Aunque ambos diseños resuelven el mismo problema, lo hacen de formas bastante distintas. En esta segunda descomposición, se ha visto el mundo como un conjunto de agentes autónomos que colaboran para llevar a cabo algún comportamiento de nivel superior. Así *Obtener actualización formateada* no existe como un algoritmo independiente; en lugar de eso, es una operación asociada con el objeto *Archivo de Actualizaciones*. La llamada a esta operación provoca la creación de otro objeto, *Actualización de Tarjeta*. De este modo, cada objeto en esta solución contiene su propio comportamiento único, y cada uno modela a algún objeto del mundo real. Desde esta perspectiva, un objeto no es más que una entidad tangible que muestra un comportamiento bien definido. Los objetos hacen cosas, y a los objetos se les pide que hagan lo que hacen en-

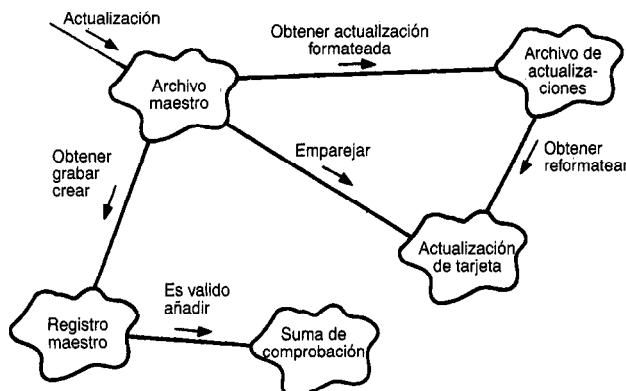


Figura 1.3. Descomposición orientada a objetos.

viéndoles mensajes. Puesto que esta descomposición está basada en objetos y no en algoritmos, se le llama descomposición *orientada a objetos*.

Descomposición algorítmica versus descomposición orientada a objetos. ¿Cuál es la forma correcta de descomponer un sistema complejo, por algoritmos o por objetos? En realidad, esta es una pregunta con truco, porque la respuesta adecuada es que ambas visiones son importantes: la visión algorítmica enfatiza el orden de los eventos, y la visión orientada a objetos resalta los agentes que o bien causan acciones o bien son sujetos de estas acciones. Sin embargo, el hecho es que no se puede construir un sistema complejo de las dos formas a la vez, porque son vistas completamente perpendiculares⁴. Hay que comenzar a descomponer un sistema sea por algoritmos o por objetos, y entonces utilizar la estructura resultante como marco de referencia para expresar la otra perspectiva.

Categorías de métodos de diseño

Resulta útil distinguir entre los términos *método* y *metodología*. Un *método* es un proceso disciplinado para generar un conjunto de modelos que describen varios aspectos de un sistema de software en desarrollo, utilizando alguna notación bien definida. Una *metodología* es una colección de métodos aplicados a lo largo del ciclo de vida del desarrollo del software y unificados por alguna aproximación

⁴ Langdom sugiere que esta «ortogonalidad» se ha estudiado desde tiempos inmemoriales. Según afirma, «C. H. Waddington ha observado que la dualidad de visiones se remonta hasta los antiguos griegos. Demócrito propuso una visión pasiva, que afirmaba que el mundo estaba compuesto por una materia llamada átomos. La visión de Demócrito coloca a las cosas en el centro de atención. Por otra parte, el portavoz clásico de la visión activa es Heráclito, que enfatizó la noción de proceso» [34].

general o filosófica. Los métodos son importantes por varias razones. En primer lugar, inculcan una disciplina en el desarrollo de sistemas de software complejos. Definen los productos que sirven como vehículo común para la comunicación entre los miembros de un equipo de desarrollo. Además, los métodos definen los hitos que necesita la dirección para medir el progreso y gestionar el riesgo.

Los métodos han evolucionado como respuesta a la complejidad creciente de los sistemas de software. En los principios de la informática, simplemente no se escribían programas grandes, porque la capacidad de las máquinas era muy limitada. Las restricciones dominantes en la construcción de sistemas se debían sobre todo al hardware: las máquinas tenían poca memoria principal, los programas tenían que enfrentarse a latencias considerables en dispositivos de almacenamiento secundario como tambores magnéticos, y los procesadores tenían tiempos de ciclo medidos en cientos de microsegundos. En los años sesenta y setenta la economía de la computación comenzó a cambiar de manera dramática a causa del descenso de los costes del hardware y el aumento de las capacidades de los computadores. Como consecuencia, resultaba más apetecible y finalmente más económico automatizar más y más aplicaciones de creciente complejidad. Los lenguajes de programación de alto nivel entraron en escena como herramientas importantes. Tales lenguajes mejoraron la productividad del desarrollador individual y del equipo de desarrollo en su conjunto, lo que irónicamente presionó a los informáticos a crear sistemas aún más complejos.

Durante los sesenta y los setenta se propusieron muchos métodos de diseño para enfrentarse a esta complejidad en aumento. El más influyente fue el diseño estructurado descendente, también conocido como *diseño compuesto*. Este método fue influido directamente por la topología de lenguajes de alto nivel tradicionales, como FORTRAN y COBOL. En estos lenguajes, la unidad fundamental de descomposición es el subprograma, y el programa resultante toma la forma de un árbol en el que los subprogramas realizan su función llamando a otros subprogramas. Este es exactamente el enfoque del diseño estructurado descendente: se aplica descomposición algorítmica para fragmentar un problema grande en pasos más pequeños.

Desde los sesenta y los setenta han aparecido computadores de capacidad muchísimo mayor. El valor del diseño estructurado no ha cambiado, pero como observa Stein, «La programación estructurada parece derrumbarse cuando las aplicaciones superan alrededor de 100.000 líneas de código» [19]. Más recientemente, se han propuesto docenas de métodos de diseño, muchos de los cuales se inventaron para resolver las deficiencias detectadas en el diseño estructurado descendente. Los métodos de diseño más interesantes y de mayor éxito han sido catalogados por Peters [20] y Yau y Tsai [21], y en una exhaustiva inspección por Teledyne—Brown Engineering [22]. Quizá no sorprenda el hecho de que muchos de ellos son en gran medida variaciones sobre un tema similar. En realidad, como sugiere Sommerville, la mayoría pueden catalogarse como pertenecientes a uno de los tres tipos siguientes [23]:

- Diseño estructurado descendente.
- Diseño dirigido por los datos (*data-driven*).
- Diseño orientado a objetos.

El diseño estructurado descendente está ejemplificado en el trabajo de Yourdon y Constantine [24], Myers [25] y Page—Jones [26]. Los fundamentos de este

método se derivan del trabajo de Wirth [27, 28] y Dahl, Dijkstra y Hoare [29]; se encuentra una importante variación del diseño estructurado en el método de Mills, Linger y Hevner [30]. Todas estas variaciones aplican descomposición algorítmica. Probablemente se ha escrito más software con estos métodos de diseño que con cualquier otro. Sin embargo, el diseño estructurado no contempla los problemas de abstracción de datos y ocultación de información, ni proporciona medios adecuados para tratar la concurrencia. El diseño estructurado no responde bien ante el cambio de tamaño cuando se aplica a sistemas extremadamente complejos, y es muy inapropiado para su uso con lenguajes de programación basados en objetos y orientados a objetos.

El diseño dirigido por los datos encuentra su mejor exponente en los primeros trabajos de Jackson [31, 32] y los métodos de Warnier y Orr [33]. En este método, la estructura de un sistema de software se deduce de la correspondencia entre las entradas y las salidas del sistema. Al igual que en el diseño estructurado, el diseño dirigido por los datos se ha aplicado con éxito a una serie de dominios complejos, particularmente sistemas de gestión de la información, que implican relaciones directas entre las entradas y las salidas del sistema, pero que no requieren demasiada atención hacia eventos en los que el tiempo sea crítico.

El diseño orientado a objetos es el método que se introduce en este libro. Su concepto subyacente es que se debería modelar los sistemas de software como colecciones de objetos que cooperan, tratando los objetos individuales como instancias de una clase que está dentro de una jerarquía de clases. El diseño orientado a objetos refleja directamente la topología de lenguajes de programación de alto nivel más recientes, como Smalltalk, Object Pascal, C++, Common Lisp Object System (CLOS) y Ada.

Nuestra experiencia nos lleva a aplicar en primer lugar el criterio *orientado a objetos* porque esta aproximación es mejor a la hora de servir de ayuda para organizar la complejidad innata de los sistemas de software, al igual que ha servido de ayuda para describir la complejidad organizada de sistemas complejos tan diversos como los computadores, plantas, galaxias o grandes instituciones sociales. Tal como se discutirá con más detalle en los Capítulos 2 y 7, la descomposición orientada a objetos tiene una serie de ventajas altamente significativas sobre la descomposición algorítmica. La descomposición orientada a objetos produce sistemas más pequeños a través de la reutilización de mecanismos comunes, proporcionando así una importante economía de expresión. Los sistemas orientados a objetos son también más resistentes al cambio y por tanto están mejor preparados para evolucionar en el tiempo, porque su diseño está basado en formas intermedias estables. En realidad, la descomposición orientada a objetos reduce en gran medida el riesgo que representa construir sistemas de software complejos, porque están diseñados para evolucionar de forma incremental partiendo de sistemas más pequeños en los que ya se tiene confianza. Es más, la descomposición orientada a objetos resuelve directamente la complejidad innata del software ayudando a tomar decisiones inteligentes respecto a la separación de intereses en un gran espacio de estados.

Los Capítulos 8 al 12 demuestran estos beneficios a través de varias aplicaciones completas, extraídas de un conjunto variado de dominios de problemas. En las notas complementarias de este capítulo se compara y contrasta aún más la visión orientada a objetos con aproximaciones más tradicionales al diseño.

El papel (rol) de la abstracción

Anteriormente se ha hecho referencia a los experimentos de Miller, a partir de los cuales se concluyó que un individuo puede comprender sólo alrededor de siete, más/menos dos, bloques de información simultáneamente. Este número parece no depender del contenido de la información. Como el propio Miller observa, «El alcance del juicio absoluto y el alcance de la memoria inmediata imponen severas limitaciones a la cantidad de información que somos capaces de recibir, procesar y recordar. Organizando el estímulo percibido simultáneamente en varias dimensiones y sucesivamente en una secuencia de bloques, conseguimos superar... este cuello de botella de la información» [35]. En palabras actuales, llamamos a este proceso *troceo* o *abstracción*.

Según dice Wulf, «(los humanos) hemos desarrollado una técnica excepcionalmente potente para enfrentarnos a la complejidad. Realizamos abstracciones. Incapaces de dominar en su totalidad un objeto complejo, decidimos ignorar sus detalles no esenciales, tratando en su lugar con el modelo generalizado e idealizado del objeto» [36]. Por ejemplo, cuando se estudia cómo funciona la fotosíntesis en una planta, se puede centrar la atención sobre las reacciones químicas en ciertas células de una hoja, e ignorar todas las demás partes, como las raíces y tallos. Aún persiste la restricción sobre el número de cosas que se pueden comprender al mismo tiempo, pero a través de la abstracción se utilizan bloques de información de contenido semántico cada vez mayor. Esto es especialmente cierto si se adopta una visión del mundo orientada a objetos, porque los objetos, como abstracciones de entidades del mundo real, representan un agrupamiento de información particularmente denso y cohesivo. El Capítulo 2 examina el significado de la abstracción con mucho mayor detalle.

El papel (rol) de la jerarquía

Otra forma de incrementar el contenido semántico de bloques de información individuales es mediante el reconocimiento explícito de las jerarquías de clases y objetos dentro de un sistema de software complejo. La estructura de objetos es importante porque ilustra cómo diferentes objetos colaboran entre sí a través de patrones de interacción llamados *mecanismos*. La estructura de clases no es menos importante, porque resalta la estructura y comportamiento comunes en el interior de un sistema. Así, en vez de estudiar cada una de las células fotosintéticas de la hoja de una planta específica, es suficiente estudiar una de esas células, porque se espera que todas las demás se comporten de modo similar.

Aunque se puede tratar como distinta cada instancia de un determinado tipo de objeto, puede asumirse que comparte la misma conducta que todas las demás instancias de ese mismo tipo de objeto. Clasificando objetos en grupos de abstracciones relacionadas (por ejemplo, clases de células vegetales frente a clases de células animales) se llega a una distinción explícita de las propiedades comunes y distintas entre diferentes objetos, lo que posteriormente ayudará a dominar su complejidad inherente [37].

La identificación de las jerarquías en un sistema de software complejo no suele ser fácil, porque requiere que se descubran patrones entre muchos objetos, cada uno de los cuales puede incluir algún comportamiento tremadamente complicado. Una vez que se han expuesto estas jerarquías, sin embargo, la estructura de un sistema complejo, y a su vez nuestra comprensión de la misma, experimenta una gran simplificación. El Capítulo 3 considera en detalle la naturaleza de las jerarquías de clases y objetos, y el Capítulo 4 describe técnicas que facilitan la identificación de esos patrones.

1.4. Del diseño de sistemas complejos

La ingeniería como ciencia y como arte

La práctica de cualquier disciplina de ingeniería —sea civil, mecánica, química, eléctrica o informática— involucra elementos tanto de ciencia como de arte. Como Petroski elocuentemente afirma, «la concepción de un diseño para una nueva estructura puede involucrar un salto de la imaginación y una síntesis de experiencia y conocimiento tan grandes como el que requiere cualquier artista para plasmar su obra en una tela o en un papel. Y una vez que el ingeniero ha articulado ese diseño como artista, debe analizarlo como científico aplicando el método científico con tanto rigor como cualquier científico lo haría» [38].

El papel del ingeniero como artista es particularmente difícil cuando la tarea es diseñar un sistema completamente nuevo. Francamente, es la circunstancia más habitual en la ingeniería del software. Especialmente en el caso de sistemas reactivos y sistemas de dirección y control, se pide con frecuencia que se escriba software para un conjunto de requerimientos completamente exclusivo, muchas veces para ser ejecutado en una configuración de procesadores que se ha construido específicamente para este sistema. En otros casos, como en la creación de marcos estructurales, herramientas para investigación en inteligencia artificial, o incluso sistemas de gestión de la información, puede ser que exista un entorno de destino estable y bien definido, pero los requerimientos pueden llevar al límite a la tecnología del software en una o más dimensiones. Por ejemplo, puede haberse recibido la solicitud de crear sistemas más rápidos, de mayor capacidad, o de una funcionalidad radicalmente mejorada. En todas estas situaciones, se intenta utilizar abstracciones y mecanismos ya probados (lo que Si-

mon llamaba «formas intermedias estables») como fundamento sobre el que construir nuevos sistemas complejos. En presencia de una gran biblioteca de componentes software reusables, el ingeniero del software debe ensamblar estas partes de formas innovadoras para satisfacer los requerimientos expresados e implícitos, al igual que el pintor o el músico deben ampliar los límites impuestos por su medio. Desgraciadamente, el ingeniero de software sólo dispone de bibliotecas tan ricas en muy contadas ocasiones, lo habitual es que deba echar mano de un conjunto relativamente primitivo de utilidades.

El significado del diseño

En todas las ingenierías, el diseño es la aproximación disciplinada que se utiliza para inventar una solución para algún problema, suministrando así un camino desde los requerimientos hasta la implantación. En el contexto de la ingeniería del software, Mostow sugiere que el propósito del diseño es construir un sistema que

- «Satisface determinada (quizás informal) especificación funcional.
- Se ajusta a las limitaciones impuestas por el medio de destino.
- Respeta requisitos implícitos o explícitos sobre rendimiento y utilización de recursos.
- Satisface criterios de diseño implícitos o explícitos sobre la forma del artefacto.
- Satisface restricciones sobre el propio proceso de diseño, tales como su longitud o coste, o las herramientas disponibles para realizar el diseño» [40].

Como sugiere Stroustrup, «el propósito del diseño es crear una estructura interna (a veces llamada también arquitectura) clara y relativamente simple... Un diseño es el producto final del proceso de diseño» [41]. El diseño conlleva un balance entre un conjunto de requisitos que compiten. Los productos del diseño son modelos que permiten razonar sobre las estructuras, hacer concesiones cuando los requisitos entran en conflicto y, en general, proporcionar un anteproyecto para la implementación.

La importancia de construir un modelo. La construcción de modelos goza de amplia aceptación entre todas las disciplinas de ingeniería, sobre todo porque construir un modelo es algo atractivo respecto a los principios de descomposición, abstracción y jerarquía [42]. Cada modelo dentro de un diseño describe un aspecto específico del sistema que se está considerando. En la medida de lo posible, se busca construir nuevos modelos sobre viejos modelos en los que ya se tiene cierta confianza. Los modelos dan la oportunidad de fallar en condiciones controladas. Se evalúa cada modelo bajo situaciones tanto previstas como improbables, y entonces se lo modifica cuando falla para que se comporte del modo esperado o deseado.

Se ha observado que, con objeto de expresar todas las sutilidades de un sistema complejo, hay que utilizar más de un solo tipo de modelo. Por ejemplo, cuando se diseña un computador de una sola placa, un ingeniero electrónico debe tomar en consideración la visión del sistema a nivel de puertas lógicas, así como la distribución física de los circuitos integrados en la placa. Esta visión a nivel de puertas forma una imagen lógica del diseño del sistema, que ayuda al ingeniero a razonar sobre el comportamiento cooperativo de las puertas. La distribución en placa representa el empaquetamiento físico de esas puertas, restringido por el tamaño de la placa, energía disponible y los tipos de circuito integrado que existen. Desde este punto de vista, el ingeniero puede razonar independientemente sobre factores como la disipación de calor y las dificultades de fabricación. El diseñador de la placa debe considerar también aspectos tanto estáticos como dinámicos del sistema en construcción. Así, el ingeniero electrónico utiliza diagramas que muestran las conexiones estáticas entre puertas individuales, y también cronogramas que muestran el comportamiento de esas puertas a lo largo del tiempo. El ingeniero puede entonces emplear herramientas como osciloscopios y analizadores digitales para validar la corrección de los modelos estático y dinámico.

Los elementos de los métodos de diseño del software. Evidentemente, no hay magia, no existe una «bala de plata» [43] que pueda dirigir infaliblemente al ingeniero del software en el camino desde los requerimientos hasta la implantación de un sistema de software complejo. De hecho, el diseño de sistemas de este tipo no se presta para nada a aproximaciones de recetario. Más bien, como se indicó anteriormente en el quinto atributo de los sistemas complejos, el diseño de tales sistemas conlleva un proceso incremental e iterativo.

A pesar de ello, los buenos métodos de diseño introducen en el proceso de desarrollo alguna de la disciplina que tanto necesita. La comunidad de la ingeniería del software ha desarrollado docenas de métodos de diseño diferentes, que a grandes rasgos pueden clasificarse en tres categorías (véase la nota del recuadro). A pesar de sus diferencias, todos estos métodos tienen elementos en común. Específicamente, cada uno incluye lo siguiente:

- | | |
|----------------|--|
| • Notación | El lenguaje para expresar cada modelo. |
| • Proceso | Las actividades que encaminan a la construcción ordenada de los modelos del sistema. |
| • Herramientas | Los artefactos que eliminan el tedio de construir el modelo y reafirman las reglas sobre los propios modelos, de forma que sea posible revelar errores e inconsistencias |

Un buen método de diseño se basa en fundamentos teóricos sólidos, pero eso no le impide ofrecer grados de libertad para la innovación artística.

Los modelos del desarrollo orientado a objetos. ¿Existe «el mejor» método de diseño? No, no hay una respuesta absoluta a esta pregunta, que al fin y al

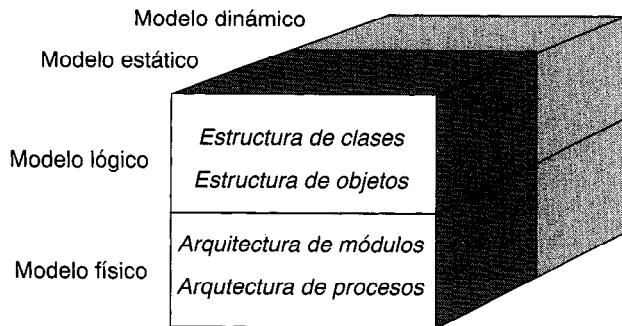


Figura 1.4. Los modelos del desarrollo orientado a objetos.

cabo no es más que una forma velada de plantear una pregunta anterior: ¿Cuál es la mejor forma de descomponer un sistema complejo? Hay que insistir: se ha encontrado un gran valor en la construcción de modelos que se centran en las «cosas» que se encuentran en el espacio del problema, formando lo que se ha dado en llamar una *descomposición orientada a objetos*.

El diseño orientado a objetos es el método que lleva a una descomposición orientada a objetos. Aplicando diseño orientado a objetos, se crea software resistente al cambio y escrito con economía de expresión. Se logra un mayor nivel de confianza en la corrección del software a través de una división inteligente de su espacio de estados. En última instancia, se reducen los riesgos inherentes al desarrollo de sistemas complejos de software.

Al ser la construcción de modelos tan importante para la construcción de sistemas complejos, el diseño orientado a objetos ofrece un rico conjunto de modelos, que se describen en la Figura 1.4. Los modelos del diseño orientado a objetos reflejan la importancia de plasmar explícitamente las jerarquías de clases y de objetos del sistema que se diseña. Estos modelos cubren también el espectro de las decisiones de diseño relevantes que hay que considerar en el desarrollo de un sistema complejo, y así animan a construir implantaciones que poseen los cinco atributos de los sistemas complejos bien formados.

El Capítulo 5 presenta cada uno de estos cuatro modelos con detalle. El Capítulo 6 explica el proceso del diseño orientado a objetos, que proporciona un conjunto ordenado de pasos para la creación y evolución de estos modelos. El Capítulo 7 examina los aspectos prácticos de dirigir un proyecto utilizando diseño orientado a objetos.

En este capítulo se ha apoyado el uso de diseño orientado a objetos para dominar la complejidad asociada al desarrollo de sistemas de software. Además, se ha sugerido una serie de beneficios fundamentales que se derivan de la aplicación de este método. Sin embargo, antes de presentar la notación y el proceso del diseño orientado a objetos es necesario estudiar los principios en los que se

basa el diseño orientado a objetos, es decir, abstracción, encapsulación, modularidad, jerarquía, tipos, concurrencia y persistencia.

Resumen

- El software es complejo de forma innata; la complejidad de los sistemas de software excede frecuentemente la capacidad intelectual humana.
- La tarea del equipo de desarrollo de software es la de crear una ilusión de sencillez.
- La complejidad toma a menudo la forma de una jerarquía; esto es útil para modelar tanto las jerarquías «es-un» (*is a*) como «parte de» (*part of*) de un sistema complejo.
- Los sistemas complejos evolucionan generalmente de formas intermedias estables.
- Existen factores de limitación fundamentales en la cognición humana; puede hacerse frente a estas restricciones mediante el uso de la descomposición, abstracción y jerarquía.
- Los sistemas complejos pueden verse centrando la atención bien en las cosas o bien en los procesos; hay razones de peso para aplicar la descomposición orientada a objetos, en la cual se ve el mundo como una colección significativa de objetos que colaboran para conseguir algún comportamiento de nivel superior.
- El diseño orientado a objetos es el método que conduce a una descomposición orientada a objetos; el diseño orientado a objetos define una notación y un proceso para construir sistemas de software complejos, y ofrece un rico conjunto de modelos lógicos y físicos con los cuales se puede razonar sobre diferentes aspectos del sistema que se está considerando.

Lecturas recomendadas

Los desafíos asociados al desarrollo de sistemas complejos de software se describen fluidamente en los trabajos clásicos de Brooks en [H 1975] y [H 1987]. Glass [H 1982], el Defense Science Board [H 1987] y el Joint Service Task Force [H 1982] ofrecen más información sobre métodos de software contemporáneos. Pueden encontrarse estudios empíricos sobre la naturaleza y causas de fallos del software en van Genuchten [H 1991], Guindon, *et. al.* [H 1987] y Jones [H 1992].

Simon [A 1962, 1982] es la referencia inicial sobre la arquitectura de sistemas complejos. Courtois [A 1985] aplica estas ideas al dominio del software. El trabajo fundamental de Alexander [I 1979] proporciona un moderno punto de vista sobre la arquitectura de estructuras físicas. Peter [I 1986] y Petroski [I 1985] examinan la complejidad en el contexto de sistemas sociales y físicos, respectivamente. Análogamente, Allen y Starr

[A 1982] examinan sistemas jerárquicos en varios dominios. Flood y Carson [A 1988] ofrecen un estudio formal de la complejidad vista desde la ciencia de la teoría de sistemas. Waldrop [A 1992] describe la ciencia emergente de la complejidad y su estudio de sistemas adaptativos complejos, comportamiento emergente y autoorganización. El informe de Miller [A 1956] proporciona evidencia empírica sobre los factores limitantes fundamentales para la cognición humana.

Existe una serie de referencias excelentes acerca del tema de la ingeniería del software. Ross, Goodenough y Irvine [H 1980], y Zelkowitz [H 1978] son dos de los artículos clásicos que resumen los elementos esenciales de la ingeniería del software. Entre otros trabajos más extensos sobre el tema se incluyen Jensen y Tonies [H 1979], Sommerville [H 1985], Vick y Ramamoorthy [H 1984], Wegner [H 1980], Pressman [H 1992], Oman y Lewis [A 1990], Berzins y Luqi [H 1991], y Ng y Yeh [H 1990]. Otros artículos relevantes para la ingeniería del software en general pueden encontrarse en Yourdon [H 1979] y Freeman y Wasserman [H 1983]. Graham [F 1991] y Berard [H 1993] presentan ambos un amplio tratamiento de la ingeniería del software orientada a objetos.

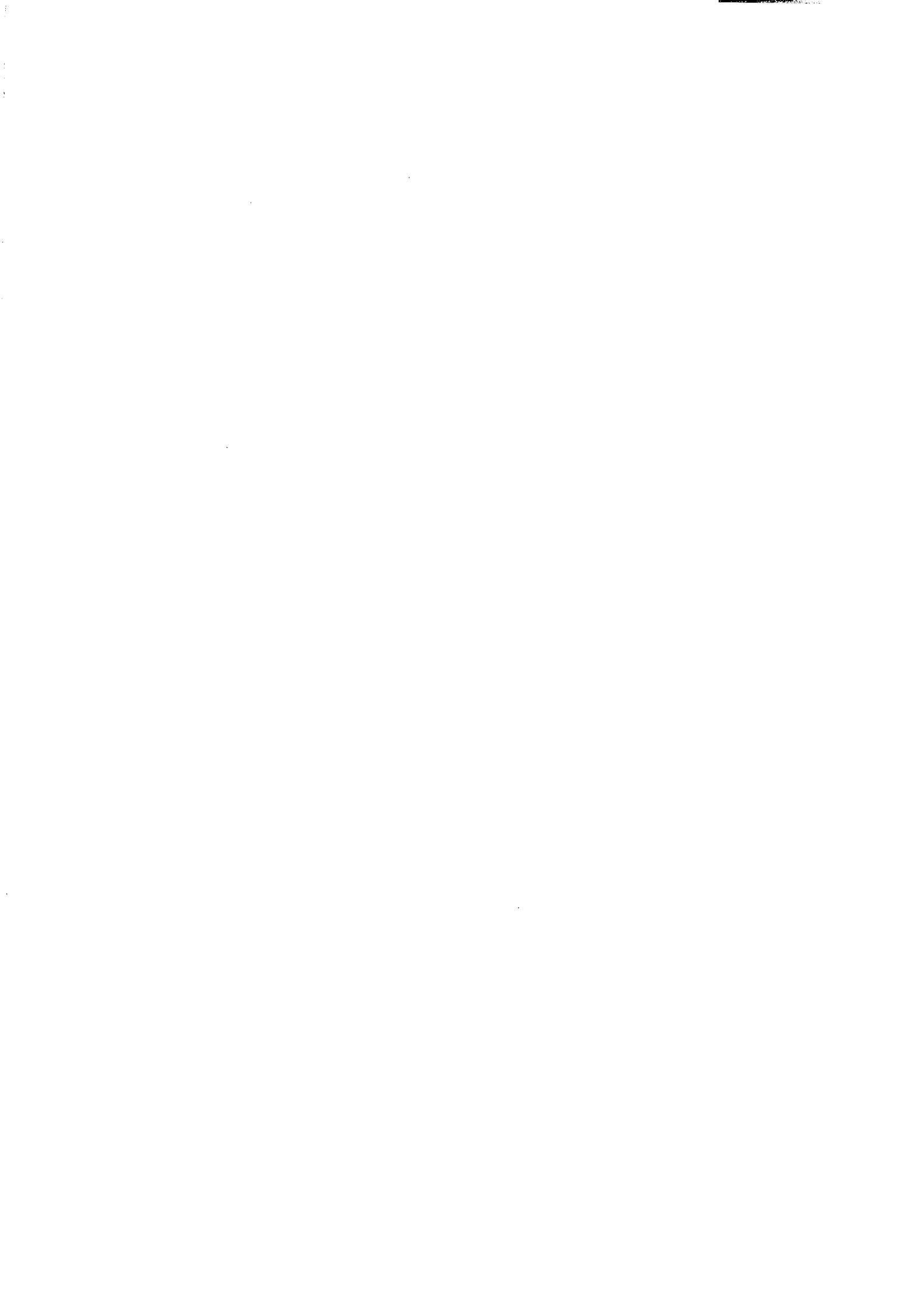
Gleick [I 1987] ofrece una introducción muy legible a la ciencia del caos.

Notas bibliográficas

- [1] Brooks, F. April 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* vol. 20(4), p. 12.
- [2] Peters, L. 1981. *Software Design*. New York, NY: Yourdon Press, p. 22.
- [3] Brooks. No Silver Bullet, p. 11.
- [4] Parnas, D. July 1985. *Software Aspects of Strategic Defense Systems*. Victoria, Canada: University of Victoria, Report DCS-47-IR.
- [5] Peter, L. 1986. *The Peter Pyramid*. New York, NY: William Morrow, p. 153.
- [6] Waldrop, M. 1992. *Complexity: The emerging Science at the Edge of Order and Chaos*. New York, NY: Simon and Schuster.
- [7] Courtois, P. June 1985. On Time and Space Decomposition of Complex Structures. *Communications of the ACM* vol. 28(6), p. 596.
- [8] Simon, H. 1982. *The Sciences of the Artificial*. Cambridge, MA: The MIT Press, p. 218.
- [9] Rechtin, E. October 1992. The Art of Systems Architecting. *IEE Spectrum*, vol. 29(10), p. 66.
- [10] Simon. *Sciences*, p. 217.
- [11] Ibid., p. 221.
- [12] Ibid., p. 209.
- [13] Gall, J. 1986. *Systemantics: How Systems Really Work and How They Fail*. Second Edition. Ann Arbor, MI: The General Systemantics Press, p. 65.
- [14] Miller, G. March 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review* vol. 63(2), p. 86.
- [15] Simon. *Sciences*, p. 81.
- [16] Dijkstra, E. 1979. Programming Considered as a Human Activity. *Classics in Software Engineering*. New York, NY: Yourdon Press, p. 5.

-
- [17] Parnas, D. December 1985. Software Aspects of Strategic Defense Systems. *Communications of the ACM* vol. 28(12), p. 1328.
 - [18] Tsai, J. and Ridge, J. November 1988. Intelligent Support for Specifications Transformation. *IEEE Software* vol. 5(6), p. 34.
 - [19] Stein, J. March 1988. Object-Oriented Programming and Database Design. *Dr. Dobb's Journal of Software Tools for the Professional Programmer*, No. 137, p. 18.
 - [20] Peters. *Software Design*.
 - [21] Yau, S. and Tsai, J. June 1986. A Survey of Software Design Techniques. *IEEE Transactions on Software Engineering* vol. SE-12(6).
 - [22] Teledyne Brown Engineering. *Software Methodology Catalog*. Report MC87-COMM/ADP-0036. October 1987. Tinton Falls, NJ.
 - [23] Sommerville, I. 1985. *Software Engineering*. Second Edition. Workingham, England: Addison-Wesley, p. 68.
 - [24] Yourdon, E. and Constantine, L. 1979. *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall.
 - [25] Myers, G. 1978. *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold.
 - [26] Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press.
 - [27] Wirth, N. January 1983. Program Development by Stepwise Refinement. *Communications of the ACM* vol. 26(1).
 - [28] Wirth, N. 1986. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall.
 - [29] Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming*. London, England: Academic Press.
 - [30] Mills, H., Linger, R., and Hevner, A. 1986. *Principles of Information System Design and Analysis*. Orlando, FL: Academic Press.
 - [31] Jackson, M. 1975. *Principles of Program Design*. Orlando, FL: Academic Press.
 - [32] Jackson, M. 1983. *System Development*. Englewood Cliffs, NJ: Prentice-Hall.
 - [33] Orr, K. 1971. *Structured Systems Development*. New York, NY: Yourdon Press.
 - [34] Langdon, G. 1982. *Computer Design*. San Jose, CA: Computeach Press, p. 6.
 - [35] Miller. *Magical Number*, p. 95.
 - [36] Shaw, M. 1981. *ALPHARD: Form and Content*. New York, NY: Springer-Verlag, p. 6.
 - [37] Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, p. 80.
 - [38] Petroski, H. 1985. *To Engineer Is Human*. St. Martin's Press: New York, p. 40.
 - [39] Dijkstra, E. January 1993. *American Programmer* vol. 6(1).
 - [40] Mostow, J. Spring 1985. Toward Better Models of the Design Process. *AI Magazine* vol. 6(1), p. 44.
 - [41] Stroustrup, B. 1991. *The C++ Programming Language*, Second Edition. Reading, MA: Addison-Wesley, p. 366.*
 - [42] Eastman, N. 1984. Software Engineering and Technology. *Technical Directions* vol. 10(1): Bethesda, MD: IBM Federal Systems Division, p. 5.
 - [43] Brooks. *No Silver Bullet*, p. 10.

* Existe versión en español por Addison-Wesley Iberoamericana.



El modelo de objetos

La tecnología orientada a objetos se apoya en los sólidos fundamentos de la ingeniería, cuyos elementos reciben el nombre global de *modelo de objetos*. El modelo de objetos abarca los principios de abstracción, encapsulación, modularidad, jerarquía, tipos, concurrencia y persistencia. Ninguno de estos principios es nuevo por sí mismo. Lo importante del modelo de objetos es el hecho de conjugar todos estos elementos de forma sinérgica.

Quede claro que el diseño orientado a objetos es fundamentalmente diferente a los enfoques de diseño estructurado tradicionales: requiere un modo distinto de pensar acerca de la descomposición, y produce arquitecturas software muy alejadas del dominio de la cultura del diseño estructurado. Estas diferencias surgen del hecho de que los métodos de diseño estructurado se basan en la programación estructurada, mientras que el diseño orientado a objetos se basa en la programación orientada a objetos. Por desgracia, la programación orientada a objetos significa cosas distintas para personas distintas. Tal como Rentsch predijo acertadamente, «Mi impresión es que la programación orientada a objetos va a ser en los ochenta lo que fue la programación estructurada en los setenta. Todo el mundo va a estar a favor de ella. Todos los fabricantes van a promocionar sus productos afirmando que la soportan. Todos los administradores hablarán bien de ella. Todos los programadores la practicarán (de forma diferente). Y nadie va a saber exactamente qué es» [1]. Las predicciones de Rentsch siguen vigentes para los noventa.

En este capítulo va a mostrarse claramente qué es y qué no es el diseño orientado a objetos, y en qué se diferencia de otros métodos de diseño a través del uso de los siete elementos del modelo de objetos.

2.1. La evolución del modelo de objetos

Tendencias en ingeniería del software

Las generaciones de los lenguajes de programación. Cuando se mira hacia atrás en la relativamente breve pero ajetreada historia de la ingeniería del software, no puede evitarse el apreciar dos amplias tendencias:

- El desplazamiento del centro de atención de la programación al por menor a la programación al por mayor.
- La evolución de los lenguajes de alto nivel.

La mayoría de los nuevos sistemas de software de dimensión industrial son más grandes y más complejos que sus predecesores de pocos años antes. Este crecimiento de la complejidad ha promovido una cantidad significativa de investigación aplicada útil en ingeniería del software, particularmente en lo referente a descomposición, abstracción y jerarquía. El desarrollo de lenguajes de programación más expresivos ha completado estos avances. La tendencia ha sido un desplazamiento desde los lenguajes que dicen al computador qué hacer, o lenguajes imperativos, hacia lenguajes que describen las abstracciones clave en el dominio del problema (lenguajes declarativos). Wegner ha clasificado algunos de los lenguajes de programación de alto nivel más populares en generaciones, dispuestas de acuerdo con las características que tales lenguajes fueron pioneros en presentar:

- Lenguajes de primera generación (1954-1958):

FORTRAN I	Expresiones matemáticas.
ALGOL 58	Expresiones matemáticas.
Flowmatic	Expresiones matemáticas.
IPL V	Expresiones matemáticas.

- Lenguajes de segunda generación (1959-1961):

FORTRAN II	Subrutinas, compilación separada.
ALGOL 60	Estructura en bloques, tipos de datos.
COBOL	Descripción de datos, manejo de ficheros.
Lisp	Procesamiento de listas, punteros, recolección de basura.

- Lenguajes de tercera generación (1962-1970):

PL/1	FORTRAN + ALGOL + COBOL.
ALGOL 68	Sucesor riguroso del ALGOL 60.
Pascal	Sucesor sencillo del ALGOL 60.
Simula	Clases, abstracción de datos.

- El Hueco Generacional (1970-1980):

Se inventaron muchos lenguajes diferentes, pero pocos perduraron [2].

En generaciones sucesivas, el tipo de mecanismo de abstracción que admitía cada lenguaje fue cambiando. Los lenguajes de la primera generación se utilizaron principalmente para aplicaciones científicas y de ingeniería, y el vocabulario de estos dominios de problema fue matemático casi por completo. Así, los lenguajes como el FORTRAN I se desarrollaron para que el programador pudiera escribir fórmulas matemáticas, liberándole de esta forma de algunas de las complicaciones del lenguaje ensamblador o del lenguaje máquina. Esta primera generación de lenguajes de alto nivel representó por lo tanto un paso de acercamiento al espacio del problema, y un paso de alejamiento de la máquina que había debajo. Entre los lenguajes de segunda generación, el énfasis se puso en las abstracciones algorítmicas. Por esta época, las máquinas eran cada vez más y más potentes, y el abaratamiento en la industria de los computadores significó que podía automatizarse una mayor variedad de problemas, especialmente para aplicaciones comerciales. En este momento, lo principal era decirle a la máquina lo que debía hacer: lee primero estas fichas personales, ordénalas después, y a continuación imprime este informe. Una vez más, esta nueva generación de lenguajes de alto nivel acercaba a los desarrolladores un paso hacia el espacio del problema y los alejaba de la máquina subyacente. A finales de los sesenta, especialmente con la llegada de los transistores y la tecnología de circuitos integrados, el coste del hardware de los computadores había caído de forma dramática, pero su capacidad de procesamiento había crecido casi exponencialmente. Ahora podían resolverse problemas mayores, pero eso exigía la manipulación de más tipos de datos. Así, lenguajes como el ALGOL 60 y posteriormente el Pascal evolucionaron soportando abstracción de datos. El programador podía describir el significado de clases de datos relacionadas entre sí (su tipo) y permitir que el lenguaje de programación apoyase estas decisiones de diseño. Esta generación de lenguajes acercó de nuevo el software un paso hacia el dominio del problema, y lo alejó otro paso de la máquina.

Los setenta ofrecieron un frenesi de actividad investigadora en materia de lenguajes de programación, con el resultado de la creación de literalmente dos millares de diferentes lenguajes con sus dialectos. En gran medida, la tendencia a escribir programas más y más grandes puso de manifiesto las deficiencias de los lenguajes más antiguos; así, se desarrollaron muchos nuevos mecanismos lingüísticos para superar estas limitaciones. Sólo algunos de estos lenguajes han sobrevivido (¿acaso ha visto el lector libros recientes sobre los lenguajes Fred, Chaos o Tranquil?); sin embargo, muchos de los conceptos que introdujeron encontraron su camino en sucesores de los lenguajes anteriores. Así, hoy en día existen Smalltalk (un sucesor revolucionario de Simula), Ada (un sucesor del ALGOL 68 y Pascal, con contribuciones de Simula, Alphard y CLU), CLOS (que surgió del Lisp, LOOPS y Flavors), C++ (derivado de un matrimonio entre C y Simula), y Eiffel (derivado de Simula y Ada). Lo que resulta del mayor interés para nosotros es la clase de lenguajes que suelen llamarse *basados en objetos y orientados a objetos*. Los lenguajes de programación basados en y orientados a objetos son los que mejor soportan la descomposición orientada a objetos del software.

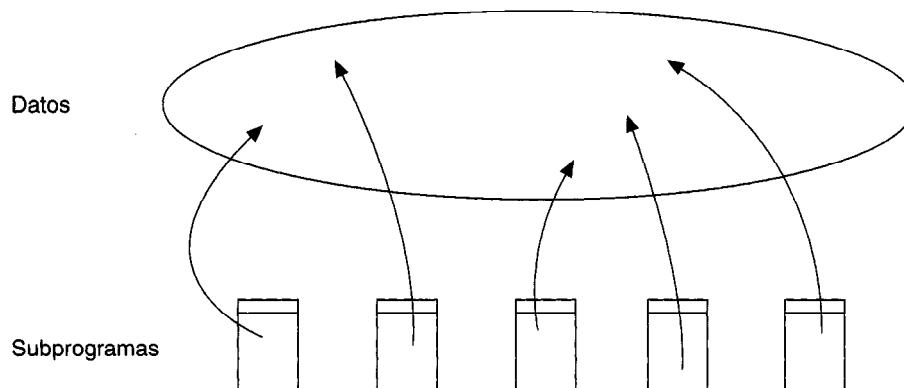


Figura 2.1. La topología de los lenguajes de programación de primera generación y principios de la segunda.

La topología de los lenguajes de primera y principios de la segunda generaciones. Para ilustrar con precisión lo que se quiere decir, pasamos a estudiar la estructura de cada generación de lenguajes de programación. En la Figura 2.1, se ve la topología de la mayoría de los lenguajes de programación de la primera generación e inicios de la segunda. Por topología se entiende los bloques físicos básicos de construcción de ese lenguaje y cómo esas partes pueden ser conectadas. En esta Figura se aprecia que, para lenguajes como FORTRAN y COBOL, el bloque básico de construcción de todas las aplicaciones es el subprograma (o párrafo, para quienes hablen COBOL). Las aplicaciones escritas en estos lenguajes exhiben una estructura física relativamente plana, consistente sólo en datos globales y subprogramas. Las flechas de la figura indican dependencias de los subprogramas respecto a varios datos. Durante el diseño pueden separarse lógicamente diferentes clases de datos, pero existen pocos elementos en estos lenguajes para reforzar estas decisiones. Un error en una parte de un programa puede tener un devastador efecto de propagación a través del resto del sistema, porque las estructuras de datos globales están expuestas al acceso de todos los subprogramas. Cuando se realizan modificaciones en un sistema grande, es difícil mantener la integridad del diseño original. Frecuentemente aparece la entropía: después de un periodo de mantenimiento aun cuando sea breve, un programa escrito en estos lenguajes suele contener una enorme cantidad de acoplamientos entre subprogramas, con significados implícitos de los datos y flujos de control retorcidos, amenazando la fiabilidad de todo el sistema y, por supuesto, reduciendo la claridad global de la solución.

La topología de los lenguajes de fines de la segunda generación y principios de la tercera. A mediados de los sesenta se reconoció finalmente a los programas como puntos intermedios importantes entre el problema y el computador

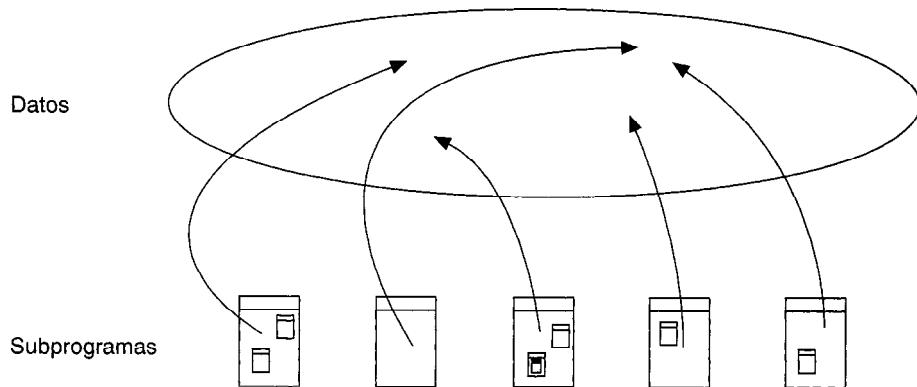


Figura 2.2. La topología de los lenguajes de programación de finales de la segunda generación y principios de la tercera.

[3]. Según apunta Shaw, «la primera abstracción del software, ahora llamada abstracción «procedimental», creció directamente de esta visión pragmática del software... Los subprogramas se inventaron antes de 1950, pero en aquel momento no se les apreció como abstracciones en toda la extensión de la palabra... En vez de eso, originalmente se les vio como dispositivos para ahorrar trabajo... Aunque rápidamente los subprogramas fueron considerados como una forma de abstraer funciones del programa» [4]. El hallazgo de que los subprogramas podían servir como un mecanismo de abstracción tuvo tres consecuencias importantes. Primero, se inventaron lenguajes que soportaban una variedad de mecanismos de paso de parámetros. Segundo, se asentaron los fundamentos de la programación estructurada, puestos de manifiesto en la capacidad de los lenguajes para anidar subprogramas y el desarrollo de teorías sobre estructuras de control y ámbito y visibilidad de las declaraciones. Tercero, surgieron los métodos de diseño estructurado, que ofrecían una guía a los diseñadores que intentaban construir grandes sistemas utilizando los subprogramas como bloques físicos básicos de construcción. Así no es extraño, tal y como muestra la Figura 2.2, que la topología de los lenguajes de finales de la segunda generación y principios de la tercera sea en gran medida una variación sobre el mismo tema de generaciones anteriores. Esta topología se enfrenta a algunos de los defectos de lenguajes precedentes, es decir, la necesidad de tener un mayor control sobre las abstracciones algorítmicas, pero sigue fallando a la hora de superar los problemas de la programación a gran escala y del diseño de datos.

La topología de los lenguajes de finales de la tercera generación-
Comenzando por el FORTRAN II, y apareciendo en la mayoría de los lenguajes de fines de la tercera generación, surgió otro importante mecanismo estructurador para resolver los problemas crecientes de la programación a gran

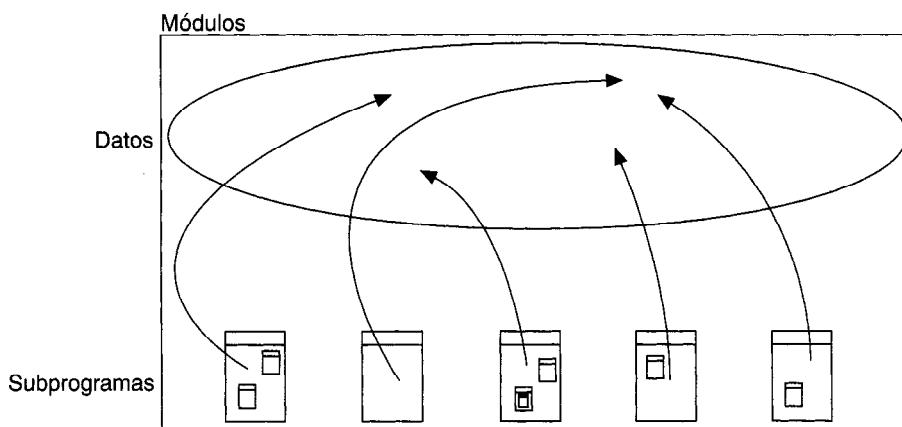


Figura 2.3. La topología de los lenguajes de programación de finales de la tercera generación.

escala. Proyectos de programación mayores significaban equipos de desarrollo mayores y, por tanto, la necesidad de desarrollar independientemente partes diferentes del mismo programa. La respuesta a esta necesidad fue el módulo compilado separadamente, que en su concepción más temprana no era mucho más que un contenedor arbitrario para datos y subprogramas, como se ve en la Figura 2.3. Los módulos no solían reconocerse como un mecanismo de abstracción importante; en la práctica se utilizaban simplemente para agrupar subprogramas de los que cabía esperar que cambiase juntos. La mayoría de los lenguajes de esta generación, aunque admitían alguna clase de estructura modular, tenían pocas reglas que exigiesen una consistencia semántica entre los interfaces de los módulos. Un desarrollador que escribía un subprograma para un módulo podía asumir que sería llamado con tres parámetros diferentes: un número en coma flotante, una matriz de diez elementos y un entero que representase un indicador de tipo booleano. En otro módulo, una llamada a este subprograma podía utilizar parámetros actuales incorrectos que violasen esas suposiciones: un entero, una matriz de cinco elementos y un número negativo. Del mismo modo, un módulo podía utilizar un bloque de datos comunes que asumiese como propio, y otro módulo violar esas suposiciones manipulando esos datos directamente. Desafortunadamente, al tener la mayoría de estos lenguajes un soporte desastroso para la abstracción de datos y la comprobación estricta de tipos, estos errores sólo podían detectarse durante la ejecución del programa.

La topología de los lenguajes de programación basados en objetos y orientados a objetos. La importancia de la abstracción de datos para dominar la complejidad está claramente establecida por Shankar: «La naturaleza de las abstracciones que pueden lograrse mediante el uso de procedimientos es ade-

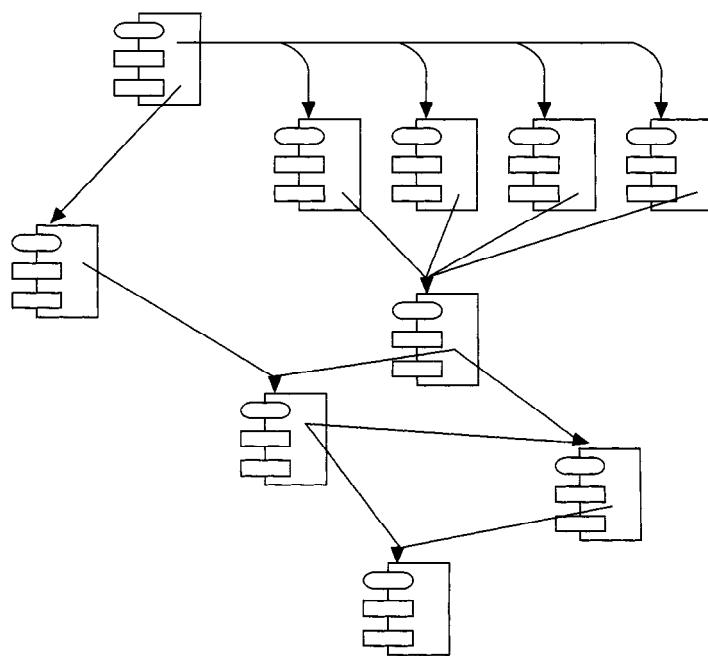


Figura 2.4. La topología de las aplicaciones de tamaño pequeño a moderado que utilizan lenguajes basados en objetos y orientados a objetos.

cuada para la descripción de operaciones abstractas, pero no es particularmente buena para la descripción de objetos abstractos. Éste es un serio inconveniente, porque en muchas aplicaciones la complejidad de los objetos de datos que hay que manipular contribuye sustancialmente a la complejidad global del problema» [5]. Este hallazgo tuvo dos consecuencias importantes. Primero, surgieron los métodos de diseño dirigido por los datos, que proporcionaron una aproximación disciplinada a los problemas de realizar abstracciones de datos en lenguajes orientados algorítmicamente. Segundo, aparecieron teorías acerca del concepto de tipo, que finalmente encontraron realización en lenguajes como el Pascal.

La conclusión natural de estas ideas apareció primero en el lenguaje Simula y fue mejorando durante el periodo de vacío generacional de los lenguajes, culminando en el desarrollo relativamente reciente de varios lenguajes como Smalltalk, Object Pascal, C++, CLOS, Ada y Eiffel. Por razones que se explicarán brevemente, estos lenguajes reciben el nombre de *basados en objetos y orientados a objetos*. La Figura 2.4 ilustra la topología de estos lenguajes para aplicaciones de tamaño pequeño y moderado. El bloque físico de construcción en estos lenguajes es el módulo, que representa una colección lógica de clases y

objetos en lugar de subprogramas, como ocurría en lenguajes anteriores. En otras palabras, «si los procedimientos y funciones son verbos y los elementos de datos son nombres, un programa orientado a procedimientos se organiza alrededor de los verbos, mientras que un programa orientado a objetos se organiza alrededor de los nombres»[6]. Por esta razón, la estructura física de una aplicación orientada a objetos de tamaño pequeño a moderado tiene el aspecto de un grafo, no el de un árbol, lo que es típico en lenguajes orientados algorítmicamente. Además, existen pocos o ningún dato global. En vez de eso, los datos y las operaciones están unidos de tal modo que los bloques lógicos de construcción fundamentales de estos sistemas ya no son algoritmos, sino clases y objetos.

Al llegar aquí ya se ha progresado más allá de la programación a gran escala y hay que enfrentarse a la programación a escala industrial. En sistemas muy complejos se encuentra que las clases, objetos y módulos proporcionan un medio esencial, pero, aun así, insuficiente de abstracción. Afortunadamente, el modelo de objetos soporta el aumento de escala. En sistemas grandes existen agrupaciones de abstracciones que se construyen en capas, una sobre otra. A cualquier nivel de abstracción se encuentran colecciones significativas de objetos que colaboran para lograr algún comportamiento de nivel superior. Si se examina cualquier agrupación determinada para ver su implantación, se desvela un nuevo conjunto de abstracciones cooperando. Ésta es exactamente la organización de la complejidad descrita en el Capítulo 1; esta topología se muestra en la Figura 2.5.

Fundamentos del modelo de objetos

Los métodos de diseño estructurado surgieron para guiar a los desarrolladores que intentaban construir sistemas complejos utilizando los algoritmos como bloques fundamentales para su construcción. Análogamente, los métodos de diseño orientados a objetos han surgido para ayudar a los desarrolladores a explotar la potencia expresiva de los lenguajes de programación basados en objetos y orientados a objetos, utilizando las clases y los objetos como bloques básicos de construcción.

En realidad, el modelo de objetos ha recibido la influencia de una serie de factores, no sólo de la programación orientada a objetos. Por contra, tal y como se discute en las notas complementarias, el modelo de objetos ha demostrado ser un concepto unificador en la informática, aplicable no sólo a los lenguajes de programación, sino también al diseño de interfaces de usuario, bases de datos e incluso arquitecturas de computadores. La razón para este gran atractivo es simplemente que una orientación a objetos ayuda a combatir la complejidad inherente a muchos tipos de sistema diferentes.

El diseño orientado a objetos representa así un desarrollo evolutivo, no revolucionario; no rompe con los avances del pasado, sino que se basa en avances ya probados. Desgraciadamente, hoy en día la mayoría de los programadores han sido educados formal e informalmente sólo en los principios del diseño es-

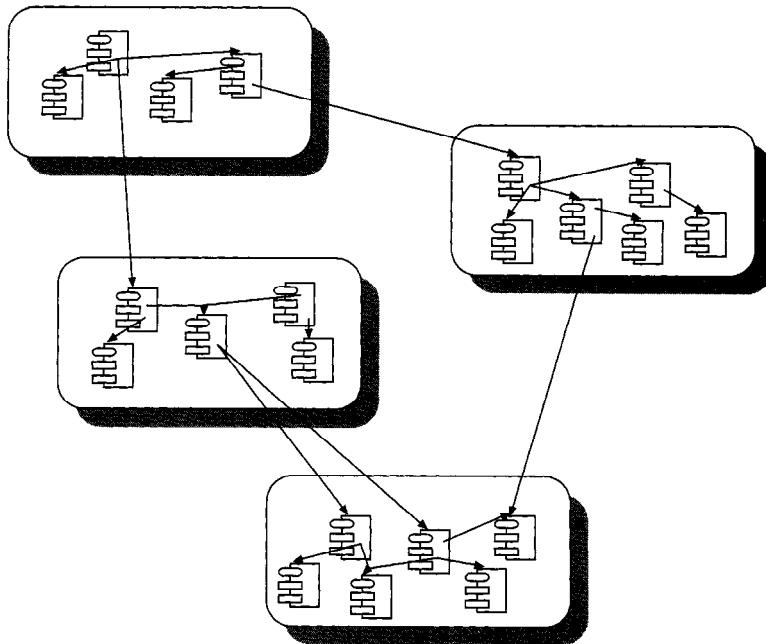


Figura 2.5. La topología de las aplicaciones a gran escala que utilizan lenguajes basados en objetos y orientados a objetos.

tructurado. Por supuesto, muchos buenos ingenieros han desarrollado y puesto en acción innumerables sistemas de software utilizando estas técnicas. Sin embargo, existen límites para la cantidad de complejidad que se puede manejar utilizando sólo descomposición algorítmica; por tanto hay que volverse hacia la descomposición orientada a objetos. Es más, si se intenta utilizar lenguajes tales como C++ y Ada como si fuesen lenguajes tradicionales orientados algorítmicamente, no sólo se pierde la potencia de la que se disponía, sino que habitualmente se acaba en una situación peor que si se hubiese utilizado un lenguaje más antiguo como C o Pascal. Ofrezcase una taladradora eléctrica a un carpintero que no sabe nada de la electricidad, y la utilizará como un martillo. Aca-bará por torcer los clavos y romperse algunos dedos, porque una taladradora eléctrica es un martillo desastroso.

POO, DOO y AOO¹

Dado que el modelo de objetos se deriva de fuentes tan dispersas, desgraciadamente ha venido acompañado por un embrollo terminológico. Un programa-

¹ Las siglas en lengua inglesa OOP, OOD y OOA también se utilizan frecuentemente en los círculos informáticos, sobre todo las primeras. (N. del T.)

dor de Smalltalk utiliza *métodos*, un programador de C++ utiliza *funciones miembro virtuales*, y un programador de CLOS utiliza *funciones genéricas*. Un programador de Object Pascal habla de *coerción o conversión forzada de tipos*; un programador de Ada llama a lo mismo una *conversión de tipos*. Para minimizar la confusión, se definirá qué es orientado a objetos y qué no lo es. El glosario ofrece un resumen de todos los términos descritos aquí, y algunos más.

Bhaskar ha observado que la expresión *orientado a objetos* «se ha esgrimido a diestro y siniestro de forma indiscriminada y con la misma reverencia que se guarda hacia ‘maternidad’, ‘tarta de manzana’ y ‘programación estructurada’» [7]. En lo que se puede estar de acuerdo es en que el concepto de objeto es central en cualquier cosa orientada a objetos. En el capítulo anterior, se definió informalmente un objeto como una entidad tangible que muestra algún comportamiento bien definido. Stefik y Bobrow definen los objetos como «entidades que combinan las propiedades de los procedimientos y los datos en el sentido de que realizan computaciones y conservan el estado local» [8]. Definir a los *objetos* como *entidades* invita a la pregunta, pero el concepto básico aquí es que los objetos sirven para unificar las ideas de las abstracciones algorítmica y de datos. Jones clarifica más este término reparando en que «en el modelo de objetos, se pone el énfasis en caracterizar nítidamente los componentes del sistema físico o abstracto que se pretende modelar con un sistema programado... Los objetos tienen una cierta «integridad» que no debería —de hecho, no puede— ser violada. Un objeto sólo puede cambiar de estado, actuar, ser manipulado o permanecer en relación con otros objetos de maneras apropiadas para ese objeto. Dicho de otro modo, existen propiedades invariantes que caracterizan un objeto y su comportamiento. Un ascensor, por ejemplo, se caracteriza por propiedades invariantes que incluyen que sólo se desplaza arriba y abajo por su hueco... Cualquier simulación de un ascensor debe incorporar estos invariantes, porque son intrínsecos al concepto de ascensor» [32].

Fundamentos del modelo de objetos

Como apuntan Yonezawa y Tokoro, «El término “objeto” surgió casi independientemente en varios campos de la informática, casi simultáneamente a principios de los setenta, para referirse a nociones que eran diferentes en su apariencia, pero relacionadas entre sí. Todas estas nociones se inventaron para manejar la complejidad de sistemas de software de tal forma que los objetos representaban componentes de un sistema descompuesto modularmente o bien unidades modulares de representación del conocimiento» [9]. Levy añade que los siguientes sucesos contribuyeron a la evolución de conceptos orientados a objetos:

- «Avances en la arquitectura de los computadores, incluyendo los sistemas de capacidades y el apoyo en hardware para conceptos de sistemas operativos.
- Avances en lenguajes de programación, como se demostró en Simula, Smalltalk, CLU y Ada.
- Avances en metodología de la programación, incluyendo la modularización y la ocultación de información» [10].

Podría añadirse a esta lista tres contribuciones más a la fundación del modelo de objetos:

- Avances en modelos de bases de datos.
- Investigación en inteligencia artificial.
- Avances en filosofía y ciencia cognitiva.

El concepto de un objeto tuvo sus inicios en el hardware hace más de veinte años, comenzando con la invención de arquitecturas basadas en descriptores y, posteriormente, arquitecturas basadas en capacidades [11]. Estas arquitecturas representaron una ruptura con las arquitecturas clásicas de Von Neumann, y surgieron como consecuencia de los intentos realizados para eliminar el hueco existente entre las abstracciones de alto nivel de los lenguajes de programación y las abstracciones de bajo nivel de la propia máquina [12]. Según sus promotores, las ventajas de tales arquitecturas son muchas: mejor detección de errores, mejora en la eficiencia de la ejecución, menos tipos de instrucciones, compilación más sencilla y reducción de los requisitos de almacenamiento. Entre los computadores que tienen una arquitectura orientada a objetos pueden citarse el Burroughs 5000, el Plessey 250 y el Cambridge CAP [13]; SWARD [14], el Intel 432 [15], el COM de Caltech [16], el IBM System/38 [17], el Rational R1000, y el BiiN 40 y 60.

En relación muy estrecha con los desarrollos de arquitecturas orientadas a objetos están los sistemas operativos orientados a objetos. El trabajo de Dijkstra con el sistema de multiprogramación THE fue el primero que introdujo la idea de construir sistemas como máquinas de estados en capas [18]. Otros sistemas operativos pioneros en orientación a objetos son Plessey/System 250 (para el multiprocesador Plessey 250), Hydra (para C.mmp de CMU), CALTSS (para el CDC 6400), CAP (para el computador Cambridge CAP), UCLA Secure UNIX (para el PDP 11/45 y 11/70), StarOS (para el Cm* de CMU), Medusa (también para el Cm* de CMU) y iMAX (para el Intel 432). La siguiente generación de sistemas operativos parece seguir esta tendencia: el proyecto Cairo de Microsoft y el proyecto Pink de Taligent son sistemas operativos orientados a objetos.

Quizás la contribución más importante al modelo de objetos deriva de la clase de lenguajes de programación que se denominan basados en objetos y orientados a objetos. Las ideas fundamentales de clase y objeto aparecieron por primera vez en el lenguaje Simula 67. El sistema Flex, seguido por varios dialectos de Smalltalk, como Smalltalk—72, —74 y —76, y por último la versión actual, Smalltalk—80, llevaron el paradigma orientado a objetos de Simula hasta su conclusión natural, haciendo que en el lenguaje todo fuese instancia de una clase. En los setenta se desarrollaron lenguajes como Alphard, CLU, Euclid, Gypsy, Mesa y Modula, que soportaban las ideas entonces emergentes de abstracción de datos. Más recientemente, la investigación en lenguajes ha dado lugar a injertos de conceptos de Simula y Smalltalk en lenguajes de alto nivel tradicionales. La unión entre conceptos orientados a objetos con C ha producido los lenguajes C++ y Objective C. La adición de mecanismos de programación orientada a objetos al Pascal ha llevado a la aparición del Object Pascal, Eiffel y Ada. Además, hay muchos dialectos del Lisp que incorporan las características orientadas a objetos de Simula y Smalltalk, como Flavors, LOOPS y, más recientemente, el Common Lisp Object System (CLOS). El apéndice trata estos y otros lenguajes con más detalle.

La primera persona en identificar formalmente la importancia de componer sistemas en capas de abstracción fue Dijkstra. Parnas introdujo después la idea de ocultación de información [20], y en los setenta varios investigadores, destacando

Liskov y Zilles [21], Guttag [22] y Shaw [23], fueron pioneros en el desarrollo de mecanismos de tipos de datos abstractos. Hoare contribuyó a esos desarrollos con su propuesta de una teoría de tipos y subclases [24].

Aunque la tecnología de bases de datos ha evolucionado un tanto independientemente de la ingeniería del software, también ha contribuido al modelo de objetos [25], sobre todo a través de las ideas de la aproximación entidad—relación (ER) al modelado de datos [26]. En el modelo ER, propuesto en primer lugar por Chen [27], el mundo se modela en términos de sus entidades, los atributos de estas y las relaciones entre esas entidades.

En el campo de la inteligencia artificial, los avances en representación del conocimiento han contribuido a una comprensión de las abstracciones orientadas a objetos. En 1975, Minsky propuso por primera vez una teoría de marcos² para representar objetos del mundo real tal como los perciben los sistemas de reconocimiento de imágenes y lenguaje natural [28]. Desde entonces, se han utilizado los marcos como fundamento arquitectónico para diversos sistemas inteligentes.

Por último, la filosofía y la ciencia cognitiva han contribuido al avance del modelo de objetos. La idea de que el mundo podía verse en términos de objetos o procesos fue una innovación de los griegos, y en el siglo XVII se encuentra a Descartes observando que los humanos aplican de forma natural una visión orientada a objetos del mundo [29]. En el siglo XX, Rand ampliaba estos temas en su filosofía de la epistemología objetivista [30]. Más recientemente, Minsky ha propuesto un modelo de inteligencia humana en el que considera la mente organizada como una sociedad formada por agentes que carecen de mente [31]. Minsky arguye que sólo a través del comportamiento cooperativo de estos agentes se encuentra lo que llamamos *inteligencia*.

Programación orientada a objetos. ¿Qué es entonces la programación orientada a objetos (o POO, como se escribe a veces)? Aquí va a definirse como sigue:

La programación orientada a objetos es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas, miembros de una jerarquía de clases unidas mediante relaciones de herencia.

Hay tres partes importantes en esta definición: la programación orientada a objetos (1) utiliza *objetos*, no algoritmos, como sus bloques lógicos de construcción fundamentales (la jerarquía «parte de» que se introdujo en el Capítulo 1); (2) cada objeto es una *instancia* de alguna *clase*; y (3) las clases están relacionadas con otras clases por medio de relaciones de *herencia* (la jerarquía «de clases» de la que se habló en el Capítulo 1). Un programa puede parecer orientado a objetos, pero si falta cualquiera de estos elementos, no es un programa ori-

² Como término equivalente a *marcos* se utiliza también con frecuencia el término original inglés, *frames*. (N. del T.)

tado a objetos. Específicamente, la programación sin herencia es explícitamente no orientada a objetos; se denomina *programación con tipos abstractos de datos*.

Según esta definición, algunos lenguajes son orientados a objetos, y otros no lo son. Stroustrup sugiere que «si el término ‘orientado a objetos’ significa algo, debe significar un lenguaje que tiene mecanismos que soportan bien el estilo de programación orientado a objetos... Un lenguaje soporta bien un estilo de programación si proporciona capacidades que hacen conveniente utilizar tal estilo. Un lenguaje no soporta una técnica si exige un esfuerzo o habilidad excepcionales escribir tales programas; en ese caso, el lenguaje se limita a permitir a los programadores el uso de esas técnicas» [33]. Desde una perspectiva teórica, uno puede fingir programación orientada a objetos en lenguajes de programación no orientados a objetos, como Pascal o incluso COBOL o ensamblador, pero es horriblemente torpe hacerlo. Cardelli y Wegner dicen que «cierto lenguaje es orientado a objetos si y sólo si satisface los siguientes requisitos:

- Soporta objetos que son abstracciones de datos con un interfaz de operaciones con nombre y un estado local oculto.
- Los objetos tienen un tipo asociado [clase].
- Los tipos [clase] pueden heredar atributos de los supertipos [superclases]» [34].

Para un lenguaje, el soportar la herencia significa que es posible expresar relaciones «es un» entre tipos, por ejemplo, una rosa roja es un tipo de flor, y una flor es un tipo de planta. Si un lenguaje no ofrece soporte directo para la herencia, entonces no es orientado a objetos. Cardelli y Wegner distinguen tales lenguajes llamándolos *basados en objetos* en vez de *orientados a objetos*. Bajo esta definición, Smalltalk, Object Pascal, C++, Eiffel y CLOS son todos ellos orientados a objetos, y Ada es basado en objetos. Sin embargo, al ser los objetos y las clases elementos de ambos tipos de lenguajes, es posible y muy deseable utilizar métodos de diseño orientado a objetos tanto para lenguajes orientados a objetos como para lenguajes basados en objetos.

Diseño orientado a objetos. El énfasis en los métodos de programación está puesto principalmente en el uso correcto y efectivo de mecanismos particulares del lenguaje que se utiliza. Por contraste, los métodos de diseño enfatizan la estructuración correcta y efectiva de un sistema complejo. ¿Qué es entonces el diseño orientado a objetos? Sugerimos que

El diseño orientado a objetos es un método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir los modelos lógico y físico, así como los modelos estático y dinámico del sistema que se diseña.

Hay dos partes importantes en esta definición: el diseño orientado a objetos (1) da lugar a una descomposición orientada a objetos y (2) utiliza diversas notaciones para expresar diferentes modelos del diseño lógico (estructura de clases

y objetos) y físico (arquitectura de módulos y procesos) de un sistema, además de los aspectos estáticos y dinámicos del sistema.

El soporte para la descomposición orientada a objetos es lo que hace el diseño orientado a objetos bastante diferente del diseño estructurado: el primero utiliza abstracciones de clases y objetos para estructurar lógicamente los sistemas, y el segundo utiliza abstracciones algorítmicas. Se utilizará el término *diseño orientado a objetos* para referirse a cualquier método que encamine a una descomposición orientada a objetos. Se utilizarán ocasionalmente las siglas DOO para denotar el método particular de diseño orientado a objetos que se describe en este libro.

Análisis orientado a objetos El modelo de objetos ha influido incluso en las fases iniciales del ciclo de vida del desarrollo del software. Las técnicas de análisis estructurado tradicionales, cuyo mejor ejemplo son los trabajos de DeMarco [35], Yourdon [36] y Gane y Sarson [37], con extensiones para tiempo real de Ward y Mellor [38] y de Hatley y Pirbhaji [39], se centran en el flujo de datos dentro de un sistema. El análisis orientado a objetos (o AOO, como se le llama en ocasiones) enfatiza la construcción de modelos del mundo real, utilizando una visión del mundo orientada a objetos:

El análisis orientado a objetos es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.

¿Cómo se relacionan AOO, DOO y POO? Básicamente, los productos del análisis orientado a objetos sirven como modelos de los que se puede partir para un diseño orientado a objetos; los productos del diseño orientado a objetos pueden utilizarse entonces como anteproyectos para la implementación completa de un sistema utilizando métodos de programación orientada a objetos.

2.2. Elementos del modelo de objetos

Tipos de paradigmas de programación

Jenkins y Glasgow observan que «la mayoría de los programadores trabajan en un lenguaje y utilizan sólo un estilo de programación. Programan bajo un paradigma apoyado por el lenguaje que usan. Frecuentemente, no se les han mostrado vías alternativas para pensar sobre un problema, y por tanto tienen dificultades para apreciar las ventajas de elegir un estilo más apropiado para el problema que tienen entre manos» [40]. Bobrow y Stefik definen un estilo de programación como «una forma de organizar programas sobre las bases de algún modelo conceptual de programación y un lenguaje apropiado para que resulten claros los programas escritos en ese estilo» [41]. Sugieren además que hay

cinco tipos principales de estilos de programación, que se listan aquí con los tipos de abstracciones que emplean:

- | | |
|-------------------------------|--|
| • Orientados a procedimientos | Algoritmos. |
| • Orientados a objetos | Clases y objetos |
| • Orientados a lógica | Objetivos, a menudo expresados como cálculo de predicados. |
| • Orientados a reglas | Reglas si-entonces (<i>if-then</i>). |
| • Orientados a restricciones | Relaciones invariantes. |

No hay un estilo de programación que sea el mejor para todo tipo de aplicaciones. Por ejemplo, la programación orientada a reglas sería la mejor para el diseño de una base de conocimiento, y la programación orientada a procedimientos sería la más indicada para el diseño de operaciones de cálculo intenso. Por nuestra experiencia, el estilo orientado a objetos es el más adecuado para el más amplio conjunto de aplicaciones; realmente, este paradigma de programación sirve con frecuencia como el marco de referencia arquitectónico en el que se emplean otros paradigmas.

Cada uno de esos estilos de programación se basa en su propio marco de referencia conceptual. Cada uno requiere una actitud mental diferente, una forma distinta de pensar en el problema. Para todas las cosas orientadas a objetos, el marco de referencia conceptual es el *modelo de objetos*. Hay cuatro elementos fundamentales en este modelo:

- Abstracción.
- Encapsulamiento.
- Modularidad.
- Jerarquía.

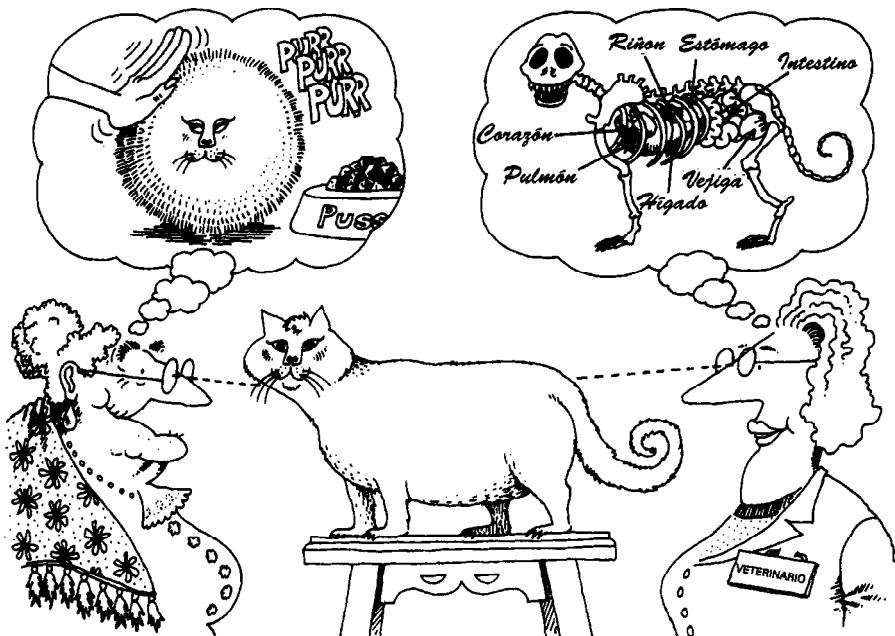
Al decir *fundamentales*, quiere decirse que un modelo que carezca de cualquiera de estos elementos no es orientado a objetos.

Hay tres elementos secundarios del modelo de objetos:

- Tipos (tipificación).
- Concurrencia.
- Persistencia.

Por *secundarios* quiere decirse que cada uno de ellos es una parte útil del modelo de objetos, pero no es esencial.

Sin este marco de referencia conceptual, puede programarse en un lenguaje como Smalltalk, Object Pascal, C++, CLOS, Eiffel o Ada, pero el diseño tendría el aspecto de una aplicación FORTRAN, Pascal o C. Se habrá perdido o, por el contrario, abusado de la potencia expresiva del lenguaje orientado a objetos que se utilice para la implementación. Más importante aún, es poco probable que se haya dominado la complejidad del problema que se tiene entre manos.



La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

Abstracción

El significado de la abstracción. La abstracción es una de las vías fundamentales por la que los humanos combatimos la complejidad. Hoare sugiere que «la abstracción surge de un reconocimiento de las similitudes entre ciertos objetos, situaciones o procesos del mundo real, y la decisión de concentrarse en esas similitudes e ignorar por el momento las diferencias» [42]. Shaw define una abstracción como «una descripción simplificada o especificación de un sistema que enfatiza algunos de los detalles o propiedades del mismo mientras suprime otros. Una buena abstracción es aquella que enfatiza detalles significativos al lector o usuario y suprime detalles que son, al menos por el momento, irrelevantes o causa de distracción» [43]. Berzins, Gray y Naumann recomiendan que «un concepto merece el calificativo de abstracción sólo si se puede describir, comprender y analizar independientemente del mecanismo que vaya a utilizarse eventualmente para realizarlo» [44]. Combinando estos diferentes puntos de vista, se define una abstracción del modo siguiente:

Una abstracción denota las características esenciales de un objeto que

lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador.

Una abstracción se centra en la visión externa de un objeto, y, por tanto, sirve para separar el comportamiento esencial de un objeto de su implantación. Abelson y Sussman llaman a esta división comportamiento/implantación una *barrera de abstracción* [45] que se consigue aplicando el principio de mínimo compromiso, mediante el cual el interfaz de un objeto muestra su comportamiento esencial, y nada más [46]. Puede citarse aquí un principio adicional que se ha denominado el *principio de mínima sorpresa*, por el cual una abstracción captura el comportamiento completo de algún objeto, ni más ni menos, y no ofrece sorpresas o efectos laterales que lleguen más allá del ámbito de la abstracción.

La decisión sobre el conjunto adecuado de abstracciones para determinado dominio es el problema central del diseño orientado a objetos. Dada la importancia de este tema, todo el Capítulo 4 está dedicado a él.

Seidewitz y Stark sugieren que «existe una gama de abstracción, desde los objetos que modelan muy de cerca entidades del dominio del problema a objetos que no tienen una verdadera razón para existir» [47]. De mayor a menor utilidad, estos tipos de abstracción incluyen:

- Abstracción de entidades Un objeto que representa un modelo útil de una entidad del dominio del problema o del dominio de la solución.
- Abstracción de acciones Un objeto que proporciona un conjunto generalizado de operaciones, y todas ellas desempeñan funciones del mismo tipo.
- Abstracción de máquinas virtuales Un objeto que agrupa operaciones, todas ellas virtuales utilizadas por algún nivel superior de control, u operaciones que utilizan todas algún conjunto de operaciones de nivel inferior.
- Abstracción de coincidencia Un objeto que almacena un conjunto de operaciones que no tienen relación entre sí.

Se persigue construir abstracciones de entidades, porque imitan directamente el vocabulario de un determinado dominio de problema.

Un *cliente* es cualquier objeto que utiliza los recursos de otro objeto (denominado *servidor*). Se puede caracterizar el comportamiento de un objeto considerando los servicios que presta a otros objetos, así como las operaciones que puede realizar sobre otros objetos. Este punto de vista obliga a concentrarse en la visión exterior del objeto, y lleva a lo que Meyer llama el *modelo contractual*³ de programación [48]: la vista exterior de cada objeto define un contrato del que pueden depender otros objetos, y que a su vez debe ser llevado a cabo por la vista interior del propio objeto (a menudo en colaboración con otros objetos).

³ También denominado modelo de programación por contratos. (N. del T.)

Así, este contrato establece todas las suposiciones que puede hacer un objeto cliente acerca del comportamiento de un objeto servidor. En otras palabras, este contrato abarca las *responsabilidades* de un objeto, es decir, el comportamiento del que se le considera responsable [49].

Individualmente, cada operación que contribuye a este contrato tiene una sola firma que comprende todos sus argumentos formales y tipo de retorno. Al conjunto completo de operaciones que puede realizar un cliente sobre un objeto, junto con las formas de invocación u órdenes que admite, se le denomina su *protocolo*. Un protocolo denota las formas en las que un objeto puede actuar y reaccionar, y de esta forma constituye la visión externa completa, estática y dinámica, de la abstracción.

Un concepto central a la idea de abstracción es el de la invariancia. Un *invariante* es una condición booleana (verdadera o falsa) cuyo valor de verdad debe mantenerse. Para cualquier operación asociada con un objeto, es necesario definir *precondiciones* (invariantes asumidos por la operación) y *postcondiciones* (invariantes satisfechos por la operación). La violación de un invariante rompe el contrato asociado con una abstracción. Si se viola una precondición, esto significa que un cliente no ha satisfecho su parte del convenio, y por tanto el servidor no puede proceder con fiabilidad. Del mismo modo, si se viola una postcondición, significa que un servidor no ha llevado a cabo su parte del contrato, y por tanto sus clientes ya no pueden seguir confiando en el comportamiento del servidor. Una «excepción» es una indicación de que no se ha satisfecho (o no puede satisfacerse) algún invariante. Como se describe posteriormente, algunos lenguajes permiten a los objetos lanzar excepciones, así como abandonar el procesamiento, avisando del problema a algún otro objeto que puede a su vez aceptar la excepción y tratar dicho problema.

Como dato curioso, los términos *operación*, *método* y *función miembro* surgiieron de tres culturas de programación diferentes (Ada, Smalltalk y C++ respectivamente). Todos significan prácticamente lo mismo, y aquí se utilizarán indistintamente.

Todas las abstracciones tienen propiedades tanto estáticas como dinámicas. Por ejemplo, un objeto archivo ocupa una cierta cantidad de espacio en un dispositivo de memoria concreto; tiene un nombre y tiene un contenido. Todas éstas son propiedades estáticas. El valor de estas propiedades es dinámico, relativo al tiempo de vida del objeto: un objeto archivo puede aumentar o contraer su tamaño, su nombre puede cambiar, su contenido puede cambiar. En un estilo de programación orientado a procedimientos, la actividad que cambia el valor dinámico de los objetos es la parte central de todos los programas: ocurren cosas cuando se llama a subprogramas y se ejecutan instrucciones. En un estilo de programación orientado a reglas, ocurren cosas cuando hay nuevos eventos que producen el disparo de ciertas reglas, que a su vez pueden disparar a otras reglas, y así sucesivamente. En un estilo de programación orientado a objetos, ocurren cosas cuando se opera sobre un objeto (en terminología de Smalltalk, cuando se *envía un mensaje* a un objeto). Así, la invocación de una operación sobre un objeto da lugar a una reacción del mismo. Las operaciones significa-

tivas que pueden realizarse sobre un objeto y las reacciones de este ante ellas constituyen el comportamiento completo del objeto.

Ejemplos de abstracción. Pasamos a ilustrar estos conceptos con algunos ejemplos. Se pretende mostrar cómo pueden expresarse abstracciones de forma concreta, más que cómo encontrar las abstracciones adecuadas para determinado problema. Esto último se trata de forma completa en el Capítulo 4.

En una granja hidropónica, las plantas se cultivan en una solución nutritiva, sin arena, gravilla ni ningún otro tipo de suelo. Mantener el ambiente adecuado en el invernadero es un trabajo delicado, y depende del tipo de planta que se cultiva y su edad. Hay que controlar factores diversos, como temperatura, humedad, luz, pH y concentraciones de nutrientes. En una granja de gran tamaño no es extraño disponer de un sistema automático que monitoriza y ajusta constantemente estos elementos. Dicho de manera simple, el propósito de un jardineró automático es llevar a cabo eficientemente, con mínima intervención humana, el cultivo de plantas para la producción correcta de múltiples cosechas.

Una de las abstracciones clave en este problema es la de sensor. En realidad, hay varios tipos diferentes de sensor. Hay que medir cualquier cosa que afecte a la producción, y por tanto hay que tener sensores para la temperatura del aire y del agua, la humedad, la luz, el pH y las concentraciones de nutrientes, entre otras cosas. Visto desde fuera, un sensor de temperatura no es más que un objeto que sabe cómo medir la temperatura en alguna posición específica. ¿Qué es una temperatura? Es algún valor numérico, dentro de un rango limitado y con cierta precisión, que representa grados en la escala Fahrenheit, Centígrada o Kelvin, la que resulte más apropiada para el problema. ¿Qué es entonces una posición? Es algún lugar identificable de la granja en el que se desea medir la temperatura; es razonable suponer que hay sólo unos pocos lugares con esas características. Lo importante de un sensor de temperatura no es tanto el lugar en el que está, sino el hecho de que tiene una posición e identidad únicas respecto a todos los demás sensores de temperatura. Ahora se está en condiciones de plantear la siguiente cuestión: ¿Cuáles son las responsabilidades de un sensor de temperatura? La decisión de diseño es que un sensor es responsable de conocer la temperatura en determinada posición, e informar de esa temperatura cuando se le solicite. Más concretamente, ¿qué operaciones puede realizar un cliente sobre un sensor de temperatura? La decisión de diseño es que un cliente puede calibrarlo, así como preguntarle cuál es la temperatura actual.

Se utilizará C++ para capturar estas decisiones de diseño. Para los lectores que no estén familiarizados con C++, o con alguno de los demás lenguajes de programación orientados a objetos que se mencionan en este libro, el apéndice ofrece una breve visión general de varios lenguajes, con ejemplos. En C++, podrían escribirse las siguientes declaraciones⁴ que plasman la abstracción descrita de un sensor de temperatura:

⁴ El texto en castellano de los ejemplos en código fuente se transcribirá evitando el uso de caracteres ASCII expandidos, tales como las letras acentuadas, para observar la mayor estandarización y transportabilidad posibles. (N. del T.)

```
// Temperatura en grados Centígrados
typedef float Temperatura;

// Número que simplemente denota la posición de un sensor
typedef unsigned int Posicion;

Class SensorTemperatura {
public:

    SensorTemperatura(Posicion);
    ~SensorTemperatura();

    void calibrar(Temperatura temperaturaFijada);

    Temperatura temperaturaActual() const;

private:
    ...
};
```

Las dos definiciones de tipos (*typedef*), `Temperatura` y `Posicion`, proporcionan nombres convenientes para más tipos primitivos, permitiendo así expresar las abstracciones en el vocabulario del dominio del problema⁵. `Temperatura` es un tipo de punto flotante que representa la temperatura en grados Centígrados. El tipo `Posicion` denota los lugares en los que pueden desplegarse los sensores de temperatura a lo largo de la granja.

La clase `SensorTemperatura` se define como una clase, no como un objeto concreto, y por tanto hay que crear una *instancia* para tener algo sobre lo que operar. Por ejemplo, podría escribirse:

```
Temperatura temperatura;

SensorTemperatura sensorInvernadero1(1);
SensorTemperatura sensorInvernadero2(2);

temperatura = sensorInvernadero1.temperaturaActual();
...
```

Considérense los invariantes asociados con la operación `temperaturaActual`: sus precondiciones incluyen la suposición de que el sensor se ha creado con una posición válida, y sus postcondiciones incluyen la suposición de que el valor devuelto está en grados centígrados.

⁵ Desgraciadamente, sin embargo, las definiciones de tipos no introducen tipos nuevos, y por eso ofrecen poca seguridad en la comprobación de tipos. Por ejemplo, en C++ la declaración siguiente se limita a crear un sinónimo para el tipo primitivo `int`:

```
typedef int Contador;
```

Como se trata en una próxima sección, otros lenguajes como Ada y Eiffel tienen una semántica más rigurosa en lo que respecta a la comprobación estricta de tipos para los tipos primitivos.

La abstracción descrita hasta aquí es pasiva; tiene que haber un objeto cliente que opere sobre el sensor de temperatura del aire para determinar su temperatura actual. Sin embargo, hay otra abstracción lícita que puede ser más o menos apropiada de acuerdo con las decisiones de diseño más amplias que se puedan tomar. Específicamente, en vez de tener un sensor de temperatura pasivo, podría hacerse que fuese activo, de forma que no se actuase sobre él sino que fuese él quien actuase sobre otros objetos cuando la temperatura en su posición cambiase en cierto número de grados respecto a un punto de referencia fijado. Esta abstracción es casi idéntica a la inicial, excepto en que sus responsabilidades han cambiado ligeramente: un sensor es ahora responsable de informar de la temperatura actual cuando cambia, no cuando se le interroga sobre ella. ¿Qué nuevas operaciones debe ofrecer esta abstracción? Un término habitual en programación que se utiliza en circunstancias de este tipo es la llamada *callback*⁶, en la que el cliente proporciona una función al servidor (la función *callback*), y el servidor llama a esta función del cliente cuando se cumplen determinadas condiciones. Así, podría escribirse lo siguiente:

```
class SensorTemperaturaActivo {
public:

    SensorTemperaturaActivo(Posicion,
                           void (*f)(Posicion, Temperatura));
    ~SensorTemperaturaActivo();

    void calibrar(Temperatura temperaturaFijada);
    void establecerPuntoReferencia(Temperatura puntoReferencia,
                                    Temperatura incremento);

    Temperatura temperaturaActual() const;

private:
    ...
};
```

Esta clase es un poco más complicada que la primera, pero representa bastante bien la nueva abstracción. Siempre que se crea un objeto sensor, hay que suministrarle su posición igual que antes, pero ahora hay que proporcionar además una función callback cuya firma incluye un parámetro *Posicion* y un parámetro *Temperatura*. Adicionalmente, un cliente de esta abstracción puede invocar la operación *establecerPuntoReferencia* para fijar un rango crítico de temperaturas. Es entonces responsabilidad del objeto *SensorTemperatura Activo* el invocar la función *callback* dada siempre que la temperatura en su posición caiga por debajo o se eleve por encima del punto de referencia elegido.

⁶ Algunos autores traducen la expresión *callback function* como *función de retorno*; aunque el término *callback* no es por el momento palabra clave, su uso ya habitual aconseja adoptar la palabra inglesa para evitar mayor confusión terminológica mientras no se imponga una traducción comúnmente aceptada. (*N. del T.*)

Cuando se invoca a la función `callback`, el sensor proporciona su posición y la temperatura en ese momento, de forma que el cliente tenga suficiente información para responder a esa situación.

Nótese que un cliente sigue teniendo la posibilidad de interrogar al sensor sobre la temperatura en cualquier momento. ¿Qué pasa si un cliente nunca establece un punto de referencia? La abstracción debe realizar alguna suposición razonable: una decisión de diseño puede ser asumir inicialmente un rango infinito de temperaturas críticas, y así nunca se realizará la llamada `callback` hasta que algún cliente establezca un punto de referencia.

El cómo lleva a cabo sus responsabilidades la clase `SensorTemperaturaActiva` es función de sus aspectos internos, y no es de la incumbencia de los clientes externos. Éstos son, pues, secretos de la clase, implantados mediante las partes privadas de la misma junto con las definiciones de las funciones miembro.

Considérese una abstracción distinta. Para cada cultivo, debe haber un plan de cultivo que describa cómo deben cambiar en el tiempo la temperatura, luz, nutrientes y otros factores para maximizar la cosecha. Un plan de cultivo es una abstracción de entidad lícita, porque forma parte del vocabulario del dominio del problema. Cada cultivo tiene su propio plan, pero los planes de todos los cultivos tienen la misma forma. Básicamente, un plan de cultivo es un mapa de fechas de actuación. Por ejemplo, en el día decimoquinto de la vida de cierto cultivo, el plan puede ser mantener la temperatura a 25° C durante 16 horas, encender las luces durante 14 de esas horas, y reducir entonces la temperatura a 18° C durante el resto del día, mientras se mantiene un pH ligeramente ácido.

Un plan de cultivo es, pues, responsable de llevar cuenta de todas las acciones interesantes relacionadas con el crecimiento del cultivo, asociadas con el momento en el que esas acciones deberían tener lugar. La decisión es también que no será necesario que un plan de cultivo lleve estas acciones a cabo; se asignará ésta responsabilidad a una abstracción diferente. De este modo, se crea una separación de intereses muy clara entre las partes lógicamente distintas del sistema, además de que se reduce el tamaño conceptual de cada abstracción individual.

Desde la perspectiva externa de un objeto de tipo plan de cultivo, un cliente debe tener la posibilidad de establecer los detalles de un plan, modificar un plan y solicitar información sobre un plan. Por ejemplo, podría haber un objeto que se situase en el límite del interfaz hombre/máquina y tradujese la información introducida por el hombre a los planes. Este es el objeto que establece los detalles de un plan de cultivo, y por tanto debe ser capaz de cambiar el estado de un objeto de tipo plan de cultivo. Debe existir también un objeto que ejecute el plan de cultivo, y debe ser capaz de leer los detalles de un plan para determinado momento.

Como apunta este ejemplo, ningún objeto está solo; cada objeto colabora con otros objetos para conseguir cierto comportamiento⁷. Las decisiones de di-

⁷ Dicho de otro modo, con disculpas para el poeta John Donne, ningún objeto es una isla (aunque una isla puede abstraerse como un objeto).

seño acerca de cómo cooperan esos objetos entre sí definen las fronteras de cada abstracción y por tanto las responsabilidades y el protocolo para cada objeto.

Podrían plasmarse las decisiones de diseño tomadas para un plan de cultivo como sigue. Primero, se suministran las siguientes definiciones de tipos, para llevar a las abstracciones más cerca del vocabulario del problema dominio:

```
// Número denotando el día del año
typedef unsigned int Dia;

// Número denotando la hora del día
typedef unsigned int Hora;

// Tipo booleano
enum Luces {OFF, ON};

// Número denotando acidez/basicidad en una escala de 1 a 14
typedef float pH;

// Número denotando concentración porcentual de 0 a 100
typedef float Concentracion;
```

A continuación, como decisión táctica de diseño, se incluye la siguiente estructura:

```
// Estructura que denota condiciones relevantes para el plan
struct Condicion{
    Temperatura temperatura;
    Luces iluminación;
    pH acidez;
    Concentración concentración;
};
```

Aquí hay algo que no llega a ser una abstracción de entidad: una Condicion es simplemente una agregación física de otras cosas, sin comportamiento intrínseco. Por esta razón, se utiliza una estructura de registro de C++, en lugar de una clase C++, que tiene una semántica más rica.

Por último, véase la propia clase de plan de cultivo:

```
class PlanCultivo {
public:

    PlanCultivo(char* nombre);
    virtual ~PlanCultivo();

    void borrar();
    virtual void establecer(Dia, Hora, const Condicion&);
```

```

const char* nombre() const;
const Condicion& condicionesDeseadas(dia, Hora) const;

protected:
    ...
};

```

Nótese que se ha asignado una nueva responsabilidad a esta abstracción: un plan de cultivo tiene un nombre, el cual puede ser fijado o examinado por un cliente. Nótese también que se declara la operación establecer como virtual, porque se espera que las subclases sobreescriban el comportamiento por defecto proporcionado por la clase PlanCultivo.

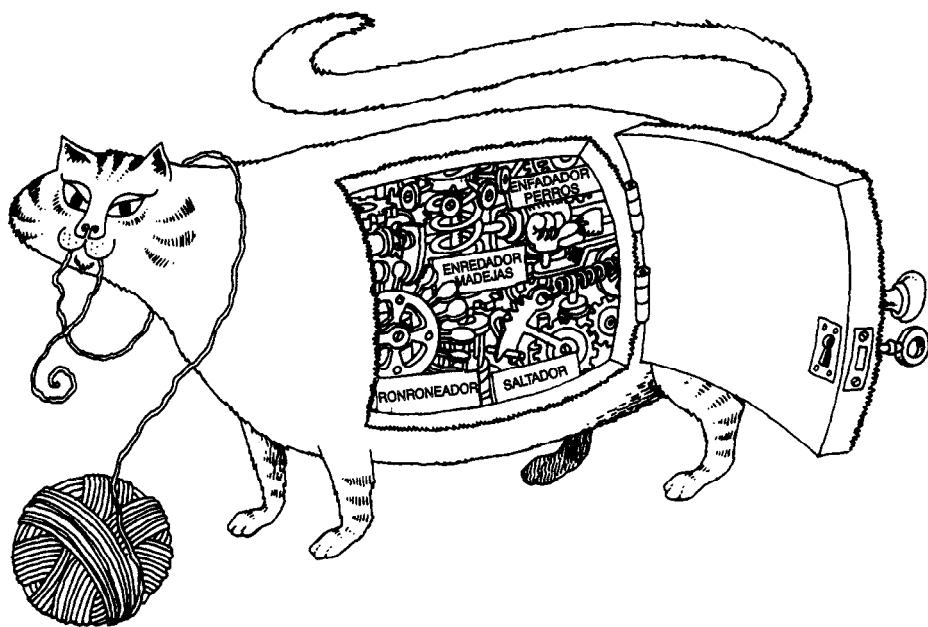
En la declaración de esta clase, la parte pública exporta las *funciones miembro constructor y destructor* (que proporciona para el nacimiento y muerte de un objeto, respectivamente), dos *modificadores* (las funciones miembro borrar y establecer), y dos *selectores* (las funciones miembro nombre y condicionesDeseadas). Se ha dejado fuera de forma intencionada a los miembros privados (designados con puntos suspensivos), porque en este punto del diseño la atención se centra sólo en las responsabilidades de la clase, no su representación.

Encapsulamiento

El significado del encapsulamiento⁸. Aunque anteriormente se describió la abstracción de la clase PlanCultivo como un esquema tiempo/acción, su implantación no es necesariamente una tabla en sentido literal o una estructura de datos con forma de esquema. En realidad, se elija la representación que se elija, será indiferente al contrato del cliente con esa clase, siempre y cuando la representación apoye el contrato. Dicho sencillamente, la abstracción de un objeto debería preceder a las decisiones sobre su implementación. Una vez que se ha seleccionado una implantación, debe tratarse como un secreto de la abstracción, oculto para la mayoría de los clientes. Como sugiere sabiamente Ingalls, «ninguna parte de un sistema complejo debe depender de los detalles internos de otras partes» [50]. Mientras la abstracción «ayuda a las personas a pensar sobre lo que están haciendo», el encapsulamiento «permite que los cambios hechos en los programas sean fiables con el menor esfuerzo» [51].

La abstracción y el encapsulamiento son conceptos complementarios: la abstracción se centra en el comportamiento observable de un objeto, mientras el *encapsulamiento* se centra en la implementación que da lugar a este comportamiento. El encapsulamiento se consigue a menudo mediante la *ocultación de información*, que es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales; típicamente, la estructura de un objeto está oculta, así como la implantación de sus métodos.

⁸ También se utiliza el término encapsulación. (*N. del T.*)



El encapsulamiento oculta los detalles de la implementación de un objeto.

El encapsulamiento proporciona barreras explícitas entre abstracciones diferentes y por tanto conduce a una clara separación de intereses. Por ejemplo, considérese una vez más la estructura de una planta. Para comprender cómo funciona la fotosíntesis a un nivel alto de abstracción, se pueden ignorar detalles como los cometidos de las raíces de la planta o la química de las paredes de las células. Análogamente, al diseñar una aplicación de bases de datos, es práctica común el escribir programas de forma que no se preocupen de la representación física de los datos, sino que dependan sólo de un esquema que denota la vista lógica de los mismos [52]. En ambos casos, los objetos a un nivel de abstracción están protegidos de los detalles de implementación a niveles más bajos de abstracción.

Liskov llega a sugerir que «para que la abstracción funcione, la implementación debe estar encapsulada» [53]. En la práctica, esto significa que cada clase debe tener dos partes: un interfaz y una implementación. El *interfaz** de una clase captura sólo su vista externa, abarcando la abstracción que se ha hecho del comportamiento común de todas las instancias de la clase. La *implementación* (implantación) de una clase comprende la representación de la abstracción así como los mecanismos que consiguen el comportamiento deseado. El interfaz de una clase es el único lugar en el que se declaran todas las suposiciones que un

* Se suele utilizar también el término con género femenino: la interfaz. (N. del T.)

cliente puede hacer acerca de todas las instancias de la clase; la implantación encapsula detalles acerca de los cuales ningún cliente puede realizar suposiciones.

Para resumir, se define el *encapsulamiento* como sigue:

El encapsulamiento es el proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar el interfaz contractual de una abstracción y su implantación.

Britton y Parnas llaman a esos elementos encapsulados los «secretos» de una abstracción [54].

Ejemplos de encapsulamiento. Para ilustrar el principio del encapsulamiento, volvamos al problema del sistema de cultivo hidropónico. Otra abstracción clave en este dominio de problema es la del calentador. Un calentador está en un nivel de abstracción claramente bajo, y por eso se podría decidir que hay sólo tres operaciones significativas que puedan realizarse sobre este objeto: activarlo, desactivarlo y saber si está funcionando. No se considera misión de esta abstracción el mantener una temperatura fijada. En lugar de eso, se elige otorgar esta responsabilidad a otro objeto, que debe colaborar con un sensor de temperatura y un calentador para lograr este comportamiento de nivel superior. Se llama a este comportamiento *de nivel superior* porque se apoya en la semántica primitiva de los sensores de temperatura y calentadores y añade alguna semántica nueva, a saber, la *histéresis*, que evita que el calentador sea encendido y apagado con demasiada rapidez cuando la temperatura está alrededor de la situación fronteriza. Decidiendo sobre esta separación de responsabilidades, se hace cada abstracción individual más cohesiva.

Comencemos con otra definición de tipos:

```
// Tipo booleano
enum Booleano { FALSE, TRUE };
```

Para la clase calentador, además de las tres operaciones mencionadas anteriormente, hay que suministrar también metaoperaciones, es decir, operaciones constructor y destructor que inicializan y destruyen instancias de esta clase, respectivamente. Ya que el sistema puede tener muchos calentadores, se utiliza el constructor para asociar cada objeto de software con un calentador físico, de forma análoga al enfoque adoptado con la clase SensorTemperatura. Dadas estas decisiones de diseño, se podría redactar la definición de la clase Calentador en C++ como sigue:

```
class Calentador {
public:
    Calentador(Posicion);
```

```

~Calentador();

void activar();
void desactivar();

Booleano encendido() const;

private:
...
};

```

Este interfaz representa todo lo que un cliente necesita saber sobre la clase Calentador.

En cuanto a la vista interna de esta clase, la perspectiva es completamente diferente. Supóngase que los ingenieros del sistema han decidido situar los computadores que controlan cada invernadero lejos del edificio (quizás para evitar el ambiente hostil), y conectar cada computador a sus sensores y actuadores mediante líneas serie. Una implantación razonable para la clase calentador puede ser utilizar un relé electromagnético que controla la llegada de energía a cada calentador físico, con los relés gobernados a su vez por mensajes enviados por esas líneas serie. Por ejemplo, para activar un calentador, se puede transmitir una cadena de comando especial, seguida por un número que identifica el calentador específico, seguido por otro número utilizado para señalizar el encendido del calentador.

Considérese la clase siguiente, que reproduce la abstracción que se ha hecho de un puerto serie:

```

class PuertoSerie {
public:

PuertoSerie();
~PuertoSerie();

void escribir(char *);
void escribir(int);

static PuertoSerie puertos[10];

private:
...
~;

```

Aquí se proporciona una clase cuyas instancias denotan puertos serie actuales, en los que pueden escribirse cadenas de caracteres (**strings**) y enteros. Además se declara una matriz de puertos serie, denotando todos los distintos puertos serie en los sistemas.

Se completa la declaración de la clase Calentador añadiendo tres atributos:

```

class Calentador {
public:
    ...
protected:
    const Posicion repPosicion;
    Booleano repEncendido;
    PuertoSerie* repPuerto;
};

```

Estos tres atributos (`repPosicion`, `repEncendido` y `repPuerto`) forman la representación encapsulada de esta clase. Las reglas del C++ hacen que la compilación del código de un cliente que intente acceder a esos objetos miembro directamente resulte en un error semántico.

Puede proporcionarse ahora la implantación de cada operación asociada con esta clase:

```

Calentador::Calentador(Posicion p)
: repPosicion(p),
  repEncendido(FALSE),
  repPuerto(&PuertoSerie::puertos[p]) {}

Calentador::~Calentador() {}

void Calentador::activar()
{
    if (!repEncendido) {
        repPuerto->escribir("*");
        repPuerto->escribir(repPosicion);
        repPuerto->escribir(1);
        repEncendido = TRUE;
    }
}

void Calentador::desactivar()
{
    if (repEncendido) {
        repPuerto->escribir("*");
        repPuerto->escribir(repPosicion);
        repPuerto->escribir(0);
        repEncendido = FALSE;
    }
}

Booleano Calentador::encendido() const
{
    return repEncendido;
}

```

Esta implantación es típica de los sistemas orientados a objetos bien estructurados: la implantación de una clase particular es en general pequeña, porque puede apoyarse en los recursos proporcionados por clases de niveles inferiores.

Supóngase que por cualquier razón los ingenieros del sistema deciden utilizar una E/S por correspondencia de memoria (*memory mapped I/O*) en lugar de líneas de comunicación en serie. No sería necesario cambiar el interfaz de esta clase; sólo se precisaría modificar su implantación. A causa de las reglas sobre obsolescencia del C++, posiblemente habría que recompilar esta clase y el cierre de sus clientes, pero al permanecer sin cambios el comportamiento funcional de esta clase, no habría que modificar en absoluto el código que utilice a esta clase a menos que un cliente concreto dependiese de la semántica espacial y temporal de la implantación original (lo que sería altamente indeseable y, por tanto, muy improbable, en cualquier caso).

Considérese ahora la implantación de la clase `PlanCultivo`. Como ya se mencionó, un plan de cultivo es en esencia un esquema momento/acción. Quizás la representación más razonable para esta abstracción sería un diccionario de pares hora/acción, utilizando una tabla por claves abierta⁹. No es preciso almacenar una acción para todas las horas del día, porque las cosas no cambian tan rápido. En vez de eso, puede almacenarse las acciones sólo para cuando cambian, y tener la implantación extrapolada entre una hora y otra.

De esta forma, la implantación encapsula dos secretos: el uso de una tabla por claves abierta (que es claramente parte del vocabulario del dominio de la solución, no del dominio del problema), y el uso de la extrapolación para reducir las necesidades de almacenamiento (de otro modo habría que almacenar muchos más pares momento/acción a lo largo de la duración de una sesión de cultivo). Ningún cliente de esta abstracción necesita saber nada sobre estas decisiones de implantación, porque no afectan materialmente el comportamiento de la clase observable exteriormente.

Un encapsulamiento inteligente hace que sean locales las decisiones de diseño que probablemente cambien. A medida que evoluciona un sistema, sus desarrolladores pueden descubrir que en una utilización real, ciertas operaciones llevan más tiempo que el admisible o que algunos objetos consumen más espacio del disponible. En esas situaciones, la representación de un objeto suele cambiarse para poder aplicar algoritmos más eficientes o para que pueda ahorrarse espacio calculando algunos datos cuando sean necesarios, en vez de tenerlos almacenados. Esta capacidad para cambiar la representación de una abstracción sin alterar a ninguno de sus clientes es el beneficio esencial de la encapsulación.

Idealmente, los intentos de acceder a la representación subyacente de un objeto deberían detectarse en el momento en que se compila el código de un cliente. El modo en que un lenguaje concreto debería enfrentarse a este asunto se debate con fervor casi religioso en la comunidad de los lenguajes de programación

⁹ Es de uso corriente el término inglés, *tabla hash*; en este texto, se utilizará el término equivalente en español, *tabla por claves*, aunque también se usa *tabla de dispersión*. (N. del T.)

orientados a objetos. Por ejemplo, Smalltalk evita que un cliente acceda directamente a las variables de instancia de otra clase; las violaciones se detectan en tiempo de compilación. Por contra, Object Pascal no encapsula la representación de una clase, así que no hay nada en el lenguaje que evite que los clientes referencien directamente los campos de otro objeto. CLOS adopta una posición intermedia; cada *slot* debe tener una de las opciones de ranura: `:reader`, `:writer` o `:accessor`, que garantizan a un cliente acceso de lectura, de escritura o ambos, respectivamente. Si no se utiliza ninguna de esas opciones, el slot está completamente encapsulado. Por convenio, el revelar el valor que se almacena en un slot se considera un derrumbamiento de la abstracción, y por eso el buen estilo CLOS requiere que, cuando se hace público el interfaz de una clase, sólo se documenten los nombres de sus funciones genéricas, y el hecho de que un slot tenga funciones que acceden a él no se revele [55]. C++ ofrece un control aún más flexible sobre la visibilidad de objetos miembro y funciones miembro. Concretamente, los miembros pueden situarse en la parte *public*, *private* o *protected* de una clase. Los miembros declarados en la parte *public* son visibles a todos los clientes; los miembros declarados en la parte *private* están completamente encapsulados; y los miembros declarados en la parte *protected* son visibles sólo para la propia clase y sus subclases. C++ también soporta la noción de «amigas» (*friends*): clases colaboradoras que, sólo entre sí, permiten ver las partes privadas (*private*).

La ocultación es un concepto relativo: lo que está oculto a un nivel de abstracción puede representar la visión externa a otro nivel de abstracción. La representación subyacente de un objeto puede revelarse, pero en la mayoría de los casos solamente si el creador de la abstracción expone la implantación de forma explícita, y entonces sólo si el cliente está dispuesto a aceptar la complejidad adicional resultante. Así, el encapsulamiento no puede evitar que un desarrollador haga cosas estúpidas: como apunta Stroustrup, «la ocultación es para prevenir accidentes, no para prevenir el fraude» [56]. Por supuesto, ningún lenguaje de programación evita que un humano vea literalmente la implantación de una clase, aunque el sistema operativo puede denegar el acceso a determinado fichero que contiene la implantación de la misma. En la práctica, hay ocasiones en las que es necesario estudiar la implantación de una clase para comprender realmente su significado, especialmente si la documentación externa es deficiente.

Modularidad

El significado de la modularidad. Como observa Myers, «el acto de fragmentar un programa en componentes individuales puede reducir su complejidad en algún grado... Aunque la fragmentación de programas es útil por esta razón, una justificación más poderosa para esta fragmentación es que crea una serie de fronteras bien definidas y documentadas dentro del programa. Estas fronteras o interfaces, tienen un incalculable valor cara a la comprensión del



La modularidad empaquetada las abstracciones en unidades discretas.

programa» [57]. En algunos lenguajes, como Smalltalk, no existe el concepto de módulo, y así la clase es la única unidad física de descomposición. En muchos otros, incluyendo Object Pascal, C++, CLOS y Ada, el módulo es una construcción adicional del lenguaje y, por lo tanto, justifica un conjunto separado de decisiones de diseño. En estos lenguajes, las clases y los objetos forman la estructura lógica de un sistema; se sitúan estas abstracciones en *módulos* para producir la arquitectura física del sistema. Especialmente para aplicaciones más grandes, en las que puede haber varios cientos de clases, el uso de módulos es esencial para ayudar a manejar la complejidad.

Liskov establece que «la modularización consiste en dividir un programa en módulos que pueden compilarse separadamente, pero que tienen conexiones con otros módulos. Utilizaremos la definición de Parnas: “Las conexiones entre módulos son las suposiciones que cada módulo hace acerca de todos los demás”» [58]. La mayoría de los lenguajes que soportan el módulo como un concepto adicional distinguen también entre el interfaz de un módulo y su implementación. Así, es correcto decir que la modularidad y el encapsulamiento van de la mano. Como en el encapsulamiento, los lenguajes concretos soportan la modularidad de formas diversas. Por ejemplo, los módulos en C++ no son más que ficheros compilados separadamente. La práctica tradicional en la comunidad de C y C++ es colocar los interfaces de los módulos en archivos cuyo nombre lleva el sufijo *.h*; se llaman *archivos cabecera*. Las implementaciones de los

módulos se sitúan en archivos cuyo nombre lleva el sufijo *.c*¹⁰. Las dependencias entre archivos pueden declararse mediante la macro `#include`. Este enfoque es por completo una convención; no es ni requerido ni promovido por el lenguaje en sí mismo. Object Pascal es algo más formal en este asunto. En este lenguaje, la sintaxis de las *units* (su nombre para los módulos) distingue entre el interfaz y la implantación de un módulo. Las dependencias entre *units* pueden declararse solamente en el interfaz del módulo. Ada va un paso más allá. Un *package* (nombre que da a los módulos) tiene dos partes: la especificación del paquete y el cuerpo del mismo. Al contrario que Object Pascal, Ada permite declarar separadamente las conexiones entre módulos, en la especificación y en el cuerpo de un paquete. Así, es posible para el cuerpo de un package depender de módulos que de otro modo no son visibles para la especificación del paquete.

La decisión sobre el conjunto adecuado de módulos para determinado problema es un problema casi tan difícil como decidir sobre el conjunto adecuado de abstracciones. Zelkowitz está totalmente en lo cierto cuando afirma que «puesto que no puede conocerse la solución cuando comienza la etapa de diseño, la descomposición en módulos más pequeños puede resultar bastante difícil. Para aplicaciones más antiguas (como la escritura de compiladores), este proceso puede llegar a estandarizarse, pero para otras nuevas (como sistemas de defensa o control de naves espaciales) puede ser bastante complicado» [59].

Los módulos sirven como los contenedores físicos en los que se declaran las clases y objetos del diseño lógico realizado. La situación no es muy distinta a la que se encuentra un ingeniero electrónico que diseña un computador a nivel de placa. Pueden utilizarse puertas NAND, NOR y NOT para construir la lógica necesaria, pero estas puertas deben estar empaquetadas físicamente en circuitos integrados estándar, como un 7400, 7402 o 7404. Al no disponer de partes estándares semejantes en software, el ingeniero del software tiene muchos más grados de libertad —como si el ingeniero electrónico tuviese una fundición de silicio a su disposición.

Para problemas muy pequeños, el desarrollador podría decidir declarar todas las clases y objetos en el mismo paquete. Para cualquier cosa que se salga de lo trivial, es mejor solución agrupar las clases y objetos que se relacionan lógicamente en el mismo módulo, y exponer solamente los elementos que otros módulos necesitan ver con absoluta necesidad. Este tipo de modularización es algo bueno, pero puede llevarse al extremo. Por ejemplo, considérese una aplicación que se ejecuta en un conjunto distribuido de procesadores y utiliza un mecanismo de paso de mensajes para coordinar las actividades de distintos programas. En un sistema grande como el que se describe en el Capítulo 12, es frecuente tener varios cientos o incluso algunos miles de tipos de mensaje. Una estrategia ingenua podría ser definir cada clase de mensaje en su propio módulo. El caso es que esta es una decisión de diseño tremadamente mala. No sólo constituye una pesadilla a la hora de documentar, sino que es terriblemente difícil para cualquier usuario encontrar las clases que necesita. Además, cuando

¹⁰ Los sufijos *.cp* y *.cpp* se utilizan habitualmente para programas en C++.

cambian las decisiones, hay que modificar o recompilar cientos de módulos. Este ejemplo muestra cómo la ocultación de información puede ser un arma de doble filo [60]. La modularización arbitraria puede ser peor a veces que la ausencia de modularización.

En el diseño estructurado tradicional, la modularización persigue ante todo el agrupamiento significativo de subprogramas, utilizando los criterios de acoplamiento y cohesión. En el diseño orientado a objetos, el problema presenta diferencias sutiles: la tarea es decidir dónde empaquetar físicamente las clases y objetos a partir de la estructura lógica del diseño, y éstos son claramente diferentes de los subprogramas.

La experiencia indica que hay varias técnicas útiles, así como líneas generales no técnicas, que pueden ayudar a conseguir una modularización inteligente de clases y objetos. Como han observado Britton y Parnas, «el objetivo de fondo de la descomposición en módulos es la reducción del coste del software al permitir que los módulos se diseñen y revisen independientemente... La estructura de cada módulo debería ser lo bastante simple como para ser comprendida en su totalidad; debería ser posible cambiar la implantación de los módulos sin saber nada de la implantación de los demás módulos y sin afectar el comportamiento de éstos; [y] la facilidad de realizar un cambio en el diseño debería guardar una relación razonable con la probabilidad de que ese cambio fuese necesario» [61]. Hay un límite pragmático a esas líneas maestras. En la práctica, el coste de recompilar el cuerpo de un módulo es relativamente pequeño: sólo hay que recompilar esa unidad y volver a enlazar la aplicación con el montador de enlaces. Sin embargo, el coste de recompilar el *interfaz* de un módulo es relativamente alto. Especialmente en lenguajes con comprobación estricta de tipos, hay que recompilar el interfaz del módulo, su cuerpo, y todos los demás módulos que dependen de ese interfaz, los módulos que dependen de estos otros módulos, y así sucesivamente. De este modo, para programas de gran tamaño (asumiendo que el entorno de desarrollo no soporte compilación incremental) un cambio en un simple interfaz de un módulo puede dar lugar a varios minutos, si no horas, de recompilación. Obviamente, un jefe de desarrollo no puede permitirse habitualmente el dejar que suceda con frecuencia un «big bang» de recompilaciones masivas. Por esta razón, el interfaz de un módulo debería ser tan estrecho como fuese posible, siempre y cuando se satisfagan las necesidades de todos los módulos que lo utilizan. Nuestro estilo es ocultar tanto como se pueda en la implantación de un módulo. Es mucho menos engorroso y desestabilizador desplazar incrementalmente las declaraciones desde la implantación del módulo hasta el interfaz que arrancar código de interfaces externas.

El desarrollador debe por tanto equilibrar dos intereses técnicos rivales: el deseo de encapsular abstracciones y la necesidad de hacer a ciertas abstracciones visibles para otros módulos. Parnas, Clements y Weiss ofrecen el siguiente consejo: «Los detalles de un sistema, que probablemente cambien de forma independiente, deberían ser secretos en módulos separados; las únicas suposiciones que deberían darse entre módulos son aquellas cuyo cambio se considera improbable. Toda estructura de datos es privada a algún módulo; a ella pueden

acceder directamente uno o más programas del módulo, pero no programas de fuera del módulo. Cualquier otro programa que requiera información almacenada en los datos de un módulo debe obtenerla llamando a programas de éste» [62]. Dicho de otro modo, hay que hacer lo posible por construir módulos cohesivos (agrupando abstracciones que guarden cierta relación lógica) y débilmente acoplados (minimizando las dependencias entre módulos). Desde esta perspectiva, puede definirse la modularidad como sigue:

La modularidad es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.

Así, los principios de abstracción, encapsulamiento y modularidad son sínrgicos. Un objeto proporciona una frontera bien definida alrededor de una sola abstracción, y tanto el encapsulamiento como la modularidad proporcionan barreras que rodean a esta abstracción.

Hay dos problemas técnicos más que pueden afectar a las decisiones de modularización. Primero, puesto que los módulos sirven usualmente como las unidades de software elementales e indivisibles a la hora de reutilizar código en diversas aplicaciones, un desarrollador debería empaquetar clases y objetos en módulos de forma que su reutilización fuese conveniente. Segundo, muchos compiladores generan código objeto en segmentos, uno para cada módulo. Por tanto, puede haber límites prácticos al tamaño de un módulo individual. Por lo que respecta a la dinámica de llamadas a subprogramas, la situación de las declaraciones en los módulos puede afectar en gran medida al grado de localidad de la referencia y, por tanto, al comportamiento de la paginación en un sistema de memoria virtual. Se da un bajo nivel de localidad cuando pueden producirse llamadas de subprogramas entre segmentos, y esto trae como consecuencia fallos de página e intensos trasiegos en la paginación, lo que en última instancia ralentiza todo el sistema.

También rivalizan varias necesidades no técnicas que pueden afectar a decisiones de modularización. La asignación de trabajo típica en un equipo de desarrollo se basa en una división por módulos, y hay que establecer las fronteras de éstos de forma que se minimicen los interfaces entre distintas partes de la organización de desarrollo. Por norma, los diseñadores experimentados se responsabilizan de los interfaces entre módulos, y los desarrolladores más jóvenes completan su implantación. A mayor escala, aparece la misma situación en las relaciones entre subcontratas. Se pueden empaquetar las abstracciones de manera que se estabilicen rápidamente los interfaces entre los módulos, tal y como se haya acordado entre las distintas compañías. El cambio de estos interfaces suele conllevar mucho llanto y rechinaz de dientes —eso sin mencionar una ingente cantidad de papeleo—, y, por eso, este factor lleva a menudo a interfaces de diseño conservador. Hablando de papeleo, los módulos sirven también normalmente como la unidad de gestión para la documentación y configuración. Tener diez módulos donde sólo haría falta uno significa algunas veces diez veces más papeleo, y por eso, desgraciadamente, hay ocasiones en que los requisitos de la documentación influyen en las decisiones de diseño de módulos (casi

siempre de la forma más negativa). La seguridad también puede ser un problema: la mayor parte del código puede considerarse de dominio público, pero otro código susceptible de ser considerado secreto —o más que secreto— está mejor colocado en módulos aparte.

Es difícil conjugar todos estos requisitos, pero no hay que perder de vista la cuestión más importante: encontrar las clases y objetos correctos y organizarlos después en módulos separados son decisiones de diseño *ampliamente independientes*. La identificación de clases y objetos es parte del diseño lógico de un sistema, pero la identificación de los módulos es parte del diseño físico del mismo. No pueden tomarse todas las decisiones de diseño lógico antes de tomar todas las de diseño físico, o viceversa; por contra, estas decisiones de diseño se dan de forma iterativa.

Ejemplos de Modularidad. Obsérvese la modularidad en el sistema de cultivo hidropónico. Supóngase que en lugar de construir algún hardware de propósito específico, se decide utilizar una estación de trabajo disponible comercialmente, y emplear un interfaz gráfico de usuario (IGU) innovador. En esta estación de trabajo, un operador podría crear nuevos planes de cultivo, modificar planes viejos, y seguir el progreso de planes activos. Ya que una de las abstracciones clave es la del plan de cultivo, se podría por tanto crear un módulo cuyo propósito fuese agrupar todas las clases asociadas con planes de cultivo individuales. En C++, podría redactarse el archivo de cabecera para este módulo (que se llamará `plancult.h`) como sigue:

```
// plancult.h

#ifndef _PLANCULT_H
#define _PLANCULT_H 1

#include "tiposcul.h"
#include "excep.h"
#include "acciones.h"

class PlanCultivo ...
class PlanCultivoFruta ...
class PlanCultivoGrano ...
...
#endif
```

Aquí se importan otros tres archivos cabecera (`planescul.h`, `excep.h` y `acciones.h`), en cuyo interfaz hay que confiar.

Las implantaciones de estas clases de planes de cultivo aparecen en la im-

plantación de este módulo, en un fichero llamado (por convenio) `plan-cult.cpp`.

Se podría definir también un módulo cuyo propósito es recoger todo el código asociado con cuadros de diálogo específicos de la aplicación. Lo más probable es que esta unidad dependa de las clases declaradas en el interfaz de `plan-cult.h`, así como de ficheros que encapsulen ciertos interfaces del IGU, y por eso debe a su vez incluir el fichero cabecera `plancult.h` y los ficheros cabecera del IGU correspondientes.

El diseño probablemente incluirá muchos otros módulos, cada uno de los cuales importa el interfaz de unidades de nivel inferior. En última instancia, hay que definir algún programa principal a partir del cual se pueda invocar esta aplicación desde el sistema operativo. En el diseño orientado a objetos, la definición de este programa principal suele ser la decisión menos importante, mientras que en el diseño estructurado tradicional, el programa principal sirve como la raíz, la piedra angular que aglutina todo lo demás. Nuestra opinión es que el punto de vista orientado a objetos es más natural, porque como observa Meyer, «la forma más apropiada de definir sistemas de software prácticos es decir que ofrecen un cierto número de servicios. Normalmente la definición de estos sistemas como funciones simples es posible, pero eso produce respuestas bastante más artificiales... Los sistemas reales no tienen una parte superior» [63].

Jerarquía

El significado de la jerarquía. La abstracción es algo bueno, pero excepto en las aplicaciones más triviales, puede haber muchas más abstracciones diferentes de las que se pueden comprender simultáneamente. El encapsulamiento ayuda a manejar esta complejidad ocultando la visión interna de las abstracciones. La modularidad también ayuda, ofreciendo una vía para agrupar abstracciones relacionadas lógicamente. Esto sigue sin ser suficiente. Frecuentemente un conjunto de abstracciones forma una jerarquía, y la identificación de esas jerarquías en el diseño simplifica en gran medida la comprensión del problema.

Se define la jerarquía como sigue:

La jerarquía es una clasificación u ordenación de abstracciones.

Las dos jerarquías más importantes en un sistema complejo son su estructura de clases (la jerarquía «de clases») y su estructura de objetos (la jerarquía «de partes»).

Ejemplos de jerarquía: herencia simple. La herencia es la jerarquía «de clases» más importante y, como se apuntó anteriormente, es un elemento esencial de los sistemas orientados a objetos. Básicamente, la herencia define una relación entre clases, en la que una clase comparte la estructura de comportamiento definida en una o más clases (lo que se denomina *herencia simple* o *herencia múltiple*, respectivamente). La herencia representa así una jerarquía de abstrac-

ciones, en la que una subclase hereda de una o más superclases. Típicamente, una subclase aumenta o redefine la estructura y el comportamiento de sus superclases.

Semánticamente, la herencia denota una relación «es un». Por ejemplo, un oso «es un» tipo de mamífero, una casa «es un» tipo de bien inmueble, y el *quick sort* «es un» tipo particular de algoritmo de ordenación. Así la herencia implica una jerarquía de generalización/especialización, en la que una subclase especializa el comportamiento o estructura, más general, de sus superclases. Realmente, ésta es la piedra de toque para la herencia: si B «no es» un tipo de A, entonces B no debería heredar de A.

Considérense los diferentes tipos de planes de cultivo que se pueden usar en el sistema de cultivo hidropónico. Por ejemplo, el plan de cultivo para las frutas es generalmente el mismo, pero es bastante diferente del plan para las hortalizas, o para las flores. A causa de este agrupamiento de abstracciones, es razonable definir un plan de cultivo estándar para las frutas que encapsule el comportamiento especializado común a todas las frutas, como el conocimiento de cuándo hay que polinizar o cuándo cosechar los frutos. Se puede declarar en C++ esta relación «es un», o «de tipos», entre esas abstracciones como sigue:

```
// Tipo producción
typedef unsigned int Produccion;

class PlanCultivoFrutas : public PlanCultivo {
public:

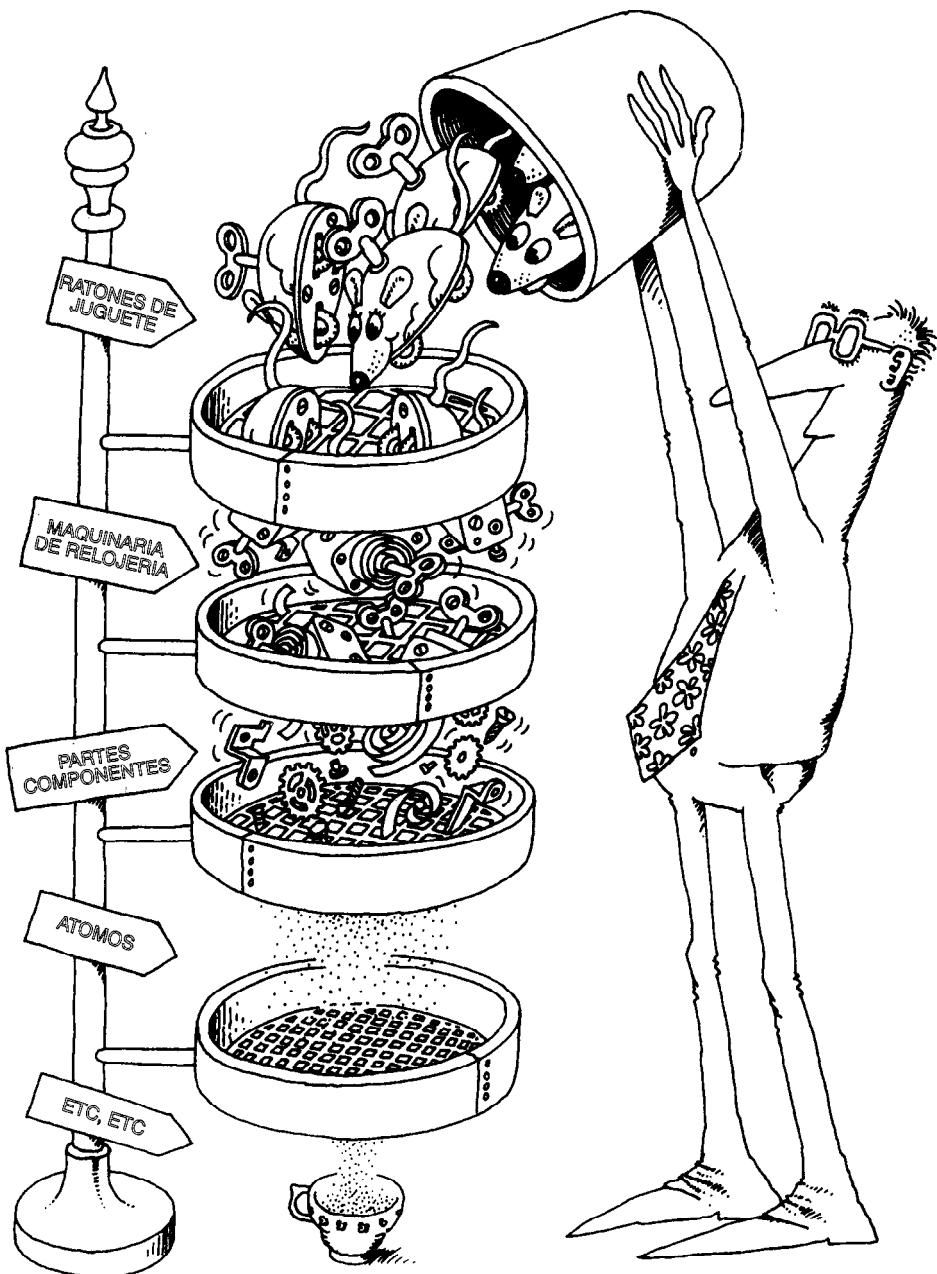
    PlanCultivoFrutas(char* nombre);
    virtual ~PlanCultivoFrutas();

    virtual void establecer(Dia, Hora, Condicion&);
    void planificarCosecha(Dia, Hora);

    Booleano yaCosechado() const;
    unsigned diasHastaCosecha() const;
    Produccion produccionEstimada() const;

protected:
    Booleano repCosechado;
    Produccion repProduccion;
};
```

Esta declaración de clase plasma la decisión de diseño por la que un `PlanCultivoFrutas` «es un» tipo de `PlanCultivo`, con algunas estructuras (los objetos miembro `repCosechado` y `repProducción`) y comportamiento (las cuatro nuevas funciones miembro, más la redefinición de la operación de la superclase `establecer`) adicionales. Utilizando esta clase, podrían declararse clases aún más especializadas, tales como la clase `PlanCultivoManzana`.



Las abstracciones forman una jerarquía.

A medida que se desarrolla la jerarquía de herencias, la estructura y comportamiento comunes a diferentes clases tenderá a migrar hacia superclases comunes. Por esta razón se habla a menudo de la herencia como una jerarquía de *generalización/especialización*. Las superclases representan abstracciones generalizadas, y las subclases representan especializaciones en las que los campos y métodos de la superclase sufren añadidos, modificaciones o incluso ocultaciones. De este modo, la herencia permite declarar las abstracciones con economía de expresión. De hecho, el ignorar las jerarquías «de clase» que existen pueden conducir a diseños deformados y poco elegantes. Como apunta Cox, «sin herencia, cada clase sería una unidad independiente, desarrollada partiendo de cero. Las distintas clases no guardarían relación entre sí, puesto que el desarrollador de cada clase proporcionaría métodos según le viniese en gana. Toda consistencia entre clases es el resultado de una disciplina por parte de los programadores. La herencia posibilita la definición de nuevo software de la misma forma en que se presenta un concepto a un recién llegado, comparándolo con algo que ya le resulte familiar» [64].

Existe una conveniente tensión entre los principios de abstracción, encapsulamiento y jerarquía. En palabras de Danforth y Tomlinson, «la abstracción de datos intenta proporcionar una barrera opaca tras de la cual se ocultan los métodos y el estado; la herencia requiere abrir este interfaz en cierto grado y puede permitir el acceso a los métodos y al estado sin abstracción» [65]. Para una clase dada, habitualmente existen dos tipos de cliente: objetos que invocan operaciones sobre instancias de la clase, y subclases que heredan de esta clase. Liskov hace notar, por tanto, que con la herencia puede violarse el encapsulamiento de tres formas: «La subclase podría acceder a una variable de instancia de su superclase, llamar a una operación privada de su superclase, o referenciar directamente a superclases de su superclase» [66]. Los distintos lenguajes de programación hacen concesiones entre el apoyo del encapsulamiento y de la herencia de diferentes formas, pero entre los lenguajes descritos en este libro, el C++ ofrece quizás la mayor flexibilidad. Específicamente, el interfaz de una clase puede tener tres partes: partes *private*, que declaran miembros accesibles sólo a la propia clase, partes *protected*, que declaran miembros accesibles sólo a la clase y sus subclases, y partes *public*, accesibles a todos los clientes.

Ejemplos de jerarquía: herencia múltiple. El ejemplo anterior ilustraba el uso de herencia simple: la subclase `PlanCultivoFrutas` tenía exactamente una superclase, la clase `PlanCrecimiento`. Para ciertas abstracciones, resulta útil la herencia de múltiples superclases. Por ejemplo, supóngase que se decide definir una clase que representa un tipo de planta. En C++, podría declararse esta clase como sigue:

```
class Planta {
public:
    Planta (char* nombre, char* especie);
```

```

virtual ~Planta();

void fijarFechaSiembra(Dia);
virtual establecerCondicionesCultivo(const Condicion&);

const char* nombre() const;
const char* especie() const;
Dia fechaSiembra() const;

protected:
    char* repNombre;
    char* repEspecie;
    Dia repSiembra;

private:
    ...
};

```

Según esta definición de clase, cada instancia de la clase `Planta` tiene un nombre, especie y fecha de siembra. Además, pueden establecerse las condiciones óptimas de cultivo para cada tipo particular de planta. Ya que se espera que este comportamiento se especialice en las subclases, se declara esta operación como `virtual` en C++¹¹. Nótese que se declara como `protected` a los tres objetos miembro; así, son accesibles sólo para la propia clase y sus subclases. Por el contrario, todos los miembros declarados en la parte `private` son accesibles sólo para la propia clase.

Este análisis del dominio del problema podría sugerir que las plantas con flor, las frutas y las hortalizas tienen propiedades especializadas que son relevantes para la aplicación. Por ejemplo, dada una planta con flor, pueden resultar importantes el tiempo esperado restante para florecer y el momento en que hay que sembrarla. Análogamente, el momento de la recolección puede ser una parte importante de la abstracción que se hace de las frutas y hortalizas. Una posibilidad para capturar esas decisiones de diseño sería hacer dos clases nuevas, una clase `Flor` y una clase `FrutaHortaliza`, ambas subclases de la clase `Planta`. Sin embargo, ¿qué pasa si se necesita modelar una planta que florece y además produce fruto? Por ejemplo, los floristas utilizan con frecuencia capullos de manzano, cerezo y ciruelo. Para esta abstracción, se necesitaría inventar una tercera clase, `FlorFrutaHortaliza`, que duplicaría información de las clases `Flor` y `FrutaHortaliza`.

Una forma mejor de expresar estas abstracciones, y con ello evitar esta redundancia, es utilizar herencia múltiple. Primero, se idean clases que capturen independientemente las propiedades únicas de las plantas con flor y de las frutas y hortalizas:

```
class FlorAditiva {
```

¹¹ En CLOS se utilizan funciones genéricas; en Smalltalk, todas las operaciones de una superclase son susceptibles de ser especializadas por una subclase, y por tanto no se requiere ninguna designación especial.

```
public:

    FlorAditiva(Dia momentoFlorecimiento, Dia momentoSiembra);
    virtual ~FlorAditiva();

    Dia momentoFlorecimiento() const;
    Dia momentoSiembra() const;

protected:
    ...
};

class FrutaHortalizaAditiva {
public:

    FrutaHortalizaAditiva(Dia momentoRecoleccion);
    virtual ~FrutaHortalizaAditiva();

    Dia momentoRecoleccion() const;

protected:
    ...
};
```

Nótese que estas dos clases no tienen superclases; son independientes. Se les llama clases aditivas (*mixin*) porque son para mezclarlas con otras clases con el fin de producir nuevas subclases. Por ejemplo, se puede definir una clase Rosa como sigue:

```
class Rosa : public Planta, public FlorAditiva...
```

Del mismo modo, puede declararse una clase zanahoria como sigue:

```
class Zanahoria : public Planta, public FrutaHortalizaAditiva {};
```

En ambos casos, se forma la subclase mediante herencia de dos superclases. Las instancias de la subclase Rosa incluyen por tanto la estructura y comportamiento de la clase Planta junto con la estructura y comportamiento de la clase FlorAditiva. Ahora, supóngase que se desea declarar una clase para una planta como el cerezo que tiene tanto flores como frutos. Se podría escribir lo siguiente:

```
class Cerezo : public Plant,
                public FlorAditiva,
                public FrutaHortalizaAditiva...
```

La herencia múltiple es conceptualmente correcta, pero en la práctica intro-

duce ciertas complejidades en los lenguajes de programación. Los lenguajes tienen que hacer frente a dos problemas: colisiones entre nombres de superclases diferentes, y herencia repetida. Las colisiones se dan cuando dos o más superclases suministran un campo u operación con el mismo nombre o prototipo. En C++, tales colisiones deben resolverse con calificación explícita; en Smalltalk, se utiliza la primera ocurrencia del nombre. La herencia repetida ocurre cuando dos o más superclases «hermanas» comparten una superclase común. En tal situación, la trama de herencias tendrá forma de rombo, y aquí surge la cuestión: ¿debe la clase que hereda de ambas tener una sola copia o muchas copias de la estructura de la superclase compartida? Algunos lenguajes prohíben la herencia repetida, otros eligen una opción de manera unilateral, y otros, como C++, permiten al programador que decida. En C++ se utilizan clases base virtuales para denotar la compartición de estructuras repetidas, mientras que las clases base no virtuales resultan en la aparición de copias duplicadas en la subclase (requiriéndose calificación explícita para distinguir entre las copias).

La herencia múltiple se sobreutiliza a menudo. Por ejemplo, el caramelo de algodón es un tipo de caramelo, pero evidentemente no es un tipo de algodón. Una vez más, se aplica la piedra de toque de la herencia: si `B` no es un tipo de `A`, entonces `B` no debería heredar de `A`. Frecuentemente, pueden reducirse tramas de herencia múltiple mal formadas a una sola superclase más la agregación de las otras clases por parte de la subclase.

Ejemplos de jerarquía: agregación. Mientras estas jerarquías «es un» denotan relaciones de generalización/especialización, las jerarquías «parte de» describen relaciones de agregación. Por ejemplo, considérese la siguiente clase:

```
class Huerta {
public:
    Huerta();
    virtual ~Huerta();

    ...

protected:
    Planta* repPlantas[100];
    PlanCultivo repPlan;
};
```

Se tiene la abstracción de un jardín, que consiste en una colección de plantas junto con un plan de cultivo.

Cuando se trata con jerarquías como éstas, se habla a menudo de *niveles de abstracción*, un concepto descrito por primera vez por Dijkstra [67]. En términos de su jerarquía «de clases», una abstracción de alto nivel está generalizada, y una abstracción de bajo nivel está especializada. Por tanto se dice que una clase `Flor` está a nivel más alto de abstracción que una clase `Planta`. En tér-

minos de su jerarquía «parte de», una clase está a nivel más alto de abstracción que cualquiera de las clases que constituyen su implantación. Así, la clase `Huerta` está a nivel más alto de abstracción que el tipo `Planta`, sobre el que se construye.

La agregación no es un concepto exclusivo de los lenguajes de programación orientados a objetos. En realidad, cualquier lenguaje que soporte estructuras similares a los registros soporta la agregación. Sin embargo, la combinación de herencia con agregación es potente: la agregación permite el agrupamiento físico de estructuras relacionadas lógicamente, y la herencia permite que estos grupos de aparición frecuente se reutilicen con facilidad en diferentes abstracciones.

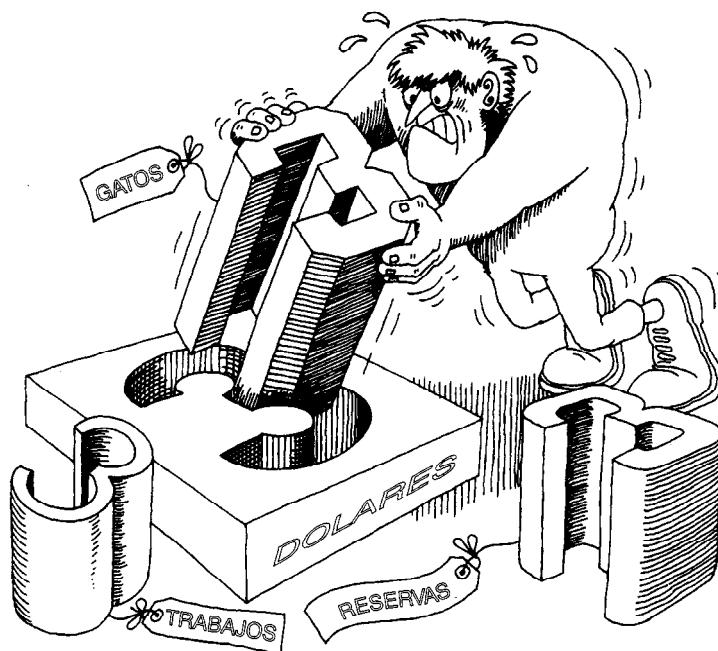
La agregación plantea el problema de la propiedad. La abstracción hecha de una huerta permite incorporar diferentes plantas a lo largo del tiempo, pero el reemplazar una planta no cambia la identidad de la huerta en su conjunto, ni el eliminar una huerta destruye necesariamente todas sus plantas (probablemente sean simplemente trasplantadas). En otras palabras, la existencia de una huerta y sus plantas son independientes: se captura esta decisión de diseño en el ejemplo de arriba, incluyendo punteros a objetos `Planta` en vez de valores. En contraste, se ha decidido que un objeto `PlanCultivo` está asociado intrínsecamente a un objeto `Huerta`, y no existe independientemente de la huerta. Por esta razón, se utiliza un valor `PlanCultivo`. Por tanto, cuando se crea una instancia de `Huerta`, se crea también una instancia de `PlanCultivo`; cuando se destruye el objeto `Huerta`, se destruye a su vez la instancia `PlanCultivo`. Se discutirá con más detalle la semántica de la propiedad por valor versus la de la propiedad por referencia en el siguiente capítulo.

Tipos (tipificación)

El significado de los tipos. El concepto de *tipo* se deriva en primer lugar de las teorías sobre los tipos abstractos de datos. Como sugiere Deutsch, «un tipo es una caracterización precisa de propiedades estructurales o de comportamiento que comparten una serie de entidades» [68]. Para nuestros propósitos, se utilizarán los términos *tipo* y *clase* de manera intercambiable¹². Aunque los conceptos de clase y tipo son similares, se incluyen los tipos como elemento separado del modelo de objetos porque el concepto de tipo pone énfasis en el significado de la abstracción en un sentido muy distinto. En concreto, se establece lo siguiente:

Los tipos son la puesta en vigor de la clase de los objetos, de modo que

¹² Un tipo y una clase no son exactamente lo mismo; algunos lenguajes en realidad distinguen estos dos conceptos. Por ejemplo, las primeras versiones del lenguaje Trellis/Owl permitían a un objeto tener simultáneamente una clase y un tipo. Incluso en Smalltalk, los objetos de las clases `SmallInteger`, `LargeNegativeInteger` y `LargePositiveInteger` son todas del mismo tipo, `Integer`, aun cuando no son de la misma clase [69]. Para la mayoría de los mortales, sin embargo, separar los conceptos de tipo y clase induce irremisiblemente a confusiones y aporta muy poco. Es suficiente decir que las clases implementan a los tipos.



La comprobación estricta de tipos impide que se mezclen abstracciones.

los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas.

Los tipos permiten expresar las abstracciones de manera que el lenguaje de programación en el que se implantan puede utilizarse para apoyar las decisiones de diseño. Wegner observa que este tipo de apoyo o refuerzo es esencial para la programación a gran escala [70].

La idea de congruencia es central a la noción de tipos. Por ejemplo, considérense las unidades de medida en física [71]. Cuando se divide distancia por tiempo, se espera algún valor que denote velocidad, no peso. Análogamente, multiplicar temperatura por una unidad de fuerza no tiene sentido, pero multiplicar masa por fuerza sí lo tiene. Ambos son ejemplos de comprobación estricta de tipos, en los que las reglas del dominio dictan y refuerzan ciertas combinaciones correctas de las abstracciones.

Ejemplos de tipos: comprobación de tipos estricta y débil. Un lenguaje de programación determinado puede tener comprobación estricta de tipos, comprobación débil de tipos, o incluso no tener tipos, y aún así ser considerado como orientado a objetos. Por ejemplo, Eiffel tiene comprobación estricta de tipos, lo que quiere decir que la concordancia se impone de manera estricta: no puede llamarse a una operación sobre un objeto a menos que en la clase o superclases

del objeto esté definido el prototipo exacto de esa operación. En lenguajes con comprobación estricta de tipos, puede detectarse en tiempo de compilación cualquier violación a la concordancia de tipos. Smalltalk, por el contrario, es un lenguaje sin tipos: un cliente puede enviar cualquier mensaje a cualquier clase (aunque ésta desconozca cómo responder al mensaje). No puede conocerse si hay incongruencias de tipos hasta la ejecución, y entonces suelen manifestarse como errores de ejecución. Los lenguajes como C++ son híbridos: tienen tendencias hacia la comprobación estricta, pero es posible ignorar o suprimir las reglas sobre tipos.

Considérese la abstracción de los distintos tipos de tanques de almacenamiento que pueden existir en un invernadero. Es probable que haya tanques para agua y para varios nutrientes; aunque uno almacena un líquido y el otro un sólido, estas abstracciones son suficientemente similares para justificar una jerarquía de clases, como ilustra el siguiente ejemplo. Primero, se introduce otro `typedef`:

```
// Número que denota el nivel de 0 a 100 por cien
typedef float Nivel;
```

En C++, los `typedef` no introducen nuevos tipos. En particular, los tipos definidos `Nivel` y `Concentracion` son ambos números en punto flotante, y pueden entremezclarse. En este aspecto, C++ tiene comprobación débil de tipos: los valores de tipos primitivos como `int` y `float` son indistinguibles dentro de ese tipo particular. En contraste, lenguajes como Ada y Object Pascal apoyan la comprobación estricta de tipos entre tipos primitivos. En Ada, por ejemplo, las construcciones del tipo derivado y el subtipo permiten al desarrollador definir tipos distintos, restringidos por rango o precisión respecto a tipos más generales.

A continuación, se muestra la jerarquía de clases para los tanques de almacenamiento:

```
class TanqueAlmacen {
public:

    TanqueAlmacen();
    virtual ~TanqueAlmacen();

    virtual void llenar();
    virtual void empezarDesaguar();
    virtual void pararDesaguar();

    Booleano estaVacio() const;
    Nivel nivel() const;

protected:
    ...
};
```

```
class TanqueAgua : public TanqueAlmacen {  
public:  
  
    TanqueAgua();  
    virtual ~TanqueAgua();  
  
    virtual void llenar();  
    virtual void empezarDesaguar();  
    virtual void pararDesaguar();  
    void empezarCalentar();  
    void pararCalentar();  
  
    Temperatura temperaturaActual() const;  
  
protected:  
    ...  
};  
  
class TanqueNutrientes : public TanqueAlmacen {  
public:  
    TanqueNutrientes();  
    virtual ~TanqueNutrientes();  
  
    virtual void empezarDesaguar();  
    virtual void pararDesaguar();  
  
protected:  
    ...  
};
```

La clase `TanqueAlmacen` es la clase base de esta jerarquía, y proporciona la estructura y comportamiento comunes a todos los tanques semejantes, como la capacidad de llenar y desaguar el tanque. `TanqueAgua` y `TanqueNutrientes` son ambas subclases de `TanqueAlmacen`. Ambas subclases redefinen parte del comportamiento de la superclase, y la clase `TanqueAgua` introduce algún comportamiento nuevo asociado con la temperatura.

Supóngase que se tienen las siguientes declaraciones:

```
TanqueAlmacen t1,t2;  
TanqueAgua a;  
TanqueNutrientes n;
```

Las variables como `t1`, `t2`, `a` y `n` no son objetos. Para ser precisos, son simplemente nombres que se utilizan para designar objetos de sus respectivas clases: cuando se dice «el objeto `t1`», realmente quiere hacerse referencia a la instancia de `TanqueAlmacen` denotada por la variable `t1`. Se explicará de nuevo este detalle sutil en el próximo capítulo.

Por lo que se refiere a la comprobación de tipos entre clases, C++ es más estricto, lo que significa que la comprobación de los tipos de las expresiones que invocan operaciones se realiza en tiempo de compilación. Por ejemplo, son correctas las siguientes sentencias:

```
Nivel niv = t1.nivel();
a.empezarDesaguar();
n.pararDesaguar();
```

En la primera sentencia, se invoca al selector `nivel`, declarado en la clase base `TanqueAlmacen`. En las dos sentencias siguientes, se invoca a un modificador (`empezarDesaguar` y `pararDesaguar`) declarado en la clase base, pero redefinido en las subclases.

Sin embargo, las siguientes sentencias no son correctas y serían rechazadas en tiempo de compilación:

```
t1.empezarCalentar(); // Incorrecto
n.pararCalentar();    // Incorrecto
```

Ninguna de las dos sentencias es correcta porque los métodos `empezarCalentar` y `pararCalentar` no están definidos para la clase de la variable correspondiente, ni para ninguna superclase de su clase. Por contra, la siguiente sentencia es correcta:

```
n.llenar();
```

Aunque `llenar` no está definido en la clase `TanqueNutrientes`, está definido en la superclase `TanqueAlmacen`, de la que la clase `TanqueNutrientes` hereda su estructura y comportamiento.

La comprobación estricta de tipos permite utilizar el lenguaje de programación para imponer ciertas decisiones de diseño, y por eso es particularmente relevante a medida que aumenta la complejidad del sistema. Sin embargo, la comprobación estricta de tipos tiene un lado oscuro. En la práctica, introduce dependencias semánticas tales que incluso cambios pequeños en el interfaz de una clase base requieren la recompilación de todas las subclases. Además, en ausencia de clases parametrizadas (que se tratarán más adelante, en el capítulo próximo y en el Capítulo 9) es problemático tener colecciones de objetos heterogéneos seguras respecto al tipo. Por ejemplo, supóngase que se necesita la abstracción de un inventario de invernadero, que recoge todos los bienes materiales asociados con un invernadero particular. Un hábito del C aplicado al C++ es utilizar una clase contenedor que almacena punteros a `void`, que representan objetos de un tipo indefinido:

```
class Inventario {
```

```

public:

Inventario();
~Inventario();

void anadir(void *);
void eliminar(void *);

void* masReciente() const;

void aplicar(Booleano (*) (void *));

private:
...
};

```

La operación `aplicar` es un iterador, que permite aplicar una operación a todos los elementos de la colección. Se tratarán los iteradores con más detalle en el próximo capítulo.

Dada una instancia de la clase `Inventario`, se puede añadir y eliminar punteros a objetos de cualquier clase. Sin embargo, esta aproximación no es segura respecto a tipos: se puede añadir legalmente a un inventario bienes materiales como tanques de almacenamiento, pero también elementos no materiales, como temperaturas o planes de cultivo, que violan la abstracción de un inventario. Análogamente, se puede añadir un objeto `TanqueAgua` igual que un objeto `SensorTemperatura`, y a menos que se ponga atención, invocar al selector `masReciente`, esperando encontrar un tanque de agua cuando lo que se devuelve es un sensor de temperatura.

Hay dos soluciones generales a estos problemas. Primero, se podría usar una clase contenedor segura respecto al tipo. En lugar de manipular punteros a `void`, se puede definir una clase `inventario` que manipula sólo objetos de la clase `BienMaterial`, que se usaría como clase aditiva para todas las clases que representan bienes materiales, como `TanqueAgua` pero no como `PlanCultivo`. Este enfoque supera el primer problema, en el que objetos de diferentes tipos se mezclaban de forma incorrecta. Segundo, se usaría alguna forma de identificación de tipos en tiempo de ejecución; esto soluciona el segundo problema, conocer qué tipo de objeto se está examinando en determinado momento. En Smalltalk, por ejemplo, se puede preguntar a un objeto por su clase. En C++, la identificación del tipo en tiempo de ejecución aún no forma parte del estándar del lenguaje¹³, pero puede lograrse un efecto similar en la práctica, definiendo una operación en la clase base que retorna una cadena o tipo enumerado que identifica la clase concreta del objeto. En general, sin embargo, la identificación de tipos en ejecución debería utilizarse sólo cuando hay una razón de peso, porque puede representar un debilitamiento del encapsulamiento. Como se verá en la

¹³ Se está considerando la adopción en C++ de la identificación de tipos en tiempo de ejecución.

siguiente sección, el uso de operaciones polimórficas puede mitigar a menudo (pero no siempre) la necesidad de identificación de tipos en ejecución.

Un lenguaje con tipos estrictos es aquel en el que se garantiza que todas las expresiones son congruentes respecto al tipo. El significado de la consistencia de tipos se aclara en el ejemplo siguiente, usando las variables declaradas previamente. Las siguientes sentencias de asignación son correctas:

```
t1 = t2;
t1 = a;
```

La primera sentencia es correcta porque la clase de la variable del miembro izquierdo de la asignación (`TanqueAlmacen`) es la misma que la de la expresión del miembro derecho. La segunda sentencia es también correcta porque la clase de la variable del miembro izquierdo (`TanqueAlmacen`) es una superclase de la variable del miembro derecho (`TanqueAgua`). Sin embargo, esta asignación desemboca en una pérdida de información (conocida en C++ como slicing, o «rebanaada»). La subclase `TanqueAgua` introduce estructuras y comportamientos más allá de los definidos en la clase base, y esta información no puede copiarse a una instancia de la clase base.

Considérense las siguientes sentencias incorrectas:

```
a = t1; // incorrectas
a = n; // incorrectas
```

La primera sentencia no es correcta porque la clase de la variable del lado izquierdo (`TanqueAgua`) es una subclase de la clase de la variable del lado derecho (`TanqueAlmacen`). La segunda sentencia es incorrecta porque las clases de las dos variables son hermanas, y no están en la misma línea de herencia (aunque tengan una superclase común).

En algunas situaciones, se necesita convertir un valor de un tipo a otro. Por ejemplo, considérese la función siguiente:

```
void comprobarNivel(const TanqueAlmacen& t);
```

Sólo en el caso de que exista la seguridad de que el argumento actual que se proporciona es de la clase `TanqueAgua` se puede efectuar una conversión forzada del valor de la clase base al de la subclase, como en la expresión siguiente:

```
if ( (TanqueAgua&) s).temperaturaActual() < 32.0) ...
```

Esta expresión es consistente respecto a tipos, aunque no es totalmente segura respecto a tipos. Por ejemplo, si la variable `t` llegase a denotar un objeto de la clase `TanqueNutrientes` en tiempo de ejecución, el ahormado fallaría durante la ejecución con resultados impredecibles. En general, la conversión de ti-

pos debería evitarse, porque frecuentemente representa una violación de la abstracción.

Como apunta Tesler, existen varios beneficios importantes que se derivan del uso de lenguajes con tipos estrictos:

- «Sin la comprobación de tipos, un programa puede ‘estallar’ de forma misteriosa en ejecución en la mayoría de los lenguajes.
- En la mayoría de los sistemas, el ciclo editar-compilar-depurar es tan tedioso que la detección temprana de errores es indispensable.
- La declaración de tipos ayuda a documentar los programas.
- La mayoría de los compiladores pueden generar un código más eficiente si se han declarado los tipos» [72].

Los lenguajes sin tipos ofrecen mayor flexibilidad, pero incluso con lenguajes de esta clase, como observan Borning e Ingalls, «en casi todos los casos, el programador conoce de hecho qué tipo de objeto se espera en los argumentos de un mensaje, y qué tipo de objeto será devuelto» [73]. En la práctica, la seguridad que ofrecen los lenguajes con tipos estrictos suele compensar con creces la flexibilidad que se pierde al no usar un lenguaje sin tipos, especialmente si se habla de programación a gran escala.

Ejemplos de tipos: ligadura estática y dinámica. Los conceptos de tipos estrictos y tipos estáticos son completamente diferentes. La noción de tipos estrictos se refiere a la consistencia de tipos, mientras que la asignación estática de tipos —también conocida como *ligadura estática* o *ligadura temprana*— se refiere al momento en el que los nombres se ligan con sus tipos. La ligadura estática significa que se fijan los tipos de todas las variables y expresiones en tiempo de compilación; la *ligadura dinámica* (también llamada *ligadura tardía*) significa que los tipos de las variables y expresiones no se conocen hasta el tiempo de ejecución. Al ser la comprobación estricta de tipos y la ligadura conceptos independientes, un lenguaje puede tener comprobación estricta de tipos y tipos estáticos (Ada), puede tener comprobación estricta de tipos pero soportar enlace dinámico (Object Pascal y C++), o no tener tipos y admitir la ligadura dinámica (Smalltalk). CLOS se encuentra a medio camino entre C++ y Smalltalk, en tanto en cuanto una implantación puede imponer o ignorar las declaraciones de tipo que pueda haber realizado un programador.

Se ilustrarán de nuevo estos conceptos en C++. Considérese la siguiente función no miembro¹⁴:

```
void equilibrarNiveles(TanqueAlmacen& t1, TanqueAlmacen& t2);
```

Llamar a la operación `equilibrarNiveles` con instancias de `Tanque`.

¹⁴ Una función no miembro es una función que no está asociada directamente con una clase. Las funciones no miembro se llaman también *subprogramas libres*. En un lenguaje orientado a objetos puro como Smalltalk, no existen subprogramas libres; cualquier operación debe estar asociada con alguna clase.

Almacen o cualquiera de sus subclases es consistente respecto al tipo, porque el tipo de los parámetros actuales es parte de la misma línea de herencia, cuya clase base es TanqueAlmacen.

En la implantación de esta función, se podría hallar la expresión:

```
if (t1.nivel() > t2.nivel())
    t2.llenar();
```

¿Cuál es el significado de la invocación al selector `nivel`? Esta operación está declarada únicamente en la clase base `TanqueAlmacen` y, por tanto, no importa de qué clase o subclase específica sea la instancia que se suministra para el parámetro formal `t1`; va a invocarse la operación de la clase base. Aquí, la llamada a `nivel` se resuelve mediante ligadura estática: en tiempo de compilación, se sabe exactamente qué operación se invocará.

Por contra, considérese la semántica de invocar al modificador `llenar`, que se resuelve mediante ligadura dinámica. Esta operación se declara en la clase base y se redefine entonces sólo en la subclase `TanqueAgua`. Si el parámetro actual para `t1` es una instancia de `TanqueAgua`, se invocará `TanqueAgua::llenar`; si el parámetro actual para `t1` es una instancia de `TanqueNutrientes`, entonces se invocará `TanqueAlmacen::llenar`¹⁵.

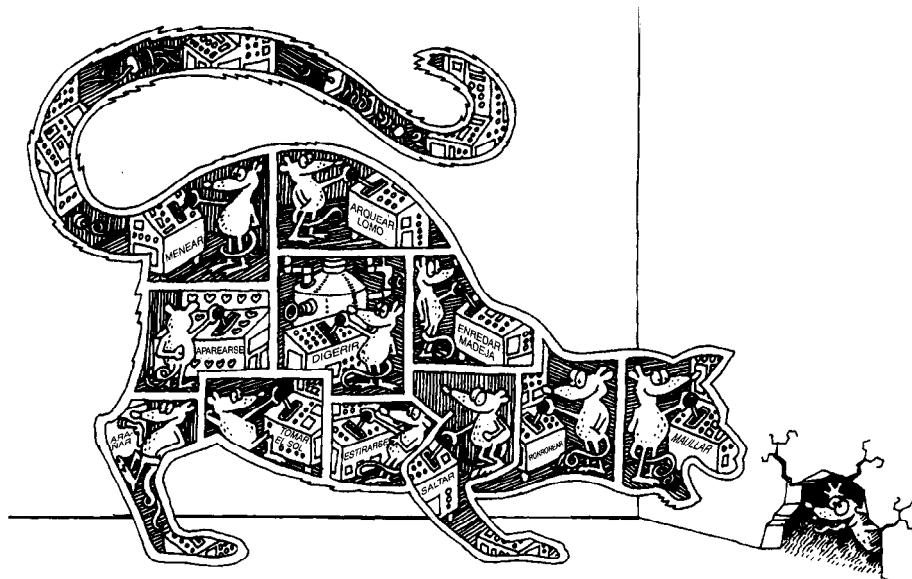
Esta característica se llama *polimorfismo*; representa un concepto de teoría de tipos en el que un solo nombre (tal como una declaración de variable) puede denotar objetos de muchas clases diferentes que se relacionan por alguna superclase común. Cualquier objeto denotado por este nombre es, por tanto, capaz de responder a algún conjunto común de operaciones [74]. El opuesto del polimorfismo es el *monomorfismo*, que se encuentra en todos los lenguajes con comprobación estricta de tipos y ligadura estática, como Ada.

Existe el polimorfismo cuando interactúan las características de la herencia y el enlace dinámico. Es quizás la característica más potente de los lenguajes orientados a objetos después de su capacidad para soportar la abstracción, y es lo que distingue la programación orientada a objetos de otra programación más tradicional con tipos abstractos de datos. Como se verá en los capítulos siguientes, el polimorfismo es también un concepto central en el diseño orientado a objetos.

Concurrencia

El significado de la concurrencia. Para ciertos tipos de problema, un sistema automatizado puede tener que manejar muchos eventos diferentes simultáneamente. Otros problemas pueden implicar tantos cálculos que excedan la capacidad de cualquier procesador individual. En ambos casos, es natural considerar el uso de un conjunto distribuido de computadores para la implantación

¹⁵ `TanqueAlmacen::llenar` es la sintaxis que utiliza C++ para calificar explícitamente el nombre de una declaración.



La concurrencia permite a diferentes objetos actuar al mismo tiempo.

que se persigue o utilizar procesadores capaces de realizar multitarea. Un solo proceso —denominado *hilo de control*— es la raíz a partir de la cual se producen acciones dinámicas independientes dentro del sistema. Todo programa tiene al menos un hilo de control, pero un sistema que implique concurrencia puede tener muchos de tales hilos: algunos son transitorios, y otros permanecen durante todo el ciclo de vida de la ejecución del sistema. Los sistemas que se ejecutan en múltiples CPUs permiten hilos de control verdaderamente concurrentes, mientras que los sistemas que se ejecutan en una sola CPU sólo pueden conseguir la ilusión de hilos concurrentes de control, normalmente mediante algún algoritmo de tiempo compartido.

Se distingue también entre concurrencia pesada y ligera. Un *proceso pesado* es aquel típicamente manejado de forma independiente por el sistema operativo de destino, y abarca su propio espacio de direcciones. Un *proceso ligero* suele existir dentro de un solo proceso del sistema operativo en compañía de otros procesos ligeros, que comparten el mismo espacio de direcciones. La comunicación entre procesos pesados suele ser costosa, involucrando a alguna forma de comunicación interproceso; la comunicación entre procesos ligeros es menos costosa, y suele involucrar datos compartidos.

Muchos sistemas operativos actuales proporcionan soporte directo para la concurrencia, y por tanto existe una gran oportunidad (y demanda) para la concurrencia en sistemas orientados a objetos. Por ejemplo, UNIX proporciona la llamada del sistema *fork*, que lanza un nuevo proceso. Análogamente, Win-

dows/NT y OS/2 tienen multitarea, y proporcionan a los programas interfaces para crear y manipular procesos.

Lim y Johnson apuntan que «el diseño de características para la concurrencia en lenguajes de POO no es muy diferente de hacerlo en otros tipos de lenguajes —la concurrencia es ortogonal a la POO en los niveles más bajos de abstracción. Se trate de POO o no, continúan existiendo todos los problemas tradicionales en programación concurrente» [75]. Realmente, construir un elemento grande de software es bastante difícil; diseñar uno que abarque múltiples hilos de control es mucho más difícil aún porque hay que preocuparse de problemas tales como interbloqueo, bloqueo activo, inanición, exclusión mutua y condiciones de competencia. Afortunadamente, como señalan también Li y Johnson, «A los niveles más altos de abstracción, la POO puede aliviar el problema de la concurrencia para la mayoría de los programadores mediante la ocultación de la misma dentro de abstracciones reutilizables» [76]. Black *et al.* sugieren, por tanto, que «un modelo de objetos es apropiado para un sistema distribuido porque define de forma implícita (1) las unidades de distribución y movimiento y (2) las entidades que se comunican» [77].

Mientras que la programación orientada a objetos se centra en la abstracción de datos, encapsulamiento y herencia, la concurrencia se centra en la abstracción de procesos y la sincronización [78]. El objeto es un concepto que une estos dos puntos de vista distintos: cada objeto (extraído de una abstracción del mundo real) puede representar un hilo separado de control (una abstracción de un proceso). Tales objetos se llaman *activos*. En un sistema basado en diseño orientado a objetos, se puede conceptualizar el mundo como un conjunto de objetos cooperativos, algunos de los cuales son activos y sirven así como centros de actividad independiente. Partiendo de esta concepción, se define la concurrencia como sigue:

La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo.

Ejemplos de concurrencia. Las discusiones anteriores sobre la abstracción introdujeron la clase `SensorTemperaturaActivo`, cuyo comportamiento requería medir periódicamente la temperatura e invocar entonces la función `callback`¹⁶ de un objeto cliente, siempre que la temperatura cambiase cierto número de grados respecto a un punto de referencia dado. No se explicó cómo implantaba la clase este comportamiento. Este hecho es un secreto de la implantación, pero está claro que se requiere alguna forma de concurrencia. En general, existen tres enfoques para la concurrencia en el diseño orientado a objetos.

Primero, la concurrencia es una característica intrínseca de ciertos lenguajes de programación. Por ejemplo, el mecanismo de Ada para expresar un proceso concurrente es la tarea (*task*). Análogamente, Smalltalk proporciona la clase `Process`, que puede usarse como la superclase de todos los objetos activos. Existen otros lenguajes concurrentes orientados a objetos, como `Actors`, `Orient`

¹⁶ Las funciones `callback` también se denominan inversas o de retorno. (*N. del T.*)

84/K y ABCL/1, que proporcionan mecanismos similares para la concurrencia y la sincronización. En todos los casos, se puede crear un objeto activo que ejecuta algún proceso concurrentemente con todos los demás objetos activos.

Segundo, se puede usar una biblioteca de clases que soporte alguna forma de procesos ligeros. Es el enfoque adoptado por la biblioteca de tareas de AT&T para C++, que proporciona las clases Sched, Timer, Task y otras. Naturalmente, la implementación de esta biblioteca es altamente dependiente de la plataforma, aunque el interfaz de la misma es relativamente transportable. En este enfoque, la concurrencia no es parte intrínseca del lenguaje (y de esta forma no constituye ninguna carga para los sistemas no concurrentes), pero aparece como si fuese intrínseca, a través de la presencia de esas clases estándar.

Tercero, pueden utilizarse interrupciones para dar la ilusión de concurrencia. Por supuesto, esto exige tener un conocimiento de ciertos detalles de bajo nivel del hardware. Por ejemplo, en la implementación realizada de la clase SensorTemperaturaActivo, se podría tener un temporizador hardware que interrumpiese periódicamente a la aplicación, y durante ese tiempo todos los sensores de ese tipo medirían la temperatura, invocando entonces a la función callback si fuese necesario.

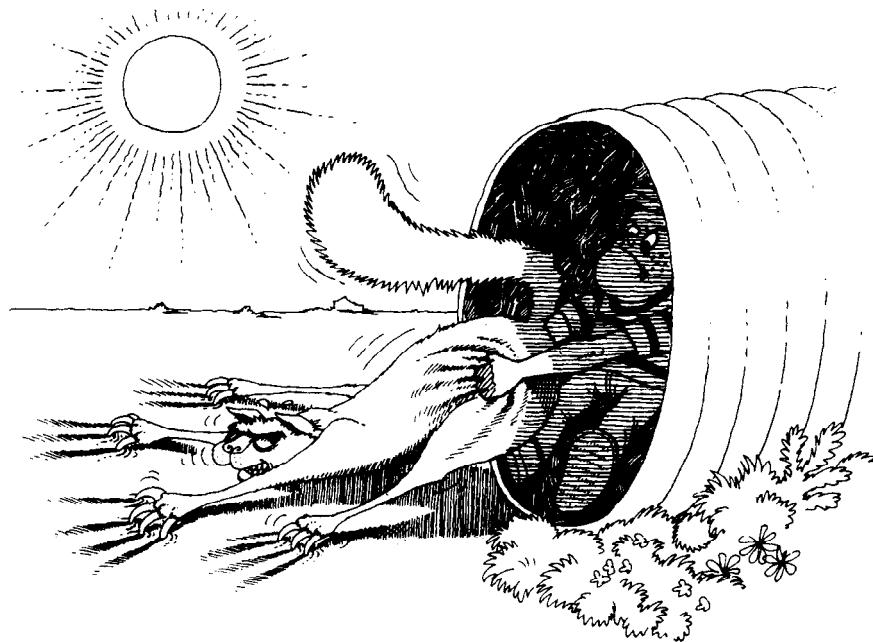
Da igual qué aproximación a la concurrencia se adopte; una de las realidades acerca de la concurrencia es que, una vez que se la introduce en un sistema, hay que considerar cómo los objetos activos sincronizan sus actividades con otros, así como con objetos puramente secuenciales. Por ejemplo, si dos objetos activos intentan enviar mensajes a un tercer objeto, hay que tener la seguridad de que existe algún mecanismo de exclusión mutua, de forma que el estado del objeto sobre el que se actúa no está corrupto cuando los dos objetos activos intentan actualizarlo simultáneamente. Éste es el punto en el que las ideas de abstracción, encapsulamiento y concurrencia interactúan. En presencia de la concurrencia, no hay suficiente con definir simplemente los métodos de un objeto; hay que asegurarse también de que la semántica de estos métodos se mantiene a pesar de la existencia de múltiples hilos de control.

Persistencia

Un objeto de software ocupa una cierta cantidad de espacio, y existe durante una cierta cantidad de tiempo. Atkinson et al. sugieren que hay un espacio continuo de existencia del objeto, que va desde los objetos transitorios que surgen en la evaluación de una expresión hasta los objetos de una base de datos que sobreviven a la ejecución de un único programa. Este espectro de persistencia de objetos abarca lo siguiente:

- «Resultados transitorios en la evaluación de expresiones.
- Variables locales en la activación de procedimientos.
- Variables propias [como en ALGOL 60], variables globales y elementos del montículo (*heap*)* cuya duración difiere de su ámbito.

* También se denomina montón. (N. del T.)



La persistencia conserva el estado de un objeto en el tiempo y en el espacio.

- Datos que existen entre ejecuciones de un programa.
- Datos que existen entre varias versiones de un programa.
- Datos que sobreviven al programa» [79].

Los lenguajes de programación tradicionales suelen tratar solamente los tres primeros tipos de persistencia de objetos; la persistencia de los tres últimos tipos pertenece típicamente al dominio de la tecnología de bases de datos. Esto conduce a un choque cultural que a veces tiene como resultado arquitecturas muy extrañas: los programadores acaban por crear esquemas *ad hoc* para almacenar objetos cuyo estado debe ser preservado entre ejecuciones del programa, y los diseñadores de bases de datos aplican incorrectamente su tecnología para enfrentarse a objetos transitorios [80].

La unificación de los conceptos de concurrencia y objetos da lugar a los lenguajes concurrentes de programación orientada a objetos. De manera similar, la introducción del concepto de persistencia en el modelo de objetos da lugar a la aparición de bases de datos orientadas a objetos. En la práctica, tales bases de datos se construyen sobre tecnología contrastada, como modelos de bases de datos secuenciales, indexadas, jerárquicas, en red o relacionales, pero ofrecen al programador la abstracción de un interfaz orientado a objetos, a través del cual las consultas y otras operaciones se llevan a cabo en términos de objetos cuyo ciclo de vida trasciende el ciclo de vida de un programa individual. Esta unificación simplifica enormemente el desarrollo de ciertos tipos de aplicación. En

particular, permite aplicar los mismos métodos de diseño a todos los segmentos de una aplicación, tanto a los propios de una base de datos como a los que no lo son, como se verá en el Capítulo 10.

Muy pocos lenguajes de programación orientados a objetos ofrecen soporte directo para la persistencia; Smalltalk es una notable excepción. En él existen protocolos para escribir y leer objetos en disco (los cuales deben ser redefinidos por subclases). Sin embargo, el almacenar objetos en simples ficheros es una solución ingenua para la persistencia, una solución que no resiste bien los cambios de escala. Más frecuentemente, la persistencia se consigue a través de un pequeño número de bases de datos orientadas a objetos disponibles comercialmente [81]. Otro enfoque razonable para la persistencia es proporcionar una piel orientada a objetos bajo la cual se oculta una base de datos relacional. Esta aproximación es más atractiva si existe una gran inversión de capital en tecnología de bases de datos relacionales que sería arriesgado o demasiado caro reemplazar. Se examinará esta situación en el Capítulo 10.

La persistencia abarca algo más que la mera duración de los datos. En las bases de datos orientadas a objetos, no sólo persiste el *estado* de un objeto, sino que su *clase* debe trascender también a cualquier programa individual, de forma que todos los programas interpreten de la misma manera el estado almacenado. Esto hace que sea un reto evidente el mantener la integridad de una base de datos a medida que crece, particularmente si hay que cambiar la clase de un objeto.

La discusión hasta aquí afecta a la persistencia en el tiempo. En la mayoría de los sistemas, un objeto, una vez creado, consume la misma memoria física hasta que deja de existir. Sin embargo, para sistemas que se ejecutan en un conjunto distribuido de procesadores, a veces hay que preocuparse de la persistencia en el espacio. En estos sistemas, es útil pensar en los objetos que puedan llevarse de una máquina a otra, y que incluso pueden tener representaciones diferentes en máquinas diferentes. Se examina este tipo de persistencia más adelante en el Capítulo 12.

Para resumir, se define la persistencia como sigue:

La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir, el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).

2.3. Aplicación del modelo de objetos

Beneficios del modelo de objetos

Como se ha mostrado, el modelo de objetos es fundamentalmente diferente de los modelos adoptados por los métodos más tradicionales de análisis estructu-

rado, diseño estructurado y programación estructurada. Esto no significa que el modelo de objetos abandone todos los sanos principios y experiencias de métodos más viejos. En lugar de eso, introduce varios elementos novedosos que se basan en estos modelos anteriores. Así, el modelo de objetos proporciona una serie de beneficios significativos que otros modelos simplemente no ofrecen. Aún más importante, el uso del modelo de objetos conduce a la construcción de sistemas que incluyen los cinco atributos de los sistemas complejos bien estructurados. Según nuestra experiencia, existen otros cinco beneficios prácticos que se derivan de la aplicación del modelo de objetos.

En primer lugar, el uso del modelo de objetos ayuda a explotar la potencia expresiva de los lenguajes de programación basados en objetos y orientados a objetos. Como apunta Stroustrup, «No siempre está claro cómo aprovechar mejor un lenguaje como C++. Se han logrado mejoras significativas de forma consistente en la productividad y la calidad del código utilizando C++ como un ‘C mejorado’ con una pizca de abstracción de datos añadida donde era claramente útil. Sin embargo, se han obtenido mejoras distintas y apreciablemente mayores aprovechando las jerarquías de clases en el proceso de diseño. Esto se llama muchas veces diseño orientado a objetos, y aquí es donde se han encontrado los mayores beneficios del uso de C++» [82]. Nuestra experiencia indica que, sin la aplicación de los elementos del modelo de objetos, las características más potentes de lenguajes como Smalltalk, Object Pascal, C++, CLOS y Ada, o bien se ignoran o bien se utilizan desastrosamente.

Por otra parte, el uso del modelo de objetos promueve la reutilización no sólo de software, sino de diseños enteros, conduciendo a la creación de marcos de desarrollo de aplicaciones reutilizables [83]. Se ha encontrado que los sistemas orientados a objetos son frecuentemente más pequeños que sus implementaciones equivalentes no orientadas a objetos. Esto no sólo significa escribir y mantener menos código, sino que la mayor reutilización del software también se traduce en beneficios de coste y planificación.

Tercero, el uso del modelo de objetos produce sistemas que se construyen sobre formas intermedias estables, que son más flexibles al cambio. Esto también significa que se puede admitir que tales sistemas evolucionen en el tiempo, en lugar de ser abandonados o completamente rediseñados en cuanto se produzca el primer cambio importante en los requerimientos.

El Capítulo 7 explica con mayor profundidad cómo el modelo de objetos reduce los riesgos inherentes al desarrollo de sistemas complejos, más que nada porque la integración se distribuye a lo largo del ciclo vital en vez de suceder como un evento principal. La guía que proporciona el modelo de objetos, al diseñar una separación inteligente de intereses también reduce los riesgos del desarrollo e incrementa la confianza en la corrección del diseño.

Finalmente, el modelo de objetos resulta atractivo para el funcionamiento de la cognición humana, porque, como sugiere Robson, «muchas personas que no tienen ni idea de cómo funciona un computador encuentran bastante natural la idea de los sistemas orientados a objetos» [84].

Aeronáutica	Fabricación integrada por computador
Análisis matemático	Hipermedia
Animación	Ingeniería petroquímica
Automatización de oficinas	Preparación de documentos
Bases de datos	Preparación de películas y escenarios
Componentes de software reutilizables	Proceso de datos de negocios
Composición de música	Reconocimiento de imágenes
Control de procesos químicos	Robótica
Control de tráfico aéreo	Simulación de cohetes y aviones
Diseño asistido por computador	Sistemas de dirección y control
Diseño de interfaces de usuario	Sistemas de telemetría
Diseño VLSI	Sistemas expertos
Electrónica médica	Sistemas operativos
Enseñanza asistida por ordenador	Software de banca y seguros
Entornos de desarrollo de software	Software de estaciones espaciales
Estrategias de inversión	Telecomunicaciones

Figura 2.6. Aplicaciones del modelo de objetos.

Aplicaciones del modelo de objetos

El modelo de objetos ha demostrado ser aplicable a una amplia variedad de dominios de problema. La Figura 2.6 enumera muchos de los dominios para los que existen sistemas que perfectamente pueden denominarse orientados a objetos. La Bibliografía proporciona una lista extensa de referencias a estas y otras aplicaciones.

Puede que el diseño orientado a objetos sea el único método entre los disponibles hoy en día, que puede emplearse para atacar la complejidad innata a muchos sistemas grandes. Siendo justos, sin embargo, el uso de diseño orientado a objetos puede no ser aconsejable en algunos dominios, no por razones técnicas, sino por razones no técnicas, como la falta de personal con entrenamiento adecuado o buenos entornos de desarrollo. Se tratarán estos problemas con más detalle en el Capítulo 7.

Problemas planteados

Para aplicar con efectividad los elementos del modelo de objetos, es necesario resolver varios problemas:

- ¿Qué son exactamente las clases y los objetos?
- ¿Cómo se identifica correctamente las clases y objetos relevantes de una aplicación concreta?
- ¿Cómo sería una notación adecuada para expresar el diseño de un sistema orientado a objetos?
- ¿Qué proceso puede conducir a un sistema orientado a objetos bien estructurado?
- ¿Cuáles son las implicaciones en cuanto a gestión que se derivan del uso de diseño orientado a objetos?

Estos problemas son el tema de los próximos cinco capítulos.

Resumen

- La madurez de la ingeniería del software ha conducido al desarrollo de métodos de análisis, diseño y programación orientados a objetos, todos los cuales tienen la misión de resolver los problemas de la programación a gran escala.
- Existen varios paradigmas de programación distintos: orientados a procedimientos, orientados a objetos, orientados a lógica, orientados a reglas y orientados a restricciones.
- Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objeto, y proporciona así fronteras conceptuales definidas con nitidez, desde la perspectiva del observador.
- El encapsulamiento es el proceso de compartimentar los elementos de una abstracción que constituyen su estructura y comportamiento. Sirve para separar el interfaz «contractual» de una abstracción y su implantación.
- La modularidad es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.
- La jerarquía es una graduación u ordenación de abstracciones.
- Los tipos son el resultado de imponer la clase de los objetos, de forma que los objetos de tipos diferentes no pueden intercambiarse o, como mucho, pueden hacerlo de formas muy restringidas.
- La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo.
- La persistencia es la propiedad de un objeto mediante la cual su existencia perdura en el tiempo y/o el espacio.

Lecturas recomendadas

El concepto del modelo de objetos fue introducido en primer lugar por Jones [F 1979] y Williams [F 1986]. La Tesis Doctoral de Kay [F 1969] estableció la dirección para gran parte del trabajo subsiguiente en programación orientada a objetos.

Shaw [J 1984] ofrece un excelente resumen acerca de los mecanismos de abstracción en lenguajes de programación de alto nivel. El fundamento teórico de la abstracción puede encontrarse en el trabajo de Liskov y Guttag [H 1986], Guttag [J 1980] y Hilsinger [J 1982]. Parnas [F 1979] es el trabajo seminal sobre la ocultación de información. El significado e importancia de la jerarquía se discute en el trabajo editado por Pattee [J 1973].

Existe gran cantidad de literatura sobre programación orientada a objetos. Cardelli y Wegner [J 1985] y Wegner [J 1987] ofrecen un excelente estudio de los lenguajes de

programación basados en objetos y orientados a objetos. Los artículos tutoriales de Stefik y Bobrow [G 1986], Stroustrup [G 1988], Nygaard [G 1986] y Grogono [G 1991] son buenos puntos de partida sobre los problemas importantes de la programación orientada a objetos. Los libros de Cox [G 1986], Meyer [F 1988], Schmucker [G 1986] y Kim y Lochovsky [F 1989] ofrecen un extenso tratamiento de estos temas.

Los métodos de diseño orientados a objetos fueron introducidos por Booch [F 1981, 1982, 1986, 1987, 1989]. Los métodos de análisis orientado a objetos fueron introducidos por Shlaer y Mellor [B 1988] y Bailin [B 1988]. Desde entonces, se han propuesto varios métodos de análisis y diseño orientados a objetos, destacando Rumbaugh [F 1991], Coad y Yourdon [B 1991], Constantine [F 1989], Shlaer y Mellor [B 1992], Martin y Odell [B 1992], Wasserman [B 1991], Jacobson [F 1992], Rubin y Goldberg [B 1992], Embley [B 1992], Wirfs-Brock [F 1990], Goldstein y Alger [C 1992], Henderson-Sellers [F 1992], Firesmith [F 1992] y Fusion [F 1992].

Pueden encontrarse casos de estudio de aplicaciones orientadas a objetos en Taylor [H 1990, C 1992], Berard [H 1993], Love [C 1993] y Pinson y Weiner [C 1990].

Puede encontrarse una excelente colección de artículos que tratan todos los aspectos de la tecnología orientada a objetos en Peterson [G 1987], Schriver y Wegner [G 1987] y Khoshafian y Abnous [G 1990]. Las actas de varias conferencias anuales sobre tecnología orientada a objetos son también excelentes fuentes de material. Algunos de los foros más importantes incluyen OOPSLA, ECOOP, TOOLS, Object World y Object Expo.

Entre las organizaciones responsables de establecer estándares para la tecnología orientada a objetos se incluyen el Object Management Group y el comité ANSI X3J7.

La referencia principal para C++ es Ellis y Stroustrup [G 1990]. Otras referencias útiles incluyen a Stroustrup [G 1991], Lippman [G 1991] y Coplien [G 1992].

Notas bibliográficas

- [1] Rentsch, T. September 1982. Object-Oriented Programming. *SIGPLAN Notices* vol. 17(12), p. 51.
- [2] Wegner, P. June 1981. *The Ada Programming Language and Environment*. Unpublished draft.
- [3] Abbott, R. August 1987. Knowledge Abstraction. *Communications of the ACM* vol. 30(8), p. 664.
- [4] Ibid., p. 664.
- [5] Shankar, K. 1984. Data design: Types, Structures, and Abstractions. *Handbook of Software Engineering*. New York, NY: Van Nostrand Reinhold, p. 253.
- [6] *Macintosh MacApp 1.1.1 Programmer's Reference*. 1986. Cupertino, CA: Apple Computer, p. 2.
- [7] Bhaskar, K. October 1983. How Object-Oriented Is Your System? *SIGPLAN Notices* vol. 18(10), p. 8.
- [8] Stefik, M. and Bobrow, D. Winter 1986. Object-Oriented Programming: Themes and Variations, *AI Magazine* vol. 6(4), p. 41.
- [9] Yonezawa, A. and Tokoro, M. 1987. Object-Oriented Concurrent Programming: An Introduction, in *Object-Oriented Concurrent Programming*. Cambridge, MA: The MIT Press, p. 2.

- [10] Levy, H. 1984. *Capability-Based Computer Systems*. Bedford, MA: Digital Press, p. 13.
- [11] Ramamoorthy, C. and Sheu, P. Fall 1988. Object-Oriented Systems. *IEEE Expert* vol. 3(3), p. 14.
- [12] Myers, G. 1982. *Advances in Computer Architecture*. Second Edition. New York, NY: John Wiley and Sons, p. 58.
- [13] Levy. *Capability-Based Computer*.
- [14] Kavi, K. and Chen, D. 1987. Architectural Support for Object-Oriented Languages. *Proceedings of the Thirty-second IEEE Computer Society International Conference*. IEEE.
- [15] *iAPX 432 Object Primer*. 1981. Santa Clara, CA: Intel Corporation.
- [16] Dally, W. J. and Kajiiya, J. T. March 1985. An Object-Oriented Architecture. *SIGARCH Newsletter* vol. 13(3).
- [17] Dahlby, S., Henry, G., Reynolds, D., and Taylor, P. 19892. The IBM System/38: A High Level Machine, in *Computer Structures: Principles and Examples*. New York, NY: McGraw-Hill.
- [18] Dijkstra, E. May 1968. The Structure of the «THE» Multiprogramming System. *Communications of the ACM* vol. 11(5).
- [19] Pashtan, A. 1982. Object-Oriented Operating Systems: An Emerging Design Methodology. *Proceedings of the ACM '82 Conference*, ACM.
- [20] Parnas, D. 1979. On the Criteria to Be Used in Decomposing Systems into Modules, in *Classics in Software Engineering*. New York, NY: Yourdon Press.
- [21] Liskov, B. and Zilles, S. 1977. An Introduction to Formal Specifications of Data Abstractions. *Current Trends in Programming Methodology: Software Specification and Design* vol. 1. Englewood Cliffs, NH: Prentice-Hall.
- [22] Guttag, J. 1980. Abstract Data Types and the Development of Data Structures, in *Programming Language Design*, New York, NY: Computer Society Press.
- [23] Shaw. Abstraction Techniques.
- [24] Nygaard, K. and Dahl, O-J. 1981. The Debvelopment of the Simula Languages, in *History of Programming Languages*. New York, NY: Academic Press, p. 460.
- [25] Atkinson, M. and Buneman, P. June 1987. Types and Persistence in Database Programming Languages. *ACM Computing Surveys* vol. 19(2), p. 105.
- [26] Rumbaugh, J. April 1988. Relational Database Design Using an Object-Oriented Methodology. *Communications of the ACM* vol. 31(4), p. 415.
- [27] Chen, P. March 1976. The Entity-Relationship Model-Toward a Unified View of Data. *ACM Transactions on Database Systems* vol. 1(1).
- [28] Barr, A. and Feigenbaum, E. 1981. *The Handbook of Artificial Intelligence*. Vol. 1. Los Altos, CA: William Kaufmann, p. 216.
- [29] Stillings, N., Feinstein, M., Garfield, J. Rissland, E., Rosenbaum, D., Weisler, S., Baker-Ward, L. 1987. *Cognitive Science: An Introduction*: Cambridge, MA: The MIT Press, p. 305.
- [30] Rand, Ayn. 1979. *Introduction to Objectivist Epistemology*. New York, NY: New American Library.
- [31] Minsky, M. 1986. *The Society of Mind*. New York, NY: Simon and Schuster.
- [32] Jones, A. 1979. The Object Model: A Conceptual Tool for Structuring Software. *Operating Systems*. New York, NY: Springer-Verlag, p. 8.
- [33] Stroustrup, B. May 1988. What Is Object-Oriented programming? *IEEE Software* vol. 5(3), p. 10.
- [34] Cardelli, L. and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. December 1985. *ACM Computing Surveys* vol. 17(4), p. 481.

- [35] DeMarco, T. 1979. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice-Hall.
- [36] Yourdon, E. 1989. *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- [37] Gane, C. and Sarson, T. 1979. *Structured Systems Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- [38] Ward, P. and Mellor, S. 1985. *Structured Development for Real-Time Systems*. Englewood Cliffs, NJ: Yourdon Press.
- [39] Hatley, D. and Pirbhai, I. 1988. *Strategies for Real-Time System Specification*. New York, NY: Dorset House.
- [40] Jenkins, M. and Glasgow, J. January 1986. Programming Styles in Nial. *IEEE Software* vol. 3(1), p. 48.
- [41] Bobrow, D. and Stefik, M. February 1986. Perspectives on Artificial Intelligence Programming. *Science* vol. 231, p. 951.
- [42] Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming*. London, England: Academic Press, p. 83.
- [43] Shaw, M. October 1984. Abstraction Techniques in Modern Programming Languages. *IEEE Software* vol. 1(4), p. 10.
- [44] Berzins, V., Gray, M., and Naumann, D. May 1986. Abstraction-Based Software Development. *Communications of the ACM* vol. 29(5), p. 403.
- [45] Abelson, H. and Sussman, G. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press, p. 126.
- [46] ibid., p. 132.
- [47] Seidewitz, E. and Stark, M. 1986. Towards a General Object-Oriented Software Development Methodology. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. NASA Lyndon B. Johnson Space Center, TX: NASA, p. D.4.6.4.
- [48] Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall.
- [49] Wirfs-Brock, R. and Wilkerson, B. October 1989. Object-Oriented Design: A Responsibility-Driven Approach. *SIGPLAN Notices* vol. 24(10).
- [50] Ingalls, D. The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*. ACM, p. 9.
- [51] Gannon, J., Hamlet, R., and Mills, H. July 1987. Theory of Modules. *IEEE Transactions on Software Engineering* vol. SE-13(7), p. 820.
- [52] Date, C. 1986. *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley, p. 180.
- [53] Liskov, B. May 1988. Data Abstraction and Hierarchy. *SIGPLAN Notices* vol. 23(5), p. 19.
- [54] Britton, K. and Parnas, D. December 8, 1981. *A-7E Software Module Guide*. Washington, D.C. Naval Research Laboratory, Report 4702, p. 24.
- [55] Gabriel, R. 1990. Private communication.
- [56] Stroustrup, B. 1988. Private communication.
- [57] Myers, G. 1978. *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold, p. 21.
- [58] Liskov, B. 1980. A Design Methodology for Reliable Software Systems, in *Tutorial on Software Design Techniques*. Thrid Edition. New York, NY: IEEE Computer society, p. 66.
- [59] Zelkowitz, M. June 1978. Perspectives on Software Engineering. *ACM Computing Surveys* vol. 10(2), p. 20.

- [60] Parnas, D., Clements, P., and Weiss, D. March 1985. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering* vol. SE-11(3), p. 260.
- [61] Britton and Parnas. *A-7E Software*, p. 2.
- [62] Parnas, D., Clements, P., and Weiss, D. 1983. Enhancing Reusability with Information Hiding. *Proceedings of the Workshop on Reusability in Programming*, Stratford, CT: ITT Programming, p. 241.
- [63] Meyer, *Object-Oriented Software Construction*, p. 47.
- [64] Cox, B. 1986. *Object-Oriented Software Construction*, p. 47.
- [65] Danforth, S. and Tomlinson, C. March 1988. Type Theories and Object-Oriented Programming. *ACM Computing Surveys* vol. 20(1), p. 34.
- [66] Liskov. 1988, p. 23.
- [67] As quoted in Liskov. 1980, p. 67.
- [68] Zilles, S. 1984. Types, Algebras, and Modeling, in *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. New York, NY: Springer-Verlag, p. 442.
- [69] Borning, A. and Ingalls, D. 1982. A Type Declaration and Inference System for Smalltalk. Palo Alto, CA: Xerox Palo Alto Research Center, p. 134.
- [70] Wegner, P. October 1987. Dimensions of Object-Based Language Design. *SIGPLAN Notices* vol. 22(12), p. 171.
- [71] Stroustrup, B. 1992. Private communication.
- [72] Tesler, L. August 1981. The Smalltalk Environment. *Byte* vol. 6(8), p. 142.
- [73] Borning and Ingalls. Type Declaration, p. 133.
- [74] Thomas, D. March 1989. What's in an Object? *Byte* vol. 14(3), p. 232.
- [75] Lim, J. and Johnson, R. April 1989. The Heart of Object-Oriented Concurrent Programming. *SIGPLAN Notices* vol. 24(4), p. 165.
- [76] Ibid., p. 165.
- [77] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. July 1986. *Distribution and Abstract Types in Emerald*. Report 86-02-04. Seattle, WA: University of Washington, p. 3.
- [78] Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming. April 1989. *SIGPLAN Notices* vol. 24(4), p. 1.
- [79] Atkinson, M., Bailey, P., Chisholm, K., Cockshott, P., and Morrison, R. 1983. An Approach to Persistent Programming. *The Computer Journal* vol. 26(4), p. 360.
- [80] Khoshafian, S. and Copeland, G. November 1986. Object Identity. *SIGPLAN Notices* vol. 21(11), p. 409.
- [81] Vbase Technical Overview. September 1987. Billerica, MA: Ontologic, p. 4.
- [82] Strostrup, B. November 1987. Possible Directions for C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, NM, p. 14.
- [83] Meyer. *Object-Oriented Software Construction*, p. 30-31.
- [84] Robson, D. August 1981. Object-Oriented Software Systems. *Byte* vol. 6(8), p. 74.



Clases y objetos

Tanto el ingeniero como el artista deben estar íntimamente familiarizados con los materiales que manejan. Cuando se usan métodos orientados a objetos para analizar o diseñar un sistema de software complejo, los bloques básicos de construcción son las clases y los objetos. Puesto que hasta ahora se han proporcionado sólo definiciones informales de estos dos elementos, este capítulo se vuelve a un estudio detallado de la naturaleza de las clases, los objetos y sus relaciones, y a lo largo del camino se ofrecen varias reglas para construir abstracciones y mecanismos de calidad.

3.1. La naturaleza de los objetos

Qué es y qué no es un objeto

La capacidad de reconocer objetos físicos es una habilidad que los humanos aprenden en edades muy tempranas. Una pelota de colores llamativos atraerá la atención de un niño, pero casi siempre, si se esconde la pelota, el niño no intentará buscarla; cuando el objeto abandona su campo de visión, hasta donde él puede determinar, la pelota ha dejado de existir. Normalmente, hasta la edad de un año un niño no desarrolla lo que se denomina el *concepto de objeto*, una habilidad de importancia crítica para el desarrollo cognitivo futuro. Muéstrese una pelota a un niño de un año y escóndase a continuación, y normalmente la buscará incluso si no está visible. A través del concepto de objeto, un niño llega a darse cuenta de que los objetos tienen una permanencia e identidad además de cualesquiera operaciones sobre ellos [1].

En el capítulo anterior se definió informalmente un objeto como una entidad tangible que exhibe algún comportamiento bien definido. Desde la perspectiva de la cognición humana, un objeto es cualquiera de las siguientes cosas:

- Una cosa tangible y/o visible.
- Algo que puede comprenderse intelectualmente.
- Algo hacia lo que se dirige un pensamiento o acción.

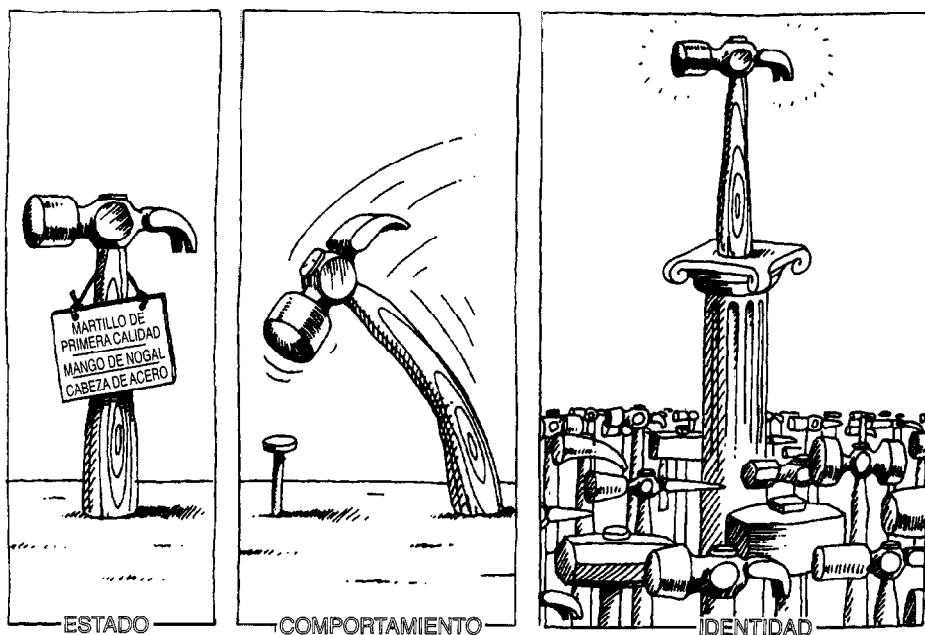
Se añade a la definición informal la idea de que un objeto modela alguna parte de la realidad y es, por tanto, algo que existe en el tiempo y el espacio. En software, el término *objeto* se aplicó formalmente en primer lugar en el lenguaje Simula; los objetos existían en los programas en Simula típicamente para simular algún aspecto de la realidad [2].

Los objetos del mundo real no son el único tipo de objeto de interés en el desarrollo del software. Otros tipos importantes de objetos son invenciones del proceso de diseño cuyas colaboraciones con otros objetos semejantes sirven como mecanismos para desempeñar algún comportamiento de nivel superior [3]. Esto nos lleva a la definición más refinada de Smith y Tockey, que sugieren que «un objeto representa un elemento, unidad o entidad individual e identificable, ya sea real o abstracta, con un papel bien definido en el dominio del problema» [4]. En términos aún más generales, se define un objeto como cualquier cosa que tenga una frontera definida con nitidez [5].

Considérese por un momento una planta de fabricación que procesa materiales compuestos para fabricar elementos tan diversos como cuadros de bicicleta y alas de aeroplano. Las fábricas se dividen con frecuencia en talleres separados: mecánico, químico, eléctrico, etc. Los talleres se dividen además en células, y en cada célula hay alguna colección de máquinas, como troqueladoras, prensas y tornos. A lo largo de una línea de fabricación, se podría encontrar tanques con materias primas, que se utilizan en un proceso químico para producir bloques de materiales compuestos, y a los que a su vez se da forma para producir elementos finales como cuadros para bicicletas y alas de aeroplano. Cada una de las cosas tangibles que se han mencionado hasta aquí es un objeto. Un torno tiene una frontera perfectamente nítida que lo separa del bloque de material compuesto sobre el que opera; un cuadro de bicicleta tiene una frontera perfectamente nítida que lo distingue de la célula o las máquinas que lo producen.

Algunos objetos pueden tener límites conceptuales precisos, pero aun así pueden representar eventos o procesos intangibles. Por ejemplo, un proceso químico en una fábrica puede tratarse como un objeto, porque tiene una frontera conceptual clara, interactúa con otros determinados objetos a lo largo de una colección bien ordenada de operaciones que se despliegan en el tiempo, y exhibe un comportamiento bien definido. Análogamente, considérese un sistema CAD/CAM para modelar sólidos. Donde se intersectan (intersecan) dos sólidos como una esfera y un cubo, pueden formar una línea irregular de intersección. Aunque no existe fuera de la esfera o el cubo, esta línea sigue siendo un objeto con fronteras conceptuales muy precisas.

Algunos objetos pueden ser tangibles, y aun así tener fronteras físicas difusas. Objetos como los ríos, la niebla o las multitudes humanas encajan en esta



Un objeto tiene estado, exhibe algún comportamiento bien definido, tiene una identidad única.

definición¹. Exactamente igual que la persona que sostiene un martillo tiende a verlo todo a su alrededor como un clavo, el desarrollador con una mentalidad orientada a objetos comienza a pensar que todo lo que hay en el mundo son objetos. Esta perspectiva es un poco ingenua, porque existen algunas cosas que claramente no son objetos. Por ejemplo, atributos como el tiempo, la belleza o el color no son objetos, ni las emociones como el amor o la ira. Por otro lado, todas estas cosas son potencialmente propiedades de otros objetos. Por ejemplo, se podría decir que un hombre (un objeto) ama a su mujer (otro objeto), o que cierto gato (un objeto más) es gris.

Así, es útil decir que un objeto es algo que tiene fronteras nítidamente definidas, pero esto no es suficiente para servir de guía al distinguir un objeto de otro, ni permite juzgar la calidad de las abstracciones. Por tanto, nuestra experiencia sugiere la siguiente definición:

Un objeto tiene estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidos en su clase común; los términos instancia y objeto son intercambiables.

¹ Esto es cierto sólo a un nivel de abstracción lo bastante alto. Para una persona que camina a través de un banco de niebla, es generalmente inútil distinguir «mi niebla» de «tu niebla». Sin embargo, considérese un mapa meteorológico: un banco de niebla sobre San Francisco es un objeto claramente distinto de un banco de niebla sobre Londres.

Estado

Semántica. Considérese una máquina expendedora de refrescos. El comportamiento usual de tales objetos es tal que cuando se introducen monedas en una ranura y se pulsa un botón para realizar una selección, surge una bebida de la máquina. ¿Qué pasa si un usuario realiza primero la selección y a continuación introduce dinero en la ranura? La mayoría de las máquinas expendedoras se inhiben y no hacen nada, porque el usuario ha violado las suposiciones básicas de su funcionamiento. Dicho de otro modo, la máquina expendedora estaba haciendo un papel (o esperando monedas) que el usuario ignoró (haciendo primero la selección). Análogamente, supóngase que el usuario ignora la luz de advertencia que dice «Sólo el dinero exacto», e introduce dinero extra. La mayoría de las máquinas son «hostiles al usuario»; se tragará tan felices las monedas sobrantes.

En todas esas circunstancias se ve cómo el comportamiento de un objeto está influenciado por su historia: el orden en el que se opera sobre el objeto es importante. La razón para este comportamiento dependiente del tiempo y los eventos es la existencia de un estado en el interior del objeto. Por ejemplo, un estado esencial asociado con la máquina expendedora es la cantidad de dinero que acaba de introducir un usuario pero que aún no se ha aplicado a una selección. Otras propiedades importantes incluyen la cantidad de cambio disponible y la cantidad de refrescos que tiene.

De este ejemplo se puede extraer la siguiente definición de bajo nivel:

El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades.

Otra propiedad de una máquina expendedora es que puede aceptar monedas. Ésta es una propiedad estática (es decir, fija), lo que significa que es una característica esencial de una máquina expendedora. En contraste, la cantidad actual de monedas que ha aceptado en un momento dado representa el valor dinámico de esta propiedad, y se ve afectado por el orden de las operaciones que se efectúan sobre la máquina. Esta cantidad aumenta cuando el usuario introduce monedas, y disminuye cuando un empleado atiende la máquina. Se dice que los valores son «normalmente dinámicos» porque en algunos casos los valores son estáticos. Por ejemplo, el número de serie de una máquina expendedora es una propiedad estática, y el valor es estático.

Una propiedad es una característica inherente o distintiva, un rasgo o cualidad que contribuye a hacer que un objeto sea ese objeto y no otro. Por ejemplo, una propiedad esencial de un ascensor es que está restringido a moverse arriba y abajo y no horizontalmente. Las propiedades suelen ser estáticas, porque atributos como éstos son inmutables y fundamentales para la naturaleza del objeto. Se dice «suelen ser» porque en algunas circunstancias las propiedades de un objeto pueden cambiar. Por ejemplo, considérese un robot autónomo que

puede aprender sobre su entorno. Puede reconocer primero un objeto que parece ser una barrera inamovible, para aprender después que este objeto es de hecho una puerta que puede abrirse. En este caso, el objeto creado por el robot a medida que este construye su modelo conceptual del mundo gana nuevas propiedades a medida que adquiere nuevo conocimiento.

Todas las propiedades tienen algún valor. Este valor puede ser una mera cantidad, o puede denotar a otro objeto. Por ejemplo, parte del estado de un ascensor puede tener el valor 3, que denota el piso actual en el que está. En el caso de la máquina expendedora, el estado de la misma abarca muchos otros objetos, tales como una colección de refrescos. Los refrescos individuales son de hecho objetos distintos; sus propiedades son diferentes de las de la máquina (pueden consumirse, mientras que una expendedora no puede), y puede operarse sobre ellos de formas claramente diferentes. Así, se distingue entre objetos y valores simples: las cantidades simples como el número 3 son «intemporales, inmutables y no instanciadas», mientras que los objetos «existen en el tiempo, son modificables, tienen estado, son instanciados y pueden crearse, destruirse y compartirse» [6].

El hecho de que todo objeto tiene un estado implica que todo objeto toma cierta cantidad de espacio, ya sea en el mundo físico o en la memoria del computador.

Ejemplo: Considérese la estructura de un registro de personal. En C++ podría escribirse:

```
struct RegistroPersonal
{
    char nombre[100];
    int numeroSeguridadSocial;
    char departamento[10];
    float salario;
};
```

Cada parte de esta estructura denota una propiedad particular de la abstracción de un registro de personal. Esta declaración denota una clase, no un objeto, porque no representa una instancia específica². Para declarar objetos de esta clase, se escribe:

```
RegistroPersonal deb, dave, karen, jim, tom, denise, kaitlyn,
krista, elyse;
```

Aquí hay nueve objetos distintos, cada uno de los cuales ocupa una cierta cantidad de espacio en memoria. Ninguno de estos objetos comparte su espacio

² Para ser precisos, esta declaración denota una estructura, una construcción de registro de C++ de nivel inferior cuya semántica es la misma que la de una clase con todos sus miembros public. Las estructuras denotan así una abstracción no encapsulada.



con ningún otro objeto, aunque todos ellos tienen las mismas propiedades; por tanto, sus estados tienen una representación común.

Es una buena práctica de ingeniería el encapsular el estado de un objeto en vez de exponerlo como en la declaración precedente. Por ejemplo, se podría reescribir la declaración de clase como sigue:

```
class RegistroPersonal {
public:
    char *nombreEmpleado() const;
    int numeroSeguridadSocialEmpleado() const;
    char *departamentoEmpleado() const;
protected:
    char nombre[100];
    int numeroSeguridadSocial;
    char departamento[10];
    float salario;
};
```

Esta declaración es ligeramente más complicada que la anterior, pero es mucho mejor por varias razones³. Específicamente, se ha escrito esta clase de forma que su representación está oculta para todos los clientes externos. Si se cambia su representación, habrá que recompilar algún código, pero semánticamente ningún cliente externo resultará afectado por este cambio (en otras palabras, el código ya existente no se estropeará). Además, se han capturado algunas decisiones acerca del espacio del problema estableciendo explícitamente algunas de las operaciones que los clientes pueden realizar sobre objetos de esta clase. En particular, se garantiza a todos los clientes el derecho a recuperar el nombre, número de la seguridad social y departamento de un empleado. Sólo clientes especiales (es decir, subclases de esta clase) tienen permiso para modificar los valores de estas propiedades. Más aún, sólo estos clientes especiales pueden modificar o recuperar el salario de un empleado, mientras que los clientes externos no pueden. Otra razón por la que esta declaración es mejor tiene que ver con la reutilización. Como se verá en una sección posterior, la herencia hace posible la reutilización de esta abstracción, y refinirla o especializarla de diversas formas.

Puede decirse que todos los objetos de un sistema encapsulan algún estado, y que todo el estado de un sistema está encapsulado en objetos. Sin embargo, encapsular el estado de un objeto es un punto de partida, pero no es suficiente para permitir que se capturen todos los designios de las abstracciones que se

³ Un problema de estilo: la clase `RegistroPersonal` tal como se ha declarado aquí no es una clase de altísima calidad, de acuerdo con las métricas que se describen más adelante en este capítulo (este ejemplo sólo sirve para ilustrar la semántica del estado de una clase). Tener una función miembro que devuelve un valor de tipo `char*` es a menudo peligroso, porque esto viola uno de los principios de seguridad de la memoria: si el método crea espacio de almacenamiento del que el cliente no se hace responsable, la recolección de memoria (basura) implicará dejar referencias (puntcos) colgadas. En sistemas de producción, es preferible usar una clase parametrizada de cadenas de longitud variable, como se encuentra en una biblioteca de clases básica como la que se describe en el Capítulo 9. Además, las clases son más simples que la `struct` del C envueltas en sintaxis del C++; como se explica en el Capítulo 4, la clasificación requiere una atención deliberada hacia estructura y comportamiento comunes.

descubren e inventan durante el desarrollo. Por esta razón, hay que considerar también cómo se comportan los objetos.

Comportamiento

El significado del comportamiento. Ningún objeto existe de forma aislada. En vez de eso, los objetos reciben acciones, y ellos mismos actúan sobre otros objetos. Así, puede decirse que

El comportamiento es cómo actúa y reacciona un objeto, en términos de sus cambios de estado y paso de mensajes.

En otras palabras, el comportamiento de un objeto representa su actividad visible y comprobable exteriormente.

Una operación es una acción que un objeto efectúa sobre otro con el fin de provocar una reacción. Por ejemplo, un cliente podría invocar las operaciones *anadir* y *extraer* para aumentar y disminuir un objeto cola, respectivamente. Un cliente podría también invocar la operación *longitud*, que devuelve un valor denotando el tamaño del objeto cola pero no altera el estado de la misma. En lenguajes orientados a objetos puros como Smalltalk, se habla de que un objeto pasa un mensaje a otro. En lenguajes como C++, que deriva de antecesores más procedimentales, se habla de que un objeto invoca una función miembro de otro. Generalmente, un mensaje es simplemente una operación que un objeto realiza sobre otro, aunque los mecanismos subyacentes para atenderla son distintos. Para nuestros propósitos, los términos *operación* y *mensaje* son intercambiables.

En la mayoría de los lenguajes de programación orientados a objetos, las operaciones que los clientes pueden realizar sobre un objeto suelen declararse como *métodos*, que forman parte de la declaración de la clase. C++ usa el término *función miembro* para denotar el mismo concepto; se utilizarán ambos vocablos de forma equivalente.

El paso de mensajes es una de las partes de la ecuación que define el comportamiento de un objeto; la definición de «comportamiento» también recoge que el estado de un objeto afecta asimismo a su comportamiento. Considérese de nuevo el ejemplo de la máquina expendedora. Se puede invocar alguna operación para hacer una selección, pero la expendedora se comportará de modo diferente según su estado. Si no se deposita suficiente dinero para nuestra selección, posiblemente la máquina no hará nada. Si se introduce suficiente dinero, la máquina tomará el dinero y suministrará lo que se ha seleccionado (alterando con ello su estado). Así, se puede decir que el comportamiento de un objeto es función de su estado así como de la operación que se realiza sobre él, teniendo algunas operaciones el efecto lateral de modificar el estado del objeto. Este concepto de efecto lateral conduce así a refinar la definición de estado:

El estado de un objeto representa los resultados acumulados de su comportamiento.

Los objetos más interesantes no tienen un estado estático; antes bien, su estado tiene propiedades cuyos valores se modifican y consultan según se actúa sobre el objeto.

Ejemplo: Considérese la siguiente declaración de una clase cola en C++:

```
class Cola {
public:

    Cola();
    Cola(const Cola&);
    virtual ~Cola();

    virtual Cola& operator=(const Cola&);
    virtual int operator==(const Cola&) const;
    int operator!=(const Cola&) const;

    virtual void borrar();
    virtual void anadir(const void*);
    virtual void extraer();
    virtual void eliminar(int donde);

    virtual int longitud() const;
    virtual int estaVacia() const;
    virtual const void *cabecera() const;
    virtual int posicion(const void *);

protected:
    ...
};
```

Esta clase utiliza el modismo habitual del C por el que se fijan y recuperan valores mediante `void*`, que proporciona la abstracción de una cola heterogénea, lo que significa que los clientes pueden añadir objetos de cualquier clase a un objeto cola. Este enfoque no es demasiado seguro respecto a los tipos, porque el cliente debe recordar la clase de los objetos que hay en la cola. Además, el uso de `void*` evita que el objeto `Cola` «posea» sus elementos, lo que quiere decir que no se puede confiar en la actuación del destructor de la cola (`~Cola()`) para destruir los elementos de la misma. En una próxima sección se estudiarán los tipos parametrizados, que mitigan estos problemas.

Ya que la declaración `Cola` representa una clase, no un objeto, hay que declarar instancias que los clientes puedan manipular:

```
Cola a, b, c, d;
```

A continuación, se puede operar sobre estos objetos como en el código que sigue:

```
a.anadir(&deb);
a.anadir(&karen);
a.anadir(&denise);
b = a;
a.extraer();
```

Después de ejecutar estas sentencias, la cola denotada por `a` contiene dos elementos (con un puntero al registro `karen` en su cabecera), y la cola denotada por `b` contiene tres elementos (con el registro `deb` en su cabecera). De este modo, cada uno de estos objetos de cola contiene algún estado distinto, y este estado afecta al comportamiento futuro de cada objeto. Por ejemplo, se puede extraer elementos de `b` de forma segura tres veces más, pero sobre `a` sólo puede efectuarse esta operación de forma segura otras dos veces.

Operaciones. Una operación denota un servicio que una clase ofrece a sus clientes. En la práctica, se ha visto que un cliente realiza típicamente cinco tipos de operaciones sobre un objeto⁴. Los tres tipos más comunes de operaciones son los siguientes:

- Modificador Una operación que altera el estado de un objeto.
- Selector Una operación que accede al estado de un objeto, pero no altera ese estado.
- Iterador Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido

Puesto que estas operaciones son tan distintas lógicamente, se ha hallado útil aplicar un estilo de codificación que subraya sus diferencias. Por ejemplo, en la declaración de la clase `cola`, se declaran primero todos los modificadores como funciones miembro `no-const` (las operaciones `borrar`, `anadir`, `extraer` y `eliminar`), seguidas por todos los selectores como funciones `const` (las operaciones `longitud`, `estavacia`, `cabecera` y `posicion`). Como se ilustrará en el Capítulo 9, el estilo que se propone es definir una clase separada que actúa como el agente responsable de iterar a lo largo de las colas.

Hay otros dos tipos de operaciones habituales; representan la infraestructura necesaria para crear y destruir instancias de una clase:

- Constructor Una operación que crea un objeto y/o inicializa su estado.
- Destructor Una operación que libera el estado de un objeto y/o destruye el propio objeto.

En C++, los constructores y destructores se declaran como parte de las definiciones de una clase (los miembros `cola` y `~cola`), mientras que en Smalltalk

⁴ Lippman sugiere una categorización ligeramente diferente: funciones de manejo, funciones de implantación, funciones de asistencia (todo tipo de modificadores), y funciones de acceso (equivalentes a selectores) [7].

y CLOS tales operaciones suelen ser parte del protocolo de una metaclasé (es decir, la cláse de una cláse).

En lenguajes orientados a objetos puros como Smalltalk, sólo pueden declararse las operaciones como métodos, ya que el lenguaje no permite declarar procedimientos o funciones separados de ninguna cláse. Por contra, los lenguajes como Object Pascal, C++, CLOS y Ada permiten al desarrollador escribir operaciones como subprogramas libres; en C++, se les llama funciones no miembro. Los *subprogramas libres* son procedimientos o funciones que sirven como operaciones no primitivas sobre un objeto u objetos de la misma o de distintas cláses. Los subprogramas libres están típicamente agrupados según las cláses sobre las que se los ha construido; por tanto, se llama a tales colecciones de subprogramas libres *utilidades de cláse*. Por ejemplo, dada la declaracióne precedente del paquete `cola`, se podría escribir la siguiente función no miembro:

```
void copiarHastaEncontrado(Cola& fuente, Cola& destino, void*
elemento)
{
    while ((!fuente.estaVacia()) &&
           (fuente.cabecera() != elemento)) {
        destino.anadir(fuente.cabecera());
        fuente.extraer();
    }
}
```

El propósito de esta operación es copiar repetidamente y entonces extraer el contenido de una cola hasta que se encuentra el elemento dado en la cabecera de la misma. Esta operación no es primitiva; puede construirse a partir de operaciones de nivel inferior que ya son parte de la cláse `cola`.

Forma parte del estilo habitual del C++ (y del Smalltalk) recoger todos los subprogramas libres relacionados lógicamente y declararlos como parte de una cláse que no tiene estado. En particular, en C++ esto se hace static.

Así, puede decirse que todos los métodos son operaciones, pero no todas las operaciones son métodos: algunas operaciones pueden expresarse como subprogramas libres. En la práctica, es preferible declarar la mayoría de las operaciones como métodos, si bien, como se observa en una sección posterior, hay a veces razones de peso para obrar de otro modo, como cuando una operación concreta afecta a dos o más objetos de cláses diferentes, y no se deriva ningún beneficio especial al declarar esa operación en una cláse y no en la otra.

Papeles (roles) y responsabilidades. Colectivamente, todos los métodos y subprogramas libres asociados con un objeto concreto forman su *protocolo*. El protocolo define así la envoltura del comportamiento admisible en un objeto, y por tanto engloba la visión estática y dinámica completa del mismo. Para la mayoría de las abstracciones no triviales, es útil dividir este protocolo mayor en

grupos lógicos de comportamiento. Estas colecciones, que constituyen una partición del espacio de comportamiento de un objeto, denotan los *papeles* que un objeto puede desempeñar. Como sugiere Adams, un papel es una máscara que se pone un objeto [8] y por tanto define un contrato entre una abstracción y sus clientes.

Unificando nuestras definiciones de estado y comportamiento, Wirfs-Brock define las *responsabilidades* de un objeto de forma que «incluyen dos elementos clave: el conocimiento que un objeto mantiene y las acciones que puede llevar a cabo. Las responsabilidades están encaminadas a transmitir un sentido del propósito de un objeto y de su lugar en el sistema. Las responsabilidades de un objeto son todos los servicios que proporciona para todos los contratos que soporta» [9]. En otras palabras, se puede decir que el estado y comportamiento de un objeto definen en conjunto los papeles que puede representar un objeto en el mundo, los cuales a su vez cumplen las responsabilidades de la abstracción.

Realmente, los objetos más interesantes pueden representar muchos papeles diferentes durante su tiempo de vida; por ejemplo [10]:

- Una cuenta bancaria puede estar en buena o mala situación, y el papel en el que esté afecta a la semántica de un reintegro.
- Para un comerciante, una acción de bolsa representa una entidad con cierto valor que puede comprarse o venderse; para un abogado, la misma acción denota un instrumento legal que abarca ciertos derechos.
- En el curso de un día, la misma persona puede representar el papel de madre, doctor, jardinero y crítico de cine.

En el caso de la cuenta bancaria, los papeles que este objeto puede desempeñar son dinámicos, pero exclusivos mutuamente: puede estar saneada o en números rojos, pero no ambas cosas a la vez. En el caso de las acciones, sus papeles se superponen ligeramente, pero cada papel es estático en relación con el cliente que interacciona con ellas. En el caso de la persona, sus papeles son bastante dinámicos, y pueden cambiar de un momento a otro.

Como se verá en los Capítulos 4 y 6, se inicia con frecuencia el análisis de un problema examinando los distintos papeles que puede desempeñar un objeto. Durante el diseño, se refinan estos papeles descubriendo las operaciones particulares que llevan a cabo las responsabilidades de cada papel.

Los objetos como máquinas. La existencia de un estado en el seno de un objeto significa que el orden en el que se invocan las operaciones es importante. Esto da lugar a la idea de que cada objeto es como una pequeña máquina independiente [11]. Realmente, para algunos objetos, esta ordenación de las operaciones respecto a los eventos y al tiempo es tan penetrante que se puede caracterizar mejor formalmente el comportamiento de tales objetos en términos de una máquina de estados finitos equivalente. En el Capítulo 5, se mostrará una notación específica para máquinas de estados finitos jerárquicas que se puede utilizar para expresar esta semántica.

Siguiendo con la metáfora de las máquinas, se pueden clasificar los objetos

como activos o pasivos. Un *objeto activo* es aquel que comprende su propio hilo de control, mientras que un *objeto pasivo* no. Los objetos activos suelen ser autónomos, lo que quiere decir que pueden exhibir algún comportamiento sin que ningún otro objeto opere sobre ellos. Los objetos pasivos, por otra parte, sólo pueden padecer un cambio de estado cuando se actúa explícitamente sobre ellos. De este modo, los objetos activos del sistema sirven como las raíces del control. Si el sistema comprende múltiples hilos de control, habrá generalmente múltiples objetos activos. Los sistemas secuenciales, por contra, suelen tener exactamente un objeto activo, tal como un objeto de tipo ventana principal responsable de manejar un bucle de eventos que despacha mensajes. En tales arquitecturas, todos los demás objetos son pasivos, y su comportamiento en última instancia es disparado por mensajes del único objeto activo. En otros tipos de arquitecturas de sistemas secuenciales (como los sistemas de procesamiento de transacciones), no existe ningún objeto activo central evidente, y, por tanto, el control tiende a estar distribuido entre los objetos pasivos del sistema.

Identidad

Semántica. Khoshafian y Copeland ofrecen la siguiente definición:

«*La identidad es aquella propiedad de un objeto que lo distingue de todos los demás objetos*» [12].

A continuación hacen notar que «la mayoría de los lenguajes de programación y de bases de datos utilizan nombres de variable para distinguir objetos temporales, mezclando la posibilidad de acceder a ellos con su identidad. La mayoría de los sistemas de bases de datos utilizan claves de identificación para distinguir objetos persistentes, mezclando el valor de un dato con la identidad.» El fracaso en reconocer la diferencia entre el nombre de un objeto y el objeto en sí mismo es fuente de muchos tipos de errores en la programación orientada a objetos.

Ejemplo. Considérense las siguientes declaraciones en C++. Primero, se va a comenzar con una estructura simple que denota un punto en el espacio:

```
struct Punto {
    int x;
    int y;
    Punto() : x(0), y(0) {}
    Punto(int valorX, int valorY) : x(valorX), y(valorY) {}
};
```

Aquí se ha elegido declarar Punto como una estructura, no como una clase en toda su dimensión. La regla que se aplica para hacer esta distinción es simple. Si la abstracción representa un simple registro de otros objetos y no tiene

un comportamiento verdaderamente interesante que se aplique al objeto en su conjunto, hágase una estructura. Sin embargo, si la abstracción exige un comportamiento más intenso que la simple introducción y recuperación de elementos altamente independientes del registro, hágase una clase. En el caso de la abstracción Punto, se define un punto como la representación de unas coordenadas (x , y) en el espacio. Por conveniencia, se suministra un constructor que proporciona un valor (0,0) por defecto, y otro constructor que inicializa un punto con un valor explícito (x , y).

A continuación se proporciona una clase que denota un elemento de pantalla. Un elemento de pantalla es una abstracción habitual en todos los sistemas basados en IGU: representa la clase base de todos los objetos que tienen una representación visual en alguna ventana, y así refleja la estructura y comportamiento comunes a todos esos objetos. He aquí una abstracción que es más que un simple registro de datos. Los clientes esperan ser capaces de dibujar, seleccionar y mover elementos de pantalla, así como interrogar sobre su estado de selección y su posición. Se puede plasmar la abstracción en la siguiente declaración de C++:

```
class ElementoPantalla {
public:

    ElementoPantalla();
    ElementoPantalla(const Punto& posicion);
    virtual ~ElementoPantalla();

    virtual void dibujar();
    virtual void borrar();
    virtual void seleccionar();
    virtual void quitarSeleccion();
    virtual void mover(const Punto& posicion);

    int estaSeleccionado() const;
    Punto posicion() const;
    int estaBajo(const Punto& posicion) const;

protected:
    ...
};
```

Esta declaración está incompleta: se han omitido intencionadamente todos los constructores y operadores necesarios para manejar la copia, asignación y pruebas de igualdad. Se considerarán estos aspectos de la abstracción en la siguiente sección.

Puesto que se espera que los clientes declaren subclases de esta clase, se ha declarado el destructor y todos sus modificadores como virtual. En particular, se espera que las subclases concretas redefinan `dibujar` para reflejar el compor-

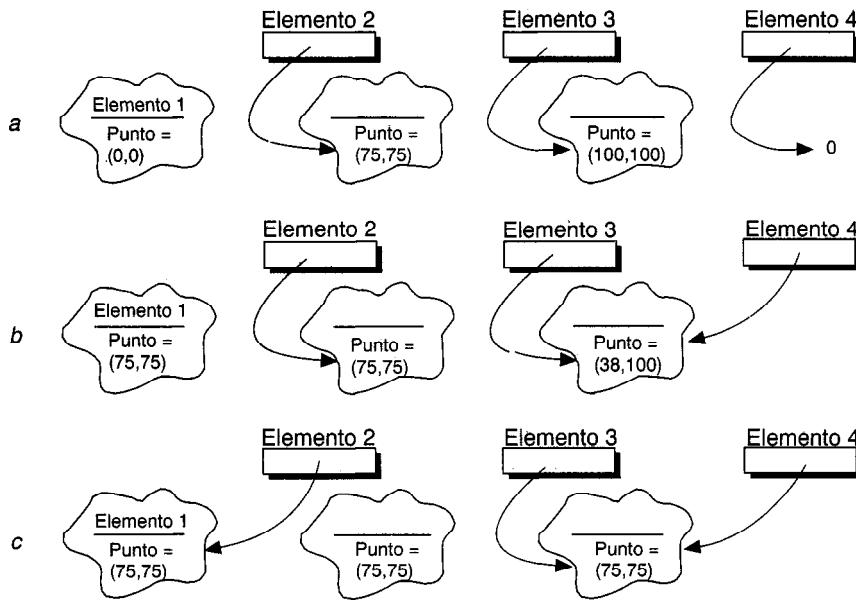


Figura 3.1. Identidad de los objetos.

tamiento de dibujar en una ventana elementos específicos de cada dominio. No se ha declarado ninguno de los selectores como virtual, porque no se espera que las subclases redefinan este comportamiento. Nótese también que el selector `estaBajo` implica algo más que la recuperación de un simple valor del estado. Aquí, la semántica de esta operación requiere que el objeto calcule si el punto dado está en cualquier lugar del interior de la trama del elemento de pantalla.

Para declarar instancias de esta clase, se podría escribir lo siguiente:

```
ElementoPantalla*elemento1;
ElementoPantalla*elemento2=newElementoPantalla(Punto(75,75));
ElementoPantalla*elemento3=newElementoPantalla(Punto(100,100));
ElementoPantalla*elemento4=0;
```

Como muestra la Figura 3.1a, la elaboración de estas declaraciones crea cuatro nombres y tres objetos distintos. Específicamente, aparecen cuatro asignaciones distintas de memoria cuyos nombres son `elemento1`, `elemento2`, `elemento3` y `elemento4`, respectivamente. Además, `elemento1` es el nombre de un objeto `ElementoPantalla` concreto, pero los otros tres nombres denotan cada uno un *puntero* a un objeto `ElementoPantalla`. Sólo `elemento2` y `elemento3` apuntan realmente a objetos `ElementoPantalla` precisos (porque sólo sus declaraciones asignan espacio a un nuevo objeto `ElementoPantalla`); `elemento4` no designa tal objeto. Es más, los nombres de los objetos apuntados por `elemento2` y `elemento3` son anónimos; sólo puede hacerse referencia a esos ob-

jetos concretos indirectamente, *desreferenciando* el valor de su puntero. Así, perfectamente se puede decir que `elemento2` apunta a un objeto `ElementoPantalla` concreto, cuyo nombre se puede expresar indirectamente como `*elemento2`. La identidad única (pero no necesariamente el nombre) de cada objeto se preserva durante el tiempo de vida del mismo, incluso cuando su estado cambia. Es como la cuestión Zen sobre los ríos: ¿es un río el mismo río de un día a otro, incluso aunque nunca fluye la misma agua a través de él? Por ejemplo, considérense los resultados de ejecutar las siguientes sentencias:

```
elemento1.mover(elemento3->posicion());
elemento4 = elemento3;
elemento4->tmove(Punto(38, 100));
```

La Figura 3.1b ilustra estos resultados. Aquí se aprecia que `elemento1` y el objeto designado por `elemento2` tienen el mismo estado en su posición, y que `elemento4` ahora designa también el mismo objeto que `elemento3`. Nótese que se usa la frase «el objeto designado por `elemento2`» en vez de decir «el objeto `elemento2`». La primera expresión es más precisa, aunque a veces se utilizarán esas frases de forma equivalente.

Aunque `elemento1` y el objeto designado por `elemento2` tienen el mismo estado, representan objetos distintos. Además, nótese que se ha cambiado el estado del objeto designado por `elemento3` operando sobre él mediante su nuevo nombre indirecto, `elemento4`. Esta es una situación que se denomina *compartición estructural*, lo que quiere decir que un objeto dado puede nombrarse de más de una manera; en otras palabras, existen alias para el objeto. La compartición estructural es fuente de muchos problemas en programación orientada a objetos. El fracaso al reconocer los efectos laterales de operar sobre un objeto a través de alias lleva muchas veces a pérdidas de memoria, violaciones de acceso a memoria y, peor aún, cambios de estado inesperados. Por ejemplo, si se destruyese el objeto designado por `elemento3` utilizando la expresión `delete elemento3`, entonces el valor del puntero `elemento4` podría no tener significado; esta situación se denomina *referencia colgada* (*dangling reference*).

Considérese también la Figura 3.1c, que ilustra los resultados de ejecutar las siguientes sentencias:

```
elemento2 = &elemento1;
elemento4->mover(elemento2->posicion());
```

La primera sentencia introduce un alias, porque ahora `elemento2` designa el mismo objeto que `elemento1`; la segunda sentencia accede al estado de `elemento1` a través del nuevo alias. Desgraciadamente, se ha introducido una pérdida de memoria: el objeto originalmente designado por `elemento2` ya no puede ser llamado, ya sea directa o indirectamente, y así su identidad se ha perdido. En lenguajes como Smalltalk y CLOS, tales objetos serán localizados por el recolector de basura y su espacio de memoria recobrado automáticamente,

pero en lenguajes como C++, su espacio no se recobrará hasta que el programa que los creó finalice. Especialmente para programas de ejecución larga, las pérdidas de memoria de este tipo son fastidiosas o desastrosas⁵.

Copia, asignación e igualdad. La compartición estructural se da cuando la identidad de un objeto recibe un alias a través de un segundo nombre. En la mayoría de las aplicaciones interesantes orientadas a objetos, el uso de alias simplemente no puede evitarse. Por ejemplo, considérense las siguientes dos declaraciones de funciones en C++:

```
void remarcar(ElementoPantalla& i);
void arrastrar(ElementoPantalla i); // Peligroso
```

Invocar la primera función con el argumento `elemento1` crea un alias: el parámetro formal `i` denota una referencia al objeto designado por el parámetro actual, y desde ahora `elemento1` e `i` nombrarán al mismo objeto en tiempo de ejecución. Por contra, la invocación de la segunda función con el argumento `elemento1` produce una copia del parámetro actual, y por tanto no existe alias: `i` denota un objeto completamente diferente (pero con el mismo estado) que `elemento1`. En lenguajes como C++ donde existe una distinción entre pasar argumentos por referencia o por valor, hay que tener el cuidado de evitar operar sobre una copia de un objeto, cuando lo que se pretendía era operar sobre el objeto original⁶. Realmente, como se verá en una próxima sección, el pasar objetos por referencia en C++ es esencial para promover un comportamiento polimórfico. En general, el paso de objetos por referencia es la práctica más deseable para objetos no primitivos, porque su semántica sólo involucra el copiar referencias, no estados, y de aquí que sea mucho más eficiente para pasar cualquier cosa que sea más grande que valores elementales.

En algunas circunstancias, sin embargo, la copia es la semántica que se pretendía utilizar, y en lenguajes como C++ es posible controlar la semántica de la copia. En particular, se puede introducir un constructor de copia en la declaración de una clase, como en el siguiente fragmento de código, que se declararía como parte de la declaración para `ElementoPantalla`:

```
ElementoPantalla(const ElementoPantalla&);
```

En C++, puede invocarse un constructor de copia explícitamente (como parte de la declaración de un objeto) o implícitamente (como cuando se pasa un objeto por valor). La omisión de este constructor especial invoca el constructor de

⁵ Considerense los efectos de la pérdida de memoria en el software que controla un satélite o un marcapasos. Reiniciar el computador en un satélite que está a varios millones de kilómetros de la Tierra es bastante poco conveniente. Análogamente, la impredecible entrada en funcionamiento de una recolección de basura automáticamente en el software de un marcapasos será probablemente fatal. Por estas razones, los desarrolladores de sistemas en tiempo real evitan a menudo la asignación indiscriminada de espacio para los objetos en la memoria dinámica.

⁶ En Smalltalk, la semántica de pasar objetos como argumentos para métodos es el equivalente del paso de argumentos por referencia en C++.

copia por defecto, cuya semántica está definida como una copia de miembros. Sin embargo, para objetos cuyo estado propio involucra a punteros o referencias a otros objetos, la copia por defecto de miembros suele ser peligrosa, porque la copia introduce entonces de forma implícita alias de nivel inferior. La regla que se aplica, por lo tanto, es que se omite un constructor de copia explícito sólo para aquellas abstracciones cuyo estado se compone de valores simples, primitivos; en todos los demás casos, normalmente se proporciona un constructor de copia explícito.

Esta práctica distingue lo que algunos lenguajes llaman copia *superficial* versus *profunda*. Smalltalk, por ejemplo, proporciona los métodos `shallowCopy` (o copia superficial, que copia el objeto pero comparte su estado) y `deepCopy` (o copia en profundidad, que copia el objeto así como su estado, y así recursivamente). La redefinición de estas operaciones para clases agregadas permite una mezcla de semánticas: la copia de un objeto de nivel más alto podría copiar la mayor parte de su estado, pero podría introducir alias para ciertos elementos de nivel más bajo.

La asignación es del mismo modo en general una operación de copia, y en lenguajes como C++, su semántica puede controlarse también. Por ejemplo, se podría añadir la siguiente declaración a la declaración realizada para `ElementoPantalla`:

```
virtual ElementoPantalla& operator=(const ElementoPantalla&);
```

Se declara este operador como virtual, porque se espera que una subclase redefina su comportamiento. Como con el constructor de copia, se puede implantar esta operación para proporcionar una semántica de copia superficial o en profundidad. La omisión de esta declaración explícita invoca el operador de asignación por defecto, cuya semántica se define como una copia de miembros.

El problema de la igualdad está en relación muy estrecha con el de la asignación. Aunque se presenta como un concepto simple, la igualdad puede significar una de dos cosas.

Primero, la igualdad puede significar que dos nombres designan el mismo objeto. Segundo, la igualdad puede significar que dos nombres designan objetos distintos cuyos estados son iguales. Por ejemplo, en la Figura 3.1c, ambos tipos de igualdad se evalúan como ciertos entre `elemento1` y `elemento2`. Sin embargo, sólo el segundo tipo de igualdad se evalúa como cierto entre `elemento1` y `elemento3`.

En C++ no hay un operador de igualdad por defecto, así que hay que establecer la semántica que se deseé introduciendo los operadores explícitos para igualdad y desigualdad como parte de la declaración de `ElementoPantalla`.

```
virtual int operator==(const ElementoPantalla&) const;
int operator!=(const ElementoPantalla&) const;
```

El estilo adoptado es declarar el operador de igualdad como virtual (porque

se espera que las subclases redefinan su comportamiento) y declarar el operador de desigualdad como no virtual (se desea siempre que el operador de desigualdad signifique la negación lógica de la igualdad; las subclases no deberían sobreescribir este comportamiento).

De manera similar, se puede definir explícitamente el significado de los operadores de ordenación, como las pruebas para menor-que o mayor-que entre dos objetos.

Espacio de vida de un objeto. El tiempo de vida de un objeto se extiende desde el momento en que se crea por primera vez (y consume así espacio por primera vez) hasta que ese espacio se recupera. Para crear explícitamente un objeto, hay que declararlo o bien asignarle memoria dinámicamente.

Declarar un objeto (como `elemento1` en el ejemplo anterior) crea una nueva instancia en la pila. Reservar espacio para un objeto (como `elemento3`) crea una nueva instancia en el montículo (*heap*). En C++, en ambos casos, siempre que se crea un objeto, se invoca automáticamente a su constructor, cuyo propósito es asignar espacio para el objeto y establecer un estado inicial estable. En lenguajes como Smalltalk, tales operaciones de constructor son realmente parte de la metaclasa del objeto, no de la clase (se examinará la semántica de las metaclasses más adelante en este capítulo).

Frecuentemente, los objetos se crean de forma implícita. Por ejemplo, en C++ el paso de un objeto por valor crea en la pila un nuevo objeto que es una copia del parámetro actual. Es más, la creación de objetos es transitiva: crear un objeto agregado también crea cualquier objeto que sea físicamente parte del conjunto. La redefinición de la semántica del constructor de copia y del operador de asignación en C++ permite un control explícito sobre el momento en que tales partes se crean y destruyen. Además, en C++ es posible redefinir la semántica del operador `new` (que asigna espacio para instancias en el heap), de forma que cada clase puede proporcionar su propia política de manejo de memoria.

En lenguajes como Smalltalk, un objeto se destruye automáticamente como parte de la recolección de basura cuando todas las referencias a él se han perdido. En lenguajes sin recolección de basura, como C++, un objeto sigue existiendo y consume espacio incluso si todas las referencias a él han desaparecido. Los objetos creados en la pila son destruidos de manera implícita siempre que el control sale del bloque en el que se declaró el objeto. Los objetos creados en el heap con el operador `new` deben ser destruidos explícitamente con el operador `delete`. Si esto no se hace bien se producirán pérdidas de memoria, como se vio anteriormente. Liberar dos veces el espacio de memoria de un objeto (habitualmente a causa de un alias) es igualmente indeseable, y puede manifestarse en corrupciones de la memoria o en una caída completa del sistema.

En C++, siempre que se destruye un objeto ya sea implícita o explícitamente, se invoca automáticamente a su destructor, cuyo propósito es devolver

el espacio asignado al objeto y sus partes, y llevar a cabo cualquier otra limpieza posterior a la existencia del objeto (como cerrar archivos o liberar recursos)⁷.

Los objetos persistentes tienen una semántica ligeramente diferente respecto a la destrucción. Como se discutió en el capítulo anterior, ciertos objetos pueden ser persistentes, lo que quiere decir que su tiempo de vida trasciende al tiempo de vida del programa que los creó. Los objetos persistentes suelen ser elementos de algún marco de referencia mayor de una base de datos orientada a objetos, y así la semántica de la destrucción (y la creación) son en gran medida función de la política de la base de datos concreta. En tales sistemas, el enfoque más habitual de la persistencia es el uso de una clase aditiva persistente. Todos los objetos para los que se desea persistencia tienen así a esta clase aditiva como superclase en algún punto de su trama de herencias de clases.

3.2. Relaciones entre objetos

Tipos de relaciones

Un objeto por sí mismo es bastante poco interesante. Los objetos contribuyen al comportamiento de un sistema colaborando con otros. Como sugiere Ingalls, «en lugar de un procesador triturador de bits que golpea y saquea estructuras de datos, tenemos un universo de objetos bien educados que cortésmente solicitan a los demás que lleven a cabo sus diversos deseos» [13]. Por ejemplo, considérese la estructura de objetos de un aeroplano, que se ha definido como «una colección de partes con una tendencia innata a caer a tierra, y que requiere esfuerzos y supervisión constantes para atajar ese suceso» [14]. Sólo los esfuerzos en colaboración de todos los objetos componentes de un aeroplano lo hacen capaz de volar.

La relación entre dos objetos cualesquiera abarca las suposiciones que cada uno realiza acerca del otro, incluyendo qué operaciones pueden realizarse y qué comportamiento se obtiene. Se ha encontrado que hay dos tipos de jerarquías de objetos de interés especial en el análisis y diseño orientados a objetos, a saber:

- Enlaces.
- Agregación.

Seidewitz y Stark las llaman relaciones de *antigüedad* y de *padre/hijo*, respectivamente [15].

⁷ Los destructores no recuperan automáticamente el espacio asignado por el operador new; los programadores deben recobrar este espacio explicitamente como parte de la destrucción.

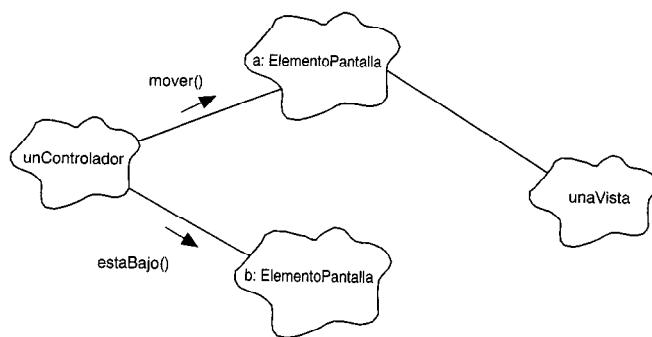


Figura 3.2. Enlaces.

Enlaces

Semántica. El término *enlace* deriva de Rumbaugh, que lo define como «una conexión física o conceptual entre objetos» [16]. Un objeto colabora con otros objetos a través de sus enlaces con éstos. Dicho de otro modo, un enlace denota la asociación específica por la cual un objeto (el cliente) utiliza los servicios de otro objeto (el suministrador o servidor), o a través de la cual un objeto puede comunicarse con otro.

La Figura 3.2 ilustra varios enlaces diferentes. En esta Figura, una línea entre dos iconos de objeto representa la existencia de un enlace entre ambos y significa que pueden pasar mensajes a través de esta vía. Los mensajes se muestran como líneas dirigidas que representan su dirección, con una etiqueta que nombría al propio mensaje. Por ejemplo, se ve que el objeto *unControlador* tiene enlaces hacia dos instancias de *ElementoPantalla* (los objetos *a* y *b*). Aunque tanto *a* como *b* tienen probablemente enlaces hacia la vista en la que aparecen, se ha elegido remarcar una sola vez tal enlace, desde *a* hasta *unaVista*. Sólo a través de estos enlaces puede un objeto enviar mensajes a otro.

El paso de mensajes entre dos objetos es típicamente unidireccional, aunque ocasionalmente puede ser bidireccional. En el ejemplo, el objeto *unControlador* sólo invoca operaciones sobre los dos objetos de pantalla (para moverlos y averiguar su posición), pero los objetos de pantalla no operan sobre el objeto controlador. Esta separación de intereses es bastante común en sistemas orientados a objetos bien estructurados, como se trata en el Capítulo 5⁸. Nótese también que, aunque el paso de mensajes es iniciado por el cliente (como *unControlador*) y dirigido hacia el servidor (como el objeto *a*), los datos pueden fluir en ambas direcciones a través de un enlace. Por ejemplo, cuando *unControlador* invoca la operación *mover* sobre *a*, los datos fluyen desde el

⁸ De hecho, esta organización de controlador, vista y elemento de pantalla es tan común que podemos identificarla como un patrón de diseño, que puede por tanto reutilizarse. En Smalltalk, esto se llama un *mecanismo MVC*, de modelo/vista/controlador. Como se discute en el capítulo siguiente, un sistema orientado a objetos bien estructurado suele tener muchos patrones identificables de este tipo.

cliente hacia el servidor. Sin embargo, cuando `unControlador` invoca la operación `estaBajo` sobre el objeto `b`, el resultado pasa del servidor al cliente.

Como participante de un enlace, un objeto puede desempeñar uno de tres papeles:

- Actor Un objeto que puede operar sobre otros objetos pero nunca se opera sobre él por parte de otros objetos; en algunos contextos, los términos *objeto activo* y *actor* son equivalentes.
- Servidor Un objeto que nunca opera sobre otros objetos; sólo otros objetos operan sobre él.
- Agente Un objeto que puede operar sobre otros objetos y además otros objetos pueden operar sobre él; un agente se crea normalmente para realizar algún trabajo en nombre de un actor u otro agente

Ciñéndose al contexto de la Figura 3.2, `unControlador` representa un objeto actor, `unaVista` representa un objeto servidor, y `a` representa un agente que lleva a cabo la petición del controlador para dibujar el elemento en la vista.

Ejemplo. En muchos tipos diferentes de procesos industriales, algunas reacciones requieren un gradiente de temperatura, en la que se eleva la temperatura de alguna sustancia, se mantiene a tal temperatura durante un tiempo determinado, y entonces se deja enfriar hasta la temperatura ambiente. Procesos diferentes requieren perfiles diferentes: algunos objetos (como espejos de telescopios) deben enfriarse lentamente, mientras que otros materiales (como el acero) deben ser enfriados con rapidez. Esta abstracción de un gradiente de temperatura tiene un comportamiento lo suficientemente bien definido para justificar la creación de una clase, como la que sigue. Primero, se introduce una definición de tipos cuyos valores representan el tiempo transcurrido en minutos:

```
// Número que denota minutos transcurridos
typedef unsigned int Minuto;
```

Este `typedef` es similar a los de `Dia` y `Hora`, que se introdujeron en el Capítulo 2. A continuación, se proporciona la clase `GradienteTemperatura`, que es conceptualmente una correspondencia tiempo/temperatura:

```
class GradienteTemperatura {
public:
    GradienteTemperatura();
    virtual ~GradienteTemperatura();

    virtual void borrar();
```

```

virtual void ligar(Temperatura, Minuto);

Temperatura temperaturaEn(Minuto);

protected:
...
};

```

En consonancia con el estilo adoptado, se han declarado varias operaciones como virtual, porque se espera que existan subclases de esta clase.

En realidad, el comportamiento de esta abstracción es algo más que una mera correspondencia tiempo/temperatura. Por ejemplo, se podría fijar un gradiente de temperatura que requiere que la temperatura sea de 120° C en el instante 60 (una hora transcurrida en la rampa de temperatura) y de 65° C en el instante 180 (tres horas del proceso), pero entonces se desearía saber cuál debería ser la temperatura en el instante 120. Esto requiere interpolación lineal, que es otra parte del comportamiento que se espera de esta abstracción.

Un comportamiento que no se requiere explícitamente de esta abstracción es el control de un calentador que lleve a cabo un gradiente de temperatura concreta. En vez de eso, se prefiere una mayor separación de intereses, en la que este comportamiento se consigue mediante la colaboración de tres objetos: una instancia de gradiente de temperatura, un calentador y un controlador de temperatura. Por ejemplo, se podría introducir la siguiente clase:

```

class ControladorTemperatura {
public:
    ControladorTemperatura(Posicion);
    ~ControladorTemperatura();

    void procesar(const GradienteTemperatura&);

    Minuto planificar(const GradienteTemperatura&) const;

private:
    ...
};

```

Esta clase usa el tipo definido `Posicion` introducido en el Capítulo 2. Nótese que no se espera que exista ninguna subclase de esta clase, y por eso no se ha hecho ninguna operación virtual.

La operación `procesar` suministra el comportamiento central de esta abstracción; su propósito es ejecutar el gradiente de temperatura dada en el calentador de la posición indicada. Por ejemplo, dadas las declaraciones siguientes:

```

GradienteTemperatura gradienteCreciente;
ControladorTemperatura controladorGradiente(7);

```

podría establecerse entonces un gradiente de temperatura particular, y decir al controlador que efectuase este perfil:

```
gradienteTemperatura.ligar(120, 60);
gradienteTemperatura.ligar(65, 180);

controladorGradiente.procesar(gradienteCreciente);
```

Considérese la relación entre los objetos `gradienteCreciente` y `controladorGradiente`: el objeto `controladorGradiente` es un agente responsable de efectuar un gradiente de temperatura, y así utiliza el objeto `gradienteCreciente` como servidor. Este enlace se manifiesta en el hecho de que el objeto `controladorGradiente` utiliza el objeto `gradienteCreciente` como argumento para una de sus operaciones.

Un comentario al respecto del estilo: a primera vista, puede parecer que se ha ideado una abstracción cuyo único propósito es envolver una descomposición funcional en el seno de una clase para que parezca noble y orientada a objetos. La operación `planificar` sugiere que no es este el caso. Los objetos de la clase `ControladorTemperatura` tienen conocimiento suficiente para determinar cuándo un perfil determinado debería ser planificado, y así se expone esta operación como un comportamiento adicional de la abstracción. En algunos procesos industriales de alta energía (como la fabricación de acero), calentar una sustancia es un evento costoso, y es importante tener en cuenta cualquier calor que subsista de un proceso previo, así como el enfriamiento normal de cualquier calentador desatendido. La operación `planificar` existe para que los clientes puedan solicitar a un objeto `ControladorTemperatura` que determine el siguiente momento óptimo para procesar un gradiente de temperatura concreta.

Visibilidad. Considérense dos objetos, A y B, con un enlace entre ambos. Con el fin de que A envíe un mensaje a B, B debe ser visible para A de algún modo. Durante el análisis de un problema, se pueden ignorar perfectamente los problemas de visibilidad, pero una vez que se comienza a idear implantaciones concretas, hay que considerar la visibilidad a través de los enlaces, porque las decisiones en este punto dictan el ámbito y acceso de los objetos a cada lado del enlace.

En el ejemplo anterior, el objeto `controladorGradiente` tiene visibilidad hacia el objeto `gradienteCreciente`, porque ambos están declarados dentro del mismo ámbito, y `gradienteCreciente` se presenta como un argumento de una operación sobre el objeto `controladorGradiente`. Realmente, ésta es sólo una de las cuatro formas diferentes en que un objeto puede tener visibilidad para otro:

- El objeto servidor es global para el cliente.
- El objeto servidor es un parámetro de alguna operación del cliente.
- El objeto servidor es parte del objeto cliente.

- El objeto servidor es un objeto declarado localmente en alguna operación del cliente.

Cómo un objeto se hace visible a otro es un problema de diseño táctico.

Sincronización. Siempre que un objeto pasa un mensaje a otro a través de un enlace, se dice que los dos objetos están *sincronizados*. Para objetos de una aplicación completamente secuencial, esta sincronización suele realizarse mediante una simple invocación de métodos, como se describe en las notas complementarias. Sin embargo, en presencia de múltiples hilos de control, los objetos requieren un paso de mensajes más sofisticado con el fin de tratar los problemas de exclusión mutua que pueden ocurrir en sistemas concurrentes. Como se describió anteriormente, los objetos activos contienen su propio hilo de control, y así se espera que su semántica esté garantizada en presencia de otros objetos activos. No obstante, cuando un objeto activo tiene un enlace con uno pasivo, hay que elegir uno de tres enfoques para la sincronización:

- | | |
|--------------|---|
| • Secuencial | La semántica del objeto pasivo está garantizada sólo en presencia de un único objeto activo simultáneamente. |
| • Vigilado | La semántica del objeto pasivo está garantizada en presencia de múltiples hilos de control, pero los clientes activos deben colaborar para lograr la exclusión mutua. |
| • Síncrono | La semántica del objeto pasivo está garantizada en presencia de múltiples hilos de control, y el servidor garantiza la exclusión mutua. |

Todos los objetos descritos hasta aquí en este capítulo son secuenciales. En el Capítulo 9 se ilustrará cada una de estas formas con mayor detalle.

Agregación

Semántica. Mientras que los enlaces denotan relaciones igual-a-igual o cliente/servidor, la agregación denota una jerarquía todo/parte, con la capacidad de ir desde el todo (también llamado el *agregado*) hasta sus partes (conocidas también como *atributos*). En este sentido, la agregación es un tipo especializado de asociación. Por ejemplo, como se muestra en la Figura 3.3, el objeto `controladorGradiente` tiene un enlace al objeto `gradienteCreciente` así como un atributo `h` cuya clase es `Calentador`. El objeto `controladorGradiente` es así el todo, y `h` es una de sus partes. En otras palabras, `h` es una parte del estado del objeto `controladorGradiente`. Dado el objeto `controladorGradiente`, es posible encontrar su calentador correspondiente `h`. Dado un objeto como `h`, es posible llegar al objeto que lo encierra (también llamado su *contenedor*) si y sólo si este conocimiento es parte del estado de `h`.

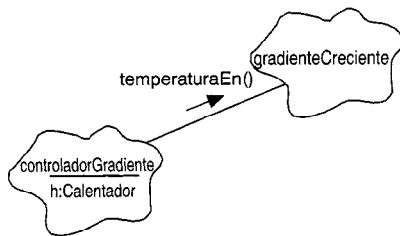


Figura 3.3. Agregación.

La agregación puede o no denotar contención física. Por ejemplo, un aeroplano se compone de alas, motores, tren de aterrizaje, etc.: es un caso de contención física. Por contra, la relación entre un accionista y sus acciones es una relación de agregación que no requiere contención física. El accionista únicamente posee acciones, pero las acciones no son de ninguna manera parte física del accionista. Antes bien, esta relación todo/parte es más conceptual y por tanto menos directa que la agregación física de las partes que forman un aeroplano.

Existen claros pros y contras entre los enlaces y la agregación. La agregación es a veces mejor porque encapsula partes y secretos del todo. A veces son mejores los enlaces porque permiten acoplamientos más débiles entre los objetos. Las decisiones de ingeniería inteligentes requieren sopesar cuidadosamente ambos factores.

Por implicación, un objeto que es atributo de otro tiene un enlace a su agregado. A través de este enlace, el agregado puede enviar mensajes a sus partes.

Ejemplo. Para continuar con la declaración de la clase `ControladorTemperatura`, podría completarse su parte private como sigue:

```
Calentador h;
```

Esto declara a `h` como una parte de cada instancia de `ControladorTemperatura`. De acuerdo con la declaración de la clase `Calentador` realizada en el capítulo anterior, hay que crear correctamente este atributo, porque su clase no suministra un constructor por defecto. Así, se podría escribir el constructor para `ControladorTemperatura` como sigue:

```
ControladorTemperatura::ControladorTemperatura(Posicion p)
: h(p) {}
```

3.3. La naturaleza de una clase

Qué es y qué no es una clase

Los conceptos de clase y objeto están estrechamente entrelazados, porque no puede hablarse de un objeto sin atención a su clase. Sin embargo, existen diferencias importantes entre ambos términos. Mientras que un objeto es una entidad concreta que existe en el tiempo y el espacio, una clase representa sólo una abstracción, la «esencia» de un objeto, como si dijésemos. Así, se puede hablar de la clase *Mamífero*, que representa las características comunes a todos los mamíferos. Para identificar a un mamífero particular en esta clase, hay que hablar de «este mamífero» o «aquel mamífero».

En términos corrientes, se puede definir una clase como «un grupo, conjunto o tipo marcado por atributos comunes o un atributo común; una división, distinción o clasificación de grupos basada en la calidad, grado de competencia o condición» [17]⁹. En el contexto del análisis y diseño orientados a objetos, se define una clase como sigue:

Una clase es un conjunto de objetos que comparten una estructura común y un comportamiento común.

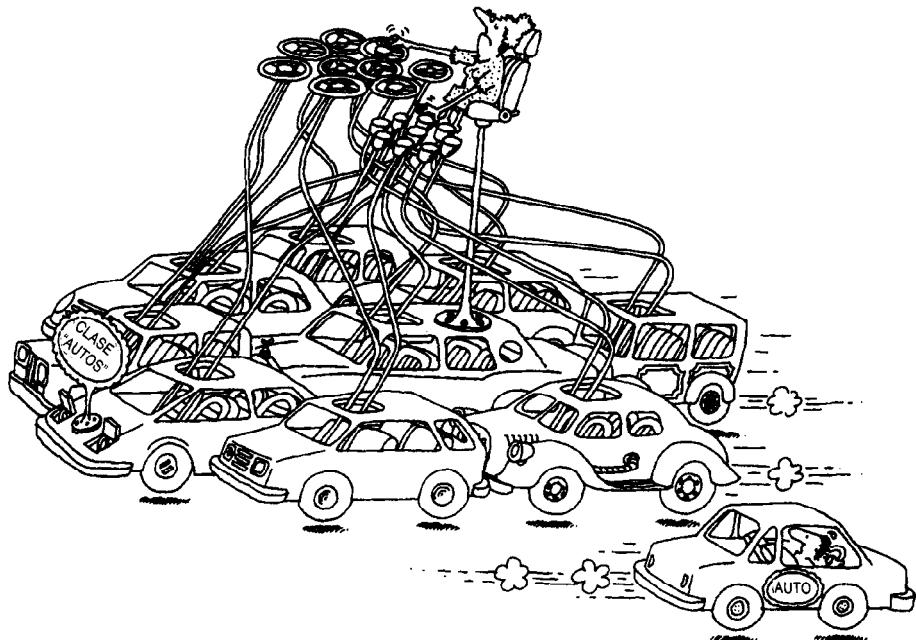
Un solo objeto no es más que una instancia de una clase.

¿Qué no es una clase? Un objeto no es una clase, aunque, curiosamente, como se describirá después, una clase puede ser un objeto. Los objetos que no comparten estructura y comportamiento similares no pueden agruparse en una clase porque, por definición, no están relacionados entre sí a no ser por su naturaleza general como objetos.

Es importante notar que la clase —tal como la define la mayoría de los lenguajes de programación— es un vehículo necesario pero no suficiente para la descomposición. A veces las abstracciones son tan complejas que no pueden expresarse convenientemente en términos de una sola declaración de clase. Por ejemplo, a un nivel de abstracción lo bastante alto, un marco de referencia de un IGU, una base de datos, o un sistema completo de inventario son conceptualmente objetos individuales, ninguno de los cuales puede expresarse como una sola clase¹⁰. En lugar de eso, es mucho mejor capturar esas abstracciones como una agrupación de clases cuyas instancias colaboran para proporcionar el comportamiento y estructura deseados. Stroustrup llama a tal agrupamiento un *componente* [18]. Por razones que se explicarán en el Capítulo 5, aquí se llama a tales grupos una *categoría de clases*.

⁹ Con permiso de *Webster's Third New International Dictionary* (C) 1986 por Merriam-Webster Inc., editor de los diccionarios Merriam-Webster (R).

¹⁰ Uno puede verse tentado a expresar tales abstracciones en una sola clase, pero la granularidad de reutilización y cambio sería nefasta. Tener un interfaz grande es mala práctica, porque la mayoría de los clientes querrán referenciar sólo un pequeño subconjunto de los servicios proporcionados. Es más, el cambio de una parte de un interfaz enorme deja obsoletos a todos los clientes, incluso a los que no tienen que ver con las partes que cambiaron. La anidación de clases no elimina estos problemas; sólo los aplaza.



Una clase representa un conjunto de objetos que comparten una estructura común y un comportamiento común.

Interfaz e implementación

Meyer [19] y Snyder [20] han sugerido que la programación es en gran medida un asunto de «contratos»: las diversas funciones de un problema mayor se descomponen en problemas más pequeños mediante subcontratas a diferentes elementos del diseño. En ningún sitio es más evidente esta idea que en el diseño de clases.

Mientras que un objeto individual es una entidad concreta que desempeña algún papel en el sistema global, la clase captura la estructura y comportamiento comunes a todos los objetos relacionados. Así, una clase sirve como una especie de contrato que vincula a una abstracción y todos sus clientes. Capturando estas decisiones en el interfaz de una clase, un lenguaje con comprobación estricta de tipos puede detectar violaciones de este contrato durante la compilación.

Esta visión de la programación como un contrato lleva a distinguir entre la visión externa y la visión interna de una clase. El *interfaz* de una clase proporciona su visión externa y por tanto enfatiza la abstracción a la vez que oculta su estructura y los secretos de su comportamiento. Este interfaz se compone principalmente de las declaraciones de todas las operaciones aplicables a instancias de esta clase, pero también puede incluir la declaración de otras clases, constan-

tes, variables y excepciones, según se necesiten para completar la abstracción. Por contraste, la *implementación* de una clase es su visión interna, que engloba los secretos de su comportamiento. La implementación de una clase se compone principalmente de la implementación de todas las operaciones definidas en el interfaz de la misma.

Se puede dividir además el interfaz de una clase en tres partes:

- **Public (*pública*)** Una declaración accesible a todos los clientes.
- **Protected** Una declaración accesible sólo a la propia clase, sus subclases, y sus clases amigas (*friends*).
- **Private (*privada*)** Una declaración accesible sólo a la propia clase y sus clases amigas.

Los distintos lenguajes de programación ofrecen diversas mezclas de partes public, protected y private, entre las que los desarrolladores pueden elegir para establecer derechos específicos de acceso para cada parte del interfaz de una clase y de este modo ejercer un control sobre qué pueden ver los clientes y qué no pueden ver.

En particular, C++ permite a los desarrolladores hacer distinciones explícitas entre estas tres partes distintas¹¹. El mecanismo de «amistad» de C++ permite a una clase distinguir ciertas clases privilegiadas a las que se otorga el derecho de ver las partes protected y private de otra. La amistad rompe el encapsulamiento de una clase, y así, al igual que en la vida real, hay que elegirla cuidadosamente. En contraste, Ada permite declaraciones public o private, pero no protected. En Smalltalk, todas las variables de instancia son private, y todos los métodos son public. En Object Pascal, tanto los campos como las operaciones son public y, por tanto, no están encapsulados. En CLOS, las funciones genéricas son public y las ranuras (*slots*) deben hacerse private, aunque su acceso puede romperse mediante la función `slot-value`.

El estado de un objeto debe tener alguna representación en su clase correspondiente, y por eso se expresa típicamente como declaraciones de constantes y variables situadas en la parte private o protected del interfaz de una clase. De este modo, se encapsula la representación común a todas las instancias de una clase, y los cambios en esta representación no afectan funcionalmente a ningún cliente externo.

El lector cuidadoso puede preguntarse por qué la representación de un objeto es parte del interfaz de una clase (aunque sea una parte no pública) y no de su implementación. Por razones prácticas, hacer lo contrario requeriría o bien hardware orientado a objetos o bien una tecnología de compiladores muy sofisticada. Específicamente, cuando un compilador procesa una declaración de un objeto como la siguiente en C++:

```
ElementoPantalla elementol;
```

¹¹ La `struct` de C++ es un caso especial, en el sentido de que una `struct` es un tipo de clase con todos sus elementos `public`.

debe saber cuánta memoria hay que asignar al objeto `elemento1`. Si se hubiera definido la representación del objeto en la implementación de la clase, habría que completar la implementación de la clase antes de poder usar ningún cliente, frustrando así el verdadero propósito de la separación entre las visiones externa e interna de la clase.

Las constantes y variables que forman la representación de una clase se conocen bajo varias denominaciones, dependiendo del lenguaje concreto que se utilice. Por ejemplo, Smalltalk usa el término *variable de instancia*, Object Pascal usa el término *campo* (field), C++ usa el término *objeto miembro* y CLOS usa el término *ranura* (slot). Se utilizarán estos términos indistintamente para denotar las partes de una clase que sirven como representación del estado de su instancia.

Ciclo de vida de las clases

Se puede llegar a la comprensión del comportamiento de una clase simple con sólo comprender la semántica de sus distintas operaciones públicas de forma aislada. Sin embargo, el comportamiento de clases más interesantes (como el movimiento de una instancia de la clase `ElementoPantalla`, o la planificación de una instancia de la clase `ControladorTemperatura`) implica la interacción de sus diversas operaciones a lo largo del tiempo de vida de cada una de sus instancias. Como se describió antes en este capítulo, las instancias de tales clases actúan como pequeñas máquinas y, ya que todas esas instancias incorporan el mismo comportamiento, se puede utilizar la clase para capturar esta semántica común de orden respecto al tiempo y los eventos. Como se discute en el Capítulo 5, puede describirse tal comportamiento dinámico para ciertas clases interesantes mediante el uso de máquinas de estados finitos.

3.4. Relaciones entre clases

Tipos de relaciones

Considérense por un momento las analogías y diferencias entre las siguientes clases de objetos: flores, margaritas, rosas rojas, rosas amarillas, pétalos y mariquitas. Pueden hacerse las observaciones siguientes:

- Una margarita es un tipo de flor.
- Una rosa es un tipo (distinto) de flor.
- Las rosas rojas y las rosas amarillas son tipos de rosas.
- Un pétalo es una parte de ambos tipos de flores.
- Las mariquitas se comen a ciertas plagas como los pulgones, que pueden infectar ciertos tipos de flores.

Partiendo de este simple ejemplo se concluye que las clases, al igual que los objetos, no existen aisladamente. Antes bien, para un dominio de problema específico, las abstracciones clave suelen estar relacionadas por vías muy diversas e interesantes, formando la estructura de clases del diseño [21].

Se establecen relaciones entre dos clases por una de dos razones. Primero, una relación entre clases podría indicar algún tipo de compartición. Por ejemplo, las margaritas y las rosas son tipos de flores, lo que quiere decir que ambas tienen pétalos con colores llamativos, ambas emiten una fragancia, etc. Segundo, una relación entre clases podría indicar algún tipo de conexión semántica. Así, se dice que las rosas rojas y las rosas amarillas se parecen más que las margaritas y las rosas, y las margaritas y las rosas se relacionan más estrechamente que los pétalos y las flores. Análogamente, existe una conexión simbiótica entre las mariquitas y las flores: las mariquitas protegen a las flores de ciertas plagas, que a su vez sirven de fuente de alimento para la mariquita.

En total, existen tres tipos básicos de relaciones entre clases [22]. La primera es la generalización/especialización, que denota una relación «*es un*» (*is a*). Por ejemplo, una rosa es un tipo de flor, lo que quiere decir que una rosa es una subclase especializada de una clase más general, la de las flores. La segunda es la relación **todo/parte** (*whole/part*), que denota una relación «*parte de*» (*part of*). Así, un pétalo no es un tipo de flor; es una parte de una flor. La tercera es la asociación, que denota alguna dependencia semántica entre clases de otro modo independientes, como entre las mariquitas y las flores. Un ejemplo más: las rosas y las velas son clases claramente independientes, pero ambas representan cosas que podrían utilizarse para decorar la mesa de una cena.

En los lenguajes de programación han evolucionado varios enfoques comunes para plasmar relaciones de generalización/especialización, todo/parte y asociación. Específicamente, la mayoría de los lenguajes orientados a objetos ofrecen soporte directo para alguna combinación de las siguientes relaciones:

- Asociación
- Herencia
- Agregación
- Uso
- **Instanciación** (creación de instancias o ejemplares)
- Metaclasa

Un enfoque alternativo para la herencia involucra un mecanismo lingüístico llamado *delegación*, en el que los objetos se consideran prototipos (también llamados *ejemplares*) que delegan su comportamiento en objetos relacionados, eliminando así la necesidad de clases [23].

De estos seis tipos diferentes de relaciones entre clases, las asociaciones son el más general, pero también el de mayor debilidad semántica. Como se discutirá más adelante en el Capítulo 6, la identificación de asociaciones entre clases es frecuentemente una actividad de análisis y de diseño inicial, momento en el cual se comienza a descubrir las dependencias generales entre las abstracciones. A medida que se continúa el diseño y la implementación, se refinarán a me-

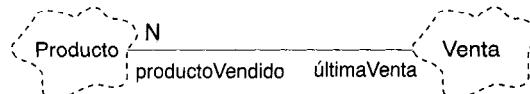


Figura 3.4. Asociación.

nudo estas asociaciones débiles orientándolas hacia una de las otras relaciones de clase más concretas.

La herencia es quizás la más interesante, semánticamente hablando, de estas relaciones concretas, y existe para expresar relaciones de generalización/especialización. Según nuestra experiencia, sin embargo, la herencia es un medio insuficiente para expresar todas las ricas relaciones que pueden darse entre las abstracciones clave en un dominio de problema dado. Se necesitan también relaciones de agregación, que suministran las relaciones todo/parte que se manifiestan en las instancias de las clases. Además, son necesarias las relaciones de uso, que establecen los enlaces entre las instancias de las clases. Para lenguajes como Ada, C++ y Eiffel, se necesitan también las relaciones de instanciación que, al igual que la herencia, soportan un tipo de generalización, aunque de forma completamente diferente. Las relaciones de metaclasificación son bastante distintas y sólo las soportan explícitamente lenguajes como Smalltalk y CLOS. Básicamente, una metaclasificación es la clase de una clase, un concepto que permite tratar a las clases como objetos.

Asociación

Ejemplo. En un sistema automatizado para un punto de venta al por menor, dos de las abstracciones clave incluyen productos y ventas. Como se muestra en la Figura 3.4, se puede mostrar una asociación simple entre estas dos clases: la clase `Producto` denota los productos que se venden como parte de una venta, y la clase `Venta` denota la transacción por la cual varios productos acaban de venderse. Por implicación, esta asociación sugiere una relación bidireccional: dada una instancia de `Producto`, deberíamos ser capaces de encontrar el objeto que denota su venta, y, dada una instancia de `Venta`, deberíamos ser capaces de localizar todos los productos vendidos en esa transacción.

Puede capturarse esta semántica en C++ utilizando lo que Rumbaugh llama punteros escondidos u ocultos (*buried pointers*) [24]. Por ejemplo, considérese la declaración muy resumida de estas dos clases:

```

class Producto;
class Venta;

class Producto {
public:
    ...
  
```

```

protected:
    Venta* ultimaVenta;
};

class Venta {
public:
    ...
protected:
    Producto** productoVendido;
};

```

Aquí se muestra una asociación uno-a-muchos: cada instancia de `Producto` puede tener un puntero a su última venta, y cada instancia de `Venta` puede tener una colección de punteros que denota los productos vendidos.

Dependencias semánticas. Como sugiere este ejemplo, una asociación sólo denota una dependencia semántica y no establece la dirección de esta dependencia (a menos que se diga lo contrario, una asociación implica relación bidireccional, como en el ejemplo) ni establece la forma exacta en que una clase se relaciona con otra (sólo puede denotarse esta semántica nombrando el papel que desempeña cada clase en relación con la otra). Sin embargo, esta semántica es suficiente durante el análisis de un problema, momento en el cual sólo es necesario identificar esas dependencias. Mediante la creación de asociaciones, se llega a plasmar y quiénes son los participantes en una relación semántica, sus papeles y, como se verá, su cardinalidad.

Cardinalidad. El ejemplo ha introducido una asociación uno-a-muchos, lo que significa que para cada instancia de la clase `Venta` existen cero o más instancias de la clase `Producto`, y por cada producto, existe exactamente una venta. Esta multiplicidad denota la *cardinalidad* de la asociación. En la práctica, existen tres tipos habituales de cardinalidad en una asociación:

- Uno a uno.
- Uno a muchos.
- Muchos a muchos.

Una relación uno a uno denota una asociación muy estrecha. Por ejemplo, en las operaciones de venta con tarjeta, se encontraría una relación uno a uno entre la clase `Venta` y la clase `TransaccionTarjetaCredito`: cada venta se corresponde exactamente con una transacción de tarjeta de crédito, y cada transacción se corresponde con una venta. Las relaciones muchos a muchos también son habituales. Por ejemplo, cada instancia de la clase `Cliente` podría iniciar una transacción con una instancia de la clase `Vendedor`, y cada uno de esos vendedores podría interactuar con muchos clientes distintos. Como se verá más adelante en el Capítulo 5, existen variaciones sobre estas tres formas básicas de cardinalidad.

Herencia

Ejemplos. Cuando se lanzan sondas espaciales, remiten informes a las estaciones terrestres con datos acerca del estado de subsistemas importantes (como energía eléctrica y sistemas de propulsión) y diferentes sensores (como sensores de radiación, espectrómetros de masas, cámaras, detectores de colisión de micrometeoritos, etc.). Globalmente, esta información retransmitida recibe el nombre de *datos de telemetría*. Los datos de telemetría se transmiten normalmente como un flujo de bits consistente en una cabecera, que incluye una marca de la hora y algunas claves que identifican el tipo de información que sigue, más varias tramas de datos procesados de los diferentes subsistemas y sensores. Puesto que esto parece ser una clara agregación de diversos tipos de datos, podríamos vernos tentados a definir un tipo de registro para cada tipo de datos de telemetría. Por ejemplo, en C++ se podría escribir:

```
class Hora...

struct DatosElectricos {
    Hora marcaHora;
    int id;
    float tensionCelulaCombustible1, tensionCelulaCombustible2;
    float corrienteCelulaCombustible1, corrienteCelulaCombustible2;
    float energiaActual;
};
```

Existen varios problemas con esta declaración. Primero, la representación de `DatosElectricos` no está encapsulada en absoluto. Así, no hay nada que evite que un cliente cambie el valor de un dato importante como `marcaHora` o `energiaActual` (que es un atributo derivado, directamente proporcional a la tensión y corriente actuales extraídas de ambas células de combustible). Más aún, la representación de esta estructura está expuesta, así que si se cambiase la representación (por ejemplo, añadiendo nuevos elementos o cambiando la alineación de bits de los que existen), todos los clientes serían afectados. Como mínimo, habría que recompilar con toda seguridad cualquier referencia a esta estructura. Más importante es que tales cambios podrían violar las suposiciones que los clientes habían hecho sobre esta representación expuesta y causar una ruptura en la lógica del programa. Además, esta estructura es enormemente carente de significado: se puede aplicar una serie de operaciones a las instancias de esta estructura como un todo (tales como transmitir los datos, o calcular una suma de comprobación para detectar errores durante la transmisión), pero no existe forma de asociar directamente esas operaciones con esta estructura. Finalmente, supóngase que el análisis de los requisitos del sistema revela la necesidad de varios cientos de tipos diferentes de datos de telemetría, incluyendo otros datos eléctricos que abarcaban la información precedente y también incluían lecturas de la tensión en varios puntos de prueba extendidos por el sis-



Una subclase puede heredar la estructura y comportamiento de su superclase.

tema. Se vería que la declaración de estas estructuras adicionales crearía una considerable cantidad de redundancia, en términos tanto de estructuras repetidas como de funciones comunes.

Una forma ligeramente mejor de capturar las decisiones sería declarar una clase para cada tipo de datos de telemetría. De este modo, podría ocultarse la representación de cada clase y asociar su comportamiento con sus datos. Aun así, este enfoque no soluciona el problema de la redundancia.

Una solución mucho mejor, sin embargo, es capturar las decisiones construyendo una jerarquía de clases, en la que las clases especializadas heredan la estructura y comportamiento definidos por clases más generalizadas. Por ejemplo:

```
class DatosTelemetria {
public:

    DatosTelemetria();
    virtual ~DatosTelemetria();

    virtual void transmitir();

    Hora horaActual() const;
```

```

protected:
    int id;
    Hora marcaHora;
};

```

Esto declara una clase con un constructor y un destructor virtual (lo que significa que se espera que haya subclases), así como las funciones `transmitir` y `horaActual`, ambas visibles para todos los clientes. Los objetos miembro `protected` `id` y `marcaHora` están algo más encapsulados, y así son accesibles solamente para la propia clase y sus subclases. Nótese que se ha declarado la función `horaActual` como un selector `public`, que posibilita a un cliente acceder a `marcaHora`, pero no cambiarlo.

A continuación, se va a reescribir la declaración de la clase `DatosElectricos`:

```

class DatosElectricos : public DatosTelemetria {
public:

    DatosElectricos(float t1, float t2, float c1, float c2);
    virtual ~DatosElectricos();

    virtual void transmitir();

    float energiaActual() const;

protected:
    float tensionCelulaCombustible1, tensionCelulaCombustible2;
    float corrienteCelulaCombustible1, corrienteCelulaCombustible2;
};

```

Esta clase hereda la estructura y comportamiento de la clase `DatosTelemetria`, pero añade cosas a su estructura (los cuatro nuevos objetos miembro `protected`), redefine su comportamiento (la función `transmitir`) y le añade cosas (la función `energiaActual`).

Herencia simple. Dicho sencillamente, la herencia es una relación entre clases en la que una clase comparte la estructura y/o el comportamiento definidos en una (*herencia simple*) o más clases (*herencia múltiple*). La clase de la que otras heredan se denomina *superclase*. En el ejemplo, `DatosTelemetria` es una superclase de `DatosElectricos`. Análogamente, la clase que hereda de otra o más clases se denomina *subclase*; `DatosElectricos` es una subclase de `DatosTelemetria`. La herencia define, por tanto, una jerarquía «de tipos» entre clases, en la que una subclase hereda de una o más superclases. Esta es de hecho la piedra de toque para la herencia. Dadas las clases A y B, si A no «es un» tipo de B, entonces A no debería ser una subclase de B. En este sentido, `DatosElectricos` es un tipo especializado de la clase `DatosTelemetria`, más gene-

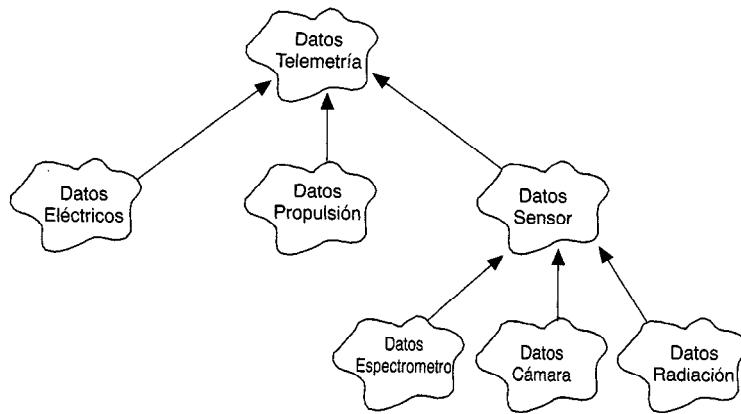


Figura 3.5. Herencia simple.

ralizada. La capacidad de un lenguaje para soportar o no este tipo de herencia distingue a los lenguajes de programación orientados a objetos de los lenguajes basados en objetos.

Una subclase habitualmente aumenta o restringe la estructura y comportamiento existentes en sus superclases. Una subclase que aumenta sus superclases se dice que utiliza herencia por extensión. Por ejemplo, la subclase `ColaVigilada` podría extender el comportamiento de su superclase `Cola` proporcionando operaciones extra que hacen que las instancias de esta clase sean seguras en presencia de múltiples hilos de control. En contraste, una subclase que restringe el comportamiento de sus superclases se dice que usa herencia por restricción. Por ejemplo, la subclase `ElementoPantallaNoSelectable` podría restringir el comportamiento de su superclase, `ElementoPantalla`, prohibiendo a los clientes la selección de sus instancias en una vista. En la práctica, no siempre está tan claro cuándo una subclase aumenta o restringe a su superclase; de hecho, es habitual que las subclases hagan las dos cosas.

La Figura 3.5 ilustra las relaciones de herencia simple que se derivan de la superclase `DatosTelemetria`. Cada línea dirigida denota una relación «es un». Por ejemplo, `DatosCamara` «es un» tipo de `DatosSensor`, que a su vez «es un» tipo de `DatosTelemetria`. Es igual que la jerarquía que se encuentra en una red semántica, una herramienta que utilizan a menudo los investigadores en ciencia cognitiva e inteligencia artificial para organizar el conocimiento acerca del mundo [25]. Realmente, como se trata después en el Capítulo 4, diseñar una jerarquía de herencias conveniente entre abstracciones es en gran medida una cuestión de clasificación inteligente.

Se espera que algunas de las clases de la Figura 3.5 tengan instancias y que otras no las tengan. Por ejemplo, se espera tener instancias de cada una de las clases más especializadas (también llamadas *clases hoja* o *clases concretas*), tales como `DatosElectricos` y `DatosEspectrometro`. Sin embargo, probablemente

no haya ninguna instancia de las clases intermedias, más generales, como `DatosSensor` o incluso `DatosTelemetria`. Las clases sin instancias se llaman *clases abstractas*. Una clase abstracta se redacta con la idea de que las subclases añadan cosas a su estructura y comportamiento, usualmente completando la implementación de sus métodos (habitualmente) incompletos. De hecho, en Smalltalk, un desarrollador puede forzar a una subclase a redefinir el método introducido por una clase abstracta utilizando el método `subclassResponsibility` para implantar un cuerpo para el método de la clase abstracta. Si la subclase no lo redefine, la invocación del método tiene como resultado un error de ejecución. C++ permite análogamente al desarrollador establecer que un método de una clase abstracta no pueda ser invocado directamente inicializando su declaración a cero. Tal método se llama *función virtual pura* (o *pure virtual function*), y el lenguaje prohíbe la creación de instancias cuyas clases exporten tales funciones.

La clase más generalizada en una estructura de clases se llama la *clase base*. La mayoría de las aplicaciones tienen muchas de tales clases base, que representan las categorías más generalizadas de abstracciones en el dominio que se trata. De hecho, especialmente en C++, las arquitecturas orientadas a objetos bien estructuradas suelen tener bosques de árboles de herencias, en vez de una sola trama de herencias de raíces muy profundas. Sin embargo, algunos lenguajes requieren una clase base en la cima, que sirve como la clase última de todas las clases. En Smalltalk, esta clase se denomina `Object`.

Una clase cualquiera tiene típicamente dos tipos de clientes [26]:

- Instancias
- Subclases

Frecuentemente resulta de utilidad definir interfaces distintos para estos dos tipos de clientes [27]. En particular, se desea exponer a los clientes instancia sólo los comportamientos visibles exteriormente, pero se necesita exponer las funciones de asistencia y las representaciones solamente a los clientes subclase. Ésta es precisamente la motivación para las partes `public`, `protected` y `private` de una definición de clase en C++: un diseñador puede elegir qué miembros son accesibles a las instancias, a las subclases o a ambos clientes. Como se mencionó anteriormente, en Smalltalk el desarrollador tiene menos control sobre el acceso: las variables instancia son visibles a las subclases, pero no a las instancias, y todos los métodos son visibles tanto a las instancias como a las subclases (se puede marcar un método como `private`, pero esta ocultación no es promovida por el lenguaje).

Existe una auténtica tensión entre la herencia y el encapsulamiento. En un alto grado, el uso de la herencia expone algunos de los secretos de una clase heredada. En la práctica, esto implica que, para comprender el significado de una clase particular, muchas veces hay que estudiar todas sus superclases, a veces incluyendo sus vistas internas.

La herencia significa que las subclases heredan la estructura de su superclase. Así, en el pasado ejemplo, las instancias de la clase `DatosElectricos` in-

cluyen los objetos miembro de la superclase (tales como `id` y `marcaHora`), así como los de las clases más especializadas (como `tensionCelulaCombustible1`, `tensionCelulaCombustible2`, `corrienteCelulaCombustible1` y `corrienteCelulaCombustible2`)¹².

Las subclases también heredan el comportamiento de sus superclases. Así, puede actuar sobre las instancias de la clase `DatosElectricos` con las operaciones `horaActual` (heredada de su superclase), `energiaActual` (definida en la propia clase), y `transmitir` (redefinida en la subclase). La mayoría de los lenguajes de programación orientados a objetos permiten que los métodos de una superclase sean redefinidos y que se añadan métodos nuevos. En Smalltalk, por ejemplo, cualquier método de una superclase puede redefinirse en una subclase. En C++, el desarrollador tiene un poco más de control. Las funciones miembro que se declaran como *virtual* (como la función `transmitir`) pueden redefinirse en una subclase; los miembros declarados de otro modo (por defecto) no pueden redefinirse (como la función `horaActual`).

Polimorfismo simple. Para la clase `DatosTelemetria`, podría implantarse la función miembro `transmitir` como sigue:

```
void DatosTelemetria::transmitir()
{
    // transmitir el id
    // transmitir la marcaHora
}
```

Se podría implantar la misma función miembro para la clase `DatosElectricos` como sigue:

```
void DatosElectricos::transmitir()
{
    DatosTelemetria::transmitir();
    // transmitir la tensión
    // transmitir la corriente
}
```

En esta implantación, se invoca primero la función correspondiente de la superclase (utilizando el nombre calificado `DatosTelemetria::transmitir`), que transmite los datos `id` y `marcaHora`, y a continuación se transmiten los datos particulares de la subclase `DatosElectricos`.

Supóngase que se tiene una instancia de cada una de esas dos clases:

```
DatosTelemetria telemetria;
DatosElectricos electricos(5.0, -5.0, 3.0, 7.0);
```

¹² Unos pocos lenguajes orientados a objetos, en su mayoría experimentales, permiten a una subclase reducir la estructura de su superclase.

Ahora, dada la siguiente función no miembro,

```
void transmitirDatosRecientes(DatosTelemetria& d, const Hora& h)
{
    if (d.horaActual() >= h)
        d.transmitir();
}
```

¿qué pasa cuando se ejecutan las dos sentencias siguientes?

```
transmitirDatosRecientes(telemetria, Hora(60));
transmitirDatosRecientes(electricos, Hora(120));
```

En la primera sentencia, se transmite una serie de bits que consta solamente de un *id* y una *marcaHora*. En la segunda sentencia, se transmite una serie de bits que consta de un *id*, una *marcaHora* y otros cuatro valores en coma flotante. ¿Cómo es esto? En última instancia, la implantación de la función *transmitirDatosRecientes* simplemente ejecuta la sentencia *d.transmitir()*, que no distingue explícitamente la clase de *d*.

La respuesta es que este comportamiento es debido al polimorfismo. Básicamente, el *polimorfismo* es un concepto de teoría de tipos en el que un nombre (como el parámetro *d*) puede denotar instancias de muchas clases diferentes en tanto en cuanto estén relacionadas por alguna superclase común. Cualquier objeto denotado por este nombre es, por tanto, capaz de responder a algún conjunto común de operaciones de diversas formas.

Como ponen de relieve Cardelli y Wegner, «los lenguajes convencionales con tipos, como Pascal, se basan en la idea de que las funciones y los procedimientos, y, por tanto, los operandos, tienen un único tipo. Tales lenguajes se dice que son monomórficos, en el sentido de que todo valor y variable puede interpretarse que tiene un tipo y sólo uno. Los lenguajes de programación monomórficos pueden contrastarse con los lenguajes polimórficos en los que algunos valores y variables pueden tener más de un tipo» [28]. El concepto de polimorfismo fue descrito en primer lugar por Strachey [29], que habló de un polimorfismo *ad hoc*, por el cual los símbolos como «+» podrían definirse para significar cosas distintas. Hoy en día, en los lenguajes de programación modernos, se denomina a este concepto *sobrecarga*. Por ejemplo, en C++, pueden declararse funciones que tienen el mismo nombre, ya que sus invocaciones pueden distinguirse por sus signaturas, que consisten en el número y tipos de sus argumentos (en C++, a diferencia del Ada, el tipo del valor que devuelve una función no se considera en la resolución de la sobrecarga). Strachey habló también de *polimorfismo paramétrico*, que hoy se denomina, sin más, *polimorfismo*.

Sin polimorfismo, el desarrollador acaba por escribir código que consiste en grandes sentencias *switch* o *case*¹³. Por ejemplo, en un lenguaje de programa-

¹³ Ésta es, de hecho, la piedra de toque para el polimorfismo. La existencia de una sentencia *switch* que selecciona una acción sobre la base del tipo de un objeto es frecuentemente un signo de advertencia de que el desarrollador ha fracasado en la aplicación efectiva del comportamiento polimórfico.

ción no orientado a objetos como Pascal, no se puede crear una jerarquía de clases para los diversos tipos de datos de telemetría; en vez de eso, hay que definir un solo registro variante monolítico que abarca las propiedades asociadas con todos los tipos de datos. Para distinguir una variante de otra, hay que examinar la etiqueta asociada con el registro. Así, un procedimiento equivalente para `transmitirDatosRecientes` podría escribirse en Pascal como sigue:

```

const
  Electrico = 1;
  Propulsion = 2;
  Espectrometro = 3;
...
procedure Transmitir_Datos_Recientes(Los_Datos:Datos;La_Hora:Hora);
begin
  if (Los_Datos.Hora_Actual >= La_Hora) then
    case Los_Datos.Tipo of
      Electrico: Transmitir_Datos_Electricos(Los_Datos);
      Propulsion: Transmitir_Datos_Propulsion(Los_Datos);
      ...
    end;
end;

```

Para añadir otro tipo de datos de telemetría, habría que modificar el registro variante y añadirlo a cualquier sentencia `case` que operase sobre instancias de este registro. Esto es especialmente propenso a errores y, además, añade inestabilidad al diseño.

En presencia de la herencia, no hay necesidad de un tipo monolítico, ya que se puede separar diferentes tipos de abstracciones. Como hacen notar Kaplan y Johnson, «el polimorfismo es más útil cuando existen muchas clases con los mismos protocolos» [30]. Con polimorfismo no son necesarias grandes sentencias `case`, porque cada objeto conoce implícitamente su propio tipo.

La herencia sin polimorfismo es posible, pero ciertamente no es muy útil. Ésta es la situación en el lenguaje Ada, en el que se pueden declarar tipos derivados, pero al ser el lenguaje monomórfico, la operación actual que se llama es conocida siempre en tiempo de compilación.

El polimorfismo y la ligadura tardía van de la mano. En presencia del polimorfismo, la ligadura de un método con un nombre no se determina hasta la ejecución. En C++, el desarrollador puede controlar si una función miembro utiliza ligadura temprana o tardía. Concretamente, si el método se declara como `virtual` se emplea ligadura tardía, y la función se considera polimórfica. Si se omite esta declaración `virtual`, el método utiliza ligadura temprana, y así puede resolverse la referencia en tiempo de compilación. Se explica en las notas complementarias el mecanismo por el que una implantación selecciona un método particular para la ejecución.

Herencia y tipos. Considérese de nuevo la redefinición del miembro `transmitir`:

```

void DatosElectricos::transmitir()
{
    DatosTelemetria::transmitir();
    // transmitir las tensiones
    // transmitir las corrientes
}

```

La mayoría de los lenguajes de programación orientados a objetos permiten a la implantación de un método de una subclase invocar directamente un método definido por alguna superclase. Como muestra este ejemplo, es también bastante habitual para la implantación de un método redefinido el invocar el método del mismo nombre definido por una clase padre. En Smalltalk se puede invocar un método que provenga de la clase inmediatamente superior utilizando la palabra clave `super`; se puede también hacer referencia al objeto que invocó a un método mediante la variable especial `self`. En C++ se puede invocar al método de cualquier antepasado accesible poniendo al nombre de método un prefijo con el nombre de la clase, formando así un *nombre calificado*, y puede hacerse referencia al objeto que invocó a un método mediante cierto puntero declarado implícitamente cuyo nombre es `this`.

En la práctica, un método redefinido suele invocar un método de una superclase ya sea antes o después de llevar a cabo alguna otra acción. De este modo, los métodos de la subclase desempeñan el papel de aumentar el comportamiento definido en la superclase¹⁴.

En la Figura 3.5, todas las subclases son también subtipos de su clase padre. Por ejemplo, las instancias de `DatosElectricos` se consideran subtipos al igual que subclases de `DatosTelemetria`. El hecho de que los tipos corran paralelos a las relaciones de herencia es común a la mayoría de los lenguajes de programación orientados a objetos con comprobación estricta de tipos, incluyendo C++. Puesto que Smalltalk es en gran medida un lenguaje sin tipos, o al menos con tipos débiles, este asunto tiene menos interés.

El paralelismo entre tipos y herencia es deseable cuando se ven las jerarquías de generalización/especialización creadas a través de la herencia como un medio para capturar la conexión semántica entre abstracciones. Una vez más, considérense las declaraciones en C++:

```

DatosTelemetria telemetria;
DatosElectricos electricos(5.0, -5.0, 3.0, 7.0);

```

Invocando un método

En los lenguajes de programación tradicionales, la invocación de un subprograma

¹⁴ En CLOS, estos diferentes papeles para los métodos se hacen explícitos declarando un método con los calificadores `:before` y `:after`, así como `:around`. Un método sin calificador se considera un método *primario* y realiza el trabajo central del comportamiento deseado. Los métodos *before* y *after* aumentan el comportamiento de un método primario; son llamados antes y después del método primario, respectivamente. Los métodos *around* forman un envoltorio alrededor de un método primario, que puede ser invocado en algún lugar dentro del método mediante la función `call-next-method`.

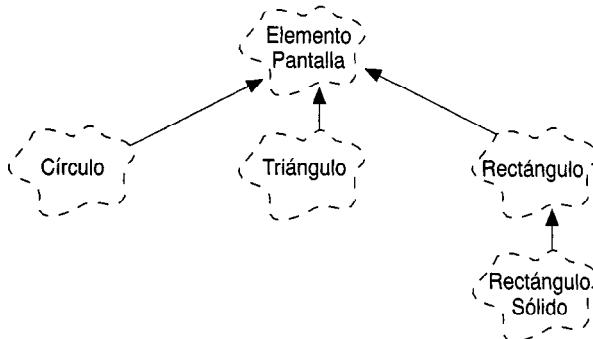


Figura 3.6. Diagrama de clases de ElementoPantalla.

es una actividad completamente estática. En Pascal, por ejemplo, para una sentencia que llama al subprograma *P*, un compilador generará típicamente código que crea una nueva trama en la pila, sitúa los argumentos correctos en la pila y entonces cambia el flujo de control para comenzar a ejecutar el código asociado con *P*. Sin embargo, en lenguajes que soportan alguna forma de polimorfismo, como Smalltalk y C++, la invocación de una operación puede requerir una actividad dinámica, porque la clase del objeto sobre el que se opera puede no conocerse hasta el tiempo de ejecución. Las cosas son aún más interesantes cuando se añade la herencia a la situación. La semántica de invocar una operación en presencia de la herencia sin polimorfismo es en gran medida la misma que para una simple llamada estática a un subprograma, pero en presencia del polimorfismo hay que utilizar una técnica mucho más sofisticada.

Considérese la jerarquía de clases de la Figura 3.6, que muestra la clase base ElementoPantalla junto con tres subclases llamadas Círculo, Triangulo y Rectángulo. Rectángulo tiene también una subclase, llamada RectánguloSólido. En la clase ElementoPantalla, supóngase que se define la variable de instancia elCentro (denotando las coordenadas para el centro del elemento visualizado), junto con las siguientes operaciones como en el ejemplo ya descrito:

- **dibujar** Dibuja el elemento.
- **mover** Mueve el elemento.
- **posicion** Devuelve la posición del elemento.

La operación *posicion* es común a todas las subclases, y por tanto no necesita ser redefinida, pero se espera que las operaciones *dibujar* y *mover* sean redefinidas, ya que sólo las subclases saben cómo dibujarse y moverse a sí mismas.

La clase Círculo debe incluir la variable de instancia elRadio y operaciones apropiadas para fijar y recuperar su valor. Para esta subclase, la operación redefinida *dibujar* dibuja un círculo del radio determinado, centrado en elCentro. Del mismo modo, la clase Rectángulo debe incluir las variables de instancia laAltura y laAnchura, junto con las operaciones apropiadas para fijar y recuperar sus valores. Para esta subclase, la operación *dibujar* dibuja un rectángulo con el ancho y alto dados, de nuevo centrado en elCentro. La subclase RectánguloSólido he-

reda todas las características de la clase `Rectangulo`, pero una vez más redefine el comportamiento de la operación `dibujar`. Específicamente, la implantación de `dibujar` para la clase `RectanguloSolido` llama en primer lugar a `dibujar` como se definió en su superclase `Rectangulo` (para dibujar el borde del rectángulo) y entonces rellena la figura.

Considérese ahora el siguiente fragmento de código:

```
ElementoPantalla* elementos[10];
...
for (unsigned indice = 0; indice < 10; indice++)
    elementos[indice]->dibujar();
```

La invocación de `dibujar` exige comportamiento polimórfico. Aquí se encuentra una matriz heterogénea de elementos, lo que significa que la colección puede tener punteros a objetos de cualquiera de las subclases `ElementoPantalla`. Supóngase ahora que se tiene algún objeto cliente que desea dibujar todos los elementos de la colección, como en el fragmento de código. La idea es iterar a través de la matriz e invocar la operación `dibujar` sobre cada objeto que se encuentra. En esta situación, el compilador no puede generar estáticamente código para invocar a la operación `dibujar` correcta, porque la clase del objeto sobre el que se opera no se conoce hasta el tiempo de ejecución. Veamos cómo se enfrentan a esta situación varios lenguajes orientados a objetos.

Puesto que Smalltalk es un lenguaje sin tipos, la selección de método es completamente dinámica. Cuando el cliente envía el mensaje `dibujar` a un elemento de la lista, esto es lo que ocurre:

- El objeto busca el mensaje en el diccionario de mensajes de su clase.
- Si se encuentra el mensaje, se invoca el código para ese método definido localmente.
- Si el mensaje no se encuentra, la búsqueda del método continúa en la superclase.

Este proceso continúa remontando la jerarquía de superclases hasta que se encuentra el mensaje, o hasta que se alcanza la clase base superior, `Object`, sin encontrar el mensaje. En el último caso, Smalltalk finalmente pasa el mensaje `doesNotUnderstand`, para señalar un error.

La clave de este algoritmo es el diccionario de mensajes, que forma parte de la representación de cada clase y está, por tanto, oculto para el cliente. Este diccionario se crea cuando se crea la clase, y contiene todos los métodos a los que las instancias de esa clase pueden responder. La búsqueda del mensaje consume tiempo; la búsqueda de métodos en Smalltalk lleva alrededor de 1,5 veces el tiempo de una llamada normal a un subprograma. Todas las implantaciones de Smalltalk con calidad de producción optimizan la selección de métodos suministrando un diccionario de mensajes con memoria intermedia, de modo que los mensajes que llegan con frecuencia pueden invocarse rápidamente. La eficacia suele mejorar en un 20%—30% [31].

La operación `dibujar` definida en la subclase `RectanguloSolido` propone un caso especial. Se dijo que esta implantación de `dibujar` llama primero al `dibujar` definido en la superclase `Rectangulo`. En Smalltalk, se especifica un método de superclase utilizando la palabra clave `super`. Entonces, cuando se pasa el mensaje `dibujar` a `super`, Smalltalk utiliza el mismo algoritmo de selección de método, excepto en que la búsqueda comienza en la superclase del objeto y no en la clase.

Estudios de Deutsch sugieren que el polimorfismo no se necesita alrededor del 85% del tiempo, así que el paso de mensajes puede reducirse a menudo a simples llamadas a procedimientos [32]. Duff remarca que, en tales casos, el desarrollador suele hacer suposiciones implícitas que permiten una ligadura temprana de la clase del objeto [33]. Desgraciadamente, los lenguajes sin tipos como Smalltalk no tienen medios convenientes para comunicar esas suposiciones implícitas al compilador.

Lenguajes más estrictos respecto a tipos como C++ dejan al desarrollador establecer esa información. Puesto que se desea evitar la selección de métodos cuando sea posible pero hay que seguir permitiendo la selección polimórfica, la invocación de un método en estos lenguajes se realiza de forma ligeramente distinta que en Smalltalk.

En C++, el desarrollador puede decidir si una operación particular va a utilizar ligadura tardía declarándola *virtual*; todos los demás métodos se ligan tempranamente, y así el compilador puede resolver estáticamente la llamada al método como una simple llamada a subprograma. En el ejemplo, se declaró *dibujar* como función miembro *virtual*, y el método *posicion* como no *virtual*, ya que no necesita ser definido por ninguna subclase. El desarrollador puede también declarar los métodos no virtuales como *inline*, lo que evita la llamada a subprograma, e intercambia espacio por tiempo.

Para manejar funciones miembro virtuales, la mayoría de las implantaciones de C++ usan el concepto de *vtable* o *tabla de métodos virtuales*, que se define para cada objeto que requiera selección polimórfica cuando el objeto se crea (y por tanto cuando se fija la clase del objeto). Esta tabla suele constar de una lista de punteros a funciones virtuales. Por ejemplo, si se crea un objeto de la clase *Rectangulo*, la *vtable* tendrá una entrada para la función virtual *dibujar*, apuntando a la implantación más cercana de *dibujar*. Si, por ejemplo, la clase *ElementoPantalla* incluyese la función virtual *rotar*, que no se redefinió en la clase *Rectangulo*, la entrada de *vtable* para *rotar* apuntaría a la implantación de *Rotar* en la clase *ElementoPantalla*. De este modo, se elimina la búsqueda en tiempo de ejecución: la referencia a una función miembro virtual de un objeto no es más que una referencia indirecta a través del puntero adecuado, que invoca inmediatamente el código correcto sin búsquedas [34].

La implantación de *dibujar* para la clase *RectanguloSolido* también introduce un caso especial en C++. Para hacer que la implantación de este método haga referencia al método *dibujar* de la superclase, C++ requiere el uso del operador de ámbito. Así, hay que escribir:

```
Rectangulo::dibujar();
```

Estudios de Stroustrup sugieren que una llamada a función virtual es aproximadamente igual de eficiente que una llamada a función normal [35]. En presencia de herencia simple, una llamada a función virtual requiere sólo tres o cuatro referencias a memoria más que una llamada a función normal; la herencia múltiple añade sólo alrededor de cinco o seis referencias a memoria.

La selección de método en CLOS es complicada por los métodos *:before*, *:after* y *:around*. La existencia de polimorfismo múltiple también complica las cosas.

La selección de método en CLOS suele utilizar el siguiente algoritmo:

- Determinar los tipos de los argumentos.

- Calcular el conjunto de métodos aplicables.
- Ordenar los métodos de más específico a más general, de acuerdo con la lista de precedencias de la clase del objeto.
- Llamar a todos los métodos :before.
- Llamar al método primario más específico.
- Llamar a todos los métodos :after.
- Devolver el valor del método primario [36].

CLOS también introduce un protocolo de metaobjetos, por el que se puede redefinir el verdadero algoritmo utilizado para la selección genérica (aunque en la práctica suele utilizarse el proceso predefinido). Como sabiamente apuntan Winston y Horn, «el algoritmo de CLOS es complicado, sin embargo, e incluso los programadores geniales de CLOS intentan apañárselas sin pensar en él, igual que los físicos se las apañan con las leyes de la mecánica newtoniana en vez de enfrentarse a la mecánica cuántica» [37].

La siguiente sentencia de asignación es correcta:

```
telemetria = electricos; // eléctricos es subtipo de telemetria
```

Aunque es correcta, esta sentencia es también peligrosa: cualquier estado adicional definido por una instancia de la subclase se ve recortado en la asignación a una instancia de la superclase. En este ejemplo, los cuatro objetos miembro `tensionCelulaCombustible1`, `tensionCelulaCombustible2`, `corrienteCelulaCombustible1` y `corrienteCelulaCombustible2` no se copiarían, porque el objeto denotado por la variable `telemetria` es una instancia de la clase `DatosTelemetria`, que no tiene esos miembros como parte de su estado.

La siguiente sentencia no es correcta:

```
electricos = telemetria; // Incorrecto: telemetria no es un
                           // subtipo de eléctricos
```

Para resumir, la asignación de un objeto `x` a un objeto `y` es posible si el tipo de `x` es el mismo que el tipo de `y` o un subtipo de él.

Lenguajes con comprobación de tipos más estricta permiten la conversión del valor de un objeto de un tipo a otro, pero normalmente sólo si hay alguna relación superclase/subclase entre los dos. Por ejemplo, en C++ se pueden escribir explícitamente operadores de conversión para una clase utilizando lo que se denomina *conversión forzada de tipo** (o *type cast*). Típicamente, como en el ejemplo, se utiliza conversión implícita de tipos para convertir una instancia de una clase más específica al hacer asignaciones a una clase más general. Tales con-

* También denominada moldeado. (*N. del T.*)

versiones se dice que son *seguras respecto al tipo*, lo que significa que se comprueba su corrección semántica en tiempo de compilación. A veces se necesita convertir una variable de una clase más general a una más específica, y por eso hay que escribir un ahormado explícito de tipos. Sin embargo, tales operaciones no son seguras respecto al tipo, porque pueden fallar en tiempo de ejecución si el objeto cuyo tipo se transforma es incompatible con el nuevo tipo¹⁵. Tales conversiones no son en realidad raras (aunque deberían evitarse a menos que hubiese razones de peso), porque el desarrollador frecuentemente conoce los tipos reales de ciertos objetos. Por ejemplo, en ausencia de tipos parametrizados, es práctica común construir clases como conjuntos y bolsas que representan colecciones de objetos, y puesto que se desea permitir colecciones de instancias de clases arbitrarias, se definen típicamente estas clases de colección de forma que operen sobre instancias de alguna clase base (un estilo mucho más seguro que el modismo `void*` utilizado anteriormente para la clase `cola`). Entonces, las operaciones de iteración definidas por tal clase sólo sabrían cómo devolver objetos de esa clase base. Sin embargo, dentro de una aplicación particular, un desarrollador debería colocar en la colección solamente objetos de alguna subclase concreta de esta clase base. Para invocar una operación específica de la clase sobre los objetos visitados durante la iteración, el desarrollador tendría que ahormar explícitamente al tipo esperado cada objeto que se visitase. Una vez más, esta operación fallaría en tiempo de ejecución si apareciese en la colección un objeto de algún tipo inesperado.

Los lenguajes con comprobación de tipos más estricta permiten a una implantación optimizar mejor la *selección* (búsqueda) de métodos, frecuentemente reduciendo el mensaje a una simple llamada a subprograma. Tales optimizaciones no guardan sorpresas si la jerarquía de tipos del lenguaje corre paralela a su jerarquía de clases (como en C++). Sin embargo, existe un lado oscuro en la unificación de estas jerarquías. Específicamente, el cambiar la estructura o comportamiento de alguna superclase puede afectar a la corrección de sus subclases. Como establece Micallef, «si las reglas de los subtipos se basan en herencia, la reimplementación de una clase de forma que su posición en el grafo de herencias cambie puede hacer a los clientes de esa clase incorrectos respecto al tipo, incluso si el interfaz externo de la clase sigue siendo el mismo» [38].

Estos problemas nos llevan a los propios fundamentos de la semántica de la herencia. Como se puso de relieve antes en este capítulo, la herencia puede utilizarse para indicar compartición o para sugerir alguna conexión semántica. Dicho de otro modo por Snyder, «se puede ver la herencia como una decisión privada del diseñador para “reusar” código porque es útil hacerlo; debería ser posible cambiar con facilidad tal decisión. Alternativamente, se puede ver la herencia como la realización de una declaración pública de que los objetos de la clase hija obedecen la semántica de la clase padre, de modo que la clase hija

¹⁵ Algunas extensiones propuestas al C++ para identificación de tipos en tiempo de ejecución ayudarán a mitigar este problema.

simplemente especializa o refina la clase padre» [39]. En lenguajes como Smalltalk y CLOS, estas dos visiones son indistinguibles. Sin embargo, en C++ el desarrollador tiene mayor control sobre las implicaciones de la herencia. Concretamente, si se establece que la superclase de una subclase dada es `public` (como en el ejemplo de la clase `DatosElectricos`) quiere decirse que la subclase es también un subtipo de la superclase, ya que ambas comparten el mismo interfaz (y, por tanto, la misma estructura y comportamiento). Alternativamente, en la declaración de una clase, se puede afirmar que una superclase es `private`, lo que significa que la estructura y comportamiento de la superclase son compartidos, pero la subclase no es un subtipo de la superclase¹⁶. Esto significa que para superclases `private`, los miembros `public` y `protected` de la superclase se convierten en miembros `private` para la subclase, y por tanto inaccesibles a subclases inferiores. Es más, no se forma ninguna relación de subtipos entre la subclase y su superclase `private`, porque las dos clases ya no presentan el mismo interfaz para otros clientes.

Considérese la siguiente declaración de clase:

```
class DatosElectricosInternos : private DatosElectricos {
public:
    DatosElectricosInternos(float t1, float t2, float c1, float
c2);
    virtual ~DatosElectricosInternos();

    DatosElectricos::energiaActual;
};
```

En esta declaración, los métodos como `transmitir` no son visibles a ningún cliente de esta clase, porque `DatosElectricos` se declara como superclase privada. Puesto que `DatosElectricosInternos` no es un subtipo de `DatosElectricos`, esto también quiere decir que no pueden asignarse instancias de `DatosElectricosInternos` a objetos de la superclase, lo que sí puede hacerse con clases que utilizan superclases públicas. Por último, nótese que se ha hecho visible la función miembro `energiaActual` nombrándola explícitamente. Sin esta alusión explícita, se trataría como `private`. Como sería de esperar, las reglas de C++ prohíben hacer a un miembro de una subclase más visible de lo que lo era en su superclase. Así, el objeto miembro `marcaHora`, declarado como miembro `protected` en la clase `DatosTelemetria`, no se podría hacer `public` nombrándolo explícitamente, como se hizo con `energiaActual`.

En lenguajes como Ada puede conseguirse el equivalente de esta distinción utilizando tipos derivados versus subtipos. En concreto, un subtipo de un tipo no define tipos nuevos, sino sólo un subtipo restringido, mientras que un tipo

¹⁶ Puede declararse también una superclase como `protected`, lo que tiene la misma semántica que una superclase `private`, excepto que los miembros `public` y `protected` de la superclase `protected` son accesibles a subclases de niveles inferiores.

derivado define un tipo nuevo e incompatible, que comparte la misma representación que su tipo padre.

Como se discute en una sección posterior, existe una gran tensión entre herencia para reutilización y agregación.

Herencia múltiple. Con herencia simple, cada subclase tiene exactamente una superclase. Sin embargo, como apuntan Vlissides y Linton, aunque la herencia simple es muy útil, «fuerza frecuentemente al programador a derivar de una sola de entre dos clases igualmente atractivas. Esto limita la aplicabilidad de las clases predefinidas, haciendo muchas veces necesario el duplicar código. Por ejemplo, no existe forma de衍生 un gráfico que es a la vez un círculo y una imagen; hay que derivar de uno o del otro y reimplantar la funcionalidad de la clase que se excluyó» [40]. La herencia múltiple la soportan directamente lenguajes como C++ y CLOS y, en un grado limitado, Smalltalk. La necesidad de herencia múltiple en los lenguajes de programación orientados a objetos es aún un tema de gran debate. En nuestra experiencia, se ha encontrado que la herencia múltiple es como un paracaídas: no siempre se necesita, pero cuando así ocurre, uno está verdaderamente feliz de tenerlo a mano.

Considérese por un momento cómo podrían organizarse varios bienes como cuentas de ahorro, bienes inmuebles, acciones y bonos. Las cuentas de ahorros y las cuentas corrientes son los bienes manejados habitualmente por un banco, de este modo podríamos clasificar a ambas como tipos de cuentas bancarias, que pueden convertirse en tipos de activos. Las acciones y los bonos se manejan de forma diferente que las cuentas bancarias, así podemos clasificar las acciones, los bonos, los fondos de inversión, y otros similares como tipos de bienes que pueden convertirse en tipos de activos.

Sin embargo, existen muchas otras vías igualmente satisfactorias para clasificar cuentas de ahorro, bienes inmuebles, acciones y bonos. Por ejemplo, en algunos contextos puede ser útil distinguir elementos asegurables como bienes inmuebles y ciertas cuentas bancarias (que en los Estados Unidos se aseguran hasta ciertos límites por parte de la Federal Depositors Insurance Corporation). También puede ser útil identificar bienes que arrojan un dividendo o interés, como cuentas bancarias, libretas de cheques y ciertas acciones y bonos.

Desgraciadamente, la herencia simple no es lo suficientemente expresiva para capturar esta trama de relaciones, así que contemplar la herencia múltiple¹⁷. La Figura 3-7 ilustra tal estructura de clases. Se ve que la clase `Valores` es un tipo de `Bienes` así como un tipo de `ElementoConIntereses`. Análogamente, la clase `CuentaBancaria` es un tipo de `Bienes`, así como un tipo de `ElementoAsegurable` y de `ElementoConIntereses`.

¹⁷ De hecho, ésta es la piedra de toque para la herencia múltiple. Si se encuentra una trama de herencias en la que las clases hoja pueden agruparse en conjuntos que denotan un comportamiento ortogonal (como los elementos asegurables y los que proporcionan intereses), y esos conjuntos no son disjuntos, eso es una indicación de que, dentro de una sola trama de herencias no existen clases intermedias a las que pueda asignarse con claridad esos comportamientos sin violar la abstracción de ciertas clases hoja proporcionándoles comportamientos que no deberían tener. Se puede remediar esta situación utilizando herencia múltiple para añadir estos comportamientos sólo donde se deseé.

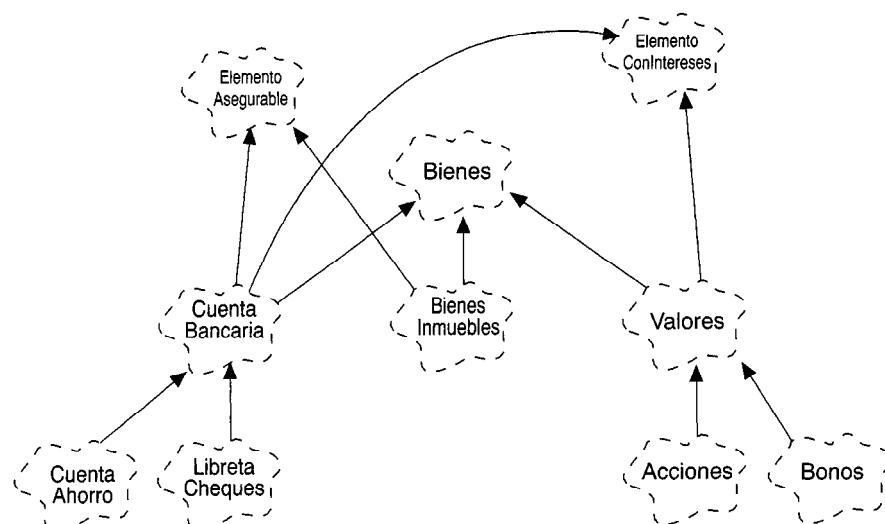


Figura 3.7. Herencia múltiple.

Para capturar estas decisiones de diseño en C++, se podrían escribir las siguientes declaraciones (muy resumidas). Se comienza con las clases base:

```
class Bienes...
class ElementoAsegurable...
class ElementoConIntereses...
```

A continuación se tienen varias clases intermedias, cada una de las cuales con varias superclases:

```
class CuentaBancaria : public Bienes,
                      public ElementoAsegurable,
                      public ElementoConIntereses...
class BienesInmuebles : public Bienes,
                       public ElementoAsegurable...
class Valores : public Bienes,
                public ElementoConIntereses...
```

Y finalmente se tienen las clases hoja restantes:

```
class CuentaAhorro : public CuentaBancaria...
class LibretaCheques : public CuentaBancaria...

class Acciones : public Valores...
class Bonos : public Valores...
```

El diseño de una estructura de clases adecuada que implique herencia, y es-

pecialmente si es herencia múltiple, es una tarea difícil. Como se explica en el Capítulo 4, éste es frecuentemente un proceso incremental e iterativo. Aparecen dos problemas cuando se tiene herencia múltiple: ¿cómo se tratan las colisiones de nombres de diferentes superclases?, y ¿cómo se maneja la herencia repetida?

Las colisiones de nombres pueden aparecer cuando dos o más superclases diferentes utilizan el mismo nombre para algún elemento de sus interfaces, como las variables de instancia o los métodos. Por ejemplo, supóngase que las clases `ElementoAsegurable` y `Bienes` tienen ambas atributos llamados `valorActual`, que denotan el valor actual del elemento. Puesto que la clase `BienesInmuebles` hereda de ambas clases, ¿qué significado tiene heredar dos operaciones con el mismo nombre? Ésta es, de hecho, la dificultad fundamental de la herencia múltiple: las colisiones pueden introducir ambigüedad en el comportamiento de la subclase que hereda de forma múltiple.

Hay tres aproximaciones básicas para resolver este tipo de desacuerdo. Primero, la semántica del lenguaje podría contemplar ese choque como algo incorrecto y rehusar la compilación de la clase. Éste es el enfoque adoptado por lenguajes como Smalltalk y Eiffel. En Eiffel, sin embargo, es posible renombrar elementos de forma que no haya ambigüedad. Segundo, la semántica del lenguaje podría contemplar el mismo nombre introducido por varias clases como referente al mismo atributo, que es el enfoque adoptado por CLOS. Tercero, la semántica del lenguaje podría permitir el desacuerdo, pero requerir que todas las referencias al nombre califiquen de forma completa la fuente de su declaración. Éste es el enfoque adoptado por C++¹⁸.

El segundo problema es la herencia repetida, que Meyer describe como sigue: «Uno de los problemas delicados planteados por la presencia de herencia múltiple es lo que sucede cuando una clase es un antecesor de otra por más de una vía. Si se permite herencia múltiple en un lenguaje, antes o después alguien escribirá una clase `D` con dos padres `B` y `C`, cada uno de los cuales tiene como padre a una clase `A` —o alguna otra situación en la que `D` herede dos (o más veces) de `A`. Esta situación se llama herencia repetida y debe tratarse de forma correcta» [41]. Como ejemplo, supóngase que se define la siguiente clase (mal concebida):

```
class FondoInversión : public Acciones,
                        public Bonos...
```

Esta clase introduce herencia repetida de la clase de `valores`, que es una superclase para `Acciones` y `Bonos`.

Existen tres enfoques para tratar el problema de la herencia repetida. Primero, se puede tratar la presencia de herencias repetidas como incorrecta. Éste es el enfoque de Smalltalk y Eiffel (donde Eiffel permite nuevamente renombrar para eliminar la ambigüedad de las referencias duplicadas). Segundo, se puede

¹⁸ En C++, las colisiones de nombres entre objetos miembro pueden resolverse calificando de forma completa todos los nombres miembro. Las funciones miembro con nombres y signaturas idénticos se consideran semánticamente la misma función.

permitir la duplicación de superclases, pero requerir el uso de nombres plenamente calificados para referirse a los miembros de una copia específica. Éste es uno de los enfoques adoptados por C++. Tercero, se pueden tratar las referencias múltiples a la misma clase como denotando a la misma clase. Éste es el enfoque de C++ cuando la superclase repetida se introduce como una clase base virtual. Existe una clase base virtual cuando una subclase nombra a otra clase como su superclase y marca esa superclase como *virtual*, para indicar que es una clase compartida. Análogamente, en CLOS las clases repetidas son compartidas, utilizando un mecanismo llamado la *lista de precedencia de clases*. Esta lista, calculada siempre que se introduce una nueva clase, incluye la propia clase y todas sus superclases, sin duplicación, y se basa en las reglas siguientes:

- Una clase siempre tiene precedencia sobre su superclase.
- Cada clase fija el orden de precedencia de sus superclases directas [42].

En este enfoque, el grafo de herencias se allana, se eliminan las clases duplicadas y la jerarquía resultante se resuelve mediante herencia múltiple [43]. Se parece a la computación de una ordenación topológica de las clases. Si se puede calcular una ordenación total de las clases, se acepta la clase que introduce la herencia repetida. Nótese que este orden total puede ser único, o puede haber varias ordenaciones posibles (y un algoritmo determinista seleccionará siempre una de esas ordenaciones). Si no se puede encontrar un orden (por ejemplo, cuando hay ciclos en las dependencias de clases), la clase se rechaza.

La existencia de herencia múltiple plantea un estilo de clases, llamadas *aditivas*. Se derivan de la cultura de programación que rodea al lenguaje Flavors: se combinan («mezclan», «añaden») pequeñas clases para construir clases con un comportamiento más sofisticado. Como observa Hendler: «una clase aditiva es sintácticamente idéntica a una clase normal, pero su intención es distinta. El propósito de tal clase es únicamente... [añadir] funciones a otras [clases] flavors —uno nunca crea una instancia de una clase aditiva» [44]. En la figura 3-7, las clases `ElementoAsegurable` y `ElementoConIntereses` son aditivas. Ninguna de esas clases puede existir por sí misma; antes bien, se usan para aumentar el significado de alguna otra clase¹⁹. Así, se puede definir una clase aditiva como la clase que incorpora un comportamiento simple y centrado y se utiliza para aumentar el comportamiento de alguna otra clase por medio de la herencia. El comportamiento de una clase aditiva suele ser completamente ortogonal al comportamiento de las clases con las que se combina. Una clase que se construye principalmente heredando de clases aditivas y no añade su propia estructura o comportamiento se llama *clase agregada*.

Polimorfismo múltiple. Considérese de nuevo la siguiente función miembro declarada para la clase `ElementoPantalla`:

```
virtual void dibujar();
```

¹⁹ En CLOS, es práctica común construir una clase aditiva utilizando sólo métodos `:before` y `:after` para aumentar el comportamiento de los métodos primarios existentes.

El propósito de esta operación es dibujar el objeto dado en algún contexto. Esta operación se declara como *virtual* y es, por tanto, polimórfica, lo que significa que siempre que se invoque esta operación para un objeto particular, se realizará una llamada a la implantación de esta operación en la subclase correspondiente, utilizando un algoritmo de selección de método como el descrito en las notas complementarias. Éste es un ejemplo de polimorfismo simple, lo que quiere decir que el método está especializado (es polimórfico) respecto a un factor, a saber, el objeto para el cual se invocó la operación.

Supóngase ahora que se precisa un comportamiento ligeramente diferente, dependiendo del dispositivo de visualización concreto que se usa. En un caso, se desearía que el método *dibujar* mostrase una representación gráfica de alta resolución; en otro, se desearía imprimir una visualización rápidamente, y así se dibujaría sólo una representación muy a grandes rasgos. Se declararían dos operaciones distintas aunque muy similares, como *dibujarGrafico* y *dibujarTexto*. Esto no es del todo satisfactorio, sin embargo, porque esta solución no responde bien al cambio de escala: si se introduce otro contexto más de dibujo habría que añadir una nueva operación para cada clase de la jerarquía *ElementoPantalla*.

En lenguajes como CLOS es posible escribir operaciones llamadas *multimétodos* que son polimórficas respecto a más de un factor o parámetro (como el elemento de pantalla y el dispositivo de visualización). En lenguajes que soportan sólo polimorfismo simple (como C++) se puede fingir este comportamiento polimórfico múltiple utilizando un recurso llamado *selección doble* (*double dispatching*).

Primero, se podría definir una jerarquía de dispositivos de visualización, enraizada en la clase *DispositivoVisual*. Después, se reescribiría la operación *dibujar* como sigue:

```
virtual void dibujar(DispositivoVisual&);
```

En la implantación de este método, se invocarían operaciones de dibujo que son polimórficas según el parámetro actual *DispositivoVisual* que se pase —de aquí el nombre «selección doble»: *dibujar* muestra primero un comportamiento polimórfico según la subclase concreta de *ElementoPantalla*, y a continuación exhibe comportamiento polimórfico según la subclase exacta del argumento *DispositivoVisual*.

Este recurso puede extenderse hasta cualquier grado de selección polimórfica.

Agregación

Ejemplo. Las relaciones de agregación entre clases tienen un paralelismo directo con las relaciones de agregación entre los objetos correspondientes a esas

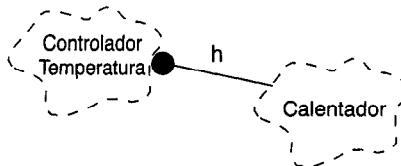


Figura 3.8. Agregación.

clases. Por ejemplo, considérese otra vez la declaración de la clase `ControladorTemperatura`:

```
class ControladorTemperatura {
public:
    ControladorTemperatura(Posicion);
    ~ControladorTemperatura();

    void procesar(const GradienteTemperatura&);

    Minuto planificar(const GradienteTemperatura&) const;

private:
    Calentador c;
};
```

Como se muestra en la Figura 3.8, la clase `ControladorTemperatura` denota el todo, y una de sus partes es una instancia de la clase `Calentador`. Esto se corresponde exactamente con la relación de agregación entre las instancias de estas clases, ilustradas en la Figura 3.3.

Contención física. En el caso de la clase `ControladorTemperatura`, hay agregación como contención *por valor*, un tipo de contención física que significa que el objeto `Calentador` no existe independientemente de la instancia `ControladorTemperatura` que lo encierra. Por el contrario, el tiempo de vida de ambos objetos está en íntima conexión: cuando se crea una instancia de `ControladorTemperatura`, se crea también una instancia de la clase `Calentador`. Cuando se destruye el objeto `ControladorTemperatura`, esto implica la destrucción del objeto `Calentador` correspondiente.

Es también posible un tipo menos directo de agregación, llamado contención *por referencia*. Por ejemplo, se podría reemplazar la parte privada de la clase `ControladorTemperatura` por la declaración siguiente²⁰:

²⁰ Alternativamente, se podría haber declarado `c` como una referencia a un objeto calentador (en C++ sería `Calentador&`), cuya semántica respecto a la inicialización y modificación es bastante distinta que para los punteros.

`Calentador* c;`

En este caso, la clase `ControladorTemperatura` sigue denotando al todo, y una de sus partes sigue siendo una instancia de la clase `Calentador`, aunque ahora hay que acceder a esa parte indirectamente. Desde ahora, los tiempos de vida de ambos objetos ya no están tan estrechamente emparejados como antes: se pueden crear y destruir instancias de cada clase independientemente. Es más, puesto que es posible que la parte sea compartida estructuralmente, hay que decidir sobre alguna política por la cual su espacio de almacenamiento sea correctamente creado y destruido por sólo uno de los agentes que comparten referencias a esa parte.

La agregación establece una dirección en la relación todo/parte. Por ejemplo, el objeto `Calentador` es una parte del objeto `ControladorTemperatura`, y no al revés. La contención por valor no puede ser cíclica (es decir, ambos objetos no pueden ser físicamente partes de otro), aunque la contención por referencia puede serlo (cada objeto puede tener un puntero apuntando al otro)²¹.

Por supuesto, como se describió en un ejemplo anterior, la agregación no precisa contención física, como se desprende de la contención por valor o por referencia. Por ejemplo, aunque los accionistas poseen acciones, un accionista no contiene físicamente las acciones que posee. Antes bien, los tiempos de vida de ambos objetos pueden ser completamente independientes, aunque siga existiendo conceptualmente una relación todo/parte (cada acción es siempre parte de los bienes del accionista) y así la representación de esta agregación puede ser muy indirecta. Por ejemplo, se podría declarar la clase `Accionista`, cuyo estado incluye una clave para una tabla de base de datos que puede utilizarse para buscar las acciones que posee un accionista particular. Esto sigue siendo agregación, aunque no contención física. En última instancia, la piedra de toque para la agregación es ésta: si y sólo si existe una relación todo/parte entre dos objetos, podremos tener una relación de agregación entre sus clases correspondientes.

La herencia múltiple se confunde a menudo con la agregación. De hecho, en C++ la herencia `protected` o `private` puede sustituirse con facilidad por agregación `protected` o `private` de una instancia de la superclase, sin pérdidas semánticas. Cuando se considera la herencia versus la agregación, recuérdese aplicar la prueba correspondiente para ambas. Si no se puede afirmar sinceramente que existe una relación «es un» entre dos clases, habría que utilizar agregación o alguna otra relación en vez de la herencia.

Uso

Ejemplo. El ejemplo anterior de los objetos `controladorGradiente` y `gradienteCreciente` ilustraba un enlace entre los dos objetos, que en su mo-

²¹ Una asociación puede reemplazarse frecuentemente por agregación cíclica o relaciones «de uso» cíclicas. Las más de las veces, sin embargo, una asociación (que por definición implica una bidireccionalidad) se refina durante el diseño para ser una sola relación de agregación o «de uso», denotando así una restricción acerca de la dirección de la asociación.

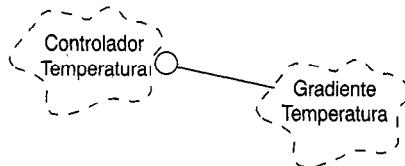


Figura 3.9. La relación «de uso».

mento se representó mediante una relación «de uso» entre sus clases correspondientes, `ControladorTemperatura` y `GradienteTemperatura`:

```

class ControladorTemperatura {
public:

    ControladorTemperatura(Posicion);
    ~ControladorTemperatura();

    void procesar(const GradienteTemperatura&);

    Minuto planificar(const GradienteTemperatura&) const;

private:
    Calentador c;
};
  
```

La clase `GradienteTemperatura` aparece como parte del prototipo de ciertas funciones miembro, y por tanto puede decirse que `ControladorTemperatura` usa los servicios de la clase `GradienteTemperatura`.

Clientes y proveedores. Las relaciones «de uso» entre clases corren paralelas a los enlaces «hermano-a-hermano» entre las instancias correspondientes de esas clases. Mientras que una asociación denota una conexión semántica bidireccional, una relación «de uso» es un posible refinamiento de una asociación, por el que se establece qué abstracción es el cliente y qué abstracción es el servidor que proporciona ciertos servicios. Se ilustra tal relación «de uso» cliente/proveedor en la Figura 3.9²².

En realidad, una clase puede utilizar a otra de diversas formas. En el ejemplo, el `ControladorTemperatura` usa a la `GradienteTemperatura` en la firma de su interfaz. El `ControladorTemperatura` podría usar también a otra clase como `Pronosticador` en su implantación de la función miembro `planificar`. Ésta no es una afirmación de una relación todo/parte: la instancia de la clase

²² Como se afirmó anteriormente, una relación «de uso» cíclica es equivalente a una asociación, aunque la afirmación inversa no es necesariamente cierta.

Pronosticador solamente es utilizada por la instancia de Controlador-Temperatura, pero no es parte de ella. Típicamente, tal relación «de uso» se manifiesta porque se declara en la implementación de alguna operación un objeto local de la clase usada.

Las relaciones «de uso» estrictas son ocasionalmente demasiado restringidas porque permiten al cliente acceder sólo al interfaz public del proveedor. A veces, por razones tácticas, hay que romper el encapsulamiento de estas abstracciones, y éste es el verdadero propósito del concepto `friend` (amiga) en C++.

Instanciación (creación de instancias)

Ejemplos. La declaración vista de la clase Cola no era demasiado satisfactoria porque su abstracción no era segura respecto a los tipos. Se puede mejorar enormemente la abstracción utilizando lenguajes como C++ y Eiffel que soporan genericidad.

Por ejemplo, se podría reescribir la declaración de clase utilizando una clase parametrizada en C++:

```
template<class Elemento>
class Cola {
public:

    Cola();
    Cola(const Cola<Elemento>&);
    virtual ~Cola();

    virtual Cola<Elemento>& operator=(const Cola<Elemento>&);
    virtual int operator==(const Cola<Elemento>&) const;
    int operator!=(const Cola<Elemento>&) const;

    virtual void borrar();
    virtual void anadir(const Elemento&);
    virtual void extraer();
    virtual void eliminar(int donde);

    virtual int longitud() const;
    virtual int estaVacia() const;
    virtual const Elemento& cabecera() const;
    virtual int posicion(const void *);

protected:
    ...
};
```

Nótese que en esta declaración ya no se añaden ni recuperan objetos por

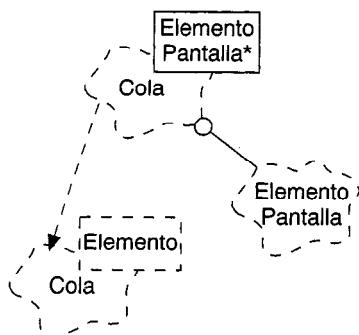


Figura 3.10. Instanciación.

medio de `void*` (que no es seguro respecto a los tipos); se hace mediante la clase `Elemento` declarada como un argumento plantilla o modelo (*template*).

Una clase parametrizada no puede tener instancias a menos que antes se la instancie. Por ejemplo, se podría declarar dos objetos de cola concretos, una cola de enteros y una cola de elementos de pantalla:

```
Cola<int> colaEnteros;
Cola<ElementoPantalla*> colaElementos;
```

Los objetos `colaEnteros` y `colaElementos` son instancias de clases claramente diferentes, y ni siquiera están unidas por ninguna superclase común, aunque ambas se derivan de la misma clase parametrizada. Por razones que se describirán en el Capítulo 9, se utiliza un puntero a la clase `ElementoPantalla` en la segunda instancia, de forma que los objetos de una subclase de `ElementoPantalla` colocados en la cola no serán cortados, sino que conservarán su comportamiento polimórfico.

Estas instanciaciones son seguras respecto a tipos. Las reglas de tipos del C++ rechazarán cualquier sentencia que intente añadir o recuperar cualquier cosa que no sean enteros en la clase `colaEnteros` y cualquier cosa que no sean instancias de `ElementoPantalla` o sus subclases en `colaElementos`.

La Figura 3.10 ilustra las relaciones entre la clase parametrizada `cola`, su instancia para `ElementoPantalla` y su instancia correspondiente `colaElementos`.

Genericidad. Existen cuatro formas básicas de construir clases como la clase parametrizada `cola`. Primero, pueden utilizarse macros. Éste es el estilo que hay que usar en versiones anteriores de C++, pero como observa Stroustrup, este «enfoque no funciona bien excepto a pequeña escala» [45], porque el mantenimiento de macros es tosco y está fuera de la semántica del lenguaje; además, cada instancia tiene como resultado una nueva copia del código. Segundo,

puede adoptarse el enfoque de Smalltalk y confiar en la herencia y la ligadura tardía [46]. Con esta aproximación, se pueden construir sólo clases de contención heterogéneas, porque no existe forma de establecer la clase específica de los elementos del contenedor; cada elemento se trata como si fuese una instancia de alguna lejana clase base. Tercero, se puede adoptar una solución utilizada habitualmente en lenguajes como Object Pascal, que tienen comprobación estricta de tipos, soportan la herencia, pero no soportan ninguna forma de clases parametrizadas. En este caso, se construyen clases contenedor generalizadas, como en Smalltalk, pero se utiliza código de comprobación de tipos explícito para reforzar la convención de que los contenidos son todos de la misma clase, que se establece cuando se crea el objeto contenedor. Este enfoque tiene un efecto significativo sobre el tiempo de ejecución. Cuarto, se puede adoptar la aproximación que introdujo CLU en primer lugar y proporcionar un mecanismo directo para clases parametrizadas, como en el ejemplo. Una *clase parametrizada* (conocida también como *clase genérica*) es una que sirve como modelo para otras clases —un modelo que puede parametrizarse con otras clases, objetos y/o operaciones. Una clase parametrizada debe ser instanciada (es decir, sus parámetros deben ser rellenados) antes de que puedan crearse los objetos. C++ y Eiffel soportan mecanismos de clases genéricas.

En la Figura 3.10, nótese que para instanciar la clase `cola`, hay que usar también la clase `ElementoPantalla`. En realidad, las relaciones de instanciación casi siempre requieren alguna relación «de uso», que hace visible a las clases actuales utilizadas para llenar el modelo o plantilla.

Meyer ha apuntado que la herencia es un mecanismo más potente que la genericidad y que gran parte de los beneficios de la genericidad puede conseguirse mediante la herencia, pero no al revés [47]. En la práctica, es de utilidad usar un lenguaje que soporte tanto la herencia como las clases parametrizadas.

Pueden utilizarse las clases parametrizadas para muchas más cosas que para construir clases contenedor. Como apunta Stroustrup, «la parametrización de tipos permitirá parametrizar las funciones aritméticas respecto al tipo numérico básico, de forma que los programadores puedan (por fin) obtener un modo uniforme de tratar con enteros, números en punto flotante de precisión simple, de doble precisión, etc.» [48].

Desde una perspectiva de diseño, las clases parametrizadas son también útiles para capturar ciertas decisiones de diseño sobre el protocolo de una clase. Mientras que una definición de clase exporta las operaciones que pueden realizarse sobre instancias de esa clase, los argumentos de un modelo sirven para importar clases (y valores) que suministran un protocolo específico. En C++, esta congruencia de tipos se realiza en tiempo de compilación, cuando se expande la instanciación. Por ejemplo, se podría declarar una clase de cola ordenada que representase colecciones de objetos ordenados según algún criterio. Esta clase parametrizada debe contar con alguna clase `Elemento`, como antes, pero debe esperar también que `Elemento` proporcione alguna operación de ordenación. Parametrizando la clase de esta forma, se logra esto de forma más débilmente acoplada: se puede sustituir el argumento formal `Elemento` con cualquier clase

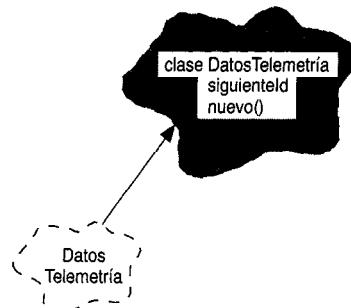


Figura 3.11. Metaclasses.

que ofrezca esta función de ordenación. En este sentido, se puede definir una clase parametrizada como una que denota una familia de clases cuya estructura y comportamiento están definidos independientemente de los parámetros formales de la clase.

Metaclases

Se ha dicho que todo objeto es una instancia de alguna clase. ¿Qué ocurre si se trata a la propia clase como un objeto que puede manipularse? Para hacerlo, hay que plantearse: ¿Qué es la clase de una clase? La respuesta es: simplemente, una metaclase. Dicho de otra forma, una *metaclase* es una clase cuyas instancias son, ellas mismas, clases. Los lenguajes como Smalltalk y CLOS soportan el concepto de metaclase directamente; C++ no. Realmente, la idea de metaclase lleva a la idea del modelo de objetos a su conclusión natural en los lenguajes de programación orientados a objetos puros.

Robson justifica la necesidad de metaclases haciendo notar que «en un sistema bajo desarrollo, una clase proporciona al programador un interfaz para comunicarse con la definición de los objetos. Para este uso de las clases, es extremadamente útil para ellas el ser objetos, de forma que puedan ser manipulados de la misma forma que todas las demás descripciones»[49].

En lenguajes como Smalltalk, el propósito principal de una metaclase es proporcionar variables de clase (compartidas por todas las instancias de la clase) y operaciones para inicializar variables de clase y crear la instancia simple de la metaclase [50]. Por convenio, una metaclase de Smalltalk contiene típicamente ejemplos que muestran el uso de las clases de la misma. Por ejemplo, como se muestra en la Figura 3.11, podría definirse en Smalltalk una variable de clase `siguienteId` para la metaclase de `DatosTelemetría`. Análogamente, se podría definir una operación para crear nuevas instancias de la clase, que quizás las generaría partiendo de algún espacio de almacenamiento ya reservado.

Aunque C++ no soporta metaclases explícitamente, la semántica de sus

constructores y destructores sirven al propósito de creación de metaclasses. Además, C++ tiene recursos para variables de clase y operaciones de metaclass. Los objetos miembro static de C++ son equivalentes a las variables de clase de Smalltalk, y las funciones miembro static son equivalentes a las operaciones de metaclass de Smalltalk.

Como se ha dicho, el soporte para metaclasses de CLOS es aún más potente que el de Smalltalk. Mediante el uso de metaclasses, se puede redefinir la propia semántica de elementos como precedencia de clases, funciones genéricas y métodos. El beneficio principal de este recurso es que permite experimentar con paradigmas alternativos de programación orientada a objetos, y facilita la construcción de herramientas de desarrollo del software, como hojeadores (browsers).

En CLOS, la clase predefinida `standard-class` es la metaclass de todas las clases sin tipo definidas mediante `defclass`. Esta metaclass define el método `make-instance`, que implanta la semántica de cómo se crean las instancias. `standard-class` también define el algoritmo para calcular la lista de precedencia de clases. CLOS permite que se redefina el comportamiento de ambos métodos.

Los métodos y las funciones genéricas también pueden tratarse como objetos en CLOS. Puesto que son algo diferentes de los tipos usuales de objetos, los objetos clase, objetos método y objetos función genérica se llaman globalmente *metaobjetos*. Cada método es una instancia de la clase predefinida `standard-method`, y cada función genérica se trata como una instancia de la clase `standard-generic-function`. Ya que el comportamiento de estas clases predefinidas puede redefinirse, es posible cambiar el significado de los métodos y las funciones genéricas.

3.5. La interacción entre clases y objetos

Relaciones entre clases y objetos

Las clases y los objetos son conceptos separados pero en íntima relación. Concretamente, todo objeto es instancia de alguna clase, y toda clase tiene cero o más instancias. Para prácticamente todas las aplicaciones, las clases son estáticas; sin embargo, su existencia, semántica y significado están fijados antes de la ejecución de un programa. Analogamente, la clase de la mayoría de los objetos es estática, lo que significa que una vez que se crea un objeto, su clase está fijada. En un agudo contraste, sin embargo, los objetos se crean y destruyen típicamente a un ritmo trepidante durante el tiempo de vida de una aplicación.

Por ejemplo, considérense las clases y objetos en la implantación de un sistema de control de tráfico aéreo. Algunas de las abstracciones más importantes son los aviones, los planes de vuelo, las pistas y los espacios aéreos. Por su pro-

pia definición, los significados de estas clases y objetos son relativamente estáticos. Deben ser estáticos, porque si no no podría construirse una aplicación que contuviese el conocimiento de hechos tan de sentido común como que los aviones pueden despegar, volar y aterrizar, y que dos aviones no deberían ocupar el mismo espacio al mismo tiempo. Por el contrario, las instancias de estas clases son dinámicas. Las pistas nuevas se construyen y las viejas se abandonan con evidente lentitud. Con mayor rapidez, se archivan nuevos planes de vuelo, y a otros viejos se les da carpetazo. Con gran frecuencia, nuevos aviones irrumpen en un espacio aéreo concreto y otros los abandonan.

El papel de clases y objetos en análisis y diseño

Durante el análisis y las primeras etapas del diseño, el desarrollador tiene dos tareas principales:

- Identificar las clases y objetos que forman el vocabulario del dominio del problema.
- Idear las estructuras por las que conjuntos de objetos trabajan juntos para lograr los comportamientos que satisfacen los requerimientos del problema.

En conjunto, se llama a esas clases y objetos las *abstracciones clave* del problema, y se denomina a esas estructuras cooperativas los *mecanismos* de la implantación.

Durante estas fases del desarrollo, el interés principal del desarrollo debe estar en la vista externa de estas abstracciones clave y mecanismos. Esta vista representa el marco de referencia lógico del sistema y, por tanto, abarca la estructura de clases y la estructura de objetos del mismo. En las etapas finales del diseño y entrando ya en la implantación, la tarea del desarrollador cambia: el centro de atención está en la vista interna de estas abstracciones clave y mecanismos, involucrando a su representación física. Pueden expresarse estas decisiones de diseño como parte de la arquitectura de módulos y la arquitectura de procesos del sistema.

3.6. De la construcción de clases y objetos de calidad

Medida de la calidad de una abstracción

Ingalls sugiere que «un sistema debería construirse con un conjunto mínimo de partes inmutables; estas partes deberían ser tan generales como fuese posible; y todas las partes del sistema deberían conservarse en un marco de referencia uniforme» [51]. Con el desarrollo orientado a objetos, estas partes son las clases y

objetos que constituyen las abstracciones clave del sistema, y el marco de referencia es proporcionado por sus mecanismos.

Según nuestra experiencia, el diseño de clases y objetos es un proceso incremental e iterativo. Francamente, excepto para las abstracciones más triviales, nunca hemos sido capaces de definir una clase perfectamente bien a la primera. Como explican los Capítulos 4 y 7, lleva tiempo suavizar las irregulares fronteras conceptuales de las abstracciones iniciales. Por supuesto, produce un coste refinar estas abstracciones, en términos de recompilación, inteligibilidad e integridad de la trama de diseño del sistema. Por tanto, sería deseable llegar tan cerca del acierto como fuese posible ya la primera vez.

¿Cómo puede saberse si una clase u objeto dado está bien diseñado? Se sugieren cinco métricas significativas:

- Acoplamiento.
- Cohesión.
- Suficiencia.
- Compleción (estado completo/plenitud).
- Ser primitivo.

El acoplamiento es una noción copiada del diseño estructurado, pero con una interpretación liberal también se aplica al diseño orientado a objetos. Stevens, Myers y Constantine definen el acoplamiento como «la medida de la fuerza de la asociación establecida por una conexión entre un módulo y otro. El acoplamiento fuerte complica un sistema porque los módulos son más difíciles de comprender, cambiar o corregir por sí mismos si están muy interrelacionados con otros módulos. La complejidad puede reducirse diseñando sistemas con los acoplamientos más débiles posibles entre los módulos» [52]. Un contraejemplo de lo que sería un acoplamiento correcto lo ofrece Page-Jones en su descripción de un sistema estéreo modular en el que la fuente de alimentación se sitúa en la caja de uno de los altavoces [53].

El acoplamiento respecto a los módulos es aplicable en el análisis y diseño orientados a objetos, pero el acoplamiento respecto a clases y objetos es igualmente importante. Sin embargo, existe tensión entre los conceptos de acoplamiento y herencia, porque la herencia introduce un acoplamiento considerable. Por un lado, son deseables clases débilmente acopladas; por el otro, la herencia —que acopla estrechamente las superclases y sus subclases— ayuda a explotar las características comunes de las abstracciones.

La idea de cohesión también proviene del diseño estructurado. Dicho sencillamente, la cohesión mide el grado de conectividad entre los elementos de un solo módulo (y para el diseño orientado a objetos, una sola clase u objeto). La forma de cohesión menos deseable es la cohesión por coincidencia, en la que se incluyen en la misma clase o módulo abstracciones sin ninguna relación. Por ejemplo, considérese una clase que comprende las abstracciones de los perros y las naves espaciales, cuyos comportamientos están bastante poco relacionados. La forma más deseable de cohesión es la cohesión funcional, en la cual los elementos de una clase o módulo trabajan todos juntos para proporcionar algún

comportamiento bien delimitado. Así, la clase `Perro` es funcionalmente cohesiva si su semántica se ciñe al comportamiento de un perro, todo el perro y nada más que el perro.

En relación muy estrecha con las ideas de acoplamiento y cohesión están los criterios de que una clase o módulo debería ser suficiente, completo y primitivo. Por *suficiente* quiere decirse que la clase o módulo captura suficientes características de la abstracción como para permitir una interacción significativa y eficiente. Lo contrario produce componentes inútiles. Por ejemplo, si se está diseñando la clase `Conjunto`, es de sabios incluir una operación que elimine un elemento del conjunto, pero esa sabiduría servirá de poco si se olvida incluir una operación que añada elementos. En la práctica, las violaciones de esta característica se detectan muy pronto; estas deficiencias asoman casi siempre que se construye un cliente que deba utilizar esa abstracción. Por *completo*, quiere decirse que el interfaz de la clase o módulo captura todas las características significativas de la abstracción. Mientras la suficiencia implica un interfaz mínimo, un interfaz completo es aquel que cubre todos los aspectos de la abstracción. Una clase o módulo completo es aquel cuyo interfaz es suficientemente general para ser utilizable de forma común por cualquier cliente. La compleción (estado completo o plenitud) es una cuestión subjetiva, y puede exagerarse. Ofrecer todas las operaciones significativas para una abstracción particular desborda al usuario y suele ser innecesario, ya que muchas operaciones de alto nivel pueden componerse partiendo de las de bajo nivel. Por esta razón, se sugiere también que las clases y módulos sean primitivos. Las operaciones *primitivas* son aquellas que pueden implantarse eficientemente sólo si tienen acceso a la representación subyacente de la abstracción. Así, añadir un elemento a un conjunto es primitiva, porque, para implantar esta operación `Anadir`, debe ser visible la representación subyacente. Por el contrario, una operación que añade cuatro elementos a un conjunto no es primitiva, porque puede implantarse con la misma eficiencia a través de la operación `Anadir` más primitiva, sin tener acceso a la representación interna. Por supuesto, la eficiencia también es una medida subjetiva. Una operación es inequívocamente primitiva si se puede implantar sólo accediendo a la representación interna. Una operación que podría implantarse sobre operaciones primitivas existentes, pero a un coste de recursos computacionales significativamente mayor, es también candidata para su inclusión como operación primitiva.

Selección de operaciones

Semántica funcional. El desarrollo del interfaz de una clase o módulo es simplemente un trabajo difícil. Típicamente, se hace un primer intento de diseño de la clase, y a continuación, a medida que uno mismo y los demás van creando clientes, se hace necesario aumentar, modificar y refinar más este interfaz. Eventualmente, pueden descubrirse patrones de operaciones o patrones

de abstracciones que llevan a la invención de nuevas clases o a la reorganización de las relaciones entre clases existentes.

Dentro de una clase dada, nuestro estilo es mantener todas las operaciones primitivas, de forma que cada una muestre un comportamiento reducido y bien definido. Se llama a tales métodos *de grano fino*. Se tiende también a separar métodos que no se comunican con otros. De este modo, es mucho más fácil construir subclases que puedan redefinir significativamente el comportamiento de sus superclases. La decisión de subcontratar un comportamiento a uno o a muchos métodos puede tomarse en función de dos razones enfrentadas: agrupar un comportamiento dado en un método conduce a un interfaz más simple pero métodos más grandes y complicados; distribuir un comportamiento entre varios métodos lleva a un interfaz más complicado, pero métodos más simples. Como observa Meyer, «un buen diseñador sabe cómo encontrar el equilibrio apropiado entre subcontratar demasiado, lo que produce fragmentación, o demasiado poco, lo que produce módulos de tamaño inmanejable» [54].

Es habitual en el desarrollo orientado a objetos diseñar los métodos de una clase como un todo, porque todos esos métodos cooperan para formar el protocolo completo de la abstracción. Así, dado un comportamiento que se desea, hay que decidir en qué clase se sitúa. Halbert y O'Brien ofrecen los siguientes criterios para que se consideren al tomar una decisión de este tipo:

- | | |
|---|--|
| <ul style="list-style-type: none">• Reutilización
(<i>reusabilidad</i>)• Complejidad• Aplicabilidad• Conocimiento de la implementación | <ul style="list-style-type: none">• ¿Sería este comportamiento más útil en más de un contexto?• ¿Qué grado de dificultad plantea el implementar este comportamiento?• ¿Qué relevancia tiene este comportamiento para el tipo en el que podría ubicarse?• ¿Depende la implementación del comportamiento de los detalles internos de un cierto tipo [55]? |
|---|--|

Suele elegirse declarar las operaciones significativas que pueden realizarse sobre un objeto como métodos en la definición de la clase (o superclase) de ese objeto. En lenguajes como C++ y CLOS, sin embargo, pueden declararse también esas operaciones como *subprogramas libres*, que pueden agruparse entonces en utilidades de clase. En terminología de C++, un *subprograma libre* es una función no-miembro. Puesto que los subprogramas libres no pueden redefinirse como se hace con los métodos, son menos generales. Sin embargo, las utilidades resultan de gran ayuda para mantener primitiva una clase y para reducir los acoplamientos entre clases, especialmente si estas operaciones de nivel superior involucran a objetos de muchas clases diferentes.

Semántica espacial y temporal. Una vez que se ha establecido la existencia de una operación particular y definido su semántica funcional, hay que decidir sobre su semántica espacial y temporal. Esto significa que hay que especificar las decisiones sobre la cantidad de tiempo que lleva completar una operación y

la cantidad de espacio de almacenamiento que necesita. Tales decisiones suelen expresarse en términos de caso mejor, medio y peor, con el caso peor especificando una cota superior de lo que es aceptable.

Anteriormente se mencionó también que siempre que un objeto pasa un mensaje a otro a través de un enlace, los dos objetos deben estar sincronizados de alguna forma. En presencia de múltiples hilos de control, esto significa que el paso de mensajes es mucho más que una mera selección de subprogramas. En la mayoría de los lenguajes que se utilizan, simplemente la sincronización entre objetos no es un problema, porque los programas contienen un único hilo de control, lo que significa que todos los objetos son secuenciales. Se habla del paso de mensajes en tales situaciones como algo simple, porque su semántica es lo más parecido posible a llamadas normales a subprogramas. Sin embargo, en lenguajes que soportan concurrencia²³, hay que ocuparse de formas más sofisticadas de paso de mensajes, para evitar con ello los problemas que se crean si dos hilos de control actúan sobre el mismo objeto sin restricciones. Como se describió anteriormente, los objetos cuya semántica se preserva en presencia de múltiples hilos de control son objetos protegidos o sincronizados.

Se ha encontrado útil en algunas circunstancias expresar la semántica de la concurrencia para cada operación individual, así como para el objeto en su conjunto, ya que diferentes operaciones pueden requerir diferentes tipos de sincronización. El paso de mensajes debe así adoptar una de las formas siguientes:

- | | |
|----------------------|--|
| • Síncrono | Una operación comienza sólo cuando el emisor ha iniciado la acción y el receptor está preparado para aceptar el mensaje; el emisor y el receptor esperarán indefinidamente hasta que ambas partes estén preparadas para continuar. |
| • Abandono inmediato | Igual que el síncrono, excepto en que el emisor abandonará la operación si el receptor no está preparado inmediatamente. |
| • De intervalo | Igual que el síncrono, excepto en que el emisor esperará a que el receptor esté listo sólo durante un intervalo de tiempo especificado. |
| • Asíncrono | Un emisor puede iniciar una acción independientemente de si el receptor está esperando o no el mensaje. |

La forma se puede seleccionar de forma individual para cada operación, pero sólo cuando se haya decidido ya sobre la semántica funcional de dicha operación.

²³ Ada y Smalltalk tienen soporte directo para la concurrencia. Lenguajes como C++ no lo tienen, pero muchas veces pueden ofrecer semántica concurrente mediante extensiones con clases dependientes de la plataforma, como la biblioteca de tareas AT&T para C++.

Elección de relaciones

Colaboraciones. La elección de las relaciones entre clases y entre objetos está ligada a la elección de operaciones. Si se decide que el objeto *x* envía un mensaje *m* al objeto *y*, entonces ya sea de forma directa o indirecta *y* debe ser accesible a *x*; de otro modo, no se podría nombrar a la operación *m* en la implantación de *x*. Por *accesible*, se entiende la capacidad de una abstracción para ver a otra y hacer referencia a recursos en su vista externa. Una abstracción es accesible a otra sólo donde sus ámbitos se superpongan y sólo donde estén garantizados los derechos de acceso (por ejemplo, las partes privadas de una clase son accesibles sólo a la propia clase y sus amigas). El acoplamiento es por tanto una medida del grado de accesibilidad.

Una línea maestra útil en la elección de relaciones entre objetos se llama la Ley de Demeter, que afirma que «los métodos de una clase no deberían depender de ninguna manera de la estructura de ninguna clase, salvo de la estructura inmediata (de nivel superior) de su propia clase. Además, cada método debería enviar mensajes sólo a objetos pertenecientes a un conjunto muy limitado de clases» [56]. El efecto básico de la aplicación de esta ley es la creación de clases débilmente acopladas, cuyos secretos de implantación están encapsulados. Tales clases están claramente libres de sobrecargas, lo que significa que para comprender el significado de una clase no es necesario comprender los detalles de muchas otras clases.

Al examinar la estructura de clases de un sistema completo, puede hallarse que su jerarquía de herencias es o bien ancha y poco profunda, estrecha y profunda, o equilibrada. Las estructuras de clases que son anchas y poco profundas suelen representar bosques de clases independientes que se pueden mezclar y combinar [57]. Las estructuras de clases que son estrechas y profundas representan árboles de clases relacionadas por un antepasado común [58]. Existen ventajas y desventajas para cada enfoque. Los bosques de clases están más débilmente acoplados, pero no pueden explotar todos los elementos comunes que existen. Los árboles de clases explotan esta comunalidad, así que las clases individuales son más pequeñas que en los bosques. Sin embargo, para comprender una clase particular suele ser necesario comprender el significado de todas las clases de las que hereda y de todas las clases que usa. La forma correcta de una estructura de clases es altamente dependiente del tipo de problema.

Hay que realizar compensaciones similares entre relaciones de herencia, agregación y uso. Por ejemplo, ¿la clase *Coche* debería heredar, contener o usar las clases llamadas *Motor* y *Rueda*? En este caso, se sugiere que una relación de agregación sería más apropiada que una relación de herencia. Meyer afirma que entre las clases *A* y *B*, «la herencia es apropiada si toda instancia de *B* puede verse también como una instancia de *A*. La relación de cliente es apropiada cuando toda instancia de *B* simplemente posee uno o más atributos de *A*» [59]. Desde otro punto de vista, si el comportamiento de un objeto es mayor que la suma de sus partes individuales, entonces es probablemente mejor la creación de una relación de agregación y no de herencia entre las clases apropiadas.

Mecanismos y visibilidad. La decisión sobre las relaciones entre objetos es principalmente una cuestión de diseñar los mecanismos por los que esos objetos interactúan. La pregunta que debe formular el desarrollador es, sencillamente: ¿Dónde debe residir cierto conocimiento? Por ejemplo, en una planta de fabricación, los materiales (llamados *lotes*) entran en células de fabricación para ser procesados. A medida que entran en ciertas células, hay que notificárselo al jefe de taller para que tome las medidas oportunas. Ahora se tiene una decisión de diseño: ¿es la entrada de un lote en una sala una operación sobre la sala, una operación sobre el lote, o una operación sobre ambos? Si se decide que es una operación sobre la sala, la sala debe ser visible para el lote. Si se decide que es una operación sobre el lote, el lote debe ser visible para la sala, porque el lote debe saber en qué sala está. Finalmente, si se considera que es una operación sobre la sala y sobre el lote, hay que disponer que la visibilidad sea mutua. Hay que decidir también sobre alguna relación de visibilidad entre la sala y el jefe (y no entre el lote y el jefe); o el jefe sabe qué sala dirige, o la sala sabe quién es su jefe.

Durante el proceso de diseño, a veces es útil establecer explícitamente cómo un objeto es visible para otro. Existen cuatro formas fundamentales por las que un objeto *x* puede hacerse visible a un objeto *y*:

- El objeto proveedor es global al cliente.
- El objeto proveedor es parámetro de alguna operación del cliente.
- El objeto proveedor es parte del objeto cliente.
- El objeto proveedor es un objeto declarado localmente en el ámbito del diagrama de objetos.

Hay una variación sobre cada una de estas ideas, y es la idea de visibilidad compartida. Por ejemplo, *y* podría ser parte de *x*, pero *y* podría ser también visible a otros objetos de formas diferentes. En Smalltalk, este tipo de visibilidad normalmente representa una dependencia entre dos objetos. La visibilidad compartida implica compartición estructural, lo que significa que un objeto no tiene acceso exclusivo sobre otro: el estado del objeto compartido puede ser alterado por más de una vía.

Elección de implementaciones

Sólo después de estabilizar el aspecto exterior de una clase u objeto dado puede pasarse a su aspecto interior. Esta perspectiva implica dos decisiones diferentes: la elección de la representación de una clase u objeto y la ubicación de la clase u objeto en un módulo.

Representación. La representación de una clase u objeto debería casi siempre ser uno de los secretos encapsulados de la abstracción. Esto posibilita cambiar la representación (por ejemplo, para alterar la semántica del tiempo y espacio) sin violar ninguna de las suposiciones funcionales que los clientes puedan

haber hecho. Como establece Wirth con razón, «la elección de la representación es frecuentemente algo bastante difícil, y no está determinado de manera unívoca por las posibilidades disponibles. Debe tomarse siempre a la luz de las operaciones que van a realizarse sobre los datos» [60]. Por ejemplo, dada una clase cuyos objetos denotan un conjunto de planes de vuelo, ¿se optimiza la representación para búsqueda rápida o para inserción y borrado rápidos? No se puede optimizar para ambos, así que la elección debe basarse en el uso esperado de esos objetos. A veces no es fácil elegir, y se acaba con familias de clases cuyos interfaces son prácticamente idénticos, pero cuyas implantaciones son radicalmente distintas, con el fin de proporcionar diferentes comportamientos respecto al espacio y el tiempo.

Una de las compensaciones más difíciles cuando se selecciona la implantación de una clase se da entre el cálculo del valor del estado de un objeto y su almacenamiento como un campo. Por ejemplo, supóngase que se tiene la clase *Cono*, que incluye el método *volumen*. La invocación de este método devuelve el volumen del objeto. Como parte la representación de esta clase, se utilizarán probablemente campos para la altura del cono y el radio de su base. ¿Habría que tener un campo adicional en el que se almacenase el volumen del objeto, o debería el método *volumen* calcularlo cada vez [60]? Si se desea que este método sea rápido, habría que almacenar el volumen como un campo. Si la eficiencia de espacio es más importante, habría que calcular el valor. Qué representación es mejor depende completamente del problema concreto. En cualquier caso, deberíamos ser capaces de elegir una implantación independientemente de la vista externa de la clase; en realidad, deberíamos ser capaces incluso de cambiar esta implantación sin ninguna preocupación hacia los clientes.

Empaqueamiento. Aparecen problemas similares en la declaración de clases y objetos dentro de los módulos. Para Smalltalk esto no es un problema, porque no existe el concepto de módulo en el lenguaje. La cosa es distinta para lenguajes como Object Pascal, C++, CLOS y Ada, que soportan la noción de módulo como una construcción separada del lenguaje. Los requerimientos competidores de visibilidad y ocultación de información suelen guiar las decisiones de diseño sobre dónde declarar clases y objetos. Generalmente, se busca construir módulos con cohesión funcional y débilmente acoplados. Hay muchos factores no técnicos que influyen estas decisiones, como cuestiones de reutilización, seguridad y documentación. Al igual que el diseño de clases y objetos, no hay que tomar a la ligera el diseño de módulos. Como hacen notar Parnas, Clements y Weiss respecto a la ocultación de información, «la aplicación de este principio no siempre es fácil. Intenta minimizar el coste esperado del software a lo largo de su periodo de uso y requiere que el diseñador estime la probabilidad de los cambios. Tales estimaciones se basan en experiencias pasadas y normalmente requieren conocimientos sobre el área de la aplicación, así como la comprensión de la tecnología del hardware y el software» [61].

Resumen

- Un objeto tiene estado, comportamiento e identidad.
- La estructura y comportamiento de objetos similares están definidos en su clase común.
- El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades.
- El comportamiento es la forma en que un objeto actúa y reacciona en términos de sus cambios de estado y paso de mensajes.
- La identidad es la propiedad de un objeto que lo distingue de todos los demás objetos.
- Los dos tipos de jerarquías de objetos son los lazos de inclusión y las relaciones de agregación.
- Una clase es un conjunto de objetos que comparten una estructura y comportamiento comunes.
- Los seis tipos de jerarquías de clase son las relaciones de asociación, herencia, agregación, «uso», instanciación y relaciones de metaclas.
- Las abstracciones clave son las clases y objetos que forman el vocabulario del dominio del problema.

Un mecanismo es una estructura por la que un conjunto de objetos trabajan juntos para ofrecer un comportamiento que satisface algún requerimiento del problema.

- La calidad de una abstracción puede medirse por su acoplamiento, cohesión, suficiencia, completud (estado completo) y por el grado hasta el cual es primitiva.

Lecturas recomendadas

MacLennan [G 1982] discute la distinción entre valores y objetos. El trabajo de Meyer [J 1987] propone la idea de programación como contrato.

Se ha escrito mucho sobre el tema de las jerarquías de clases, con énfasis particular sobre enfoques de la herencia y el polimorfismo. Los artículos de Albano [G 1983], Allen [A 1982], Brachman [J 1983], Hailpern y Nguyen [G 1987] y Wegner y Zdonik [J 1988] ofrecen una excelente fundamentación teórica para todos los conceptos y problemas importantes. Cook y Palsberg [J 1989] y Touretzky [G 1986] ofrecen tratamientos formales de la semántica de la herencia. Wirth [J 1987] propone una aproximación relacionada con las extensiones del tipo registro, tal como se usan en el lenguaje Oberon. Ingalls [G 1986] proporciona una útil discusión sobre el polimorfismo múltiple. Grogono [G 1989] estudia la interacción entre polimorfismo y comprobación de tipos, y Ponder y Buch [G 1992] advierten de los peligros del polimorfismo

- incontrolado. Se ofrece una guía práctica sobre el uso efectivo de la herencia por parte de Meyer [G 1988] y Halberd y O'Brien [G 1988]. LaLonde y Pugh [J 1985] examinan los problemas de la enseñanza del uso efectivo de la especialización y la generalización.
- La naturaleza de los papeles y responsabilidades de las abstracciones están más detallados por Rubin y Goldberg [B 1992] y Wirfs-Brock, Wilkerson y Wiener [F 1990]. Las medidas de bondad para el diseño de clases son consideradas asimismo por Coad [F 1991].
- Meyer [G 1986] examina las relaciones entre genericidad y herencia, vistas desde el lenguaje Eiffel. Stroustrup [G 1988] propone un mecanismo para tipos parametrizados en C++. El protocolo de metaobjetos de CLOS es descrito en detalle por Kiczales, Rivieres y Bobrow [G 1991].
- Existe una alternativa a las jerarquías basadas en clases, y es mediante la delegación, utilizando ejemplares. Este enfoque es examinado en detalle por Stein [G 1987].

Notas bibliográficas

- [1] Lefrancois, G. 1977. *Of Children: An Introduction to Child Development*. Second Edition. Belmont, CA: Wadsworth, p. 244-246.
- [2] Nygaard, K. and Dahl, O-J. 1981. The Development of the Simula Languages, in *History of Programming Languages*. New York, NY: Academic Press, p. 462.
- [3] Halbert, D. and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software* vol 4(5), p. 73.
- [4] Smith, M. and Tockey, S. 1988. *An Integrated Approach to Software Requirements Definition Using Objects*. Seattle, WA: Boeing Commercial Airplane Support Division, p. 132.
- [5] Cox, B. 1986. *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley, p. 29.
- [6] MacLennan, B. December 1982. Values and Objects in Programming Languages. *SIGPLAN Notices* vol. 17(12), p. 78.
- [7] Lippman, S. 1989. *C++ Primer*. Reading, MA: Addison-Wesley, p. 185.
- [8] Adams, S. 1993. Private communication.
- [9] Wirfs-Brock, R., Wilkerson, B., and Wiener, L. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey: Prentice Hall, p. 61.
- [10] Rubin, K. 1993. Private communication.
- [11] *Macintosh MacApp 1.1.1 Programmer's Reference*. 1986. Cupertino, CA: Apple Computer, p. 4.
- [12] Khoshafian, S. and Copeland, G. November 1986. Object Identity. *SIGPLAN Notices* vol. 21(11), p. 406.
- [13] Ingalls, D. 1981. Design Principles behind Smalltalk. *Byte* vol. 6(8), p. 290.
- [14] Gall, J. 1986. *Systemantics: How Systems Really Work and How They Fail*. Second Edition. Ann Arbor, MI: The General Systemantics Press, p. 158.
- [15] Seidewitz, E. and Stark, M. 1986. Towards a General Object-Oriented Software Development Methodology. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. NASA Lyndon B. Johnson Space Center, TX: NASA, p. D4.6.4.

-
- [16] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenzen, W. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall, p. 459.
 - [17] Webster's Third New International Dictionary of the English Language, unabridged. 1986. Chicago, Illinois: Merriam-Webster.
 - [18] Strostrup, B. 1991. *The C++ Programming Language*, Second Edition. Reading, MA: Addison-Wesley, p. 422.
 - [19] Meyer, B. 1987. *Programming as Contracting*. Report TR-EI-12/CO. Goleta, CA: Interactive Software Engineering.
 - [20] Snyder, A. November 1986. Encapsulation and Inheritance in Object-Oriented Programming Languages. *SIGPLAN Notices* vol. 11(2), p. 214.
 - [21] LaLonde, W. April 1989. Designing Families of Data Types Using Exemplars. *ACM Transactions on Programming Languages and Systems* vol. 11(2), p. 214.
 - [22] Rumbaugh, J. April 1988. Relational Database Design Using an Object-Oriented Methodology. *Communications of the ACM* vol. 31(4), p. 417.
 - [23] Lieberman, H. November 1986. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *SIGPLAN Notices* vol. 21(11).
 - [24] Rumbaugh, 1991, p. 312.
 - [25] Brachman, R. October 1983. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Computer* vol. 16(10), p. 30.
 - [26] Micallef, J. April/May 1988. Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages. *Journal of Object-Oriented Programming* vol. 1(1), p. 15.
 - [27] Snyder. Encapsulation, p. 39.
 - [28] Cardelli, L. and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. December 1985. *ACM Computing Surveys* vol. 17(4), p. 475.
 - [29] As quoted in Harland, D., Szypiewski, M., and Wainwright, J. October 1985. An Alternative View of Polymorphism. *SIGPLAN Notices* vol. 20(10).
 - [30] Deutsch, P. 1983. Efficient Implementation of the Smalltalk-80 System, in *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, p. 300.
 - [32] Ibid., p. 299.
 - [33] Duff, C. August 1986. Designing and Efficient Language. *Byte* vol. 11(8), p. 216.
 - [34] Strostrup, B. 1988. Private communications.
 - [35] Strostrup, B. November 1987. Possible Directions for C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, New Mexico, p. 8.
 - [36] Keene, S. 1989. *Object-Oriented Programming in Common Lisp*. Reading, MA: Addison-Wesley, p. 44.
 - [37] Winston, P. and Horn, B. 1989. *Lisp*. Third Edition. Reading: MA: Addison-Wesley, p. 510.
 - [38] Micallef, J. April/May 1988. Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages. *Journal of Object-Oriented Programming* vol. 1(1), p. 25.
 - [39] Snyder, Encapsulation, p. 41.
 - [40] Vlissedes, J. and Linton, M. 1988. Applying Object-Oriented Design to Structures Graphics. *Proceedings of USENIX C++ Conference* Berkeley, CA: USENIX Association, p. 93.
 - [41] Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall, p. 274.

- [42] Keene. *Object-Oriented Programming*, p. 118.
- [43] Snyder. Encapsulation, p. 43.
- [44] Hendler, J. October 1986. Enhancement for Multiple Inheritance. *SIGPLAN Notices* vol. 21(10), p. 100.
- [45] Stroustrup, 1987, p. 3.
- [46] Stroustrup, B. 1988. Parameterized Types for C++. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association, p. 1.
- [48] Stroustrup, 1988, p. 4.
- [49] Robson, D. August 1981. Object-Oriented Software Systems. *Byte* vol. 6(8), p. 86.
- [50] Goldberg, A. and Robson, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley, p. 287.
- [51] Ingalls, D. August 1981. Design Principles behind Smalltalk. *Byte* vol. 6(8), p. 286.
- [52] Stevens, W., Myers, G., and Constantine, L. 1979. Structured Design, in *Classics in Software Engineering*. New York, NY: Yourdon Press, p. 209.
- [53] Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press, p. 59.
- [54] Meyer. 1987, p. 4.
- [55] Halbert, D. and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software* vol. 4(5), p. 74.
- [56] Sakkinen, M. December 1988. Comments on "the Law of Demeter" and C++. *SIGPLAN Notices* vol. 23(12), p. 38.
- [57] Lea, D. August 12, 1988. *User's Guide to GNU C++ Library*. Cambridge, MA: Free Software Foundation, p. 12.
- [58] Ibid.
- [59] Meyer. 1988, p. 332.
- [60] Wirth, M. 1986. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall, p. 37.
- [61] Keene. *Object-Oriented Programming*, p. 68.
- [62] Parnas, D., Clements, P., and Weiss, D. 1989. Enhancing Reusability with Information Hiding. *Software Reusability*. New York, NY: ACM Press, p. 143.

Clasificación

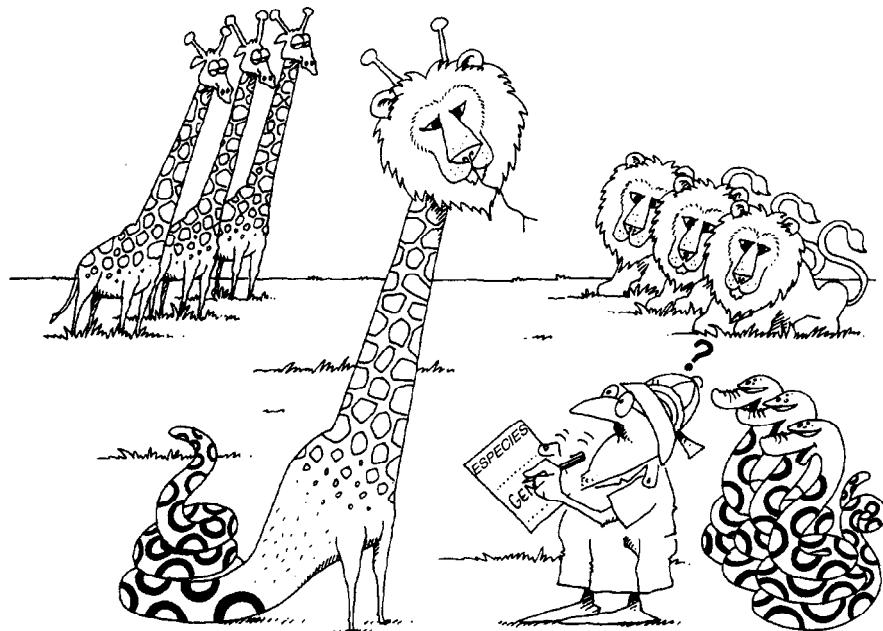
La clasificación es el medio por el que ordenamos el conocimiento. En el diseño orientado a objetos, el reconocimiento de la similitud entre las cosas nos permite exponer lo que tienen en común en abstracciones clave y mecanismos, y eventualmente nos lleva a arquitecturas más pequeñas y simples. Desgraciadamente, no existe un camino trillado hacia la clasificación. Para el lector acostumbrado a encontrar respuestas en forma de receta, afirmamos inequívocamente que no hay recetas fáciles para identificar clases y objetos. No existe algo que podamos llamar una estructura de clases «perfecta», ni el conjunto de objetos «correcto». Al igual que en cualquier disciplina de ingeniería, nuestras elecciones de diseño son un compromiso conformado por muchos factores que compiten.

En una conferencia sobre ingeniería del software, se preguntó a varios desarrolladores qué reglas aplicaban para identificar clases y objetos. Stroustrup, el diseñador del C++, respondió: «Eso es como el Santo Grial. No hay panaceas.» Gabriel, uno de los diseñadores del CLOS, afirmó: «Esa es una pregunta fundamental para la que no hay respuesta sencilla. Intento cosas» [1]. Afortunadamente, existe un vasto legado de experiencia sobre la clasificación en otras disciplinas. Partiendo de enfoques más clásicos, surgieron técnicas de análisis orientado a objetos que ofrecen varias recomendaciones prácticas y reglas útiles para identificar las clases y objetos relevantes para un problema concreto. Estos heurísticos centran el interés de este capítulo.

4.1. La importancia de una clasificación correcta

Clasificación y diseño orientado a objetos

La identificación de clases y objetos es la parte más difícil del diseño orientado a objetos. La experiencia muestra que la identificación implica descubrimiento



La clasificación es el medio por el cual ordenamos el conocimiento.

e invención. Mediante el descubrimiento, se llega a reconocer las abstracciones clave y los mecanismos que forman el vocabulario del dominio del problema. Mediante la invención, se idean abstracciones generalizadas así como nuevos mecanismos que especifican cómo colaboran los objetos. En última instancia, el descubrimiento y la invención son ambos problemas de clasificación, y la clasificación es fundamentalmente un problema de hallar analogías. Cuando se clasifica, se persigue agrupar cosas que tienen una estructura común o exhiben un comportamiento común.

La clasificación inteligente es en realidad una parte de cualquier ciencia verdadera. Como observan Michalski y Stepp, «un problema omnipresente en ciencia es construir clasificaciones significativas de objetos o situaciones observados. Tales clasificaciones facilitan la comprensión humana de las observaciones y el subsecuente desarrollo de una teoría científica» [2]. La misma filosofía se aplica a la ingeniería. En el dominio de la arquitectura de la construcción y del urbanismo, Alexander hace notar que, para el arquitecto, «su acto de diseño, ya sea humilde o enormemente complejo, está completamente gobernado por los patrones que tiene en la mente en ese momento, y su capacidad para combinar esos patrones para formar un nuevo diseño» [3]. No es de extrañar, pues, que la clasificación sea relevante para todos los aspectos del diseño orientado a objetos. La clasificación ayuda a identificar jerarquías de generalización, especialización y agregación entre clases. Reconociendo los patrones comunes

de interacción entre objetos, se llegan a idear mecanismos que sirven como alma de la implantación. La clasificación también proporciona una guía para tomar decisiones sobre modularización. Se puede decidir ubicar ciertas clases y objetos juntos en el mismo módulo o en módulos diferentes, dependiendo de la similitud que se encuentra entre estas declaraciones; el acoplamiento y la cohesión son simplemente medidas de esta similitud. La clasificación también desempeña un papel en la asignación de procesos a los procesadores. Se ubican ciertos procesos juntos en el mismo procesador o en diferentes procesadores, dependiendo de intereses de empaquetamiento, eficacia o fiabilidad.

La dificultad de la clasificación

Ejemplos de clasificación. En el capítulo anterior se definió un objeto como algo que tiene una frontera definida con nitidez. Sin embargo, las fronteras que distinguen un objeto de otro son a menudo bastante difusas. Por ejemplo, fíjese el lector en su pierna. ¿Dónde comienza la rodilla, y dónde termina? En el reconocimiento del habla humana, ¿cómo saber que ciertos sonidos se conectan para formar una palabra, y no son en realidad parte de otras palabras circundantes? Considérese también el diseño de un sistema de procesamiento de textos. ¿Constituyen los caracteres una clase, o son las palabras completas una mejor elección? ¿Cómo tratar selecciones arbitrarias, no contiguas de texto? Y qué hay sobre las oraciones, párrafos, o incluso documentos completos: ¿son relevantes para el problema estas clases de objetos?

El hecho de que la clasificación inteligente es difícil no es en absoluto una noticia nueva. Puesto que existen paralelismos con los mismos problemas en el diseño orientado a objetos, considérense por un momento los problemas de la clasificación en otras dos disciplinas científicas: biología y química.

Hasta el siglo dieciocho, la opinión científica más extendida era que todos los organismos vivos podían clasificarse del más simple hasta el más complejo, siendo la medida de la complejidad algo muy subjetivo (no es de extrañar que los humanos fuesen situados habitualmente en lo más alto de esta lista). A mediados del siglo dieciocho, sin embargo, el botánico sueco Carl von Linneo sugirió una taxonomía más detallada para categorizar los organismos, de acuerdo con lo que se llama *géneros* y *especies*. Un siglo más tarde, Darwin propuso la teoría de que la selección natural fue el mecanismo de la evolución, en virtud de la cual las especies actuales evolucionaron de otras más antiguas. La teoría de Darwin dependía de una clasificación inteligente de las especies. Como el propio Darwin afirma, los naturalistas «intentan disponer las especies, géneros y familias en cada clase, en lo que se denomina el sistema natural. Pero ¿qué es lo que se quiere decir con este sistema? Algunos autores lo toman como un mero esquema para poner juntos aquellos objetos vivos que son más parecidos, y para separar los que son más distintos» [4]. En la biología contemporánea, la clasificación denota «el establecimiento de un sistema jerárquico de categorías sobre las bases de presuntas relaciones naturales entre organismos» [5]. La categoría

más general en una taxonomía biológica es el reino, seguido en orden de especialización creciente por el filum, subfilum, clase, orden, familia, género y, finalmente, especie. Históricamente, un organismo concreto se sitúa en una categoría específica de acuerdo con su estructura corporal, características estructurales internas y relaciones evolutivas. Más recientemente, la clasificación se ha enfocado como la agrupación de organismos que comparten una herencia genética común: los organismos que tienen ADN similar se incluyen en el mismo grupo. La clasificación por ADN es útil para distinguir organismos que son estructuralmente similares, pero genéticamente muy diferentes. Por ejemplo, las investigaciones actuales sugieren que el pez pulmón y la vaca tienen una relación más cercana que el pez pulmón y la trucha [6].

Para un informático, la biología puede parecer una disciplina madura hasta la pesadez, con criterios bien definidos para clasificar organismos. Esto simplemente no es verdad. Como indica el biólogo May, «a nivel puramente de hechos, no sabemos ni siquiera dentro de un orden de magnitud con cuántas especies de plantas y animales compartimos el globo: actualmente se han clasificado menos de 2 millones, y las estimaciones del número total varían entre menos de 5 millones a más de 50 millones» [7]. Además, criterios diferentes para clasificar los mismos organismos arrojan resultados distintos. Martin sugiere que «todo depende de para qué quiere uno la clasificación. Si se desea reflejar con precisión las relaciones genéticas entre las especies, ofrecerá una respuesta. Pero si en vez de eso se quiere decir algo sobre niveles de adaptación, se obtendrá una respuesta distinta» [8]. La moraleja de todo esto es que incluso en disciplinas rigurosamente científicas, la clasificación es altamente dependiente de la razón por la que se clasifica.

Las mismas enseñanzas pueden aprenderse de la química [9]. En tiempos remotos, se creía que todas las sustancias eran una combinación de tierra, aire, fuego y agua. Para los estándares actuales (a menos que uno sea un alquimista) esto no representa una clasificación muy buena. A mediados del siglo diecisiete, el químico Robert Boyle propuso que los elementos eran las abstracciones primivas de la química, a partir de las cuales podían realizarse compuestos más complejos. No fue hasta un siglo después, en 1789, cuando el químico Lavoisier publicó la primera lista de elementos, que contenía alrededor de veintitrés de ellos, algunos de los cuales se descubrió posteriormente que no eran elementos en absoluto. El descubrimiento de nuevos elementos continuó y la lista creció, pero finalmente, en 1869, el químico Mendeleiev propuso la ley periódica que proporcionó un criterio preciso para organizar todos los elementos conocidos, y pudo predecir las propiedades de elementos aún sin descubrir. La ley periódica no era el final de la historia de la clasificación de los elementos. A principios del siglo veinte, se descubrieron elementos con propiedades químicas similares, pero pesos atómicos diferentes, conduciendo a la idea de los isótopos de los elementos.

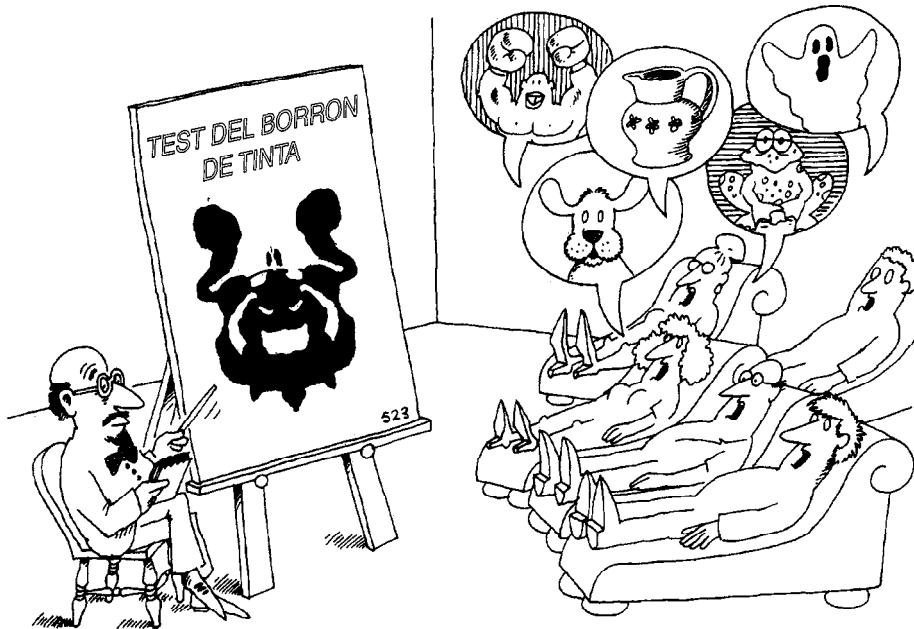
La lección aquí es simple: como afirma Descartes, «el descubrimiento de un orden no es tarea fácil...; sin embargo, una vez que se ha descubierto el orden, no hay dificultad alguna en comprenderlo» [10]. Los mejores diseños de soft-

ware parecen simples, pero, como muestra la experiencia, exige gran cantidad de esfuerzo el diseño de una arquitectura simple.

La naturaleza incremental e iterativa de la clasificación. No se ha dicho todo esto para defender planificaciones de desarrollo de software de mucha duración, aunque para el director o el usuario final, sí que parece algunas veces que los ingenieros del software necesitan siglos para completar su trabajo. Más bien se han contado estas historias para poner de relieve que la clasificación inteligente es un trabajo intelectualmente difícil, y que la mejor forma de realizarlo es a través de un proceso incremental e iterativo. Esta naturaleza incremental e iterativa es evidente en el desarrollo de tecnologías de software tan diversas como los interfaces gráficos de usuario, los estándares de bases de datos e incluso los lenguajes de cuarta generación. Como Shaw ha observado en ingeniería del software, «el desarrollo de abstracciones individuales sigue frecuentemente un patrón común. Primero, los problemas se resuelven *ad hoc*. A medida que se acumula la experiencia, se va viendo que algunas soluciones funcionan mejor que otras, y se transfiere informalmente una especie de folklore de persona a persona. Eventualmente, las soluciones útiles se comprenden de forma más sistemática, y se codifican y analizan. Esto permite el desarrollo de modelos que admiten una implantación automática y de teorías que permiten generalizar la solución. Esto a su vez da lugar a un nivel de práctica más sofisticado y nos permite atacar problemas más difíciles, a los que con frecuencia se brinda un enfoque *ad hoc*, comenzando el ciclo de nuevo» [11].

La naturaleza incremental e iterativa de la clasificación tiene un impacto directo en la construcción de jerarquías de clases y objetos en el diseño de un sistema de software complejo. En la práctica, es común establecer una determinada estructura de clases en fases tempranas del diseño y revisar entonces esa estructura a lo largo del tiempo. Sólo en etapas más avanzadas del diseño, una vez que se han construido los clientes que utilizan tal estructura, se puede evaluar de forma significativa la calidad de la clasificación. Sobre la base de esta experiencia, se puede decidir crear nuevas subclases a partir de otras existentes (derivación). Se puede dividir una clase grande en varias más pequeñas (factorización), o crear una clase mayor uniendo otras más pequeñas (composición). Ocasionalmente, se puede incluso descubrir aspectos comunes que habían pasado desapercibidos, e idear una nueva clase (abstracción) [12].

¿Por qué, entonces, es tan difícil la clasificación? Sugerimos que existen dos razones importantes. Primero, no existe algo que pueda llamarse una clasificación «perfecta», aunque por supuesto, algunas clasificaciones son mejores que otras. Según dicen Coombs, Raffia y Thrall, «potencialmente, hay al menos tantas formas de dividir el mundo en sistemas de objetos como científicos para emprender esa tarea» [13]. Cualquier clasificación es relativa a la perspectiva del observador que la realiza. Flood y Carson ponen el ejemplo de que el Reino Unido «podría ser visto como una economía por los economistas, como una sociedad por los sociólogos, como un trozo de naturaleza amenazada desde el punto de vista de los ecologistas, como una atracción turística para algunos



Diferentes observadores pueden clasificar el mismo objeto de distintas formas.

americanos, como una amenaza militar para los gobernantes de la Unión Soviética, y como la verde, verde hierba del hogar para el más romántico de entre nosotros los británicos» [14]. Segundo, la clasificación inteligente requiere una tremenda cantidad de perspicacia creativa. Birtwistle, Dahl, Myhrhaug y Nygard observan que «a veces la respuesta es evidente, a veces es una cuestión de gustos, y otras veces, la selección de componentes adecuados es un punto crucial del análisis» [15]. Este hecho recuerda una adivinanza: «¿En qué se parecen un rayo láser y un pez de la familia de las carpas doradas? ...en que ninguno de los dos puede silbar» [16]. Sólo una mente creativa puede encontrar similitudes entre cosas tan poco relacionadas entre sí.

4.2. Identificando clases y objetos

Enfoques clásicos y modernos

El problema de la clasificación ha sido del interés de innumerables filósofos, lingüistas, científicos del conocimiento y matemáticos, incluso desde antes de Platón. Es razonable estudiar sus experiencias y aplicar lo aprendido al diseño

orientado a objetos. Históricamente, sólo han existido tres aproximaciones generales a la clasificación:

- Categorización clásica.
- Agrupamiento conceptual.
- Teoría de prototipos [17]

Categorización clásica. En la aproximación clásica a la categorización, «Todas las entidades que tienen una determinada propiedad o colección de propiedades en común forman una categoría. Tales propiedades son necesarias y suficientes para definir la categoría» [18]. Por ejemplo, las personas casadas constituyen una categoría: o se está casado o no se está, y el valor de esta propiedad es suficiente para decidir a qué grupo pertenece determinada persona. Por otra parte, las personas altas no forman una categoría, a menos que pueda haber un acuerdo respecto a algún criterio absoluto por el que se distinga la propiedad alto de la propiedad bajo.

Un problema de clasificación

La Figura 4.1 contiene diez elementos, etiquetados de *A* a *J*, cada uno de los cuales representa un tren. Cada tren contiene una máquina (a la derecha) y de dos a cuatro vagones, de diferentes formas y con distintas cargas. Antes de seguir leyendo, invierta el lector los próximos minutos en disponer esos trenes en cualquier número de grupos que considere significativos. Por ejemplo, podría crear tres grupos: uno para trenes cuyas máquinas tienen ruedas negras, uno para trenes cuyas máquinas tienen ruedas blancas, y otro para trenes cuyas máquinas tienen ruedas blancas y ruedas negras.

Este problema proviene del trabajo de Stepp y Michalski sobre agrupamiento conceptual [19]. Como en la vida real, no hay respuesta «correcta». En sus experimentos, los sujetos aportaron alrededor de noventa y tres clasificaciones diferentes. La más popular fue la basada en la longitud del tren, formando tres grupos (trenes con dos, tres y cuatro vagones). La segunda más popular fue por el color de las ruedas de la máquina, como la que se ha sugerido. De las noventa y tres, alrededor de cuarenta eran totalmente únicas.

Nuestro uso de este ejemplo confirma el trabajo de Stepp y Michalski. La mayoría de nuestros sujetos han utilizado las dos clasificaciones más populares, aunque hemos encontrado algunas agrupaciones más creativas. Por ejemplo, un sujeto dispuso esos trenes en dos grupos: uno representaba trenes etiquetados con letras que contenían líneas rectas (*A*, *E*, *F*, *H* e *I*) y el otro grupo representaba trenes etiquetados con letras que contenían líneas curvas. Este es verdaderamente un ejemplo de pensamiento no lineal: creativo, aunque sea extraño.

Una vez que usted haya completado esta tarea, cambiemos los requerimientos (de nuevo, como en la vida real). Supóngase que los círculos representan sustancias tóxicas, los rectángulos representan madera, y todas las demás formas representan pasajeros. Intente clasificar los trenes de nuevo, y vea cómo este nuevo conocimiento cambia su clasificación.

Entre nuestros sujetos, el agrupamiento de los trenes cambió de forma signifi-

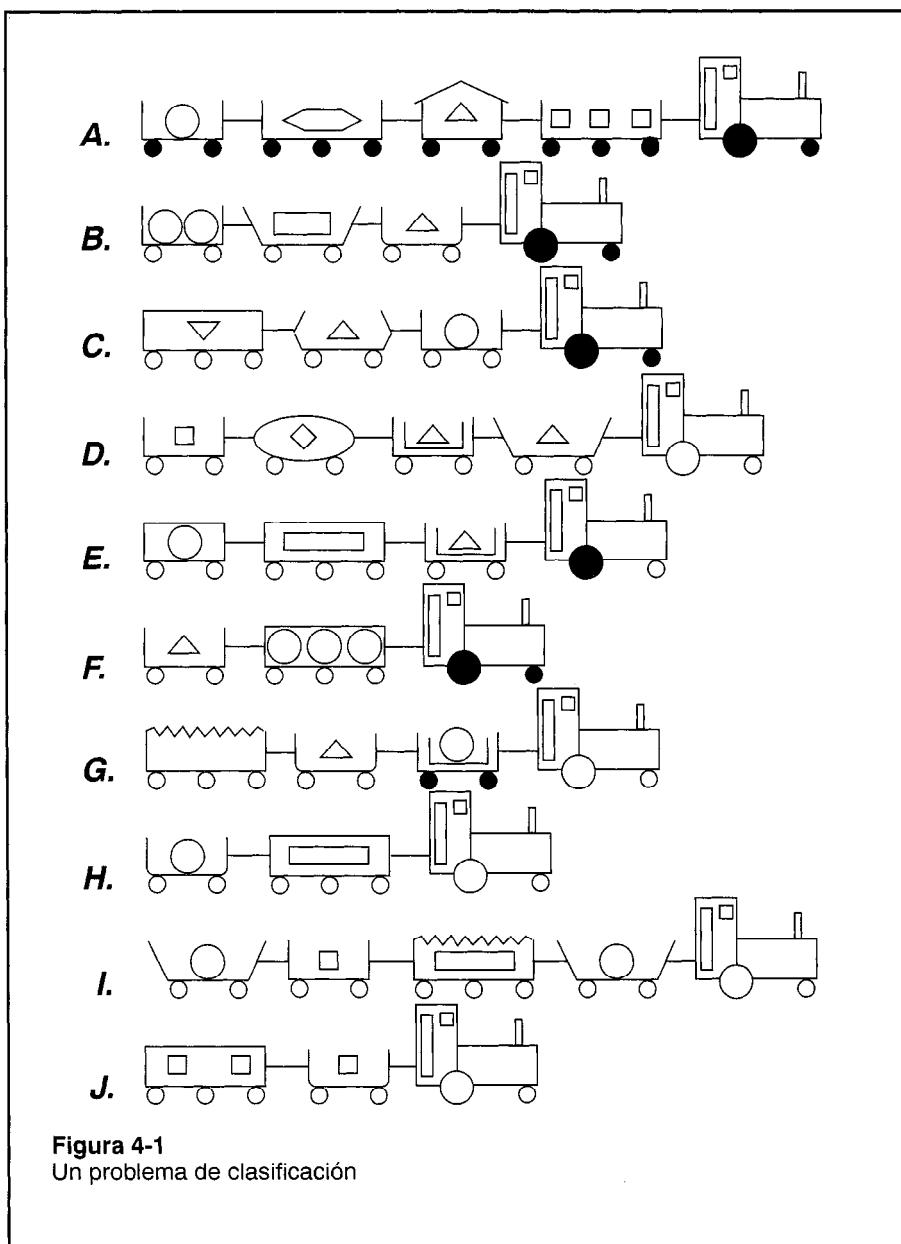
cativa. La mayoría de ellos clasificó los trenes según transportasen o no cargas tóxicas. De este simple experimento se concluye que un mayor conocimiento significativo sobre un dominio facilita, hasta cierto punto, el conseguir una clasificación inteligente.

La *categorización clásica* proviene en primer lugar de Platón, y después de Aristóteles por medio de su clasificación de plantas y animales, en la que utiliza una técnica bastante parecida a la del juego infantil de las Veinte Preguntas (¿Es animal, mineral o vegetal? ¿Tiene pelo o tiene plumas? ¿Puede volar? ¿Tiene olfato?) [20]. Filósofos posteriores, entre los que destacan Aquino, Descartes y Locke, adoptaron este enfoque. Como afirmó Aquino, «podemos nombrar una cosa según el conocimiento que tenemos sobre su naturaleza a partir de sus propiedades y efectos» [21].

La aproximación clásica a la categorización se refleja también en las teorías modernas sobre el desarrollo de los niños. Piaget observó que, alrededor de la edad de un año, el niño suele desarrollar el concepto de permanencia de los objetos; poco después, adquiere la capacidad de clasificar esos objetos, utilizando al principio categorías básicas como perros, gatos y juguetes [22]. Más tarde, el niño descubre categorías más generales (como animales) y más específicas (como sabuesos) [23].

Para resumir, la aproximación clásica emplea propiedades relacionadas como criterio de similitud entre objetos. Concretamente, se puede dividir los objetos en conjuntos disjuntos dependiendo de la presencia o ausencia de una propiedad particular. Minsky sugiere que «los conjuntos más útiles de propiedades son aquellas cuyos miembros no interactúan demasiado. Esto explica la popularidad universal de esta combinación particular de propiedades: tamaño, color, forma y sustancia. Ya que esos atributos interactúan entre sí muy escasamente, pueden juntarse en cualquier combinación imaginable para hacer un objeto que puede ser grande o pequeño, rojo o verde, de madera o de cristal, y tener forma de esfera o cubo» [24]. En sentido general, las propiedades pueden denotar algo más que meras características medibles; pueden también abarcar comportamientos observables. Por ejemplo, el hecho de que un pájaro pueda volar pero un pez no pueda es una propiedad que distingue un águila de un salmón.

Las propiedades particulares que habría que considerar en una situación dada dependen mucho del dominio. Por ejemplo, el color de un coche puede ser importante para el propósito de un control de inventario en una planta de fabricación de automóviles, pero no es relevante en absoluto para el software que controla los semáforos en un área metropolitana. Ésta es de hecho la razón por la que se dice que no hay medidas absolutas para la clasificación, aunque una estructura de clases determinada puede ser más adecuada para una aplicación que para otra. Como sugiere James, «ningún esquema de clasificación representa el orden o estructura real de la naturaleza mejor que los demás. La naturaleza se somete con indiferencia a cualquier división que se desee hacer sobre las cosas existentes. Algunas clasificaciones pueden ser más significativas que



otras, pero sólo respecto a nuestros intereses, no porque representen la realidad de forma más exacta o adecuada» [25].

La categorización clásica impregna gran parte del pensamiento contemporáneo occidental, pero tal como sugiere el ejemplo mencionado de clasificación de personas altas y bajas, este enfoque no siempre es satisfactorio. Kosok observa que «las categorías naturales tienden a ser confusas: la mayoría de los pájaros vuela, pero algunos no lo hacen. Las sillas pueden estar hechas de madera, plástico o metal, y pueden tener casi cualquier número de patas, según el capricho del diseñador. Parece prácticamente imposible proponer una lista de propiedades para cualquier categoría natural que excluya a todos los ejemplos que no están en la categoría e incluya a todos los que sí están» [26]. Éstos son realmente problemas fundamentales de la categorización clásica, que el agrupamiento conceptual y la teoría de prototipos intentan resolver.

Agrupamiento conceptual. El *agrupamiento conceptual*¹ es una variación más moderna del enfoque clásico, y deriva en gran medida de los intentos de explicar cómo se representa el conocimiento. Como afirman Stepp y Michalski, «en este enfoque, las clases (agrupaciones de entidades) se generan formulando primero descripciones conceptuales de estas clases y clasificando entonces las entidades de acuerdo con las descripciones» [27]. Por ejemplo, se puede establecer un concepto como «una canción de amor». Éste es más bien un concepto que una propiedad, porque la «cantidad de amor» de cualquier canción no es algo que se pueda medir empíricamente. Sin embargo, si se decide que cierta canción tiene más de canción de amor que de otra cosa, se la coloca en esta categoría. Así, el agrupamiento conceptual representa más bien un agrupamiento probabilístico de los objetos.

El agrupamiento conceptual está en estrecha relación con la teoría de conjuntos difusos (multivalor), en la que los objetos pueden pertenecer a uno o más grupos, en diversos grados de adecuación. El agrupamiento conceptual realiza juicios absolutos de la clasificación centrándose en la «mayor adecuación».

Teoría de prototipos. La categorización clásica y el agrupamiento conceptual son suficientemente expresivos para utilizarse en la mayoría de las clasificaciones que se necesite realizar en el diseño de sistemas de software complejos. Sin embargo, existen aún algunas situaciones en las que estas aproximaciones no son adecuadas. Esto conduce al enfoque más reciente de la clasificación, llamado *teoría de prototipos*, que deriva principalmente del trabajo de Rosch y sus colegas en el campo de la psicología cognitiva [28].

Existen algunas abstracciones que no tienen ni propiedades ni conceptos delimitados con claridad. Tomando la explicación de Lakoff sobre este problema, «Wittgenstein apuntó que una categoría como juego no encaja en el molde clásico, porque no hay propiedades comunes compartidas por todos los juegos...

¹ En el original en inglés *Conceptual clustering* (N. del T.)

Aunque no hay una sola colección de propiedades que compartan todos los juegos, la categoría de los juegos está unida por lo que Wittgenstein llama parecidos familiares... Wittgenstein observó también que no había una frontera fijada para la categoría juego. La categoría podía extenderse, podían introducirse nuevos tipos de juegos, siempre que se pareciesen a juegos anteriores de forma apropiada» [29]. Ésta es la razón por la que el enfoque se denomina *teoría de prototipos*: una clase de objetos se representa por un objeto prototípico, y se considera un objeto como un miembro de esta clase si y sólo si se parece a este prototipo de forma significativa.

Lakeoff y Johnson aplican la teoría de prototipos al problema anterior de la clasificación de sillas. Observan que «entendemos que las sillas para jugar al bingo («con cojín»), las sillas de barbero y las sillas de diseño son sillas, no porque comparten algún conjunto de propiedades definitorias con el prototipo, sino más bien porque mantienen suficiente parecido familiar con el prototipo... No hace falta que exista un núcleo fijo de propiedades de las sillas prototípicas que comparten la silla de jugar al bingo («con cojín») y la silla de barbero, sino que son ambas sillas porque, cada una a su manera, está lo bastante cerca del prototipo. Las propiedades de interacción son sobresalientes entre los tipos de propiedades que cuentan a la hora de determinar si hay suficiente parecido familiar» [30].

Esta noción de propiedades de interacción está en el centro de los conceptos de la teoría de prototipos. En agrupamiento conceptual, se agrupan las cosas según el grado de su relación con prototipos concretos.

Aplicación de teorías clásicas y modernas. Para el desarrollador que se encuentra en las trincheras, luchando contra requerimientos cambiantes y rodeado por recursos limitados y plazos muy breves, esta discusión puede parecer algo muy alejado de los campos de batalla reales. Verdaderamente, estas tres aproximaciones a la clasificación tienen aplicación directa en el diseño orientado a objetos.

Según nuestra experiencia, identificamos las clases y objetos en primer lugar de acuerdo con las propiedades relevantes para nuestro dominio particular. Aquí se hace hincapié en la identificación de las estructuras y comportamiento que son parte del vocabulario del espacio del problema. Muchas de tales abstracciones suelen estar a nuestra disposición [31]. Si este enfoque fracasa en la producción de una estructura de clases satisfactoria, hay que pasar a considerar la agrupación de objetos por conceptos. Aquí se concentra la atención en el comportamiento de objetos que colaboran. Si ambos intentos fallan al capturar nuestra comprensión del dominio del problema, entonces se considera la clasificación por asociación, a través de la cual las agrupaciones de objetos se definen según el grado en el que cada uno se parece a algún objeto prototípico.

Más directamente, éstos tres enfoques de la clasificación proporcionan el fundamento teórico del análisis orientado a objetos, que ofrece una serie de prácticas y reglas pragmáticas que se pueden aplicar para identificar clases y objetos en el diseño de un sistema de software complejo.

Análisis orientado a objetos

Las fronteras entre análisis y diseño son difusas, aunque el objetivo principal de ambos es bastante diferente. En el análisis se persigue modelar el mundo *descubriendo* las clases y objetos que forman el vocabulario del dominio del problema, y en el diseño se *inventan* las abstracciones y mecanismos que proporcionan el comportamiento que este modelo requiere².

En las siguientes secciones, se examinan varios enfoques contrastados para el análisis que son de relevancia para los sistemas orientados a objetos.

Enfoques clásicos Hay una serie de diseñadores de metodologías que han propuesto varias fuentes de clases y objetos, derivadas de los requerimientos del dominio del problema. Se llama a estos enfoques *clásicos* porque derivan sobre todo de los principios de la categorización clásica.

Por ejemplo, Shlaer y Mellor sugieren que las clases y objetos candidatos provienen habitualmente de una de las fuentes siguientes [32]:

- | | |
|--|--|
| <ul style="list-style-type: none"> • Cosas tangibles • Papeles (roles) • Eventos • Interacciones | <ul style="list-style-type: none"> Coches, datos de telemetría, sensores de presión. Madre, profesor, político. Aterrizaje, interrupción, petición. Préstamo, reunión, intersección. |
|--|--|

Desde la perspectiva del modelado de bases de datos, Ross ofrece una lista similar [33]:

- | | |
|--|---|
| <ul style="list-style-type: none"> • Personas • Lugares • Cosas • Organizaciones • Conceptos • Eventos | <ul style="list-style-type: none"> Humanos que llevan a cabo alguna función. Áreas reservadas para personas o cosas. Objetos físicos, o grupos de objetos, que son tangibles. Colecciones formalmente organizadas de personas, recursos, instalaciones y posibilidades que tienen una misión definida, cuya existencia es, en gran medida, independiente de los individuos Principios o ideas no tangibles <i>per se</i>; utilizados para organizar o llevar cuenta de actividades de negocios y/o comunicaciones. Cosas que suceden, habitualmente a alguna otra cosa, en una fecha y hora concretas, o como pasos dentro de una secuencia ordenada. |
|--|---|

² La notación y proceso descritos en este libro son perfectamente aplicables a las fases tradicionales del desarrollo de análisis y diseño, como se trata más adelante en el Capítulo 6. Realmente, es ésta la razón por la que se ha cambiado el título de esta segunda edición y se ha denominado *Análisis y diseño orientado a objetos*.

Coad y Yourdon sugieren otro conjunto más de fuentes de objetos potenciales [34]:

- | | |
|---|--|
| <ul style="list-style-type: none"> ● Estructuras ● Otros sistemas
 ● Dispositivos
 ● Eventos recordados ● Papeles desempeñados
 ● Posiciones
 ● Unidades de organización | Relaciones «de clases» y «de partes».
Sistemas externos con los que la aplicación interactúa.
Dispositivos con los que la aplicación interactúa.
Sucesos históricos que hay que registrar.
Los diferentes papeles que juegan los usuarios en su interacción con la aplicación.
Ubicaciones físicas, oficinas y lugares importantes para la aplicación.
Grupos a los que pertenecen los usuarios. |
|---|--|

A un nivel alto de abstracción, Coad introduce la idea de áreas temáticas, que son básicamente grupos lógicos de clases que se relacionan con alguna función de nivel superior.

Análisis del comportamiento. Mientras estos enfoques clásicos se centran en cosas tangibles del dominio del problema, hay otra escuela de pensamiento en el análisis orientado a objetos que se centra en el comportamiento dinámico como fuente primaria de clases y objetos³. Estos enfoques tienen más que ver con el agrupamiento conceptual: se forman clases basadas en grupos de objetos que exhiben comportamiento similar.

Wirfs-Brock, por ejemplo, hace hincapié en las responsabilidades, que denotan «el conocimiento que un objeto tiene y las acciones que un objeto puede realizar. Las responsabilidades están encaminadas a comunicar una expresión del propósito de un objeto y su lugar en el sistema. Las responsabilidades de un objeto son todos los servicios que suministra para todos los contratos que soporta» [36]. De este modo, se agrupan cosas que tienen responsabilidades comunes, y se forman jerarquías de clases que involucran a superclases que incorporan responsabilidades generales y subclases que especializan su comportamiento.

Rubin y Goldberg ofrecen una aproximación a la identificación de clases y objetos derivados de funciones del sistema. Tal como proponen, «el enfoque que utilizamos pone en primer lugar el énfasis en la comprensión de lo que sucede en el sistema. Éstos son los comportamientos del sistema. Seguidamente asignamos estos comportamientos a partes del sistema, y tratamos de entender quién inicia esos comportamientos y quién participa en ellos... Los iniciadores y los participantes que desempeñan papeles significativos se reconocen como objetos, y se les asignan las responsabilidades de actuación para esos papeles» [37].

El concepto de Rubin acerca del comportamiento de un sistema está en re-

³ Shlaer y Mellor han extendido sus trabajos previos para centrarse también en el comportamiento. En particular, estudian el ciclo de vida de cada objeto como una vía para comprender sus fronteras [35].

lación estrecha con la idea de punto funcional, introducida en 1979 por Albrech. Un punto funcional «se define como una función de las actividades* del usuario final» [38]. Una función de actividades representa algún tipo de salida, pregunta, entrada, archivo o interfaz. Aunque en esta definición se adivinan sus raíces en el ámbito de los sistemas de información, la idea de punto funcional se generaliza a todo tipo de sistema automatizado: un punto funcional es cualquier comportamiento apreciable exteriormente y comprobable del sistema.

Análisis de dominios. Los principios que se han discutido hasta aquí se aplican típicamente al desarrollo de aplicaciones concretas e independientes. El análisis de dominios, por contra, busca identificar las clases y objetos comunes a todas las aplicaciones dentro de un dominio dado, tales como mantenimiento de registros de pacientes, operaciones bursátiles, compiladores o sistemas de aviónica para misiles. Si uno está a mitad de un diseño y le faltan ideas sobre las abstracciones clave que existen, un pequeño análisis del dominio puede ayudarnos indicándonos las abstracciones clave que han demostrado ser útiles en otros sistemas relacionados con el nuestro. El análisis de dominios funciona bien porque, excepto en situaciones especiales, existen muy pocos tipos de sistemas de software que sean verdaderamente únicos.

La idea del análisis de dominios fue introducida por Neighbors. Se define el análisis de dominios como «un intento de identificar los objetos, operaciones y relaciones que los expertos del dominio consideran importantes acerca del mismo» [39]. Moore y Bailin sugieren los siguientes pasos en el análisis de dominios:

- «Construir un modelo genérico “fantasma” del dominio consultando con expertos de ese dominio.
- Examinar sistemas existentes en el dominio y representar esta comprensión en un formato común.
- Identificar analogías y diferencias entre los sistemas consultando con expertos del dominio.
- Refinar el modelo genérico para acomodar sistemas ya existentes» [40].

El análisis de dominios puede aplicarse entre aplicaciones similares (análisis vertical de dominios), así como entre partes relacionadas de la misma aplicación (análisis horizontal de dominios). Por ejemplo, cuando se comienza a diseñar un nuevo sistema de monitorización de pacientes, es razonable examinar la arquitectura de sistemas existentes para comprender qué abstracciones y mecanismos clave se emplearon anteriormente y evaluar cuáles fueron útiles y cuáles no. Del mismo modo, un sistema de contabilidad debe proporcionar muchos tipos diferentes de informes. Considerando estos informes dentro de la misma aplicación como un solo dominio, un análisis de dominios puede conducir al desarrollador a una comprensión de las abstracciones y mecanismos principales que sirven a todos los tipos diferentes de informe. Las clases y ob-

* *Business function* en el original en inglés. (N. del T.)

jetos resultantes reflejan un conjunto de abstracciones y mecanismos clave generalizados para el problema de generación de informes inmediato; por tanto, el diseño resultante será probablemente más simple que si cada informe se hubiese analizado y diseñado separadamente.

¿Quién es exactamente un experto en un dominio? Con frecuencia, un experto del dominio es simplemente un usuario, como un ingeniero ferroviario o un controlador en una red de ferrocarriles, o una enfermera o doctor en un hospital. No es preciso que un experto del dominio sea ingeniero del software; más habitualmente, es simplemente una persona profundamente familiarizada con todos los elementos de un problema particular. Un experto del dominio habla el vocabulario del dominio del problema.

Algunos gestores pueden verse interesados por la idea de una comunicación directa entre desarrolladores y usuarios finales (para algunos de ellos, una idea aún más aterradora es ¡permitir que un usuario final vea a un desarrollador!). Para sistemas muy complejos, el análisis de dominios puede conllevar un proceso formal, utilizando los recursos de muchos expertos y desarrolladores durante varios meses. En la práctica, un análisis formal de esta naturaleza es raramente necesario. Muchas veces, todo lo que se necesita para aclarar un problema de diseño es una breve reunión entre un experto del dominio y un desarrollador. Es verdaderamente asombroso ver lo que una pizca de conocimiento del dominio puede hacer para ayudar a un desarrollador a tomar decisiones de diseño inteligentes. De hecho, a nosotros nos resulta extremadamente útil tener muchos de estos encuentros a lo largo del diseño de un sistema. El análisis de dominios casi nunca es una actividad monolítica; está mejor enfocado si se decide conscientemente analizar un poco, y después diseñar un poco.

Análisis de casos de uso. Aisladamente, las prácticas del análisis clásico, análisis del comportamiento y análisis de dominios dependen de una gran cantidad de experiencia personal por parte del analista. Para la mayoría de los proyectos de desarrollo, esto es inaceptable, porque tal proceso no es ni determinístico ni predecible con fiabilidad.

Sin embargo, existe una actividad que puede acoplarse con los tres enfoques anteriores, para dirigir el proceso de análisis de modo significativo. Esta práctica es el análisis de casos de uso, formalizado en primer lugar por Jacobson. Jacobson define un caso de uso como «una forma o patrón o ejemplo concreto de utilización, un escenario que comienza con algún usuario del sistema que inicia alguna transacción o secuencia de eventos interrelacionados» [41].

Brevemente, puede aplicarse el análisis de casos de uso tan tempranamente como en el análisis de requerimientos, momento en el que los usuarios finales, otros expertos del dominio y el equipo de desarrollo enumeran los escenarios fundamentales para el funcionamiento del sistema (no se necesita trabajar sobre esos escenarios desde el principio, se puede simplemente enumerarlos). Estos escenarios en conjunto describen las funciones del sistema en esa aplicación. El análisis procede entonces como un estudio de cada escenario, utilizando técnicas de presentación (*Storyboard*) similares a las que se usan en la industria del

cine y la televisión [42]. A medida que el equipo pasa por cada escenario, debe identificar los objetos que participan él, las responsabilidades de cada objeto, y cómo esos objetos colaboran con otros, en términos de las operaciones que invoca cada uno sobre el otro. De este modo, el equipo se ve forzado a idear una clara separación de intereses entre todas las abstracciones. Según continúa el proceso de desarrollo, estos escenarios iniciales se expanden para considerar condiciones excepcionales así como comportamientos secundarios del sistema (eso de lo que Goldstein y Alger hablan como temas periféricos [43]). Los resultados de estos escenarios secundarios o bien introducen abstracciones nuevas o bien añaden, modifican o reasignan las responsabilidades de abstracciones ya existentes. Como se discute más adelante en el Capítulo 6, los escenarios sirven también como base para las pruebas del sistema.

Fichas CRC. Las fichas CRC han surgido como una forma simple, pero maravillosamente efectiva de analizar escenarios⁴. Propuestas en primer lugar por Beck y Cunningham como una herramienta para la enseñanza de programación orientada a objetos [44], las fichas CRC han demostrado ser una herramienta de desarrollo muy útil que facilita las «tormentas de ideas» y mejora la comunicación entre desarrolladores. Una ficha CRC no es más que una tarjeta con una tabla de 3×5 ⁵, sobre la cual el analista escribe —a lápiz— el nombre de una clase (en la parte superior de la tarjeta), sus responsabilidades (en una mitad de la tarjeta) y sus colaboradores (en la otra mitad). Se crea una ficha para cada clase que se identifique como relevante para el escenario. A medida que el equipo avanza por ese escenario, puede asignar nuevas responsabilidades a una clase ya existente, agrupar ciertas responsabilidades para formar una nueva clase, o (más frecuentemente) dividir las responsabilidades de una clase en otras de grano más fino, y quizás distribuir estas responsabilidades a una clase diferente.

Las fichas CRC pueden disponerse espacialmente para representar patrones de colaboración. Desde el punto de vista de la semántica dinámica del escenario, las fichas se disponen para mostrar el flujo de mensajes entre instancias prototípicas de cada clase; desde el punto de vista de la semántica estática del escenario, las fichas se colocan para representar jerarquías de generalización/especialización o de agregación entre las clases.

Descripción informal en español⁶. Una alternativa radical para el análisis orientado a objetos clásico fue propuesta primeramente por Abbott, quien sugirió redactar una descripción del problema (o de parte del problema) en lenguaje natural (por ejemplo, español), y subrayar entonces los nombres y los verbos [45]. Los nombres representan objetos candidatos, y los verbos representan operaciones candidatas sobre ellos. Esta técnica se presta a la automatización, y

⁴ CRC viene de Clases/Responsabilidades/Colaboradores.

⁵ Si el presupuesto de desarrollo puede afrontarlo, mejor comprar fichas de 5×7 . Las tarjetas con líneas son elegantes, un chorro de tarjetas de colores muestra que uno es un desarrollador muy fresco.

⁶ Descripción informal en inglés, en el original. (*N. del T.*)

se han construido sistemas de este tipo en el Tokyo Institute of Technology y en Fujitsu [46].

El enfoque de Abbott es útil porque es simple y porque obliga al desarrollador a trabajar en el vocabulario del espacio del problema. Sin embargo, de ninguna manera es un enfoque riguroso, y desde luego no se adapta bien a escalas mayores que las de problemas claramente triviales. El lenguaje humano es un vehículo de expresión terriblemente impreciso, así que la calidad de la lista resultante de objetos y operaciones depende de la habilidad del autor para redactar. Además, todo nombre puede transformarse en verbo, y al revés; por tanto, es fácil dar un sesgo a la lista de candidatos para enfatizar bien los objetos o bien las operaciones.

Análisis estructurado. Una segunda alternativa para el análisis orientado a objetos clásico utiliza los productos del análisis estructurado como vía de entrada al diseño orientado a objetos. Esta técnica es atractiva sólo porque hay gran número de analistas con experiencia en análisis estructurado, y existen muchas herramientas CASE que soportan la automatización de estos métodos. Personalmente, desaconsejamos el uso del análisis estructurado como punto de partida para el diseño orientado a objetos, pero para algunas organizaciones es la única alternativa pragmática.

En esta aproximación, se comienza con un modelo esencial del sistema, tal como lo describen los diagramas de flujo de datos y otros productos del análisis estructurado. Estos diagramas suministran un modelo formal razonable del problema. Partiendo de este modelo, puede pasarse a identificar las clases y objetos significativos en el dominio del problema de tres formas distintas.

McMenamin y Palmer proponen comenzar con un análisis del diccionario de datos y proceder a analizar el diagrama de contexto del modelo. En palabras suyas, «con la lista de elementos de datos esenciales, piénsese sobre lo que nos dicen o nos describen. Si fuesen adjetivos en una sentencia, por ejemplo, ¿a qué sustantivos afectarían? Las respuestas a esta pregunta dan lugar a la lista de objetos candidatos» [47]. Estos objetos candidatos se derivan típicamente del entorno circundante, de las entradas y salidas esenciales y de los productos, servicios y otros recursos que el sistema maneja.

Las siguientes dos técnicas conllevan el análisis de diagramas de flujos de datos individuales. Dado un diagrama de flujo de datos concreto (utilizando la terminología de Ward/Mellor [48]), los objetos candidatos pueden derivarse de lo siguiente:

- Entidades externas.
- Almacenes de datos.
- Almacenes de control.
- Transformaciones de control.

Las clases candidatas derivan de dos fuentes:

- Flujos de datos.
- Flujos de control.



Esto deja las transformaciones de datos, que se asignan o como operaciones sobre objetos existentes o como el comportamiento de un objeto que se inventa para servir como agente responsable de esta transformación.

Seidewitz y Stark sugieren otra técnica, que llaman *análisis de abstracciones*. El análisis de abstracciones se fundamenta en la identificación de las entidades centrales, cuya naturaleza es similar a la de las transformaciones centrales en el diseño estructurado. Tal y como ellos afirman, «en el análisis estructurado, los datos de entrada y salida se examinan y se siguen hacia adentro hasta que alcanzan el nivel más alto de abstracción. Los procesos entre las entradas y las salidas forman la transformación central. En el análisis de abstracciones el diseñador hace lo mismo, pero también examina la transformación central para determinar qué procesos y estados representan el mejor modelo abstracto de lo que hace el sistema» [49]. Tras identificar la entidad central en un diagrama de flujo de datos específico, el análisis de abstracciones procede a identificar todas las entidades de apoyo siguiendo los flujos de datos aferentes y eferentes a la entidad central, y agrupando los procesos y estados hallados en el camino. En la práctica, Seidewitz y Stark han encontrado que el análisis de abstracciones es una técnica difícil de aplicar con éxito, y como alternativa recomiendan los métodos de análisis orientado a objetos [50].

Hay que hacer hincapié en que el diseño estructurado, tal como se empareja normalmente con el análisis estructurado, es completamente ortogonal a los principios del diseño orientado a objetos. Nuestra experiencia indica que el uso del análisis estructurado como vía de entrada para el diseño orientado a objetos falla a menudo cuando el desarrollador no es capaz de resistir la incitación a recaer en el abismo de la mentalidad del diseño estructurado. Otro peligro habitual es el hecho de que muchos analistas tiendan a escribir diagramas de flujo de datos que reflejan un diseño en vez de un modelo esencial del problema. Es tremadamente difícil construir un sistema orientado a objetos partiendo de un modelo tan obviamente predispuesto hacia la descomposición algorítmica. Ésta es la razón por la que nosotros preferimos utilizar análisis orientado a objetos como entrada al diseño orientado a objetos: existe simplemente menos peligro de contaminar el diseño con nociones algorítmicas preconcebidas.

Si uno se ve obligado a utilizar análisis estructurado como punto de partida, por las razones (muy respetables) que sean⁷, sugerimos que se intente dejar de escribir diagramas de flujo de datos tan pronto como empiecen a oler a diseño en vez de a modelo esencial. Además, es una práctica saludable huir de los productos del análisis estructurado una vez que el diseño ya está en marcha. Recuérdese que los productos del desarrollo, incluyendo los diagramas de flujo de datos, no son fines en sí mismos; deberían ser vistos simplemente como herramientas para el camino que ayudan a la comprensión intelectual, por parte del desarrollador, del problema y su implantación. Normalmente, se escribe un diagrama de flujo de datos y después se inventan los mecanismos que implantan el comportamiento deseado. Hablando en términos prácticos, el verdadero acto

⁷ Las razones políticas o históricas no son respetables en absoluto.

de diseño cambia la comprensión que el desarrollador tiene sobre el problema, haciendo el modelo original obsoleto en algún sentido. El mantenimiento del modelo original coherente y actualizado respecto al diseño es una labor sumamente trabajosa, no se presta a la automatización y, francamente, no añade mucho valor a lo que se está haciendo. Así, sólo habría que retener los productos del análisis que estuviesen a un nivel de abstracción suficientemente alto. Capturan un modelo esencial del problema, y así se prestan a cualquier número de diseños diferentes.

4.3. Abstracciones y mecanismos clave

Identificación de las abstracciones clave

Búsqueda de las abstracciones clave. Una *abstracción clave* es una clase u objeto que forma parte del vocabulario del dominio del problema. El valor principal que tiene la identificación de tales abstracciones es que dan unos límites al problema; enfatizan las cosas que están en el sistema y, por tanto, son relevantes para el diseño, y suprimen las cosas que están fuera del sistema y, por tanto, son superfluas. La identificación de abstracciones clave es altamente específica de cada dominio. Como establece Goldberg, la «elección apropiada de objetos depende, por supuesto, de los propósitos a los que servirá la aplicación y de la granularidad de la información que va a manipularse» [51].

Como se mencionó anteriormente, la identificación de las abstracciones clave conlleva dos procesos: descubrimiento e invención. Mediante el descubrimiento, se llega a reconocer las abstracciones utilizadas por expertos del dominio; si el experto del dominio habla de ella, entonces la abstracción suele ser importante [52]. Mediante la invención, se crean nuevas clases y objetos que no son forzosamente parte del dominio del problema, pero son artefactos útiles en el diseño o la implantación. Por ejemplo, un cliente que utiliza un cajero automático habla en términos de cuentas, depósitos y reintegros; estas palabras son parte del vocabulario del dominio del problema. Un desarrollador de un sistema semejante utiliza esas mismas abstracciones, pero introduce también algunas nuevas, como bases de datos, manejadores de pantallas, listas, colas y demás. Estas abstracciones clave son artefactos del diseño particular, no del dominio del problema.

Quizás la vía más potente para identificar abstracciones clave sea examinar el problema o el diseño y ver si existe alguna abstracción que sea similar a las clases y objetos que ya existen. Como se trata más adelante en el Capítulo 6, en ausencia de tales abstracciones reutilizables se recomienda el uso de escenarios para guiar el proceso de identificación de clases y objetos.

Refinamiento de abstracciones clave. Una vez que se identifica determi-

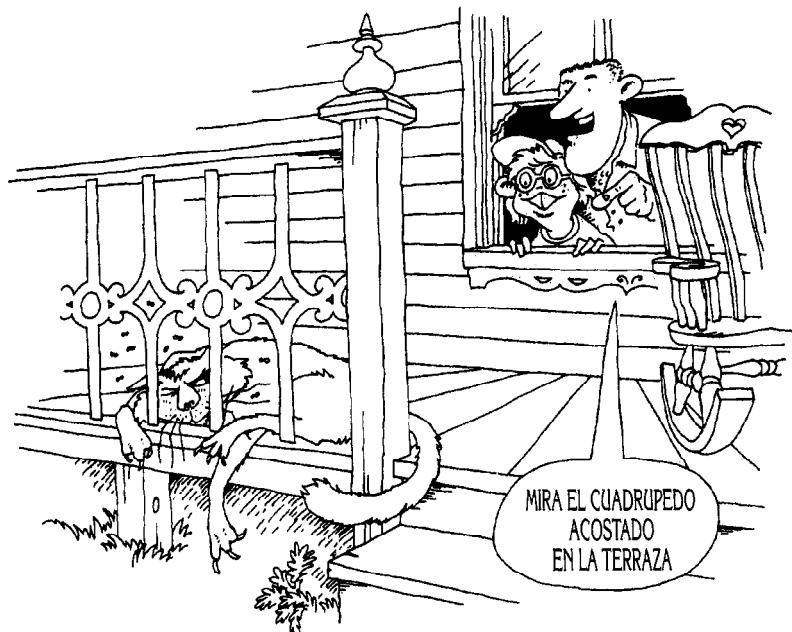
nada abstracción clave como candidata, hay que evaluarla de acuerdo a las métricas descritas en el capítulo anterior. Como sugiere Stroustrup, «frecuentemente esto quiere decir que el programador debe centrarse en las preguntas: ¿cómo se crean los objetos de esta clase? ¿Pueden los objetos de esta clase copiarse y/o destruirse? ¿Qué operaciones pueden hacerse en esos objetos? Si no hay buenas respuestas a tales preguntas, el concepto probablemente no estaba “limpio” desde el principio, y podría ser buena idea pensar un poquito más sobre el problema y la solución propuesta en lugar de empezar inmediatamente a “codificar alrededor” de los problemas» [53].

Dada una nueva abstracción, hay que ubicarla en el contexto de las jerarquías de clases y objetos que se han diseñado. Hablando en términos prácticos, esto no es ni una actividad ascendente ni descendente. Como observan Halbert y O'Brien, «uno no siempre diseña tipos en una jerarquía de tipos comenzando por un supertipo y creando a continuación los subtipos. Frecuentemente, uno crea varios tipos aparentemente dispares, se da cuenta de que están relacionados, y entonces factoriza sus características comunes en uno o más supertipos... Normalmente se necesitan varias pasadas arriba y abajo para producir un diseño del programa correcto y completo» [54]. Esto no es una licencia para hacer chapuzas, sino una observación, basada en la experiencia, de que el diseño orientado a objetos es incremental e iterativo. Stroustrup hace una observación parecida cuando dice que «las reorganizaciones más habituales en una jerarquía de clases son la factorización de las partes comunes de dos clases en una nueva clase, y la división de una clase en otras dos nuevas» [55].

La colocación de clases y objetos en los niveles correctos de abstracción es difícil. A veces se puede encontrar una subclase general, y elegir moverla hacia arriba en la estructura de clases, incrementando así el grado de compartición. Esto se llama *promoción de clases* [56]. Análogamente, se puede apreciar que una clase es demasiado general, dificultando así la herencia por las subclases a causa de un vacío semántico grande. Esto recibe el nombre de *conflicto de grano* [57]. En ambos casos, se intenta identificar abstracciones cohesivas y débilmente acopladas, para mitigar estas dos situaciones.

La mayoría de los desarrolladores suele tomarse a la ligera la actividad de dar un nombre correcto a las cosas —de forma que reflejen su semántica—, a pesar de que es importante en la captura de la esencia de las abstracciones que se describen. El software debería escribirse tan cuidadosamente como la prosa en español⁸, con consideración tanto hacia el lector como hacia el computador [58]. Considérense por un momento todos los nombres que pueden necesitarse simplemente para identificar un solo objeto: se tiene el nombre del propio objeto, el nombre de su clase y el nombre del módulo en el que se declara esa clase. Multiplíquese esto por miles de objetos y posiblemente cientos de clases, y aparecerá un problema bastante real.

⁸ Prosa en inglés en el original. (*N. del T.*).



Las clases y objetos deberían estar al nivel de abstracción adecuado: ni demasiado alto ni demasiado bajo.

Aquí se ofrecen las siguientes sugerencias:

- Los objetos deberían nombrarse empleando frases construidas con nombres propios, como `elSensor` o simplemente `forma`.
- Las clases deberían nombrarse empleando frases construidas con nombres comunes, como `Sensores` o `Formas`.
- Las operaciones de modificación deberían nombrarse empleando frases construidas con verbos activos, como `dibujar` o `moveIzquierda`.
- Las operaciones de selección deberían implicar una interrogación, o bien nombrarse con verbos del tipo «*ser-estar*», como `extensionDe` o `estaAbierto`.
- El uso de caracteres de subrayado y estilos de uso de mayúsculas son en gran medida cuestiones de gusto personal. Da igual el estilo cosmético que se use, pero al menos los programas propios deberían ser autoconsistentes.

Identificación de mecanismos

Búsqueda de mecanismos. En el capítulo anterior se utilizó el término *mecanismo* para describir cualquier estructura mediante la cual los objetos colla-

borasen para proporcionar algún comportamiento que satisficiera un requerimiento del problema. Mientras el diseño de una clase incorpora el conocimiento de cómo se comportan los objetos individuales, un mecanismo es una decisión de diseño sobre cómo cooperan colecciones de objetos. Los mecanismos representan así patrones de comportamiento.

Por ejemplo, considérese un requisito del sistema para un automóvil: la pulsación del acelerador debería hacer que el motor funcione más rápido, y soltar el acelerador debería hacer que el motor funcione con más lentitud. La forma en que se consigue esto realmente es completamente indiferente para el conductor. Puede emplearse cualquier mecanismo siempre y cuando produzca el efecto deseado, y qué mecanismo se selecciona es por tanto en gran medida una opción de diseño. Más concretamente, podría considerarse cualquiera de los siguientes diseños:

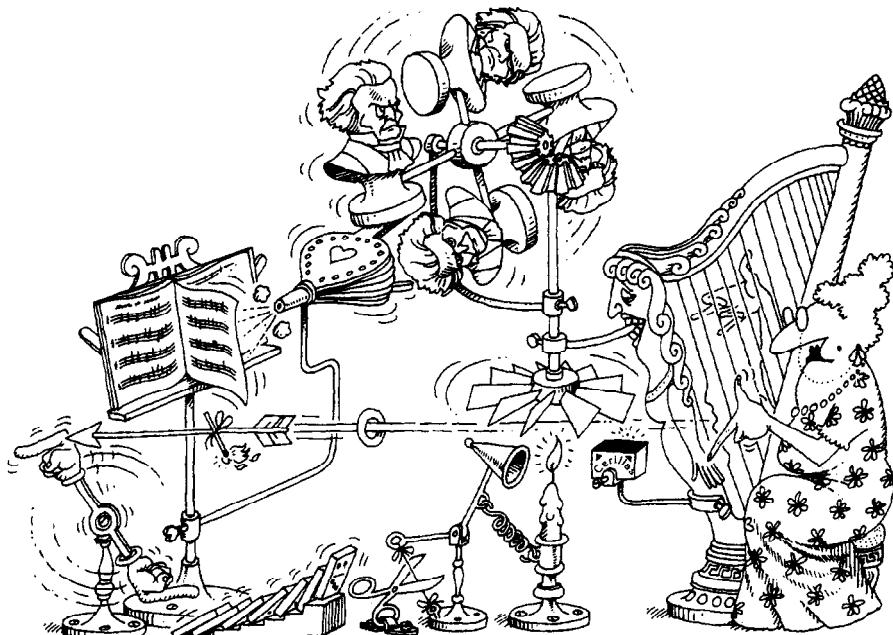
- Una conexión mecánica desde el acelerador hasta el carburador (el mecanismo más común).
- Una conexión electrónica desde un sensor de presión bajo el acelerador hasta un computador que controla el carburador (un mecanismo de transmisión por cable).
- No hay conexión; el tanque de gasolina se coloca en el techo del coche, y la gravedad hace que el combustible fluya hacia el motor. Su ritmo de flujo está regulado por una abrazadera que comprime el tubo de gasolina; la presión sobre el acelerador reduce la tensión en la abrazadera, haciendo que la gasolina fluya más rápido (mecanismo de bajo coste).

El mecanismo que elige un desarrollador entre un conjunto de alternativas es frecuentemente el resultado de otros de factores, como coste, fiabilidad, facilidad de fabricación y seguridad.

Del mismo modo que es una falta de educación el que un cliente viole el interfaz con otro objeto, tampoco es aceptable que los objetos transgredan los límites de las reglas de comportamiento dictadas por un mecanismo particular. Efectivamente, sería sorprendente para un conductor si al pisar un acelerador se encendiesen las luces del coche en lugar de acelerarse la marcha del motor.

Mientras las abstracciones clave reflejan el vocabulario del dominio del problema, los mecanismos son el alma del diseño. Durante el proceso de diseño, el desarrollador debe considerar no sólo el diseño de clases individuales, sino también cómo trabajan juntas las instancias de esas clases. Una vez más, el uso de escenarios dirige este proceso de análisis. Una vez que un desarrollador decide sobre un patrón concreto de colaboración, se distribuye el trabajo entre muchos objetos definiendo métodos convenientes en las clases apropiadas. En última instancia, el protocolo de una clase individual abarca todas las operaciones que se requieren para implantar todo el comportamiento y todos los mecanismos asociados con cada una de sus instancias.

Los mecanismos representan así decisiones de diseño estratégicas, como el diseño de una estructura de clases. En contraste, sin embargo, el interfaz de una clase individual es más bien una decisión de diseño táctica. Estas decisiones es-



Los mecanismos son los medios por los cuales los objetos colaboran para proporcionar algún comportamiento de nivel superior.

tratégicas deben tomarse explícitamente; de otro modo se acabará por tener una muchedumbre de objetos que prácticamente no cooperan, todos ellos presionando y empujando para hacer su trabajo con poca consideración hacia los demás objetos. Los programas más elegantes, compactos y rápidos incorporan mecanismos cuidadosamente discurridos.

Los mecanismos son realmente uno de entre la variedad de patrones que se encuentran en los sistemas de software bien estructurados. En el límite inferior de la cadena de alimentación, están los modismos. Un *modismo** es una expresión peculiar de algún lenguaje de programación o cultura de aplicaciones, que representa una convención generalmente aceptada para el uso del lenguaje⁹. Por ejemplo, en CLOS ningún programador usaría caracteres de subrayado en nombres de función o variable, aunque sea práctica común en Ada [59]. Parte del esfuerzo en el aprendizaje de un lenguaje de programación está en el aprendizaje de sus modismos, que normalmente se transmiten como folklore de programador a programador. Sin embargo, como apunta Coplien, los modismos desempeñan un importante papel en la codificación de patrones de bajo nivel.

* *Idiom* en el original en inglés. (N. del T.)

⁹ Una característica distintiva de un modismo es que la ignorancia o violación de uno de ellos tiene consecuencias sociales inmediatas: a uno lo marcan como a un cantamañas o, peor aún, como a un intruso, que no merece respeto alguno.

Hace notar que «muchas tareas de programación habituales son idiomáticas», y, por tanto, la identificación de tales modismos permite «el uso de construcciones de C++ para expresar funcionalidad más allá del propio lenguaje, dando la ilusión de que son parte del mismo» [60].

En el extremo superior de la cadena de alimentación están los marcos de referencia. Un *marco de referencia** es una colección de clases que ofrecen un conjunto de servicios para un dominio particular; un marco de referencia exporta así una serie de clases y mecanismos individuales que los clientes pueden utilizar o adaptar. Como se discutirá en el Capítulo 9, los marcos de referencia representan reutilización a lo grande.

Mientras que los modismos son parte de una cultura de programación, los marcos de referencia suelen ser producto de aventuras comerciales. Por ejemplo, el MacApp de Apple (y su sucesor, Bedrock) son marcos de referencia de aplicaciones, escritos en C++, para construir aplicaciones de acuerdo con los estándares de interfaces de usuario de Macintosh. Análogamente, la Microsoft Foundation Library y la biblioteca ObjectWindows de Borland son marcos de referencia para construir aplicaciones según los estándares de interfaces de usuario de Windows.

Ejemplos de mecanismos. Considérese el mecanismo de dibujo utilizado habitualmente en interfaces gráficos de usuario. Varios objetos deben colaborar para presentar una imagen a un usuario: una ventana, una vista, el modelo que se va a visualizar, y algún cliente que sabe cuándo (pero no cómo) hay que visualizar ese modelo. Primero, el cliente dice a la ventana que se dibuje a sí misma, lo que al fin y al cabo resulta en una imagen que se muestra al usuario. En este mecanismo, el modelo está completamente separado de la ventana y la vista en la que se presenta: las vistas pueden enviar mensajes a los modelos, pero los modelos no pueden enviar mensajes a las vistas. Smalltalk usa una variante de este mecanismo, y lo llama el paradigma *modelo-vista-controlador (MVC)* [61]. Se emplea un mecanismo similar en casi cualquier marco de referencia de interfaz gráfico de usuario orientado a objetos.

Los mecanismos representan así un nivel de reutilización que es mayor que la reutilización de las clases individuales. Por ejemplo, el paradigma MVC se utiliza de manera extensiva en el interfaz de usuario de Smalltalk. El paradigma MVC a su vez se construye sobre otro mecanismo, el mecanismo de dependencia, incorporado al comportamiento de la clase base de Smalltalk `Model`, y que afecta por tanto a gran parte de la biblioteca de clases de Smalltalk.

Pueden encontrarse ejemplos de mecanismos prácticamente en cualquier dominio. Por ejemplo, la estructura de un sistema operativo puede describirse al nivel más alto de abstracción según el mecanismo que se utilice para distribuir programas. Un diseño determinado podría ser monolítico (como MS-DOS), o puede emplear un núcleo o kernel (como UNIX) o una jerarquía de procesos (como en el sistema operativo THE) [62]. En inteligencia artificial, se han ex-

* *Framework* en el original en inglés. (N. del T.)

plorado diversos mecanismos para el diseño de sistemas de razonamiento. Uno de los paradigmas de uso más extendido es el de la pizarra, en el que las fuentes individuales de conocimiento actualizan independientemente una pizarra. No hay un control central en tal mecanismo, pero cualquier cambio en la pizarra puede disparar a un agente para que explore alguna nueva vía de resolución de un problema [63]. Coad ha identificado análogamente una serie de mecanismos comunes en sistemas orientados a objetos, entre los que se incluyen patrones de asociación temporal, de registro de eventos y multidifusión [64]. En todos los casos, estos mecanismos no se manifiestan como clases individuales, sino como la estructura de clases que colaboran.

Esto completa nuestro estudio de la clasificación y los conceptos que sirven como fundamentos del diseño orientado a objetos. Los siguientes tres capítulos se centran en el propio método, incluyendo su notación, proceso y pragmática.

Resumen

- La identificación de clases y objetos es el problema fundamental en el diseño orientado a objetos; la identificación implica descubrimiento e invención.
- La clasificación es fundamentalmente un problema de agrupación.
- La clasificación es un proceso incremental e iterativo, que se complica por el hecho de que un conjunto dado de objetos puede clasificarse de muchas formas igualmente correctas.
- Los tres enfoques de la clasificación son la categorización clásica (clasificación por propiedades), agrupamiento conceptual (clasificación por conceptos) y teoría de prototipos (clasificación por asociación con un prototipo).
- Los escenarios son una potente herramienta para el análisis orientado a objetos, y pueden utilizarse para guiar los procesos de análisis clásico, análisis del comportamiento y análisis de dominios.
- Las abstracciones clave reflejan el vocabulario del dominio del problema y pueden ser descubiertas en el dominio del problema, o bien ser inventadas como parte del diseño.
- Los mecanismos denotan decisiones estratégicas de diseño respecto a la actividad de colaboración entre muchos tipos diferentes de objetos.

Lecturas recomendadas

El problema de la clasificación es intemporal. En su obra titulada *El Estadista*, Platón introduce el enfoque clásico a la categorización, a través de la cual se agrupan los objetos con propiedades similares. En las *Categorías*, Aristóteles prosigue con el tema y

analiza las diferencias entre clases y objetos. Varios siglos después, Aquino, en su *Summa Theologica*, y luego Descartes, en *Reglas para la Dirección de la Mente*, examinan la filosofía de la clasificación. Entre los filósofos objetivistas contemporáneos se cuenta Rand [I 1979].

Se discuten alternativas a la visión objetivista del mundo en Lakoff [I 1980] y Goldstein y Alger [C 1992].

La clasificación es una habilidad humana básica. Las teorías sobre su adquisición en la primera infancia tienen un pionero en Piaget, y son resumidas por Maier [A 1969]. Lefrancois [A 1977] ofrece una introducción muy legible a estas ideas y proporciona un discurso excelente sobre la adquisición del concepto de objeto por los niños.

Los científicos del conocimiento han explorado en gran detalle los problemas de la clasificación. Newell y Simon [A 1972] ofrecen una incomparable fuente de material sobre las capacidades humanas de clasificación. Hay información más general en Simon [A 1982], Hofstadter [I 1979], Siegler y Richards [A 1982] y Stillings, Feinstein, Garfield, Rissland, Rosenbaum, Weisler y Baker-Ward [A 1987]. Lakoff [A 1987], un lingüista, ofrece información sobre cómo diferentes lenguajes humanos evolucionaron para afrontar los problemas de la clasificación y qué revela esto sobre la mente. Minsky [A 1986] enfoca el tema por la dirección opuesta y comienza con una teoría sobre la estructura de la mente.

El agrupamiento conceptual, un enfoque hacia la representación del conocimiento mediante la clasificación, es descrito en detalle por Michalski y Stepp [A 1983, 1986], Peckham y Maryanski [J 1988] y Sowa [A 1984]. El análisis de dominios, un enfoque para encontrar abstracciones y mecanismos clave examinando el vocabulario del dominio del problema, se describe en la inteligible colección de artículos de Prieto-Díaz y Arango [A 1991]. Iscoe [B 1988] ha realizado varias contribuciones importantes a este campo. Puede encontrarse información adicional en Iscoe, Browne y Weth [B 1989], Moore y Bailin [B 1988] y Arango [B 1989].

La clasificación inteligente requiere a menudo observar el mundo de formas innovadoras, y estas habilidades pueden aprenderse (o, al menos, reforzarse). VonOech [I 1990] sugiere algunos caminos hacia la creatividad. Coad [A 1993] ha desarrollado un juego de tablero (el *Object Game*) que desarrolla la habilidad en la identificación de clases y objetos.

Aunque ese campo está aún en pañales, se está realizando algún trabajo muy prometedor en la catalogación de patrones en los sistemas de software, dando lugar a una taxonomía de modismos, mecanismos y marcos de referencia. Algunas referencias interesantes son Coplien [G 1992], Coad [A 1992], Johnson [A 1992], Shaw [A 1989, 1990, 1991], y Wirfs-Brock [C 1991]. El influyente trabajo de Alexander [I 1979] aplica los patrones al campo de la arquitectura de la construcción y al del urbanismo.

Los matemáticos han intentado desarrollar enfoques empíricos de la clasificación, dando lugar a lo que se denomina *teoría de la medida*. Stevens [A 1946] y Coombs, Raiffa y Thrall [A 1954] proporcionan el trabajo seminal en este tema.

La Classification Society of North America publica una revista semestral, que contiene diversos artículos sobre los problemas de la clasificación.

Notas bibliográficas

- [1] Citado por Swain, M. June 1988. Programming Paradigms. *Dr. Dobb's Journal of Software Tools*, No. 140, p. 110.

- [2] Michalski, R. and Stepp, R. 1983. Learning from Observation: Conceptual Clustering in *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, CA: Tioga, p. 332.
- [3] Alexander, C. 1979. *The Timeless Way of Building*. New York, NY: Oxford University Press, p. 203.
- [4] Darwin, D. 1984. *The Origin of Species. Vol. 49 of Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 207.
- [5] *The New Encyclopedia Britannica*. 1985. Chicago, IL: Encyclopedia Britannica. vol. 3, p. 356.
- [6] Gould, S. June 1992. *We Are All Monkey's Uncles. Natural History*.
- [7] May, R. September 16, 1988. How Many Species Are There on Earth? *Science* vol. 241, p. 1441.
- [8] Citado por Lewin, R. November 4, 1988. Family Relationships Are a Biological Conundrum. *Science* vol. 242, p. 671.
- [9] *The New Encyclopedia Britannica* vol. 3, p. 156.
- [10] Descartes, R. 1984. Rules for the Direction of the Mind. Vol. 31 of *Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 32.
- [11] Shaw, M. May 1989. Larger Scale Systems Require Higher-Level Abstractions. *SIGSOFT Engineering Notes* vol. 14(3), p. 143.
- [12] Goldstein, T. May 1989. The Object-Oriented Programmer. *The C++ Report* vol. 1(5).
- [13] Coombs, C., Raiffa, H., and Thrall, R. 1954. Some Views on Mathematical Models and Measurement Theory. *Psychological Review* vol. 61(2), p. 132.
- [14] Flood, R. and Carson, E. 1988. *Dealing with Complexity*. New York, NY: Plenum Press, p. 8.
- [15] Birtwistle, G., Dahl, O-J., Myhrhaug, B., and Nygard, K. 1979. *Simula begin*. Lund, Sweden: Studenlitteratur, p. 23.
- [16] Heinlein, R. 1966. *The Moon Is a Harsh Mistress*. New York, NY: The Berkeley Publishing Group, p. 11.
- [17] Sowa, J. 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley, p. 16.
- [18] Lakoff, G. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind*. Chicago, IL: The University of Chicago Press, p. 161.
- [19] Stepp, R. and Michalski, R. February 1986. Conceptual Clustering of Structured Objects: A Goal-Oriented Approach. *Artificial Intelligence* vol. 28(1), p. 53.
- [20] Wegner, P. 1987. The Object-Oriented Classification Paradigm, in *Research Directions in Object-Oriented Programming*. Cambridge, MA: The MIT Press, p. 480.
- [21] Aquinas, T. 1984. *Summa Theologica. Vol. 19 of Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 71.
- [22] Maier, H. 1969. *Three Theories of Child Development: The Contributions of Erik H. Erickson, Jean Piaget, and Robert R. Sears, and Their Applications*. New York, NY: Harper and Row, p. 111.
- [23] Lakoff. *Women, Fire*, p. 32.
- [24] Minsky, M. 1986. *The Society of Mind*. New York, NY: Simon and Schuster, p. 199.
- [25] *The Great Ideas: A Syntopicon of Great Books of the Western World*. 1984. Vol. 1 of *Great Books of the Western World*. Chicago, IL: Encyclopedia Britannica, p. 293.
- [26] Kosko, B. 1992. *Neural Networks and Fuzzy Systems*. Englewood Cliffs, NJ: Prentice-Hall, p. xx.
- [27] Stepp, p. 44.

- [28] Lakoff. *Women, Fire, and Dangerous Things*, p. 16.
- [29] Ibid., p. 16.
- [30] Lakoff, G. and Johnson, M. 1980. *Metaphors We Live By*. Chicago, IL: The University of Chicago Press, p. 122.
- [31] Meyer, B. 1988. Private communication.
- [32] Shlaer, S. and Mellor, S. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press, p. 15.
- [33] Ross, R. 1987. *Entity Modeling: Techniques and Application*. Boston, MA: Database Research Group, p. 9.
- [34] Coad, P. and Yourdon, E. 1990. *Object-Oriented Analysis*. Englewood Cliffs, NJ: Prentice-Hall, p. 62.
- [35] Shlaer, S. and Mellor, S. 1992. *Object Lifecycles: Modeling the World in States*. Englewood Cliffs, New Jersey: Yourdon Press.
- [36] Wirfs-Brock, R., Wilkerson, B., and Wiener, L. 1990. *Designing Object-Oriented Software*. Englewoods Cliffs, New Jersey: Yourdon Press.
- [37] Rubin, K. and Goldberg, A. September 1992. Object Behavior Analysis. *Communications of the ACM*, vol. 35(9), p. 48.
- [38] Dreger, B. 1989. *Function Point Analysis*. Englewoods Cliffs, NJ: Prentice Hall, p. 4.
- [39] Arango, G. May 1989. Domain Analysis: From Art Form to Engineering Discipline. *SIGSOFT Engineering Notes* vol. 14(3), p. 153.
- [40] Moore, J. and Bailin, S. 1988. *Position Paper on Domain Analysis*. Laurel, MD: CTA, p. 2.
- [41] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. 1992. *Object-Oriented Software Engineering*. Workingham, England: Addison-Wesley, p. viii.
- [42] Zahniseer, R. July/August 1990. Building Software In Groups. *American Programmer*, vol. 3(7-8).
- [43] Goldstein, N. and Alger, J. 1992. *Developing Object-Oriented Software for the Macintosh*. Reading, Massachusetts: Addison-Wesley, p. 161.
- [44] Beck, K. and Cunningham, W. October 1989. A Laboratory for Teaching Object-Oriented Thinking. *SIGPLAN Notices* vol. 24(10).
- [45] Abbott, R. November 1983. Program Design by Informal English Descriptions. *Communications of the ACM* vol. 26(11).
- [46] Saeki, M., Horai, H., and Enomoto, H. May 1989. Software Development Process from Natural Language Specification. *Proceedings of the 11th International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.
- [47] McMenamin, S. and Palmer, J. 1984. *Essential Systems Analysis*. New York, NY: Yourdon Press, p. 267.
- [48] Ward, P. and Mellor, S. 1985. *Structured Development for Real-time Systems*. Englewood Cliffs, NJ: Yourdon Press.
- [49] Seidewitz, E. and Stark, M. August 1986. *General Object-Oriented Software Development*, Report SEL-86-002. Greenbelt, MD: NASA Goddard Space Flight Center, p. 5-2.
- [50] Seidewitz, E. 1990. Private communication.
- [51] Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley, p. 77.
- [52] Thomas, D. May/June 1989. In Search of an Object-Oriented Development Process. *Journal of Object-Oriented Programming* vol. 2(1), p. 61.
- [53] Stroustrup, B. 1986. *The C++ Programming Language*. Reading, MA: Addison-Wesley, p. 7.

- [54] Halbert, D. and O'Brien, P. September 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software* vol. 4(5), p. 75.
- [55] Stefik, M. and Bobrow, D. Winter 1986. Object-Oriented Programming: Themes and Variations, *AI Magazine* vol. 6(4), p. 60.
- [56] Stroustrup, B. 1991. *The C+ Programming Language*, Second Edition. Reading, Massachusetts: Addison-Wesley, p. 377.
- [57] Stefik and Bobrow. Object-Oriented Programming, p. 58.
- [58] Lins, C. 1989. A First Look at Literate Programming. *Structured Programming*.
- [59] Gabriel, R. 1990. Comunicación privada.
- [60] Coplien, J. 1992. *Advanced C++ Programming Styles and Idioms*. Reading, Massachusetts: Addison-Wesley.
- [61] Adams, S. July 1986. MetaMethods: The MVC Paradigm, in *HOOPLA: Hooray for Object-Oriented Programming Languages*. Everette, WA: Object-Oriented Programming for Smalltalk Applications Developers Association vol. 1(4), p. 6.
- [62] Russo, V., Johnston, G., and Campbell, R. September 1988. Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. *SIGPLAN Notices* vol. 23(11), p. 249.
- [63] Englemore, R. and Morgan, T. 1988. *Blackboard Systems*. Wokingham, England: Addison-Wesley, p. v.
- [64] Coad, P. September 1992. Object-Oriented Patterns. *Communications of the ACM*, vol. 35(9).



El método

Qué innovación conduce a un diseño de éxito y qué innovación conduce a un fracaso no es del todo predecible. Cada oportunidad para diseñar algo nuevo, ya sea un puente o un avión o un rascacielos, presenta al ingeniero muchas alternativas que pueden parecer incontables. El ingeniero puede decidir copiar todas las características aparentemente positivas que pueda de diseños ya existentes que hayan resistido a las fuerzas del hombre y la naturaleza, pero puede decidir también mejorar aquellos aspectos de los diseños anteriores que parezcan deficientes.

HENRY PETROSKI*
To Engineer is Human

* Petrosky, H. 1985. *To Engineer is Human*. New York, NY: St. Martin's Press, p. 73.



La notación

La acción de dibujar un diagrama no constituye análisis ni diseño. Un diagrama se limita a capturar una descripción del comportamiento del sistema (en el análisis), o la visión y los detalles de una arquitectura (en el diseño). Si se observa el trabajo de cualquier ingeniero —de software, civil, mecánico, químico, arquitecto o cualquier otra disciplina— se advertirá pronto que el único y exclusivo lugar en el que se concibe un sistema es en la mente del diseñador. Puesto que este diseño no permanece en el tiempo, a menudo se capture mediante medios tan avanzados como pizarras, servilletas o el reverso de los envoltorios [1].

No obstante, el disponer de una notación expresiva y bien definida es importante para el proceso de desarrollo de software. En primer lugar, una notación estándar posibilita al analista o desarrollador el describir un universo o formular una arquitectura y comunicar estas decisiones entonces a otros de forma no ambigua. Si se dibuja un circuito eléctrico, el símbolo del transistor será identificado por virtualmente todos los ingenieros electrónicos del mundo. De la misma forma, si un arquitecto en Nueva York bosqueja los planos de una casa, un constructor en San Francisco no tendrá mayores problemas para entender dónde situar puertas, ventanas y tomas de corriente a partir de los detalles de los diagramas. Segundo, como Whitehead afirma en su fundamental trabajo matemático, «al aliviar al cerebro de todo el trabajo innecesario, una buena notación lo libera para concentrarse en problemas más avanzados» [2].

Tercero, una notación expresiva hace posible eliminar buena parte del tedio de comprobar la consistencia y corrección de las decisiones adoptadas, ya que pueden utilizarse herramientas automáticas. Como afirma un informe del Defense Science Board, «el desarrollo de software es y será siempre una tecnología que exige trabajo intensivo [...]. Aunque nuestras máquinas pueden hacer el trabajo sucio y ayudarnos a tener bajo control nuestros edificios, el desarrollo de un concepto es la quintaesencia de la actividad humana [...]. La parte del desarrollo de software que no va a desaparecer es la creación de estructuras conceptuales; la parte que puede desaparecer es la labor de expresarlas» [3].

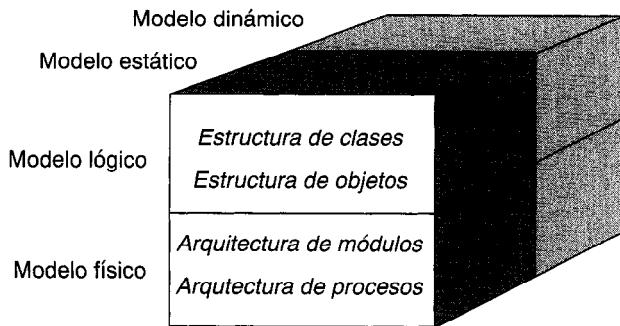


Figura 5.1. Los modelos del desarrollo orientado a objetos.

5.1. Elementos de la notación

La necesidad de vistas múltiples

Es imposible capturar todos los detalles sutiles de un sistema de software complejo en una sola vista. Como observan Kleyn y Gingrich, «uno debe comprender tanto la estructura como la función de los objetos involucrados. Uno debe comprender la estructura taxonómica de las clases, los mecanismos de herencia utilizados, los comportamientos individuales de los objetos y el comportamiento dinámico del sistema en su conjunto. El problema es un tanto análogo al de ver un evento deportivo tal como un partido de tenis o fútbol. Se requieren muchos ángulos de cámara diferentes para proporcionar una comprensión de lo que está sucediendo. Cada cámara revela aspectos particulares de la acción que no podrían ser transmitidos por una sola cámara» [4].

Aunque ya apareció en el Capítulo 1, la Figura 5.1 indica los diferentes modelos que se han juzgado relevantes en el desarrollo orientado a objetos. Dado un proyecto, los productos del análisis y el diseño se expresan mediante estos modelos. En conjunto, estos distintos modelos son ricos semánticamente: son suficientemente expresivos para permitir a un desarrollador capturar todas las decisiones estratégicas y tácticas interesantes que habitualmente se toman durante el análisis de un sistema así como durante la formulación de su arquitectura, y son suficientemente completos como para servir de planos para la implantación en prácticamente cualquier lenguaje de programación orientado a objetos.

El hecho de que esta notación sea detallada no significa que todos sus aspectos deban ser utilizados en todas las ocasiones. De hecho, un subconjunto de ella es suficiente para expresar la semántica de un gran porcentaje de problemas de análisis y diseño; uno de nuestros colegas se refiere a este subconjunto como la *notación Booch Lite*. Se enfatizará este subconjunto durante la presentación de la notación que va a realizarse en este capítulo. ¿Por qué molestar entonces

con el detalle más allá de este subconjunto? Muy sencillo, este detalle es necesario para expresar ciertas decisiones tácticas importantes; además, algunos de estos elementos existen para facilitar la creación de herramientas de ingeniería directa e inversa, las cuales proporcionan una integración entre herramientas CASE como *front-end*¹ que soportan esta notación y entornos de desarrollo de software que se centran en la manipulación de productos de lenguajes orientados a objetos.

Como señala Weinberg, «en otros campos del diseño, tales como la arquitectura, el boceto aproximado es el dispositivo gráfico más frecuentemente empleado, y los esquemas detallados y precisos se utilizan muy raramente mientras no finalice la parte creativa del trabajo de diseño» [5]. Recuérdese, una notación no es más que un vehículo para capturar los razonamientos acerca del comportamiento y la arquitectura de un sistema; una notación no es un fin por sí misma. Consiguientemente, deberían aplicarse solamente los elementos de la notación que son necesarios para transmitir el significado deseado, y ninguno más. Así como sobreespecificar un conjunto de requisitos no es aconsejable, también es peligroso sobreespecificar una solución para un problema. Por ejemplo, en un plano, un arquitecto puede mostrar la situación general de un conmutador de la luz en una habitación, pero su localización exacta no se establecerá hasta que el director de obra y el propietario realicen un repaso al sistema eléctrico, una vez que la casa ya ha tomado forma. Sería tonto especificar las coordenadas tridimensionales precisas del conmutador en los planos (a menos, por supuesto, que éste fuese un detalle funcionalmente importante para el propietario: tal vez la familia del propietario es significativamente más alta o baja que la media). Así, si los analistas, diseñadores e implantadores de un sistema basado en software están altamente adiestrados y han establecido ya una relación de trabajo estrecha, puede que el esquema aproximado sea suficiente (a pesar de que será necesario de todas formas dejar constancia de la visión arquitectónica para quienes se ocupen del mantenimiento del sistema). Si, en el otro extremo, los implantadores no están tan entrenados, o si los desarrolladores están separados geográficamente, o en el tiempo, o por contrato, será necesario reflejar más detalles durante el proceso de desarrollo. La notación que se presenta en este capítulo cubre ambas situaciones.

Los distintos lenguajes de programación utilizan a veces términos diferentes para expresar el mismo concepto. La notación que se presenta aquí es fuertemente independiente del lenguaje, característica que cualquier buena notación de desarrollo debería contemplar. Por supuesto, algunos elementos de la notación no tienen equivalencia en determinados lenguajes, y deberían evitarse si uno de estos lenguajes va a ser el utilizado en la implementación. Por ejemplo, no pueden declararse subprogramas libres en Smalltalk, y, por tanto, las utilidades de clases no se utilizarán generalmente en un sistema implementado en Smalltalk. De la misma forma, C++ no soporta metaclasses, y por tanto este ele-

¹ El término *front-end* se refiere generalmente a un interfaz de usuario, es decir, a lo que «da la cara» frente al usuario. Este vocablo suele utilizarse sin traducir. (*N. del T.*)

mento de la notación puede ignorarse. Por otra parte, no hay nada malo en adaptar esta notación de formas específicas para cada lenguaje. Por ejemplo, la calificación asociada con una operación puede ser adaptada para CLOS con el fin de identificar métodos primarios, así como los métodos :before, :after y :around. Análogamente, una herramienta para C++ puede ignorar la especificación de clases de la notación, y utilizar ficheros cabecera de C++ directamente.

El único propósito de este capítulo es describir la sintaxis y semántica de la notación presentada para el análisis y diseño orientados a objetos. Se proporcionarán algunos pequeños ejemplos de esta notación, utilizando el problema del sistema de cultivo hidropónico que se introdujo en el Capítulo 2. Este Capítulo 5 no explica el proceso por el cual se derivaron estas figuras; ese es el tema del Capítulo 6. En el Capítulo 7, se discuten los aspectos prácticos de este proceso, y en los Capítulos 8 al 12 se demuestra la aplicación práctica de esta notación a través de una serie de ejemplos de aplicación.

Modelos y vistas

En el Capítulo 3 se explicó el significado de las clases y los objetos y sus relaciones. Como sugiere la Figura 5.1, podemos plasmar nuestras decisiones de análisis y diseño observando estas clases y objetos y sus colaboraciones de acuerdo con dos dimensiones: su visión física/lógica, y su visión estática/dinámica. Son necesarias ambas dimensiones para especificar la estructura y el comportamiento de un sistema orientado a objetos.

Para cada dimensión, se define una serie de diagramas que denotan una vista de los modelos del sistema. En este sentido, los modelos del sistema reflejan «toda la verdad» sobre sus clases, relaciones y otras entidades, y cada diagrama representa una proyección de estos modelos. En el estado estable, todos esos diagramas deben ser consistentes con el modelo y por lo tanto consistentes entre sí mismos.

Por ejemplo, considérese una aplicación que comprende varios cientos de clases; las clases forman parte del modelo de la aplicación. Es imposible —y de hecho innecesario— producir un solo diagrama que muestre todas estas clases y todas sus interrelaciones. Resulta más adecuado ver este modelo a través de varios diagramas de clases, cada uno de los cuales presenta una vista del modelo. Un diagrama puede reflejar el entramado de herencias de algunas clases fundamentales; otro puede describir el cierre transitivo de todas las clases utilizadas por una clase concreta. A veces cuando el modelo es estable (cuando le aplicamos el término ya utilizado, *estado estable*), cada uno de estos diagramas permanece semánticamente consistente con el resto y con el modelo. Por ejemplo, si en un universo dado (el cual se describe en un diagrama de objetos), el objeto **A** pasa el mensaje **M** al objeto **B**, entonces **M** debe estar definido para la clase de **B** ya sea directa o indirectamente. En el diagrama de clases correspon-

diente debe existir una relación apropiada entre las clases de A y B, de modo que esas instancias de la clase A puedan de hecho invocar el mensaje m.

Por simplicidad, a través de todos los diagramas, todas las entidades con el mismo nombre y dentro del mismo ámbito se consideran referencias al mismo elemento del modelo. Por ejemplo, si la clase c aparece en dos diagramas diferentes para el mismo sistema, ambas son referencias a la misma clase c. La excepción a esta regla la constituyen las operaciones, cuyos nombres pueden ser sobrecargados.

Para distinguir un diagrama de otro, debemos dar un nombre cuyo propósito es indicar el concepto central o el propósito del diagrama. Pueden añadirse a un diagrama otras etiquetas y notas para aclarar mejor sus contenidos, como se describirá en una próxima sección; tales notas, en general, no contienen semántica adicional.

Modelos lógicos versus modelos físicos

La visión lógica de un sistema sirve para describir la existencia y significado de las abstracciones principales y los mecanismos que forman el espacio del problema, o para definir la arquitectura del sistema. El modelo físico de un sistema describe la composición concreta en cuanto a hardware y software del contexto o implantación del sistema.

Durante el análisis, deben plantearse las siguientes cuestiones principales:

- ¿Cuál es el comportamiento que se desea del sistema?
- ¿Cuáles son las misiones y responsabilidades de los objetos que llevan a cabo este comportamiento?

Como se describió en el capítulo anterior, se utilizan universos para expresar las decisiones tomadas acerca del comportamiento de un sistema. En el modelo lógico, los diagramas de objetos sirven de vehículos primarios para describir universos. Durante el análisis, se pueden utilizar también diagramas de clases para plasmar una abstracción de esos objetos en términos de sus misiones y responsabilidades comunes.

Durante el diseño, deben plantearse las siguientes cuestiones principales relativas a la arquitectura del sistema:

- ¿Qué clases existen, y cómo se relacionan estas clases?
- ¿Qué mecanismos se utilizan para regular la forma en que los objetos colaboran?
- ¿Dónde debería declararse cada clase y objeto?
- ¿A qué procesador debería asignarse un proceso, y para un procesador dado, cómo deberían planificarse sus múltiples procesos?

Se utilizan los siguientes diagramas, respectivamente, para responder a estas preguntas:

- Diagramas de clases.

- Diagramas de objetos.
- Diagramas de módulos.
- Diagramas de procesos.

Modelos estáticos versus modelos dinámicos

Los cuatro diagramas introducidos hasta ahora son fuertemente estáticos. Sin embargo, los eventos suceden dinámicamente en todos los sistemas basados en software: los objetos se crean y se destruyen, los objetos envían mensajes a otros de forma ordenada, y en algunos sistemas, eventos externos disparan operaciones sobre ciertos objetos. Evidentemente, la descripción de un evento dinámico en un medio estático tal como una hoja de papel es un problema difícil, pero afecta prácticamente a todas las disciplinas científicas. En el desarrollo orientado a objetos, se expresa la semántica dinámica de un problema o su implantación mediante dos diagramas adicionales:

- Diagramas de transición de estados.
- Diagramas de interacción.

Cada clase puede tener un diagrama de transición de estados que indica el comportamiento de las instancias de esa clase de cara al orden de sucesión de los eventos. De la misma forma, en conjunción con un diagrama de objetos que representa un escenario, se puede suministrar un guión o diagrama de interacción para mostrar el turno u orden temporal en que se evalúan los mensajes.

El papel de las herramientas

Dado el soporte automático para cualquier notación, una de las cosas que las herramientas pueden hacer es ayudar a los malos diseñadores a crear diseños horribles mucho más rápido de lo que jamás habían podido. Los grandes diseños provienen de los grandes diseñadores, nunca de las grandes herramientas. Las herramientas simplemente fortalecen al individuo, liberándole para concentrarse sobre los aspectos verdaderamente creativos del análisis o el diseño. Así, hay algunas cosas que las herramientas hacen muy bien y otras cosas que no pueden hacer en absoluto. Por ejemplo, cuando se utiliza un diagrama de objetos para mostrar un escenario con un mensaje que pasa de un objeto a otro, una herramienta puede asegurarse de que el mensaje es realmente parte del protocolo del objeto; éste es un ejemplo de comprobación de la consistencia. Cuando se enuncian restricciones, tales como «no hay más de tres instancias de esta clase», uno puede esperar que una herramienta apoye estas convenciones; éste es un ejemplo de comprobación de restricciones. De la misma forma, una herramienta puede advertir si ciertas clases o métodos de determinada clase nunca se utilizan; es un ejemplo de comprobación de la completitud. Además, una herramienta sofisticada puede decir cuánto tiempo lleva completar cierta opera-

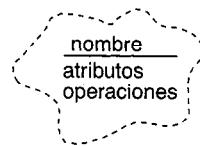


Figura 5.2. Icono de clase.

ción, o si un estado de un diagrama de transición de estados es alcanzable o no; éste es un ejemplo de análisis. En el otro extremo, una herramienta no puede decir que convendría inventar una nueva clase o simplificar la estructura de clases; eso exige discernimiento humano. Se podría considerar la posibilidad de intentar utilizar un sistema experto como herramienta para ese propósito, pero esto requiere por un lado una persona experta simultáneamente en desarrollo orientado a objetos y en el dominio del problema, y por otro lado la capacidad de articular heurísticos de clasificación, así como de plasmar mucho sentido común en forma de conocimiento manejable. No cabe esperar que estas herramientas aparezcan en el futuro próximo; mientras tanto, hay que crear algunos sistemas reales.

5.2. Diagrama de clases

Elementos esenciales: las clases y sus relaciones

Se utiliza un *diagrama de clases* para mostrar la existencia de clases y sus relaciones en la visión lógica de un sistema. Un solo diagrama de clases representa una vista de la estructura de clases de un sistema. Durante el análisis, se utilizan diagramas de clases para indicar las misiones y responsabilidades comunes de las entidades que caracterizan el comportamiento de un sistema. Durante el diseño, se utilizan diagramas de clases para plasmar la estructura de las clases que forman la arquitectura del sistema.

Los dos elementos esenciales de un diagrama de clases son las clases y sus relaciones básicas.

Clases. La Figura 5.2 muestra el icono que se utiliza para representar una clase en un diagrama de clases. Su contorno es el de una nube; algunos lo llaman *mancha amorf*².

² La selección de iconos para cualquier notación es una tarea difícil, y no debe tomarse a la ligera. En realidad, el diseño de iconos es en gran medida un arte, no una ciencia, y requiere un balance cuidadoso entre las demandas que plantean la expresividad y la simplicidad. La elección del icono en forma de nube se deriva del trabajo de Intel en la documentación de su arquitectura original orientada a objetos, la iAPX432 [6]. La intención de este icono es sugerir las fronteras de una abstracción, un concepto que no tiene necesariamente bordes sencillos o sim-

Se requiere un nombre para cada clase; si el nombre es particularmente largo, puede abreviarse éste o bien utilizar un ícono mayor. Todo nombre de clase debe ser único para la categoría de clases que engloba a esta. En muchos lenguajes, los más significativos de los cuales son C++ y Smalltalk, puede además restringirse esta semántica para exigir que todos los nombres de clase sean únicos para el sistema.

En ciertos diagramas de clases, es útil exponer algunos de los atributos y operaciones asociados con una clase. El término «algunos» obedece a que para todas, excepto la clase más trivial, es chapucero y en realidad innecesario mostrar todos esos miembros en un diagrama, incluso cuando se usa un ícono rectangular. En este sentido, los atributos y operaciones mostrados representan una visión abreviada de la especificación completa de la clase, que sirve como el único punto de declaración para todos sus miembros. Si se necesita mostrar muchos de estos miembros, se puede agrandar el ícono de clase; si se decide no mostrar tales miembros de todas formas, se puede eliminar la línea de separación y mostrar solamente el nombre de clase.

Como se describió en el Capítulo 3, un atributo denota una parte de un objeto agregado, y por eso se utiliza tanto durante el análisis como durante el diseño para expresar una propiedad singular de la clase³. Según la siguiente sintaxis independiente del lenguaje, un atributo puede tener un nombre, una clase o ambos, y opcionalmente un valor por defecto:

- A Nombre del atributo solamente.
 - : C Clase del atributo solamente.
 - A : C Nombre y clase del atributo.

bles. Las líneas discontinuas que forman el contorno del icono de clase indican que generalmente los clientes sólo operan sobre instancias de una clase, no sobre la clase misma. Una alternativa aceptable a esta silueta es un rectángulo:

Ésta es la práctica habitual de Rumbaugh [7]. Aunque es más fácil de dibujar a mano, los rectángulos son símbolos intensamente sobreutilizados y de ahí que intuitivamente ya no signifiquen nada. Por otra parte, la elección por parte de Rumbaugh de los rectángulos para las clases y de los rectángulos redondeados para los objetos chocan con otros símbolos de su notación (los rectángulos se utilizan para los actores en los diagramas de flujo de datos, y los rectángulos redondeados se utilizan para denotar los estados en los diagramas de transición de estados). En la práctica, el ícono de nube se presta más para adornos tales como los que se requieren para clases abstractas o para clases parametrizadas, las cuales se examinarán más adelante en este capítulo. Por estas razones, la nube es la forma preferida para su utilización en diagramas de clase y objetos. Especialmente en presencia de soporte automatizado para la notación, el argumento a favor de los rectángulos basándose en su simplicidad de dibujo es discutible. Sin embargo, para facilitar el dibujo manual de diagramas, y para ofrecer un puente hacia el trabajo de Rumbaugh, aquí se permite el rectángulo como alternativa aceptable para representar clases y el rectángulo redondeado para representar objetos.

³ Para ser precisos, un atributo equivale a una asociación de agregación con contención física, cuya etiqueta es el nombre de atributo y cuya cardinalidad es exactamente uno.



Figura 5.3. Marca de las clases abstractas.

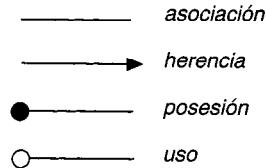


Figura 5.4. Iconos de relación entre clases.

- A : C = E

Nombre, clase y valor por defecto del atributo.

El nombre de atributo debe ser no ambiguo en el contexto de la clase.

Como se describió también en el Capítulo 3, una operación denota algún servicio proporcionado por la clase. Las operaciones se nombran habitualmente cuando aparecen dentro de un ícono de clase, y se distinguen de los atributos añadiendo paréntesis o, si fuese necesario para el propósito del diagrama, proporcionando la descripción completa de la operación:

- N()

Nombre de la operación solamente.

- R N(Argumentos)

Clase de retorno de la operación, nombre y parámetros formales (si los hay).

Los nombres de operación deben ser no ambiguos en el contexto de la clase, de acuerdo con las reglas de la sobrecarga en el lenguaje de implantación elegido.

Como regla general de la notación, la sintaxis para elementos tales como atributos y operaciones puede adaptarse para utilizar la sintaxis del lenguaje de implantación elegido. Esto simplifica la notación al aislar las peculiaridades de varios lenguajes. Por ejemplo, en C++ puede que se desee declarar ciertos atributos como estáticos o ciertas operaciones como virtuales o virtuales puras⁴; en CLOS, puede desearse designar ciertas operaciones como métodos :around. En uno u otro caso, se utiliza la sintaxis específica del lenguaje dado para mostrar estos detalles. Como se describió en el Capítulo 3, una clase abstracta es aquella para la cual no pueden crearse instancias. Ya que tales clases son tan importantes para concebir buenos entramados de clases, se introduce una marca o adorno especial para designar una clase como abstracta, como se muestra en la Figura 5.3. En concreto, se señala el ícono de clase con la letra A (de «abstracta») situada en el interior de un triángulo, en cualquier punto del interior del ícono de clase. Esta marca sigue un principio general de la notación: los «adornos» son elementos secundarios de información acerca de alguna entidad en un modelo de un sistema. Todos los adornos y añadidos de índole similar utilizan el mismo ícono triangular de manera consistente.

Relaciones de clase. Las clases raramente permanecen en solitario; antes

⁴ En C++, static denota un miembro de una clase; virtual denota una operación polimórfica, y pure virtual denota una operación cuya implantación es responsabilidad de las subclases.

bien, como se explicó en el Capítulo 3, colaboran con otras clases de diversas maneras. Las conexiones esenciales entre clases incluyen las relaciones de asociación, herencia, posesión («has») y uso (utilización, «using»), cuyos iconos se resumen en la Figura 5.4. Cada una de esas relaciones puede incluir una etiqueta textual que documenta el nombre de la relación o sugiere su propósito. Los nombres de relación no tienen que ser globales, pero deben ser únicos dentro de su contexto.

El icono de asociación conecta dos clases y denota una conexión semántica. Las asociaciones se etiquetan a menudo con expresiones sustantivas, como *Empleo*, denotando la naturaleza de la relación. Una clase puede tener una asociación consigo misma (llamada asociación *reflexiva*). También es posible tener más de una asociación entre el mismo par de clases. Adicionalmente, puede expresarse la cardinalidad de las asociaciones, como se describió en el Capítulo 3, utilizando la sintaxis de los siguientes ejemplos:

- 1 Exactamente uno.
 - N Número ilimitado (cero o más).
 - 0..N Cero o más.
 - 1..N Uno o más.
 - 0..1 Cero o uno.
 - 3...7 Rango especificado.
 - 1..3, 7 Rango especificado o número exacto.

Se aplica el adorno de la cardinalidad al extremo de destino de una asociación, y denota el número de enlaces entre cada instancia de la clase origen y las instancias de la clase destino. A menos que se señale explícitamente, la cardinalidad de una asociación se considera sin especificar.

Las tres asociaciones fundamentales que quedan se dibujan como refinamientos del ícono de asociación, más general. En realidad, durante el desarrollo, ésta es precisamente la forma como las asociaciones tienden a evolucionar. Primero se afirma la existencia de una conexión semántica entre dos clases y, entonces, a medida que se adoptan decisiones tácticas acerca de la naturaleza concreta de su interrelación, frecuentemente se refina como relación de herencia, posesión o utilización.

El ícono de herencia denota una relación de generalización/especialización (la relación «es un», descrita en el Capítulo 3), y aparece como una asociación con una cabeza de flecha. La flecha apunta a la superclase, y el extremo opuesto de la asociación designa la subclase. De acuerdo con las reglas del lenguaje de implantación elegido, la subclase hereda la estructura y comportamiento de su superclase. También según estas reglas, una clase puede tener una (herencia simple) o más (herencia múltiple) superclases; los conflictos de nombre entre las superclases se resuelven también de acuerdo con las reglas del lenguaje elegido. En general, se prohíbe la existencia de ciclos en las relaciones de herencia. Además, las relaciones de herencia no pueden llevar indicaciones de cardinalidad.

El icono de posesión denota una relación todo/parte (la relación «tiene un», conocida también como *agregación*), y aparece como una asociación con un

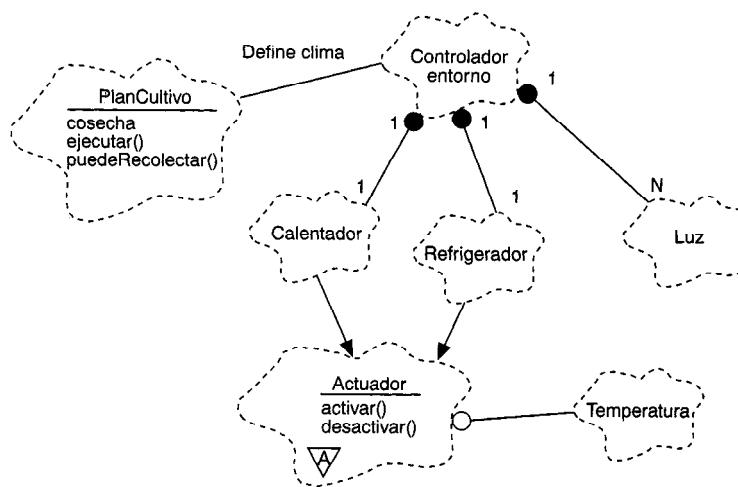


Figura 5.5. Diagrama de clases del sistema de cultivo hidropónico.

círculo (relleno) en el extremo que señala al agregado. La clase que está en el otro extremo denota la parte cuyas instancias están contenidas por el objeto agregado. Es posible la agregación reflexiva y cíclica; la agregación no requiere una inclusión física.

El ícono de utilización denota una relación cliente/proveedor, y aparece como una asociación con una circunferencia (en oposición al círculo del ícono de posesión) en el extremo que denota al cliente. Como se dijo en el Capítulo 3, esta relación indica que el cliente de alguna forma depende del servidor para que éste le proporcione determinados servicios. Se utiliza típicamente para indicar la decisión de que hay operaciones de la clase cliente que invocan operaciones de la clase servidora, o utiliza algún tipo de función cuyo retorno o cuyos argumentos son instancias de la clase servidora.

Ejemplo. Los íconos descritos hasta ahora constituyen los elementos esenciales de todos los diagramas de clases. En conjunto, proporcionan al desarrollador una notación suficiente para describir los aspectos fundamentales de la estructura de clases de un sistema.

En la Figura 5.5 se ofrece un ejemplo de esta notación, dibujado para el problema del sistema de cultivo hidropónico. Este diagrama describe sólo una pequeña parte de la estructura de clases del sistema hidropónico. Aquí se ve la clase **PlanCultivo**, que incluye un atributo llamado `cosecha` junto con una operación que lo modifica, `ejecutar`, y una operación selectora, `puedeRecolectar`. Hay una asociación entre esta clase y la clase **ControladorEntorno**, asociación en la que instancias del plan definen el clima que instancias del controlador se encargan de monitorizar y modificar.

Este diagrama también indica que la clase **ControladorEntorno** es un agre-

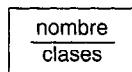


Figura 5.6. Icono de categoría de clases.

gado, cuyas instancias contienen exactamente un calentador, un refrigerador y cualquier número de luces. Las clases Calentador y Refrigerador, por su parte, son ambas subclases de la clase abstracta Actuador, que proporciona el protocolo activar y desactivar, y que utiliza la clase Temperatura.

Elementos esenciales: categorías de clases

Como se explicó en el Capítulo 3, la clase es un vehículo necesario pero no suficiente para la descomposición. Una vez que el sistema ha crecido hasta tener más de alrededor de una docena de abstracciones, se puede comenzar a identificar agrupamientos (*clusters*) de clases que son en sí mismos cohesivos, pero están débilmente acoplados en relación con otros agrupamientos. Se representan estos agrupamientos como categorías de clases.

La mayoría de los lenguajes orientados a objetos no tienen ningún soporte sintáctico para las categorías de clases. Por lo tanto, el proporcionar una notación para las categorías de clases permitirá expresar un importante elemento arquitectónico que de otra forma no puede expresarse directamente en el lenguaje de implementación⁵.

Las clases y las categorías de clases pueden aparecer en el mismo diagrama. De manera más habitual, para representar la arquitectura lógica de alto nivel del sistema, se utilizan algunos diagramas de clases que contienen únicamente categorías de clases.

Categorías de clases. Las categorías de clases sirven para partir el modelo lógico de un sistema. Una categoría de clases es un agregado que contiene clases y otras categorías de clases, en el mismo sentido en que una clase es un agregado que contiene operaciones y otras clases. Cada clase del sistema debe estar en una sola categoría de clases o en el nivel superior del sistema. Al contrario que las clases, una categoría de clases no contribuye directamente al sistema con estados u operaciones; lo hace sólo indirectamente, a través de las clases que contiene.

La Figura 5.6 muestra el icono que se utiliza para representar una categoría de clases. Al igual que para una clase, se requiere un nombre para cada catego-

⁵ El entorno de programación Smalltalk soporta el concepto de categorías de clases. De hecho, ésta fue una de las fuentes de inspiración para introducir categorías en la notación. Sin embargo, en Smalltalk las categorías de clases no tienen contenido semántico: existen únicamente como una conveniencia para organizar la biblioteca de clases de Smalltalk. En C++, las categorías de clases están relacionadas con el concepto de «componentes» desarrollado por Stroustrup, que no son por ahora una característica del lenguaje, aunque se están considerando aspectos de semántica del espacio de nombres para su adopción [8].

ría de clases; si el nombre es particularmente largo, puede abreviarse o puede agrandarse el ícono. Como en las reglas del C++ para nombrar clases, cada nombre de categoría de clases en el modelo lógico debe ser único y distinto de todos los demás nombres de clase.

Para ciertos diagramas de clases, es útil exponer algunas de las clases contenidas en una categoría de clases particular. Una vez más, se dice «algunas» porque la mayoría de las categorías de clases contienen más de un puñado de clases, y de esta forma sería pesado enumerar todas sus clases. Así, como con los atributos y las operaciones mostrados en el ícono de clase, se puede listar los nombres de clases interesantes contenidas en la categoría en cuestión. En este sentido, esta lista de clases representa una visión abreviada de la especificación de la categoría de clase, que sirve como único punto de declaración de todas sus clases. Si se necesita mostrar muchas de tales clases, se puede agrandar el ícono de categoría de clase; si se elige no mostrar ninguna de esas clases, puede omitirse la línea de separación y mostrar sólo el nombre de categoría de clases.

Una categoría de clases representa un espacio de nombres encapsulado. Al igual que en la calificación de nombres del C++, se puede utilizar el nombre de una categoría de clases para calificar sin ambigüedad el nombre de cualquier clase contenida en una categoría. Por ejemplo, dada la clase *c* contenida en la categoría de clases *A*, el nombre completamente calificado es *A::c*. Ya que las clases y las categorías de clases pueden anidarse, como se discutirá posteriormente, puede extenderse esta calificación hasta la profundidad que sea necesario.

Algunas de las clases englobadas por una categoría de clases pueden ser públicas (*public*), lo que significa que son exportadas por la categoría de clases y por lo tanto utilizables fuera de esta categoría. Otras clases pueden ser parte de la implantación, es decir, no pueden ser utilizadas por ninguna otra clase fuera de la categoría de clases. Durante el análisis y el diseño de arquitectura, esta distinción es bastante importante, porque permite especificar una separación de intereses muy clara entre las clases exportadas que proporcionan los servicios de la categoría de clases y las clases que forman la implantación real de esos servicios. De hecho, durante el análisis, típicamente se ignoran los detalles privados de una categoría de clases. Por convención, todas las clases de una categoría de clases se consideran *public*, a menos que explícitamente se definan de otro modo. La restricción del acceso es un concepto avanzado, que se discute en una sección posterior.

Una categoría de clases puede utilizar otra categoría de clases no anidada u otra clase, y una clase puede utilizar una categoría de clases. Por consistencia, se aplica el mismo ícono de la relación «utiliza a» mostrado en la Figura 5.4 para indicar tales conexiones de importación entre categorías de clases. Por ejemplo, considérese una relación «utiliza» de la categoría de clases *A* a la *B*. Esta relación significa que las clases contenidas en *A* pueden heredar de, contener instancias de (poseer), utilizar, y cualquier otra asociación solamente con las clases que *B* exporta.

Surge un problema práctico cuando una categoría de clases contiene una se-

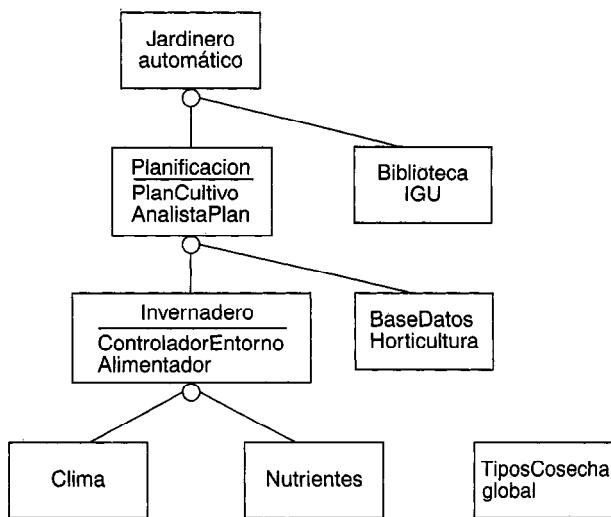


Figura 5.7. Diagrama de clases de nivel superior para el sistema de cultivo hidropónico.

rie de clases comunes, tales como clases contenedoras (*container class*) fundamentales u otras clases básicas que hacen de cimientos, tales como la clase *object* en Smalltalk. Tales clases acaban siendo usadas por prácticamente todas las demás categorías de clases en el sistema, y por tanto hacen que los diagramas de clase del nivel más alto lleguen a quedar abarrotados. Para tratar este inconveniente, se permite que una categoría de clases se señale con la palabra clave *global* situada en la esquina inferior izquierda, indicando que esa categoría puede ser utilizada por todas las demás.

Los diagramas de clases de más alto nivel que contienen solamente categorías de clases representan la arquitectura de alto nivel del sistema. Tales diagramas son extraordinariamente útiles para visualizar las capas y particiones del sistema. Una capa denota la colección de categorías de clases al mismo nivel de abstracción. Así, las capas representan agrupaciones de categorías de clases, del mismo modo que las categorías de clases representan agrupaciones (*clusters*) de clases. Un uso habitual de las capas es aislar las capas más altas de los detalles de las capas más bajas. En contraste, una partición denota cada una de las categorías de clases similares que conviven al mismo nivel de abstracción. Bajo este punto de vista, las capas representan cortes horizontales a través de la arquitectura, y las particiones representan cortes verticales.

Ejemplo. La Figura 5.7 muestra un ejemplo del diagrama de clases del nivel superior para el sistema de cultivo hidropónico. Este es un típico sistema en capas en el cual las abstracciones que están cerca de las fronteras del sistema físico (es decir, los sensores y actuadores de clima y nutrientes) están a los ni-

veles más bajos, y las abstracciones centradas en el usuario están más cerca del extremo superior. La categoría de clases llamada `TiposCosecha` es global, indicando que sus servicios están disponibles para todas las demás categorías de clases. Nótese también que la categoría de clases `Planificación` muestra dos de sus clases interesantes, `PlanCultivo` (la cual se vio en la Figura 5.5) y `AnalistaPlan`. Si se examina con más detalle cualquiera de las ocho categorías de clases mostradas aquí, se hallarán todas sus clases correspondientes.

Conceptos avanzados

Los elementos que se han presentado hasta ahora constituyen las partes esenciales de la notación⁶. Sin embargo, existen a menudo diversas decisiones estratégicas y tácticas que debemos plasmar que requieren una extensión de esta notación básica. Como norma general, hay que atenerse a los elementos esenciales de la notación, y aplicar sólo aquellos conceptos avanzados necesarios para expresar detalles de análisis o diseño que son fundamentales para visualizar o comprender el sistema.

Clases parametrizadas. Algunos lenguajes de programación orientados a objetos, los más conocidos de los cuales son C++, Eiffel y Ada, proporcionan clases parametrizadas. Como se discutió en el Capítulo 3, una clase parametrizada denota una familia de clases cuya estructura y comportamiento están definidas independientemente de sus parámetros de clase formales. Hay que hacer corresponder estos parámetros formales con parámetros actuales (el proceso de instanciación) para formar una clase concreta en esa familia; por clase *concreta* se entiende una que puede tener instancias.

Las clases parametrizadas son suficientemente diferentes de las clases convencionales como para incorporar una marca especial. Como muestra el ejemplo de la Figura 5.8, una clase parametrizada se visualiza como una clase simple, pero con una caja de línea discontinua en la esquina superior derecha indicando sus parámetros formales. Una clase instanciada se señala con una caja de línea continua denotando sus parámetros actuales, coincidiendo sus posiciones con las de los parámetros formales correspondientes. En ambos casos, se pueden incluir opcionalmente los parámetros actuales y formales como texto dentro de la caja.

La relación de instanciación entre una clase parametrizada y su clase instanciada se representa con una línea discontinua que apunta a la clase parametrizada. La mayoría de las veces, la clase instanciada requiere una relación «utiliza» hacia otras clases concretas (tales como `PlanCultivo` en este ejemplo) para utilizarlas como parámetro actual.

Una clase parametrizada no puede tener instancias y no puede ser ella misma utilizada como parámetro. Una clase instanciada define una nueva clase dis-

⁶ En conjunto, todos los elementos esenciales forman la ya nombrada forma «Booch lite» de la notación.

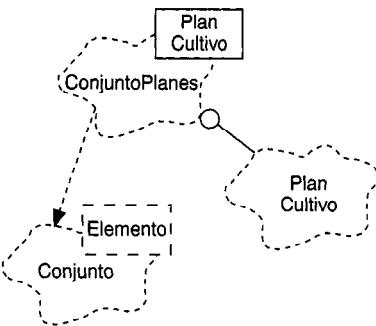


Figura 5.8. Clases parametrizadas.

tinta de todas las demás clases concretas de la misma familia cuyos parámetros actuales sean distintos de los de ella.

Metaclases. Lenguajes como Smalltalk y CLOS proporcionan metaclases. Como se discutió en el Capítulo 3, una metaclass es la clase de una clase. En Smalltalk, el uso más habitual de las metaclases es proporcionar variables de instancia de clase y operaciones, de manera similar al uso en C++ de miembros estáticos, o definir operaciones de fabricación que generen instancias de la clase correspondiente. En CLOS, las metaclases desempeñan un importante papel en la capacidad de adaptar la semántica del lenguaje [9].

Las metaclases son también suficientemente diferentes de las clases normales como para proporcionarles un adorno especial. Como se muestra en la Figura 5.9, una metaclass aparece como una clase simple, pero con un ícono cuyo fondo es gris. La meta-relación se representa como una línea gruesa de color gris con cabeza de flecha, y apunta de una clase a su metaclass. En este ejemplo,

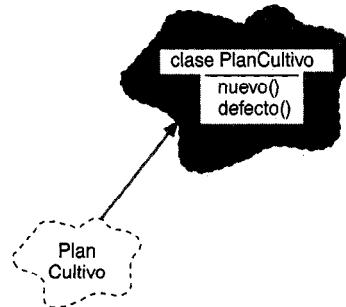


Figura 5.9. Metaclases.

la metaclassa suministra las operaciones de fabricación *nuevo* y *defecto*⁷ para generar nuevas instancias de la clase `PlanCultivo`.

Una metaclassa no puede tener ninguna instancia ella misma, pero puede heredar de, contener instancias de, utilizar y asociarse de otros modos con otras clases.

Las metarrelaciones tienen otro uso más. En algunos diagramas de clases, es útil mostrar un objeto que sirve como miembro estático (*static*) de alguna clase. Para mostrar la clase de este objeto, se puede dibujar una metarrelación desde el objeto a su clase. Esto es consistente con el uso inicial: una metarrelación muestra la conexión entre alguna entidad (sea un objeto o una clase) y su clase.

Utilidades de clase. Debido a herencias de su pasado, lenguajes híbridos, como C++, Object Pascal y CLOS, permiten al desarrollador aplicar un estilo de programación tanto procedimental como orientado a objetos. Esto contrasta con el estilo de Smalltalk, donde todo el lenguaje está en última instancia organizado en torno a las clases. En los lenguajes híbridos, es posible escribir funciones que no son miembros de ninguna clase, también llamadas *subprogramas libres*. Los subprogramas libres suelen aparecer durante el análisis y el diseño en los límites que separan el sistema orientado a objetos de sus interfaces procedimentales con entidades del mundo real.

Las utilidades de clase se manifiestan de una o dos formas. Primero, una utilidad de clase puede denotar uno o más subprogramas libres; el nombre de la utilidad de clase no tiene entonces más significado que el de dar un nombre adecuado a un grupo lógico de tales funciones no-miembro. Segundo, una utilidad de clase puede nombrar a una clase que solamente tiene operaciones y variables de instancia de clase; en C++, esto denotaría una clase que sólo tiene miembros *static*⁸. Tales clases no tienen instancias significativas, principalmente porque no pueden existir estados asociados con ninguna instancia. En cierta manera, la clase por sí misma actúa como la única instancia sobre la que puede operarse.

Como se muestra en la Figura 5.10, una utilidad de clase se representa como un ícono para una clase normal al que se le añade una sombra. En este ejemplo, la utilidad de clase `MetricasPlan` proporciona dos operaciones interesantes, `cosechaEsperada` y `TiempoParaRecoleccion`. La utilidad de clase construye estas dos operaciones mediante los servicios de las clases de nivel inferior `Plancultivo` y `BaseDatosCosecha`. Como indica el diagrama, `MetricasPlan` depende de `BaseDatosCosecha` para recuperar información histórica sobre algunos cultivos interesantes. Por su parte, la clase `AnalistaPlan` utiliza los servicios de `MetricasPlan`.

La Figura 5.10 ilustra un motivo habitual para usar utilidades de clase: he

⁷ La expresión inglesa *default* se utiliza en Informática para aludir a valores o características tomados por omisión, es decir, *por defecto*. Aquí se ha traducido por el término «defecto» con el significado descrito anteriormente, y no el de uso corriente (fallo o error de fabricación). (*N. del T.*)

⁸ Esta convención lingüística también se usa habitualmente por programadores de Smalltalk para lograr el mismo efecto que en C++.

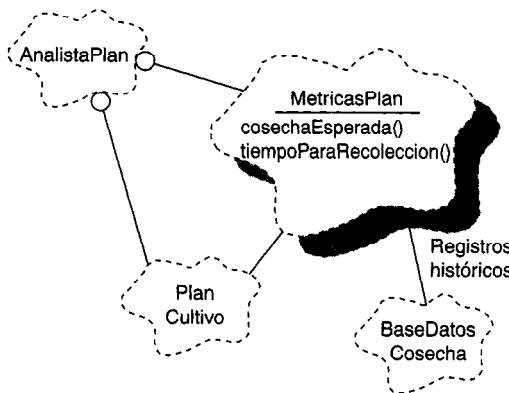


Figura 5.10. Utilidades de clase.

aquí una utilidad de clase que proporciona algunos servicios algorítmicos comunes construidos a través de dos abstracciones dispares de nivel inferior. Mejor que asociar estas operaciones con una clase de nivel superior como *AnalistaPlan*, se elige recoger ambas en una utilidad de clase, de forma que hay una separación clara de intereses entre estas utilidades procedimentales más simples y la abstracción más complicada del analista. Además, el recoger estos subprogramas libres en una estructura lógica aumenta su posibilidad de reutilización, porque esto provoca una granularidad más fina de la abstracción.

Como muestra el ejemplo, las clases pueden asociarse con y utilizar, pero no heredar de o contener una instancia de una utilidad de clase. Igualmente, una utilidad de clase puede asociarse con, utilizar o contener instancias estáticas de otras clases, pero no heredar de ellas.

Igual que las clases convencionales, las utilidades de clase pueden parametrizarse y en su momento ser instanciadas. Para denotar tales utilidades de clase, se puede aplicar los mismos añadidos gráficos que para las clases parametrizadas e instanciadas, como se ve en la Figura 5.8. Se puede también utilizar la misma relación de instanciación mostrada en esa figura para denotar la relación entre una utilidad de clase parametrizada y su instanciación.

Anidamiento. Las clases pueden anidarse físicamente en otras clases, y las categorías también pueden anidarse en otras categorías, hasta cualquier nivel de anidamiento, habitualmente para conseguir algún control sobre el espacio de nombres. En cualquier caso, este anidamiento corresponde a la declaración de la entidad anidada que se produce en el contexto que la engloba. Como se ve en la Figura 5.11, se indica el anidamiento mediante el anidamiento físico de los iconos; el nombre calificado de la clase anidada es *Alimentador::PerfilNutrientes*. De acuerdo con las reglas del lenguaje de implantación elegido, las clases pueden contener instancias de clases anidadas o utilizar

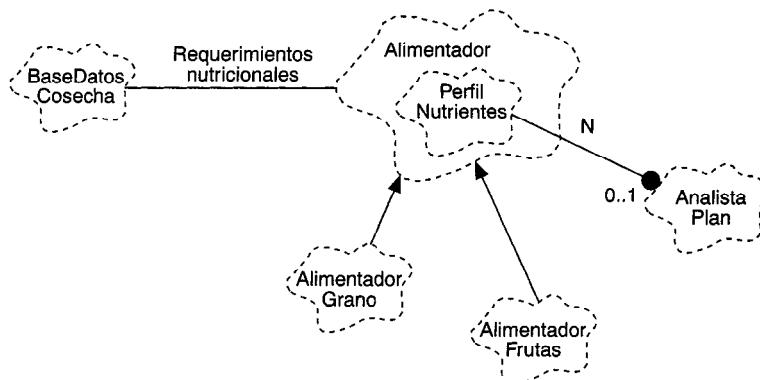


Figura 5.11. Anidamiento.

clase anidadas. Habitualmente, los lenguajes no permiten heredar de clases anidadas.

El anidamiento de las clases tiende a ser una decisión táctica de diseño. El anidamiento de categorías de clases, sin embargo, suele ser una decisión de estrategia arquitectónica. En ambos casos, rara vez existe una razón que obligue a anidar clases o categorías de clases hasta profundidades mayores de uno o dos niveles.

Control de exportación. Todos los lenguajes interesantes de programación orientada a objetos ofrecen una separación clara entre el interfaz y la implantación de una clase. Como se describió en el Capítulo 3, la mayoría permiten además al desarrollador especificar un acceso de grano más fino al interfaz. Por ejemplo, en C++ los miembros pueden ser `public` (accesibles a todos los clientes), `protected` (accesibles sólo a las subclases, las clases amigas o a la propia clase) o `private` (accesibles sólo a la propia clase y sus amigas). Ciertos elementos podrían ser también parte de la implantación de una clase, y ser, por tanto, inaccesibles incluso para los amigos de la clase⁹. Análogamente, en Ada, los elementos de una clase pueden ser `public` o `private`. En Smalltalk, todas las variables de instancia son `private` por defecto, y todas las operaciones son `public`. El acceso se garantiza explícitamente por parte de la propia clase, y no se toma por la fuerza por parte del cliente.

Se puede especificar acceso marcando la relación apropiada con los siguientes símbolos:

- <sin marcas> Acceso público (la opción por defecto, `public`).
- | Acceso protegido (`protected`).
- || Acceso privado (`private`).
- ||| Acceso de implementación.

⁹ Por ejemplo, considérese un objeto o clase declarado en un fichero `.cpp` y, por tanto, accesible solamente a las funciones miembro implementadas en él.

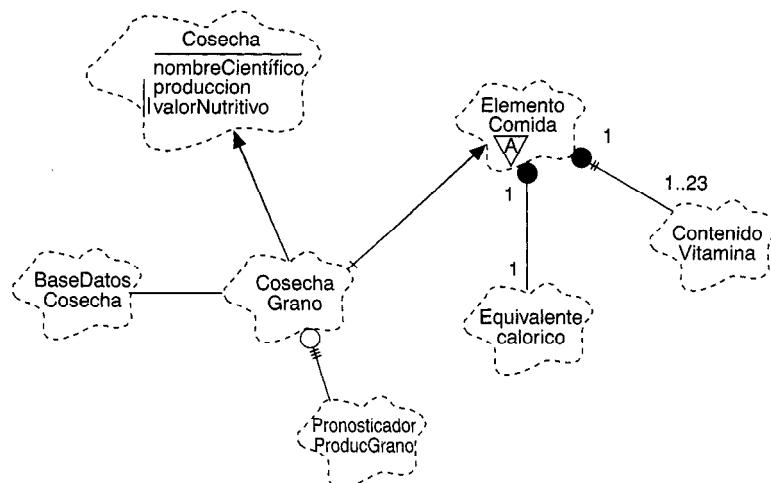


Figura 5.12. Control de exportación.

Se colocan estas marcas en el extremo de partida de una relación. Por ejemplo, en la Figura 5.12, se advierte que la clase **CosechaGrano** tiene herencia múltiple de la clase **Cosecha** (una superclase pública) y de la clase abstracta **ElementoComida** (una superclase protegida). **ElementoComida** a su vez contiene de 1 a 23 instancias privadas de la clase **ContenidoVitamina**, y una instancia pública de la clase **EquivalenteCalorico**. Nótese que **EquivalenteCalorico** podía haberse escrito como un atributo de la clase **ElementoComida**, porque los atributos son equivalentes a la agregación cuya cardinalidad es exactamente 1:1. Continuando, se ve que la clase **CosechaGrano** usa la clase **Pronosticador_ProducGrano** como parte de su implantación. Esto típicamente significa que hay algún método de la clase **CosechaGrano** que usa los servicios de **Pronosticador_ProducGrano** en su implantación.

Además de las clases mostradas en este ejemplo, las asociaciones sencillas pueden marcarse también con símbolos de acceso. La instanciación y las relaciones de metaclasses no pueden señalarse de esta forma.

Estos símbolos de acceso se aplican también a entidades anidadas en todas sus formas. Concretamente, en un ícono de clase puede indicarse la accesibilidad de los atributos, operaciones y clases anidadas, prefijando el nombre del elemento anidado con uno de los símbolos de acceso. Por ejemplo, en la Figura 5.12, se ve que la clase **Cosecha** tiene un atributo público (**nombreCientífico**), uno protegido (**producción**) y uno privado (**valorNutritivo**). Se aplica esta misma notación a las clases y categorías de clases anidadas en otras categorías de clases. Por defecto, todas las clases y categorías de clases anidadas son públicas, pero se puede indicar un acceso restringido añadiendo la marca que denota acceso de implantación.

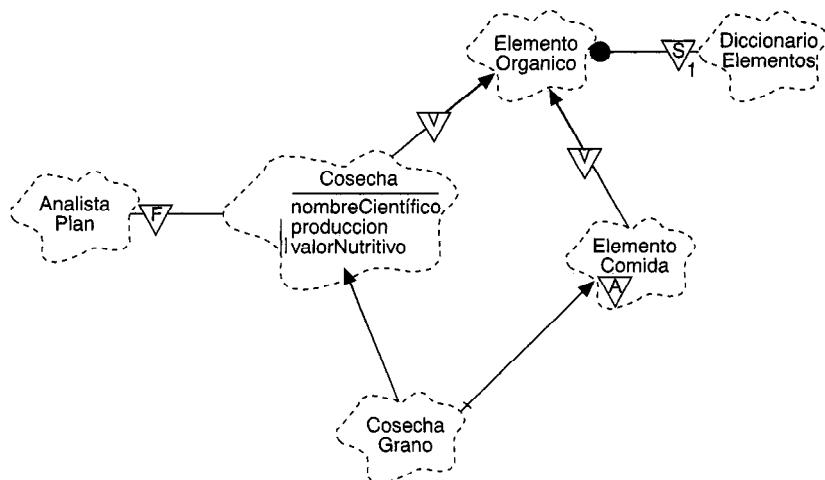


Figura 5.13. Propiedades.

Propiedades. En ciertos lenguajes, hay algunas calificaciones de relación tan penetrantes y de semántica tan fundamental que justifican el uso de símbolos especiales. En C++, por ejemplo, hay tres de tales propiedades:

- **static** La designación de un objeto o función miembro de una clase.
- **virtual** La designación de una clase base compartida en una trama de herencias con forma de rombo.
- **friend** La designación de una clase que concede a otra derechos de acceso a sus partes no públicas.

Por consistencia, se dibujan estas marcas utilizando el mismo ícono triangular que se usa para la marca de clase abstracta, pero con los símbolos **s**, **v** y **f**, respectivamente.

Considérese el ejemplo de la Figura 5.13, que proporciona una visión diferente de las clases mostradas en la figura previa. Aquí se ve que la clase base **ElementoOrganico** contiene una instancia de la clase **DiccionarioElementos**, y que esta instancia la posee la propia clase, no sus instancias individuales. En general, se puede aplicar el adorno **static** a cualquier extremo de una asociación o al extremo de partida de una relación de posesión.

Observando la clase **CosechaGrano**, se ve que su trama de herencias toma una forma de rombo (bifurcación-unión). Por defecto en C++, una trama romboidal genera para la clase hoja duplicados del estado de la clase base compartida. Con el fin de que la clase **cosechaGrano** tenga una sola copia del estado multi-heredado de **ElementoOrganico**, puede especificarse herencia **virtual**, como se muestra en la Figura 5.13. Sólo puede aplicarse el adorno **virtual** a una relación de herencia.

La amistad puede aplicarse al servidor de cualquier relación, denotando que el servidor ha concedido el derecho de amistad al cliente. Por ejemplo, en la Figura 5.13 se ve que la clase `AnalistaPlan` es una amiga de la clase `Cosecha`, y, por tanto, tiene acceso a sus miembros no públicos, incluyendo los atributos `produccion` y `nombreCientifico`.

Contención física. Como se hizo notar en el Capítulo 3, la agregación, tal como se manifiesta en la relación «de posesión» o «tiene» (*has*), es una forma restringida de la relación de asociación (más general). La agregación denota una jerarquía todo/parte, y también implica la capacidad de ir del agregado a sus partes. Esta jerarquía todo/parte no significa necesariamente contención física: una sociedad profesional tiene una serie de miembros, pero de ningún modo «posee» a sus miembros. Por contra, un registro individual de una historia de la cosecha contiene físicamente información subordinada, como el nombre de cosecha, producción y planes de nutrición aplicados.

La elección de la agregación suele ser una decisión de análisis o diseño arquitectónico; la elección de agregación por contención física es normalmente una cuestión táctica, de detalle. Sin embargo, la determinación de la contención física es importante por dos razones: primero, tiene una semántica que desempeña un papel en la construcción y destrucción de las partes de un agregado, y segundo, la especificación de la contención física es necesaria para la generación de código significativo a partir del diseño y para la ingeniería inversa a partir de la implantación.

La contención física se indica con una marca en el extremo de partida de una relación «de posesión» o de pertenencia (*has*); la ausencia de esta marca significa que la decisión respecto a la contención física está sin especificar. En lenguajes híbridos, se distingue entre dos tipos de contención física:

- Por valor Denota contención física de un valor de la parte.
- Por referencia Denota contención física de un puntero o referencia a la parte.

En lenguajes de programación orientados a objetos puros, entre los que destaca Smalltalk, toda contención es por referencia.

Puesto que la contención física y sus nociones correspondientes de comparición estructural son suficientemente diferentes de la semántica de las propiedades tratadas anteriormente, se ha elegido un estilo de señalización ligeramente distinto. En concreto, se usa un cuadrado relleno para denotar agregación por valor, y un cuadrado hueco para denotar agregación por referencia. Como se verá en una sección posterior, este estilo de marca es consistente con los adornos que representan semánticas físicas semejantes en los diagramas de objetos.

Considérese el ejemplo de la Figura 5.14. Aquí se ve la clase `Historia-Cosecha`, cuyas instancias contienen físicamente N instancias de la clase `Plan-Nutricion` y N instancias de la clase `EventoClima`. La contención por valor implica que la construcción y destrucción de estas partes ocurre como consecuencia

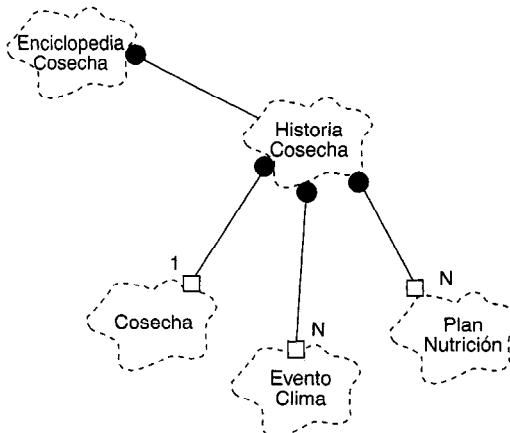


Figura 5.14. Contención física.

de la construcción y destrucción del agregado. En concreto, la contención por valor asegura que los tiempos de vida del agregado y sus partes son iguales. En contraste, cada instancia de **HistoriaCosecha** contiene físicamente sólo una referencia o puntero a una instancia de **Cosecha**. Esto quiere decir que los tiempos de vida de los dos objetos son independientes, aunque el uno se sigue considerando parte física del otro. Esto contrasta también con la relación de agregación que se da entre las clases **EncyclopediaCosecha** e **HistoriaCosecha**. Aquí no se ha especificado contención física. La semántica de esta parte del diagrama dice que estas dos clases participan realmente en una relación todo/parte, y que es posible viajar de una instancia de **EncyclopediaCosecha** a una instancia de **HistoriaCosecha**, aunque esto no puede ser resultado de la contención física. En vez de eso, pueden existir algunos mecanismos mucho más elaborados que implanten esa asociación; por ejemplo, puede ser necesario que **EncyclopediaCosecha** inicie una búsqueda sobre algún otro agente, como un actor que ejerza como base de datos, para buscar la instancia apropiada de **HistoriaCosecha** y devolver una referencia compartida a ella.

Papeles (roles) y claves. En el capítulo anterior se describió la importancia que tiene la identificación de los diversos papeles que un objeto desempeña en colaboración con otros objetos; en el siguiente capítulo, se estudiará cómo la identificación de roles ayuda a guiar el proceso de análisis.

Brevemente, el papel (*rol*) de una abstracción es la cara que presenta al mundo en determinado momento. Un papel denota el propósito o capacidad por la que una clase se asocia con otra. Como muestra el ejemplo de la Figura 5.15, se nombra el papel de una clase mediante un etiquetado textual a cualquier asociación, colocado en una posición adyacente a la clase que ofrece ese papel. Aquí se aprecia que las instancias de la clase **AnalistaPlan** y **Alimen-**

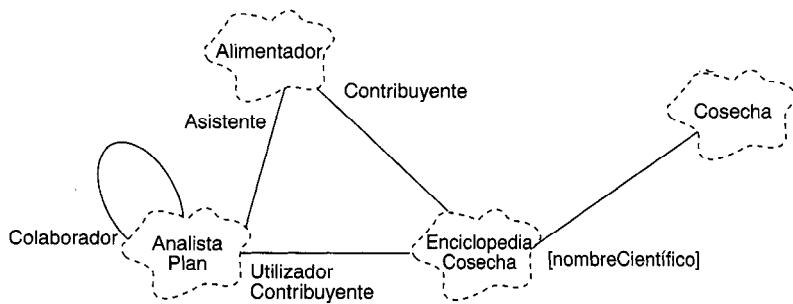


Figura 5.15. Papeles (roles) y claves.

tador son ambas contribuyentes del objeto `EnciclopediaCosecha` (lo que significa que ambas añaden información a la enciclopedia), y que los objetos `AnalistaPlan` son también usuarios (lo que significa que buscan información en la enciclopedia). En cada caso, el papel del cliente identifica el comportamiento y protocolo particulares que utiliza con su servidor mientras ejecuta ese papel. Nótese también la asociación reflexiva en la clase `AnalistaPlan`; aquí se muestra que múltiples instancias de esta clase pueden colaborar entre sí, y que tienen un protocolo particular que utilizan cuando colaboran, el cual se distingue de su comportamiento en su asociación con, por ejemplo, `EnciclopediaCosecha`.

El ejemplo también muestra una asociación entre las clases `EnciclopediaCosecha` y `Cosecha`, pero con un tipo diferente de marca, representando esta a una clave, que aparece como un identificador entre corchetes. Una clave es un atributo cuyo valor identifica un objeto de manera única. En este ejemplo, la clase `EnciclopediaCosecha` usa el atributo `nombreCientífico` como clave para acudir a entradas individuales en el conjunto de elementos manejado por instancias de `EnciclopediaCosecha`. En general, una clave debe ser un atributo del objeto que es parte del objeto agregado situado en el extremo de destino de la asociación. Son posibles las claves múltiples, pero los valores de clave deben ser únicos.

Restricciones. Como se trató en el Capítulo 3, una restricción es la expresión de alguna condición semántica que hay que preservar. Dicho de otro modo, una restricción es un invariante de la clase o relación que hay que preservar mientras el sistema esté en un *estado estable*. Se hace hincapié en el término *estado estable* porque pueden darse circunstancias transitorias en las que el estado del sistema esté cambiando (y por tanto esté de manera temporal en un estado no autoconsistente), durante las cuales es imposible preservar todas las restricciones del sistema. Las restricciones son garantías que se aplican sólo cuando el estado del sistema es estable.

En cuanto a notación, se usa para las restricciones un adorno similar al usado para papeles y claves: en concreto, se sitúa una expresión entre llaves, adyacente

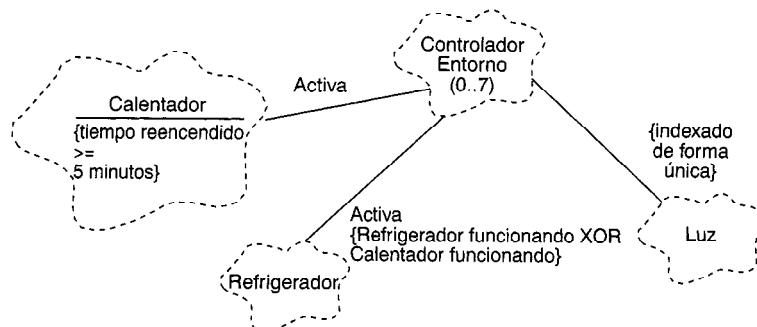


Figura 5.16. Restricciones.

a la clase o relación a la que se aplica la restricción. Como indica el ejemplo de la Figura 5.16, se puede aplicar restricciones a clases individuales, asociaciones enteras, y participantes en una asociación.

En este diagrama, se ve una restricción de cardinalidad sobre la clase `ControladorEntorno`, indicando que no puede haber más de 7 instancias de esta clase en el sistema. En ausencia de una restricción de cardinalidad, una clase puede tener cero o más instancias. El adorno de clase abstracta descrito anteriormente es un caso especial (que denota una cardinalidad cero), pero ya que se da tan frecuentemente en tramas de clases, se le ha otorgado un símbolo especial (la marca triangular).

La clase `Calentador` tiene un tipo de restricción distinto. Aquí se ve que hay una sentencia de histéresis en el funcionamiento del calentador: un calentador no puede volver a activarse antes de que hayan pasado cinco minutos desde la última vez que se desconectó. Se añade esta restricción a la clase `Calentador`, porque se quiere indicar con esto un invariante preservado por las propias instancias de la clase.

En este diagrama aparecen también dos tipos diferentes de restricciones de asociación. En la asociación entre las clases `ControladorEntorno` y `Luz`, se requiere que las luces individuales se indexen de forma única respecto al resto en el contexto de esta asociación. Se tiene también una restricción que se extiende sobre la asociación del controlador con las clases `Calentador` y `Refrigerador`, estableciendo el invariante de que el `ControladorEntorno` no puede activar el calentador y el refrigerador al mismo tiempo. Se coloca esto como una restricción sobre la asociación en lugar de una restricción sobre la clase `Calentador` o `Refrigerador`, porque es un invariante que no pueden preservar los calentadores o refrigeradores por sí mismos.

Si es necesario, pueden escribirse expresiones de restricción que nombren a otras asociaciones, utilizando la sintaxis para nombres calificados que se usa en otras partes de la notación. Por ejemplo, `Calentador::Activa` nombra inequívocamente una de las asociaciones del controlador. En la notación, tales expresiones se usan con frecuencia en circunstancias en las que una clase tiene una

asociación (pongamos agregación) con otras dos o más clases, pero sus instancias pueden asociarse solamente con una de estas instancias de destino en un momento dado.

Las restricciones son útiles también para la expresión de clases, atributos y asociaciones secundarias¹⁰. Por ejemplo, considérense las clases `Adulto` y `Chiquillo`, las cuales podrían ser ambas subclases de la clase abstracta `Persona`. Para la clase `Persona`, se podría suministrar el atributo `fechaNacimiento`, y podría también incluirse un atributo llamado `edad`, quizás porque la edad es importante en nuestro modelo del mundo real. Sin embargo, el atributo `edad` es secundario; puede calcularse a partir de `fechaNacimiento`. Así, en el modelo, podrían incluirse ambos atributos, pero podría incluirse también una restricción de atributos que estableciese esa derivación. Es una decisión táctica qué atributo se deriva del otro, pero la restricción puede registrar cualquier decisión que se tome.

Análogamente, se podría tener una asociación entre las clases `Adulto` y `Chiquillo`, llamada `Padre`. Podría incluirse también otra asociación llamada `Cuidador`, porque conviniese a los propósitos del modelo (quizás se están modelando las relaciones legales entre padre e hijo en el análisis de un sistema de bienestar social). `Cuidador` es secundario; deriva de las consecuencias de la asociación `Padre`, y se podría establecer este invariante como una restricción sobre la asociación `Cuidador`.

Asociaciones atribuidas y notas. El último concepto avanzado específico de los diagramas de clases se interesa por el problema de modelar propiedades de las asociaciones; la solución notacional a este problema específico se generaliza a un elemento de los diagramas que puede aplicarse a cualquier diagrama de la notación.

Considérese el ejemplo de la Figura 5.17. Se ve una asociación muchos a muchos entre `Cosecha` y `Nutriente`, que significa que cada cosecha depende de `N` nutrientes, y cada nutriente puede aplicarse a `N` cosechas diferentes. La clase `PlanNutricion` es verdaderamente una propiedad de esta relación muchos a muchos, cuyas instancias denotan una correspondencia específica entre una cosecha y sus nutrientes. Para indicar este hecho semántico, se dibuja una línea discontinua desde la asociación `Cosecha/Nutriente` (la asociación atribuida) a su propiedad, la clase `PlanNutricion` (el atributo de la asociación). Una asociación dada puede tener como mucho un atributo de estas características, y el nombre de tal asociación debe ir en consonancia con el nombre de la clase utilizada como atributo.

La propia idea de las asociaciones atribuidas tiene una generalización. Concretamente, durante el análisis y el diseño existe una cantidad muy grande de suposiciones y decisiones aparentemente aleatorias que puede recoger cada de-

¹⁰ En términos utilizados por Rumbaugh, se les llama *entidades derivadas*, para las cuales él propone un solo símbolo de señalización. Nuestro enfoque general de las restricciones es suficiente para expresar la semántica de clases, atributos y asociaciones derivados, y tiene las ventajas de reutilizar un elemento notacional ya existente, así como de identificar sin ambigüedades la entidad de la que se parte para realizar la derivación.

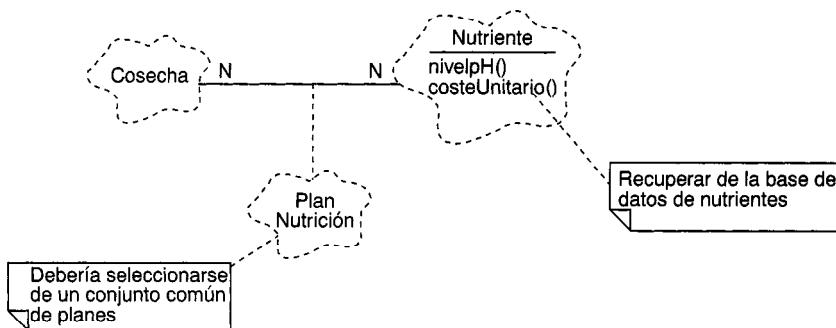


Figura 5.17. Asociaciones atribuidas y notas.

sarrollador; estas interioridades se pierden a menudo, porque no hay normalmente un lugar en el que recogerlas, excepto mantenerlas en la cabeza del desarrollador (práctica decididamente poco fiable). Así, es útil añadir notas arbitrarias a cualquier elemento del diagrama, cuyo texto capture estas suposiciones y decisiones. En la Figura 5.17, hay dos de tales notas. Una nota, unida a la clase `PlanNutricion`, dice algo sobre la unicidad esperada de sus instancias. La otra nota, asignada a una operación específica de la clase `Nutriente`, captura las expectativas que se tienen sobre cómo se implantará esa operación.

Para tales notas se usa un ícono distintivo en forma de nota y se conecta al elemento al que afecta mediante una línea discontinua como la usada antes. Las notas, en gran medida una cuestión de herramientas, pueden contener cualquier información, incluyendo simple texto, fragmentos de código o referencias a otros documentos. Una nota puede estar sin conexión, lo que significa que se aplica al diagrama en su conjunto¹¹.

Especificaciones

Una especificación es una forma no gráfica que se usa para proporcionar la definición completa de una entidad de la notación, como una clase, una asociación, una operación individual o incluso un diagrama completo. El hojear a través de los diagramas de clases permite al lector visualizar un sistema grande con relativa facilidad; sin embargo, esta visión gráfica por sí sola no es suficiente: hay que disponer de alguna substancia tras los dibujos, y esta es la motivación a la que obedecen las especificaciones en general.

Como ya se ha dicho, un diagrama no es más que una vista dentro de un modelo del sistema que se desarrolla. Una especificación sirve por tanto como el modelo fundamental de partida para cualquier entidad de la notación. El

¹¹ El ícono que se usa es similar al ícono de nota usado en gran variedad de sistemas de ventanas, especialmente los que incluyen el aspecto y sensación Macintosh. Nuestra inspiración concreta para este elemento de la notación se deriva de las sugerencias de Gamma, Help, Johnson y Vlissides [10].

conjunto de hechos sintácticos y semánticos que se reflejan en cada diagrama es por tanto un subconjunto de, aunque debe ser consistente con, los hechos establecidos en las especificaciones del modelo. Obviamente, las herramientas que soporten la notación pueden desempeñar un importante papel en el mantenimiento de los diagramas y especificaciones en sincronización.

En esta sección se examinarán en primer lugar los elementos esenciales de las dos especificaciones importantes en la notación, y se considerarán entonces sus propiedades avanzadas. No se prestará atención a la presentación exacta de cada especificación —eso es cuestión del aspecto y sensación de herramientas particulares que soporten la notación— ni será necesario presentar la especificación de todos los elementos, tales como las especificaciones de metaclasses o de tipos individuales de relaciones. La mayoría de tales especificaciones son subconjuntos obvios de las especificaciones más importantes, como las de las clases, o bien no aportan información más allá de lo que ya se ha descrito en sus imágenes gráficas equivalentes. Lo que resulta particularmente importante en los párrafos siguientes es la exposición de los elementos de especificación que no tienen analogía en los diagramas; las especificaciones contienen ciertas informaciones que se expresan mejor textualmente, y por tanto no tienen representación gráfica.

Elementos comunes. Todas las especificaciones tienen al menos las siguientes entradas:

Nombre:	identificador
Definición:	texto

El significado del nombre de entidad es obvio; su unicidad depende de la semántica del propio elemento. Por ejemplo, los nombres de clases deben ser únicos al menos para la categoría de clases en que se encierran, mientras que los nombres de operación tienen un ámbito que es local a la clase que las contiene.

Una definición es texto que identifica al concepto o función representado por la entidad, y es apropiado para su inclusión en el diccionario de datos, como se verá en el siguiente capítulo.

Estas son las entradas mínimas para toda especificación. Las herramientas pueden ciertamente definir sus propias entradas, para satisfacer las necesidades de su entorno de computación particular. Además, es importante decir que aunque ciertas especificaciones pueden tener una serie de entradas diferentes, los desarrolladores no están obligados a utilizarlas todas, o a seguir la estúpida regla de que el desarrollo no puede pasar a la fase siguiente hasta que se han llenado todas las partes de una especificación. Las notaciones son una ayuda al desarrollo, no un fin por sí mismas, y por tanto deben utilizarse sólo cuando añadan valor a las actividades del análisis y el diseño.

Especificaciones de clases. Cada clase del modelo tiene exactamente una especificación de clase que proporciona al menos las siguientes entradas:

Responsabilidades:	texto
Atributos:	lista de atributos
Operaciones:	lista de operaciones
Restricciones:	lista de restricciones

Como se discutió en el capítulo anterior, las responsabilidades de una clase son las afirmaciones sobre sus obligaciones de ofrecer determinado comportamiento. El siguiente capítulo explica cómo se usa esta entrada como un recipiente para las responsabilidades de una clase, que se descubren o inventan durante el desarrollo.

Las diversas entradas de atributos, operaciones y restricciones están en paralelo con sus duplicados gráficos. Las operaciones individuales son lo bastante interesantes como para justificar sus propias especificaciones, que se presentan en la sección siguiente.

Estos primeros elementos esenciales pueden suministrarse con términos del lenguaje de implementación dado. En particular, puede ser suficiente escribir declaraciones de clase de C++ o especificaciones de paquete de Ada para capturar esta información.

Como se discutió en el Capítulo 3, el comportamiento de ciertas clases interesantes se expresa mejor con frecuencia mediante el uso de máquinas de estados, y así se puede añadir otra entrada esencial para tales clases:

Máquina de estados: referencia a la máquina de estados

Los usos avanzados de la notación requieren las siguientes entradas adicionales para las especificaciones de clase:

Control de exportación: public | implantación
Cardinalidad: expresión

Estos elementos se corresponden con sus imágenes gráficas avanzadas.

Las clases parametrizadas e instanciadas deben incluir la siguiente entrada:

Parámetros: lista de parámetros genéricos formales
o actuales

Las siguientes entradas muy avanzadas no tienen correspondencia gráfica; sirven para capturar ciertos aspectos funcionales de una clase:

Persistencia: transitorio | persistente
Concurrencia: secuencial | vigilada | síncrona | activa
Complejidad espacial: expresión

El primero de estos tres elementos capture la propiedad que denota cuándo las instancias de la clase son o no persistentes. Como se vio en el Capítulo 2, una entidad persistente es aquella cuyo estado trasciende el tiempo de vida del ob-

jeto que la encierra, mientras que una entidad transitoria es aquella cuyo estado y ciclo de vida son idénticos.

Como se dijo también en el Capítulo 2, la concurrencia de una clase es una declaración sobre su semántica en presencia de múltiples hilos de control. Por defecto, el objeto es secuencial, y denota una clase cuya semántica se garantiza sólo en presencia de un único hilo de control. Una clase vigilada es aquella cuya semántica se garantiza en presencia de múltiples hilos de control, pero que requiere colaboración entre todos los hilos de los clientes para conseguir la exclusión mutua. Una clase sincronizada es lo mismo, excepto en que la exclusión mutua la proporciona la propia clase. Por último, una clase activa incorpora su propio hilo de control.

La complejidad espacial de una clase es una declaración sobre el almacenamiento relativo o absoluto consumido por cada objeto de la clase. Se puede usar esta entrada para estimar un tamaño para cada clase, o para registrar la complejidad espacial de la construcción de las instancias de la clase.

Especificaciones de operaciones. Para cada operación miembro de una clase, y para todos los subprogramas libres, se define una especificación de operación que ofrece al menos las siguientes entradas:

Clase de retorno: referencia a una clase
Argumentos: lista de argumentos formales

Estos elementos deben escribirse en el lenguaje de implantación determinado. Dependiendo de la adaptación de la notación a los lenguajes específicos, puede también incluirse el siguiente elemento esencial:

Calificación: texto

En C++, por ejemplo, la calificación incluiría una declaración de las propiedades static, virtual, pure virtual y const de la operación.

Algunos usos avanzados de la notación requieren las siguientes entradas adicionales para la especificación de operaciones:

Control de exportación: public | protected | private | implantación

Los valores que son significativos para el control de exportación son dependientes del lenguaje. En Object Pascal, por ejemplo, los atributos y operaciones son siempre public; en Ada, las operaciones pueden ser public o private, pero en C++ pueden aplicarse los cuatro valores.

El uso avanzado de la notación incluye asimismo el elemento siguiente:

Protocolo: texto

Este elemento sigue la práctica de Smalltalk: el protocolo de una operación no

tiene impacto semántico, sino que sirve simplemente para nombrar una agrupación lógica de operaciones, como `initialize-release` o `model access`.

Las siguientes entradas muy avanzadas no tienen equivalencia gráfica, y sirven para capturar formalmente la semántica de una operación:

```

Precondiciones:
texto | referencia a código fuente | referencia a diagrama de
objetos
Semántica:
texto | referencia a código fuente | referencia a diagrama de
objetos
Postcondiciones:
texto | referencia a código fuente | referencia a diagrama de
objetos
Excepciones:
lista de excepciones

```

Las precondiciones, semántica y postcondiciones de una operación deben declararse en cualquiera de una serie de formas, incluyendo texto (ya sean expresiones informales o formales), referencias a código fuente potencialmente ejecutable o sentencias de afirmación, o referencias a diagramas de objetos que sirven como escenarios de la semántica dada. La entrada de excepciones lista las excepciones que pueden elevarse (*lanzarse**, en términos de C++) por parte de la operación; cada elemento de esta lista es el nombre de una clase que nombra la excepción.

Las últimas entradas muy avanzadas sirven para capturar ciertos aspectos funcionales de una operación:

Concurrencia:	secuencial vigilada síncrona
Complejidad espacial:	expresión
Complejidad temporal:	expresión

Los dos primeros elementos son iguales que para las especificaciones de clases. La complejidad temporal de una operación es una declaración sobre el tiempo absoluto o relativo que requiere el completar una operación. Se puede usar esta entrada para estimar un tiempo para cada operación o para registrar la complejidad temporal en la práctica en términos de eficacia en el caso mejor, medio y/o peor.

5.3. Diagramas de transición de estados

Aspectos esenciales: los estados y las transiciones de estados

Un *diagrama de transición de estados* se utiliza para mostrar el espacio de estados de una clase determinada, los eventos que provocan una transición de un

* *Thrown* en el original en inglés. (N. del T.)

estado a otro, y las acciones que resultan de ese cambio de estado. Se ha adoptado la notación usada por Harel [11] para los diagramas de transición de estados; su trabajo ofrece un enfoque simple pero altamente expresivo que es mucho mejor que las meras máquinas de estados finitos convencionales¹². Un solo diagrama de transición de estados representa una vista del modelo dinámico de una sola clase o del sistema completo. No todas las clases tienen un comportamiento significativo respecto al orden de los eventos; también se pueden suministrar diagramas de transición de estados que muestren el comportamiento respecto al orden de los eventos del sistema en su conjunto. Durante el análisis, se usan diagramas de transición de estados para indicar el comportamiento dinámico del sistema. Durante el diseño, se usan diagramas de transición de estados para capturar el comportamiento dinámico de las clases individuales o de colaboraciones de clases.

Los dos elementos esenciales de un diagrama de transición de estados son los estados y las transiciones entre estados.

Estados. El estado de un objeto representa los resultados acumulados de su comportamiento. Por ejemplo, cuando se instala un teléfono por primera vez, está en estado ocioso, lo que significa que no hay ningún comportamiento anterior de interés especial y que el teléfono está listo para iniciar o recibir llamadas. Cuando alguien levanta el auricular, se dice que el teléfono está ahora descolgado y en estado de marcar; en este estado, no se espera que el teléfono suene; se espera la posibilidad de iniciar una conversación con una o más personas que están en otros teléfonos. Cuando el teléfono está colgado, si suena y a continuación se levanta el auricular, el teléfono estará en estado de recepción, y se espera la posibilidad de hablar con la persona que inició la conversación.

En cualquier momento concreto, el estado de un objeto abarca todas sus propiedades (habitualmente estáticas), junto con los valores actuales (habitualmente dinámicos) de cada una de esas propiedades. Por propiedades se entiende la totalidad de los atributos del objeto y de sus relaciones con otros objetos. Se puede generalizar el concepto del estado de un objeto individual para aplicarlo a la clase del objeto, porque todas las instancias de la misma clase cohabitan en el mismo espacio de estados, que contiene un número indefinido pero finito de estados posibles (aunque no siempre deseables o esperados). La Figura 5.18 muestra el ícono que se utiliza para representar un estado específico.

Se requiere un nombre para cada estado; si el nombre es especialmente largo, puede abreviarse o bien puede agrandarse el ícono. Todo nombre de estado debe ser único en el ámbito que lo encierra, es decir, la clase que lo contiene. Los estados asociados con el sistema en su conjunto tienen un ámbito global, y el ámbito de un estado anidado (un concepto avanzado) se extiende al estado que lo contiene. Todos los iconos de estado con el mismo nombre en un diagrama determinado se considera que se refieren al mismo estado.

¹² Se complementa su trabajo con las contribuciones de Rumbaugh [12] y Bear, Allen, Coleman y Hayes [13], quienes adaptan el trabajo de Harel al dominio de la computación orientada a objetos.

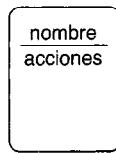


Figura 5.18. Icono de estado.

Para ciertos estados, es útil exponer las acciones asociadas a ellos. Como se muestra en la figura, por razones de consistencia se usa la misma notación que para los atributos y operaciones asociadas con una clase. Si fuese necesario, se puede agrandar el icono de estado; si no hay ninguna acción, se puede eliminar la línea separatoria¹³. La asociación de acciones con un estado es un concepto avanzado, que se discutirá en una sección posterior.

Transiciones entre estados. Un evento es algún suceso que puede causar un cambio de estado en un sistema. Este cambio de estado se llama *transición de estados*, que se representa con el icono mostrado en la Figura 5.19. Cada transición de estados conecta dos estados. Un estado puede tener una transición hacia sí mismo, y es común tener muchas transiciones distintas partiendo del mismo estado, aunque cada una de ellas debe ser única, lo que significa que nunca se darán circunstancias que pudieran disparar más de una transición desde el mismo estado.



Figura 5.19. Icono de transición de estado.

Por ejemplo, en el sistema de cultivo hidropónico, los siguientes eventos juegan un papel en el comportamiento del sistema:

- Se planta una nueva cosecha.
- Una cosecha llega a estar lista para la recolección.
- La temperatura en un invernadero cae a causa de las inclemencias del tiempo.
- Un refrigerador falla.
- El tiempo pasa.

Como se verá en el capítulo siguiente, la identificación de eventos como éstos ayuda a definir los límites del comportamiento de un sistema y a asignar responsabilidades a las clases individuales que llevan a cabo ese comportamiento.

Cada uno de los primeros cuatro eventos de arriba es probable que dispare alguna acción, como iniciar o detener la ejecución de un plan de cultivo espe-

¹³ Por consistencia con la notación de Harel, la línea separatoria puede eliminarse totalmente.

cífico, activar un calentador o hacer sonar una alarma para el jardinero. El paso del tiempo es otra cosa: aunque el paso de segundos o minutos puede no ser significativo para el sistema (el crecimiento observable en una planta suele moverse en escalas mucho mayores de tiempo), el paso de horas o días puede ser una señal para que el sistema encienda o apague las luces o cambie la temperatura de un invernadero, con el fin de crear un día artificial necesario para el crecimiento de las plantas.

Una acción es una operación que, a todos los efectos prácticos, ocupa un tiempo cero. Por ejemplo, hacer sonar una alarma para el jardinero es una acción. Una acción denota típicamente la invocación de un método, el disparo de otro evento, o el inicio o parada de una actividad. Una actividad, por contra, es alguna operación que lleva algún tiempo completar. Por ejemplo, calentar el invernadero es una actividad, disparada mediante el encendido del calentador que debe permanecer así por tiempo indefinido hasta que se lo apague explícitamente.

Aunque es conceptualmente puro, el modelo de Harel para la difusión de eventos debe adaptarse para encajar en el modelo de objetos. Durante el análisis, se pueden nombrar eventos y acciones de modo amplio, con el fin de capturar la comprensión que se tiene sobre el espacio del problema. Sin embargo, una vez que se comienza a asignar esos conceptos a clases, hay que imponer una estrategia particular para su implementación.

Un evento puede ser un nombre simbólico (o un objeto nombrado), una clase, o el nombre de alguna operación. Por ejemplo, el evento `fallo del refrigerador` podría denotar bien un literal o bien el nombre de un objeto. Se puede adoptar la estrategia de que todos los eventos son sólo nombres simbólicos y que cada clase con comportamiento interesante respecto al orden de los eventos proporciona una operación que puede recibir tales nombres y efectuar la acción apropiada. Esta es la estrategia que se toma a menudo en Smalltalk en las arquitecturas modelo-vista-controlador, donde los eventos son nombres simbólicos que son procesados por métodos de actualización. Para mayor generalidad, se puede tratar los eventos como objetos, y definir así una jerarquía de clases de eventos que proporcionan la abstracción que se hace de los eventos específicos. Por ejemplo, podría definirse una clase general de eventos llamada `FalloDispositivo` y subclases especializadas como `FalloRefrigerador` y `FalloCalentador`. Cuando se envía entonces un evento, podría enviarse una instancia de una clase hoja (como `FalloRefrigerador`) o de una superclase más general (como `FalloDispositivo`). Si se especifica entonces la acción que toma el sistema solamente en presencia de un evento `FalloRefrigerador`, entonces se ignorarían de forma intencionada todos los demás tipos de fallos de dispositivo. Por otro lado, si se especifica la acción que toma el sistema en presencia de un evento `FalloDispositivo`, se dispararía la misma acción con independencia del fallo de dispositivo específico que se hubiese enviado. De este modo, se puede hacer que las transiciones de estado exhiban comportamiento polimórfico respecto a la clase de evento que las haya disparado. Por último, se podría definir un evento simplemente como una operación, tal como `Plan-`

`Cultivo::ejecutar()`. Esta aproximación es similar a la de tratar los eventos como nombres simbólicos, excepto en que ya no se requiere una operación de selección de eventos explícita.

La elección que se realice entre estas tres estrategias es irrelevante para el método, siempre y cuando se elija una y se aplique consistentemente en todas las partes del sistema. Típicamente, se usa una nota para indicar qué estrategia aplica cada máquina de estados finitos.

Una acción puede escribirse usando la sintaxis que se muestra en los siguientes ejemplos:

- `calentador.activar()` Una operación.
- `FalloDispositivo` Disparo de un evento.
- `comenzar calentar` Comenzar una actividad.
- `parar calentar` Terminar alguna actividad.

En el caso de una operación o evento, el nombre debe estar en el ámbito del diagrama y, donde sea necesario, debe calificarse con el nombre de clase u objeto conveniente. En el caso de comenzar y terminar una actividad, esa actividad puede denotar una operación (como `Actuador::desactivar()`) o un nombre simbólico (como para los eventos). Típicamente, se usan nombres simbólicos cuando la actividad corresponde a alguna función del sistema, como `recolectar cosecha`.

En todo diagrama de transición de estados debe haber exactamente un estado de partida por defecto, que se designa escribiendo una transición sin etiqueta al estado desde un ícono especial, que aparece como un círculo relleno. Con menos frecuencia, se necesita designar un estado de parada. Normalmente, una máquina de estados asociada con una clase o con el estado en conjunto nunca alcanza un estado de parada; la máquina de estados simplemente deja de existir cuando se destruye el objeto que la comprende. Se designa un estado de parada dibujando una transición sin etiqueta desde el estado a un ícono especial, que tiene la forma de un círculo relleno dentro de otro círculo hueco ligeramente mayor.

Ejemplo. Los iconos descritos hasta aquí constituyen los elementos esenciales de todos los diagramas de transición de estados. En conjunto, proveen al desarrollador de suficiente notación para describir máquinas de estados finitos lisas y llanas, adecuadas para aplicaciones con un número limitado de estados. Los sistemas que tienen un gran número de ellos o que exhiben un comportamiento respecto al orden de los eventos particularmente complicado, incorporando transiciones condicionales o transiciones basadas en estados visitados previamente, requieren el uso de los conceptos más avanzados de los diagramas de transición de estados.

En la Figura 5.20, se ofrece un ejemplo de esta notación esencial, obtenido nuevamente del problema del sistema de cultivo hidropónico. Aquí se ve un diagrama de transición de estados para la clase `ControladorEntorno`, introducido en la Figura 5.5.

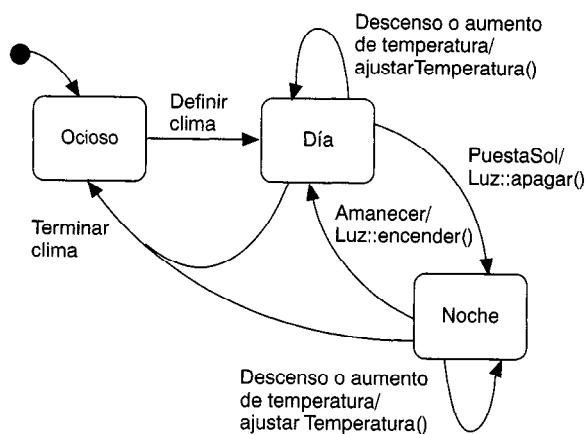


Figura 5.20. Diagrama de transición de estados del controlador de entorno.

En este diagrama se ha elegido una estrategia en la que los eventos se designan como nombres simbólicos. Aquí se ve que los objetos de esta clase comienzan en el estado `Ociooso`; entonces cambian de estado con la recepción del evento `DefinirClima`, para el que no hay acción explícita (para los propósitos de este diagrama, se ha hecho la suposición simplificadora de que este evento ocurrirá sólo durante el día). El comportamiento dinámico de esta clase alterna entonces entre los estados `Día` y `Noche`, disparado por los eventos `PuestaSol` y `Amanecer`, respectivamente, cuya acción es cambiar la iluminación de forma consecuente. En ambos estados, un evento de caída o elevación de la temperatura invoca una acción para ajustar la temperatura (la operación `ajustarTemperatura()`, local a esta clase). Se vuelve al estado `Ociooso` siempre que se recibe un evento `Terminar clima`.

Conceptos avanzados

Los elementos de los diagramas de transición de estados que se han descrito hasta el momento no son suficientes para muchos tipos de sistema complejo, y por esta razón se expande la notación para incluir la semántica de los diagramas de estado de Harel.

Acciones de estado y transiciones de estado condicionales. Como se ve en la Figura 5.18, las acciones pueden asociarse con estados. En particular, se puede especificar alguna acción que va a llevarse a cabo cuando se entra o se sale de un estado, utilizando la sintaxis de los ejemplos siguientes:

- entrada comenzarAlarma Comenzar una actividad al entrar.
- salida desactivar() Invocar una operación al salir.

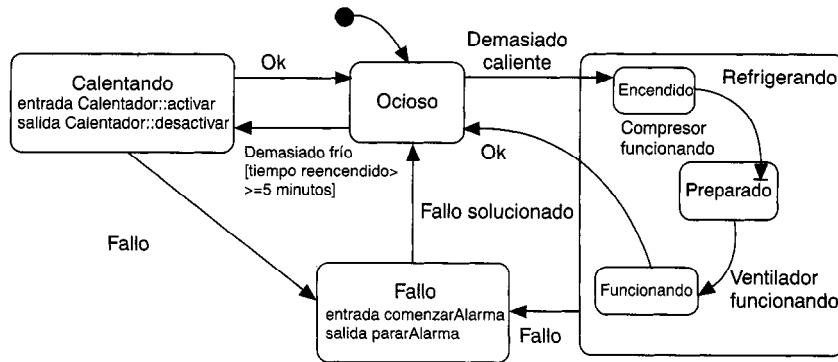


Figura 5.21. Acciones, transiciones condicionales y estados anidados.

Al igual que para las transiciones de estado, se puede especificar cualquier acción tras las palabras clave `entrada` y `salida`.

Las actividades pueden asociarse con un estado, usando la sintaxis del siguiente ejemplo:

- `hacer Refrigerar` Efectuar una actividad mientras se está en el estado.

Esta sintaxis es en gran medida una taquigrafía para el comienzo explícito de la actividad al entrar al estado y el final explícito de la actividad al salir.

En la Figura 5.21 se ve un ejemplo de este concepto avanzado. Aquí se ve que al entrar al estado `Refrigerando`, se invoca la operación `Refrigerador::activar()`, y al salir de este estado, se invoca la operación `Refrigerador::desactivar()`. En caso de entrar y salir del estado `Fallo`, se arranca y se detiene una alarma, respectivamente.

Considérese también la transición de estado de `Ocio` a `Calentando`. Aquí se efectúa transición si la temperatura es demasiado fría, pero sólo si han pasado más de cinco minutos desde que se apagó el calentador. Éste es un ejemplo de transición de estados condicional (o vigilada); se representa una condición como una expresión booleana situada entre llaves.

Generalmente, una transición de estado dada tendrá o bien un evento o bien un evento y una condición. Se permite también que una transición de estado no tenga eventos asociados. En tal caso, la transición se dispara automáticamente inmediatamente después de que se ha completado la acción del estado de partida; como consecuencia se efectúan también las acciones de salida. Si la transición de estado es condicional, entonces la transición será disparada sólo en el caso de que la expresión se evalúe como cierta.

El orden de evaluación en transiciones de estado condicionales es importante. Dado el estado `s` con la transición `T` sobre el evento `E` con condición `c` y acción `A`, se aplica el siguiente orden:

- Ocurre el evento `E`.

- Se evalúa la condición c.
- Si c se evalúa como cierta, entonces se dispara T y se invoca la acción A.

Esto significa que si una condición se evalúa como falsa, la transición de estado no puede dispararse hasta que vuelva a ocurrir el evento y se vuelva a evaluar la condición. Los efectos laterales al evaluar la condición o al llevar a cabo una acción de salida no afectarán al disparo de una transición de estado. Por ejemplo, supóngase que el evento E ocurre, c resulta ser cierta, y entonces la ejecución de una condición de salida cambia el mundo de forma que c pasa a ser falsa; la transición de estados sucederá de todas formas.

Se pueden incluir expresiones que usen la siguiente sintaxis:

- en Refrigerando

Expresión del estado actual.

Aquí se suministra un nombre de estado (que puede calificarse); esta expresión se evalúa como cierta si y sólo si el sistema está en el estado actual. Este tipo de expresión es especialmente útil cuando un estado externo necesita disparar una transición condicional sobre la base de algún estado anidado de nivel más bajo.

También se pueden escribir expresiones condicionales que denoten restricciones de tiempo, como en el ejemplo siguiente:

- cuentaatras(Calentando, 30) Expresión de restricción temporal.

Esta condición se evalúa como cierta si el sistema estaba en el estado Calentando y había estado en él por más de 30 segundos. Este tipo de expresión es común con transiciones de estado sin eventos en muchos sistemas en tiempo real, porque protege contra la permanencia demasiado larga en un estado. Se puede usar también esta expresión para establecer un límite inferior para el tiempo de permanencia en un estado. Si se asigna la misma restricción temporal a todas las transiciones de estado que tienen eventos que provocan la salida de ese estado, esa asignación sería equivalente a exigir que el sistema esté en un estado dado durante, como mínimo, el tiempo especificado por la restricción temporal¹⁴.

¿Qué pasa si llega un evento pero el estado actual no tiene transiciones que lleven a un nuevo estado, ya sea porque no existe tal transición para ese evento concreto o porque ninguna de las condiciones adecuadas se evalúa como cierta? Por defecto, esto debería considerarse un fallo: la indiferencia silenciosa ante un evento suele ser una indicación de un análisis incompleto del problema. En general, un estado debería documentar los eventos a los que ignora intencionalmente.

Estados anidados. La capacidad de anidar estados otorga profundidad a los diagramas de transición de estados; este es el aspecto clave de los diagramas de

¹⁴ Harel sugiere una notación «garabato»* para expresar simultáneamente los límites inferior y superior de las restricciones temporales, pero no se discutirá aquí la generalización que propone, porque las expresiones de cuenta atrás son lo bastante expresivas.

* Squiggle en el original inglés. (N. del T.)

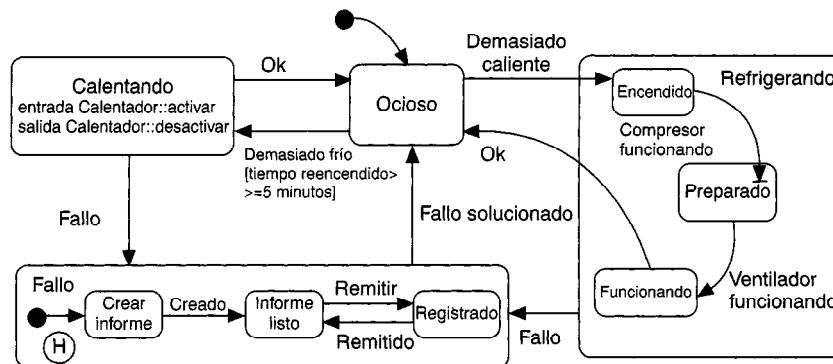


Figura 5.22. Historia.

estados de Harel que mitiga la explosión combinatoria de estados y transiciones que sucede a menudo en sistemas complejos.

En la Figura 5.22, se ha expandido el estado **Refrigerando** para revelar sus estados anidados; por simplicidad, se han omitido todas las acciones, incluyendo las acciones de entrada y salida del estado, como se ven en la Figura 5.21.

Los estados continentales como **Refrigerando** se llaman *superestados*, y sus estados anidados, como **Funcionando**, se llaman *subestados*. El anidamiento puede llegar a cualquier profundidad, y así los subestados pueden ser superestados de otros subestados de nivel inferior. Dado el superestado **Refrigerando** con sus tres subestados, la semántica del anidamiento implica una relación *xor* («o» exclusivo): si el sistema está en el estado **Refrigerando** (el superestado), entonces debe estar también en exactamente uno de los tres subestados, **Encendido**, **Preparado** O **Funcionando**.

Por simplicidad en el dibujo de los diagramas de transición de estados con profundidad, se puede hacer una ampliación o reducción con respecto a un estado particular. La reducción suprime subestados, y la ampliación revela subestados. Cuando se hace reducción, las transiciones de estado hacia y desde subestados aparecen con una flecha romana, como en el caso de la transición hacia el subestado **Preparado**¹⁵.

Se permite que las transiciones de estado se originen y terminen a cualquier nivel. Considérense entonces las diversas formas de transición de estado:

- La transición de un estado a otro estado hermano (como desde **Fallo** hasta **Ociooso** o desde **Preparado** a **Funcionando**) es la forma más simple de transición; sigue la semántica descrita en la sección anterior sobre acciones de estado y transiciones condicionales de estado.
- Puede haber transición directamente hacia un subestado (como desde

¹⁵ En la Figura 5.21, para ser precisos, las transiciones de estado para **Demasiado caliente** y **ok** relativas al estado **Refrigerando** deberían aparecer también romas, porque se efectúan también hacia y desde subestados.

Ocioso a Encendido) o directamente desde un subestado (tal como la transición desde Funcionando hasta Ocioso) o ambas.

- La especificación de una transición directamente desde un superestado (como desde Refrigerando hasta Fallo mediante el evento Fallo) significa que la transición de estado se aplica a todos los subestados del superestado. La transición se pasa a lo largo de todos los niveles, hasta que se anula. Esta semántica reduce en gran medida el embottellamiento de transiciones comunes desde subestados.
- La especificación de una transición directa hacia un estado con subestados (como por ejemplo hacia Fallo) implica realmente pasar al nuevo estado, pero implica también el paso hacia el subestado por defecto de este superestado.

Historia. Frecuentemente cuando se transita directamente a un estado con subestados, se desea volver al estado visitado más recientemente; esta semántica puede indicarse con el icono de historia, que aparece como la letra H dentro de un círculo y situado en cualquier parte directamente dentro del estado. Por ejemplo, en la Figura 5.22 se ha expandido el estado Fallo para revelar sus subestados. La primera vez que se pasa a este estado, se pasa también a su estado de partida por defecto, Crear informe, indicado por la transición sin etiqueta desde el círculo relleno; en última instancia, se crea el registro de informes, y se pasa al estado Informe listo. Tras remitir el fallo, se vuelve a este estado. La siguiente vez que se haga transición hasta el estado Fallo, no se crea de nuevo el registro de informes; en vez de eso, se desea empezar en el estado Informe listo. Ya que este fue el último estado visitado, y ya que se incluyó el icono de historia, ésta es precisamente la semántica que se obtendrá todas las veces siguientes cuando se entre en el estado Fallo.

La historia se aplica sólo al nivel concreto en el que aparece. Puede hacerse que afecte a todos los niveles inferiores de estados anidados añadiendo un asterisco al icono de historia. Se pueden conseguir tipos intermedios de transición histórica aplicando la historia sólo a subestados individuales.

Estados ortogonales Los diagramas de estados de Harel introducen el concepto de estados ortogonales, que representan una descomposición «and» de estados. Dado un sistema en estado A con subestados ortogonales B y C, esto quiere decir que el sistema está en el estado A, así como en ambos estados B y C.

Esta semántica es innecesaria en gran medida una vez que se ha hecho la correspondencia entre diagramas de estados y el modelo de objetos, como se ha hecho ya en esta sección. En concreto, los objetos hermanos cuyas clases tienen comportamiento dependiente del orden de los eventos representan implícitamente una descomposición «and»: el sistema está en el estado denotado por ambos objetos simultáneamente. Por esta razón, se omite la noción de estados ortogonales de Harel.



Figura 5.23. Icono de un objeto.

Especificaciones

Al igual que para los diagramas de clases, cada entidad de un diagrama de transición de estados puede tener una especificación que suministra su definición completa. Sin embargo, al contrario que las especificaciones para las clases, las especificaciones para los estados y las transiciones de estado no añaden información nueva sobre lo que ya se ha descrito en esta sección, y por tanto no se necesita tratar su especificación textual.

5.4. Diagramas de objetos

Elementos esenciales: los objetos y sus relaciones

Un *diagrama de objetos* se utiliza para mostrar la existencia de objetos y sus relaciones en el diseño lógico de un sistema. Dicho de otro modo, un diagrama de objetos representa una instantánea en el tiempo de una corriente (de otro modo transitoria) de eventos sobre una cierta configuración de objetos. Los diagramas de objetos son por tanto prototípicos; cada uno representa las interacciones o relaciones estructurales que pueden darse entre un conjunto determinado de instancias de clases, con indiferencia de qué objetos de nombre concreto participan en la colaboración. En este sentido, un solo diagrama de objetos representa una vista de la estructura de objetos de un sistema. Durante el análisis se usan los diagramas de objetos para indicar la semántica de escenarios primarios y secundarios que proporcionan una traza del comportamiento del sistema. Durante el diseño se usan diagramas de objetos para ilustrar la semántica de los mecanismos en el diseño lógico de un sistema.

Los dos elementos esenciales de un diagrama de objetos son los objetos y sus relaciones.

Objetos. La Figura 5.23 muestra el icono que se usa para representar un objeto en un diagrama de objetos. Al igual que se hacía en los diagramas de clases, se puede dibujar opcionalmente una linea horizontal para dividir el texto del interior del icono en dos regiones, una que denota el nombre del objeto y otra que ofrece una visión opcional de los atributos del objeto.

El nombre de un objeto sigue la sintaxis de los atributos, y puede escribirse en cualquiera de las tres formas siguientes, o utilizando la sintaxis del lenguaje de implantación elegido:

- A Solamente nombre del objeto.
- : C Solamente nombre de la clase.
- A : C Nombre y clase del objeto.

Si el texto es especialmente largo, puede abreviarse o puede agrandarse el icono. Si hay varios iconos de objeto en el mismo diagrama que utilicen el mismo nombre de objeto sin calificar, entonces todos ellos denotan el mismo objeto; de otro modo, cada ícono de objeto denota una ocurrencia distinta del objeto¹⁶. Si varios íconos de objeto usan el mismo nombre en diferentes diagramas, entonces denotan a objetos diferentes, a menos que su nombre se califique explícitamente.

El significado de los nombres sin calificar depende del contexto del diagrama de objetos. En concreto, los diagramas de objetos definidos al nivel más alto del sistema tienen un ámbito global; pueden definirse otros diagramas de objetos para las categorías de clases, clases individuales o métodos individuales, teniendo así el ámbito que corresponda. La calificación puede usarse también según sea necesario para referirse explícitamente a objetos globales, variables de instancia de clase (en C++, objetos miembro static), parámetros de métodos, atributos y objetos definidos localmente del mismo ámbito.

Si no se especifica nunca la clase de un objeto, ya sea explícitamente mediante la sintaxis descrita arriba o implícitamente a través de la especificación del objeto, entonces la clase del objeto se considera anónima, y no puede haber comprobación semántica alguna, ni sobre el significado de las operaciones realizadas sobre o por parte del objeto, ni sobre la relación del objeto con cualquier otro objeto del diagrama. Si se especifica sólo un nombre de clase, el objeto se dice que es anónimo; cada ícono de este tipo sin nombre de objeto denota un objeto anónimo distinto.

En cualquier caso, el nombre dado para la clase de un objeto debe ser el de la clase verdadera (o cualquiera de sus superclases) en el ámbito del diagrama utilizado para instanciar el objeto, incluso si tales clases fuesen abstractas. Estas reglas posibilitan escribir escenarios que hacen referencia a objetos sin conocer la subclase precisa en cuestión.

Para algunos objetos, es útil exponer algunos de sus atributos. Una vez más se dice «algunos» porque los íconos de objeto sólo representan una vista de la estructura del objeto. La sintaxis para los atributos es la misma descrita en la sección anterior para las clases y sus atributos, e incluye la capacidad de especificar una expresión por defecto para cada atributo. Los nombres de atributo deben hacer referencia a un atributo definido en la clase del objeto o cualquiera

¹⁶ Los íconos de objeto con el mismo nombre no calificado, pero de clases diferentes, pueden aparecer en el mismo diagrama, en tanto en cuanto esas clases estén relacionadas a través de alguna superclase antepasada común. Esto posibilita representar la propagación de operación de una subclase a una superclase, y viceversa.

mensajes**Figura 5.24.** Ícono de relación entre objetos.

de sus superclases. La sintaxis para los elementos puede adaptarse para utilizar la del lenguaje de implantación elegido.

Los diagramas de objetos también pueden incluir iconos que denotan utilidades de clase y metaclasses, ya que ambas entidades denotan cosas parecidas a objetos, sobre las que se puede operar y que operan sobre otros objetos.

Relaciones entre objetos. Como se explicó en el Capítulo 3, los objetos interactúan a través de sus enlaces con otros objetos, representados por el ícono de la Figura 5.24. Un enlace es una instancia de una asociación, al igual que un objeto es una instancia de una clase.

Puede existir un enlace entre dos objetos (incluyendo utilidades de clase y metaclasses) si y sólo si existe una asociación entre sus clases correspondientes. Esta asociación de clases puede manifestarse de cualquier modo, lo que significa que la relación de clase podría ser una asociación sin más, una relación de herencia o, por ejemplo, una relación «de posesión» (*tiene*). La existencia de una asociación entre dos clases denota por tanto una vía de comunicación (es decir, un enlace) entre instancias de las clases, por la que un objeto puede enviar mensajes a otro. Todas las clases tienen implícitamente una asociación consigo mismas, y por tanto es posible que un objeto se envíe un mensaje a sí mismo.

Dado un objeto *A* con un enlace *L* hacia un objeto *B*, *A* puede invocar cualquier operación aplicable a la clase de *B* que sea accesible para *A*; la inversa es cierta para las operaciones invocadas por *B* sobre *A*. Cualquier objeto que invoque la operación se conoce como el *cliente*; cualquier objeto que suministre la operación se conoce como el *proveedor* (servidor). En general, el emisor de un mensaje sabe quién es el receptor, pero el receptor no sabe necesariamente quién es el emisor.

En estado estable, debe existir consistencia entre la estructura de clases y la estructura de objetos de un sistema. Si se tiene una operación *M* a la que se invoca a través del enlace *L* sobre el objeto *B*, entonces la especificación de *B* (o la especificación de una superclase apropiada) debe contener la declaración de *M*.

Como se muestra en la Figura 5.24, se puede adornar un enlace con una serie de mensajes. Cada mensaje consta de los siguientes tres elementos:

- D Un símbolo de sincronización que denota la dirección de la invocación.
- M Una invocación de operación o despacho de evento.
- S Opcionalmente, un número de secuencia.

Se indica la dirección de un mensaje adornándolo con una línea dirigida, que apunta al objeto servidor. Este símbolo particular denota la forma más simple de paso de mensajes, cuya semántica se garantiza sólo en presencia de un

único hilo de control. Como se discute en una sección posterior, existen formas más avanzadas de sincronización que son apropiadas para múltiples hilos de control.

La invocación de una operación es el tipo de mensaje más común. Una invocación de operación sigue la sintaxis para operaciones tal como se definió anteriormente, excepto en que se puede incluir parámetros actuales que encajan con el prototipo de la operación:

- N() Solamente el nombre de la operación.
- R N(argumentos) Objeto de retorno, nombre y argumentos actuales de la operación.

El emparejamiento de los argumentos actuales con los formales se realiza en base a la posición. Si el objeto de retorno de la operación y los argumentos actuales usan nombres sin calificar que se corresponden con otros nombres sin calificar del diagrama de objetos, su propósito es denotar al mismo objeto, y así sus clases respectivas deben ser apropiadas a la signatura de la operación. De este modo, se pueden representar interacciones que involucran a objetos pasados por parámetro a ciertas operaciones o devueltos por ellas.

También es posible un mensaje que denote el despacho de un evento. Un despacho de evento sigue la sintaxis para los eventos tal como se la definió anteriormente, y así puede representar un nombre simbólico, un objeto o el nombre de alguna operación. En todos los casos, el nombre de evento debe estar definido para los diagramas de transición de estados apropiados a la clase del objeto servidor. La gestión de eventos como operaciones puede incluir parámetros actuales como se dice más arriba.

En ausencia de un número de secuencia explícito, pueden pasarse los mensajes en cualquier momento en relación a los otros mensajes representados en un diagrama de objetos particular. Para mostrar un orden explícito de los eventos, opcionalmente se puede poner como prefijo un número de secuencia (comenzando en el uno) en una invocación de operación o en un despacho de evento. Este número de secuencia se usa para indicar el orden relativo de los mensajes. Los mensajes con el mismo número de secuencia no están ordenados relativamente entre sí; los mensajes con números de secuencia menores se despiden antes que los mensajes con números de secuencia mayores. Los números de secuencia duplicados o inexistentes permiten una ordenación parcial de los mensajes.

Ejemplo. La Figura 5.25 muestra un ejemplo de un diagrama de objetos para el sistema de cultivo hidropónico, cuyo contexto es la categoría de clases Planificacion, descrita por primera vez en la Figura 5.7. Este diagrama pretende ilustrar un escenario que muestra una traza de la ejecución de una función común del sistema, a saber, la determinación de una predicción del coste neto de recolección para una cosecha específica.

La realización de esta función del sistema requiere la colaboración de varios objetos diferentes. Se ve en este diagrama que la acción del escenario comienza

único hilo de control. Como se discute en una sección posterior, existen formas más avanzadas de sincronización que son apropiadas para múltiples hilos de control.

La invocación de una operación es el tipo de mensaje más común. Una invocación de operación sigue la sintaxis para operaciones tal como se definió anteriormente, excepto en que se puede incluir parámetros actuales que encajan con el prototipo de la operación:

- $N()$ Solamente el nombre de la operación.
- $R\ N(argumentos)$ Objeto de retorno, nombre y argumentos actuales de la operación.

El emparejamiento de los argumentos actuales con los formales se realiza en base a la posición. Si el objeto de retorno de la operación y los argumentos actuales usan nombres sin calificar que se corresponden con otros nombres sin calificar del diagrama de objetos, su propósito es denotar al mismo objeto, y así sus clases respectivas deben ser apropiadas a la signatura de la operación. De este modo, se pueden representar interacciones que involucran a objetos pasados por parámetro a ciertas operaciones o devueltos por ellas.

También es posible un mensaje que denote el despacho de un evento. Un despacho de evento sigue la sintaxis para los eventos tal como se la definió anteriormente, y así puede representar un nombre simbólico, un objeto o el nombre de alguna operación. En todos los casos, el nombre de evento debe estar definido para los diagramas de transición de estados apropiados a la clase del objeto servidor. La gestión de eventos como operaciones puede incluir parámetros actuales como se dice más arriba.

En ausencia de un número de secuencia explícito, pueden pasarse los mensajes en cualquier momento en relación a los otros mensajes representados en un diagrama de objetos particular. Para mostrar un orden explícito de los eventos, opcionalmente se puede poner como prefijo un número de secuencia (comenzando en el uno) en una invocación de operación o en un despacho de evento. Este número de secuencia se usa para indicar el orden relativo de los mensajes. Los mensajes con el mismo número de secuencia no están ordenados relativamente entre sí; los mensajes con números de secuencia menores se despiden antes que los mensajes con números de secuencia mayores. Los números de secuencia duplicados o inexistentes permiten una ordenación parcial de los mensajes.

Ejemplo. La Figura 5.25 muestra un ejemplo de un diagrama de objetos para el sistema de cultivo hidropónico, cuyo contexto es la categoría de clases Planificacion, descrita por primera vez en la Figura 5.7. Este diagrama pretende ilustrar un escenario que muestra una traza de la ejecución de una función común del sistema, a saber, la determinación de una predicción del coste neto de recolección para una cosecha específica.

La realización de esta función del sistema requiere la colaboración de varios objetos diferentes. Se ve en este diagrama que la acción del escenario comienza

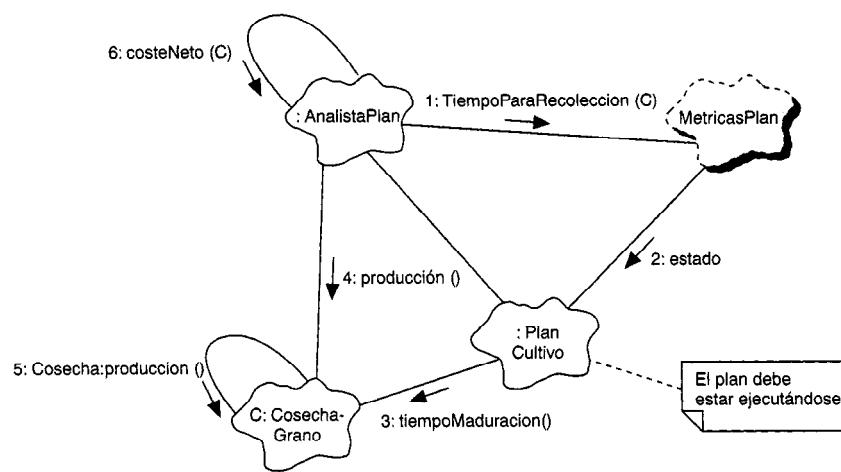


Figura 5.25. Diagrama de objetos del sistema de cultivo hidropónico.

con algún objeto `AnalistaPlan` que invoca la operación `tiempoParaRecolección()` sobre la utilidad de clase `MetricasPlan`. Nótese que el objeto `C` se pasa como argumento actual a esta operación. A continuación, la utilidad de clase `MetricasPlan` llama a `estado()` sobre cierto objeto `PlanCultivo` que no se nombra; el diagrama incluye una nota de desarrollo que indica que hay que comprobar que el plan dado está de hecho ejecutándose. El objeto `PlanCultivo` a su vez invoca la operación `tiempoMaduración()` sobre el objeto `CosechaGrano` seleccionado, preguntando por el momento en el que se espera que la cosecha esté madura. Después de que se complete esta operación selectora, el control vuelve al objeto `AnalistaPlan`, que entonces a `c.producción()` directamente, quien a su vez propaga esta operación a la superclase de la cosecha (la operación `Cosecha::producción()`). El control vuelve de nuevo al objeto `AnalistaPlan`, que completa el escenario invocando la operación `costeNeto()` sobre sí mismo.

Este diagrama indica un enlace entre los objetos `AnalistaPlan` y `PlanCultivo`. Aunque no hay paso de mensajes, la presencia de este enlace sirve para subrayar la existencia de una dependencia semántica entre los dos objetos.

Conceptos avanzados

Los elementos que se han presentado hasta ahora constituyen las partes esenciales de la notación para diagramas de objetos. Sin embargo, hay una serie de cuestiones de desarrollo particularmente enredadas que requieren que se extienda ligeramente esta notación básica. Como se advirtió en la discusión sobre los diagramas de clases, hay que hacer hincapié de nuevo en que estas características avanzadas deberían aplicarse sólo según fuese necesario para capturar la semántica que se pretende para el escenario.

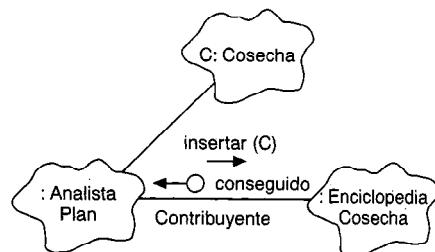


Figura 5.26. Papeles (roles).

Papeles, claves y restricciones. En una sección anterior, se hizo notar que las asociaciones de un diagrama de clases pueden adornarse con un papel que denote el propósito o carácter de la relación que asocia una clase con otra. Para ciertos diagramas de objetos, es útil declarar este papel en el enlace correspondiente entre dos objetos. Con frecuencia, este adorno ayuda a explicar por qué un objeto opera sobre otro.

La Figura 5.26 ofrece un ejemplo de esta característica avanzada. Se ve que algún objeto **AnalistaPlan** inserta una cosecha específica en un objeto anónimo **EnciclopediaCosecha**, y lo hace actuando en el papel de **Contribuyente**.

Usando la misma notación que se introdujo en los diagramas de clases, se puede indicar las claves o restricciones asociadas con un objeto o un enlace.

Flujo de datos. Como se describió en el Capítulo 3, los datos pueden fluir en la misma dirección que un mensaje o en dirección contraria. Ocasionalmente, el mostrar explícitamente la dirección del flujo de datos ayuda a explicar la semántica de un escenario particular. Copiando la notación del diseño estructurado, se usa el ícono que aparece en la Figura 5.26 para mostrar que se retorna el valor **conseguido** cuando se completa el mensaje **insertar**.

Se puede usar tanto un objeto como un valor en un flujo de datos.

Visibilidad. En ciertos escenarios complicados es útil reflejar la forma exacta en que un objeto tiene visibilidad de otro. Aunque las asociaciones de los diagramas de clases denotan las dependencias semánticas que pueden existir entre las clases de dos objetos, no indican exactamente cómo pueden ver esas instancias a las otras. Por esta razón, se pueden señalar los enlaces del diagrama de objetos con iconos que representan la visibilidad de un objeto hacia otro. Este adorno también es importante para herramientas que soportan generación precoz de código e ingeniería inversa.

La Figura 5.27 es un refinamiento de la Figura 5.25, e incluye algunos de estos adornos, que son similares a los iconos que se utilizaron para representar contención física en los diagramas de clases, pero con el añadido de una letra que designa el tipo de visibilidad. Por ejemplo, la marca **s** que se ve en el enlace

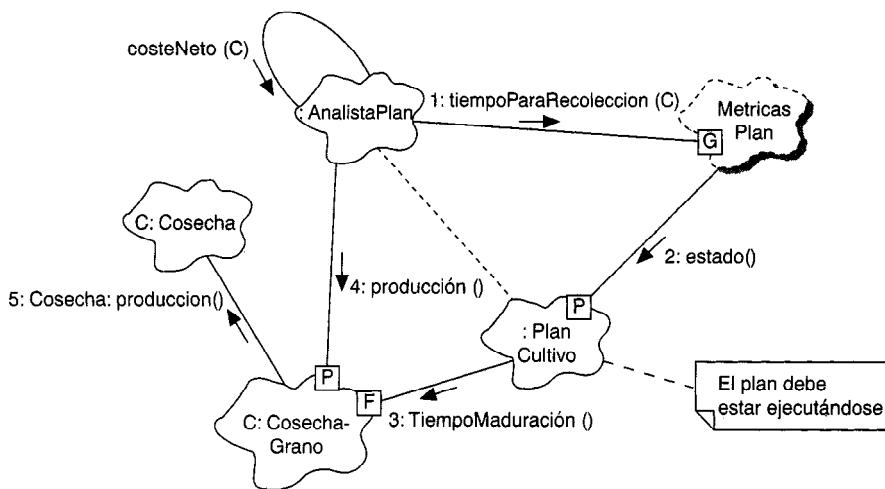


Figura 5.27. Visibilidad.

del objeto **AnalistaPlan** a la utilidad de clase **MetricasPlan** denota que la utilidad de clase es global a la declaración del objeto analista. El objeto **c** es visible al objeto **AnalistaPlan** y al objeto **PlanCultivo** por dos vías diferentes. Desde la perspectiva del objeto **AnalistaPlan**, el objeto **c** de tipo **CosechaGrano** es visible como parámetro de alguna operación del analista (el adorno **P**); desde la perspectiva del objeto **PlanCultivo**, el objeto **c** de la clase **CosechaGrano** es visible como un campo (es decir, una parte del objeto agregado que es el plan).

Generalizando, pueden usarse las siguientes marcas para indicar visibilidad:

- **G** El objeto proveedor es global al cliente.
- **P** El objeto proveedor es parámetro de alguna operación del cliente.
- **F** El objeto proveedor es parte del cliente objeto (F de *field*, es decir, campo).
- **L** El objeto proveedor es un objeto declarado localmente en el ámbito del diagrama de objetos.

De forma consistente con los adornos para contención física en los diagramas de clases, estos adornos pueden escribirse como una caja hueca con una letra (lo que representa que la identidad del objeto es compartida) o como una caja rellena con una letra (lo que representa que la identidad del objeto no es compartida estructuralmente).

La ausencia de un adorno de visibilidad significa que la visibilidad precisa entre los dos objetos se deja sin especificar. En la práctica, es común adornar con estos símbolos de visibilidad sólo unos pocos enlaces clave en un diagrama de objetos. El uso más habitual de estos símbolos es representar relaciones de todo/parte (agregación) entre dos objetos; el segundo uso más corriente es representar objetos transitorios que se pasan como parámetros al escenario del diagrama de objetos.

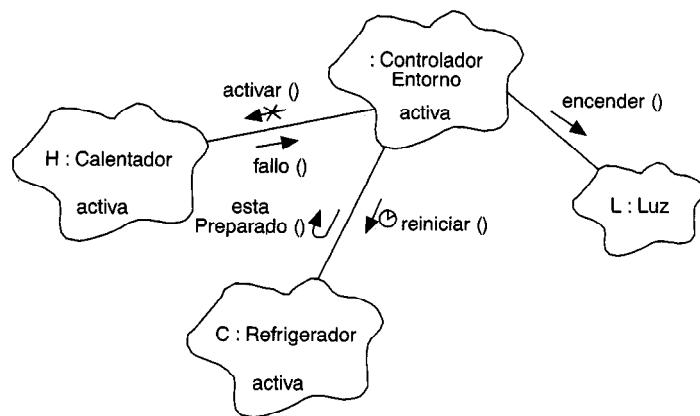


Figura 5.28. Objetos activos y sincronización.

Objetos activos y sincronización. Como se hizo notar en el Capítulo 3, algunos objetos pueden ser activos, lo que quiere decir que incorporan su propio hilo de control. Otros objetos pueden tener una semántica puramente secuencial, mientras que otros más podrían no ser activos, aunque garantizando su semántica en presencia de múltiples hilos de control.

En todas esas circunstancias, hay que afrontar dos cuestiones: cómo poner de manifiesto los objetos activos que denotan raíces de control en un escenario, y cómo representar diferentes formas de sincronización entre tales objetos.

En la discusión anterior sobre las características avanzadas de las especificaciones de clases, se hizo notar que las clases pueden presentar uno de cuatro posibles estilos de concurrencia: secuencial, vigilada, síncrona y activa. Por implicación, todas las instancias de una clase toman la misma semántica concurrente que su clase; todos los objetos son secuenciales a menos que se diga lo contrario. Se puede revelar explícitamente el estilo de concurrencia de un objeto en un diagrama de objetos adornando su ícono de objeto con los nombres secuencial, vigilada, síncrona o activa, colocados en la esquina inferior izquierda del ícono. Por ejemplo, en la Figura 5.28 se ve que H, C y la instancia anónima de la clase ControladorEntorno son todos objetos activos e incorporan por tanto su propio hilo de control. Los objetos sin este tipo de marca (como L) se asume que son secuenciales.

El símbolo de sincronización de mensajes que se introdujo anteriormente (la línea simple dirigida) representa simple paso de mensajes secuencial. En presencia de múltiples hilos de control, sin embargo, hay que especificar otras formas de sincronización.

Aun siendo de concepción sencilla, el ejemplo de la Figura 5.28 ilustra los diferentes tipos de sincronización de mensajes que pueden aparecer en un diagrama de objetos. El mensaje `encender()` es un ejemplo de paso de mensajes simple, y se representa con la línea dirigida. La semántica del paso de mensajes

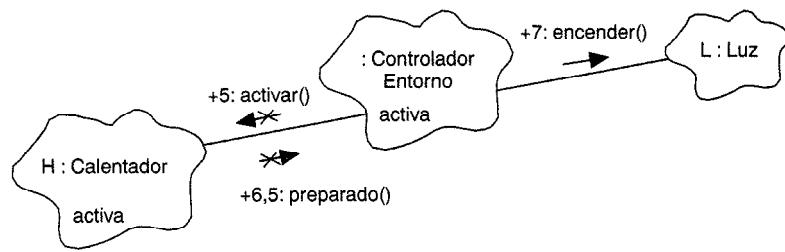


Figura 5.29. Presupuestos de tiempo.

simple está garantizada sólo en presencia de un hilo único de control. En contraste, todos los otros mensajes conllevan alguna forma de sincronización de procesos; todas esas formas avanzadas de sincronización se aplican solamente a los servidores que no son secuenciales.

Por ejemplo, el mensaje `activar()` es síncrono, lo que significa que el cliente esperará por siempre hasta que el servidor acepte el mensaje. El paso síncrono de mensajes es equivalente al mecanismo de cita entre tareas de Ada. El mensaje `estaPreparado()` denota paso de mensajes con abandono inmediato, lo que quiere decir que el cliente abandonará el mensaje si el servidor no lo atiende inmediatamente. El mensaje `reiniciar()` denota una sincronización de intervalo (de espera) o *timeout*: el cliente abandonará el mensaje si el servidor no puede atenderlo dentro de un espacio de tiempo determinado.

En cada uno de estos últimos tres casos, el cliente debe esperar a que el servidor procese por completo el mensaje (o abandonar el mensaje) antes de recuperar el control. En el caso del mensaje `fallo()`, la semántica es distinta. Éste es un ejemplo de mensaje asíncrono, lo que significa que el cliente envía el evento al servidor para que lo procese, el servidor pone el mensaje en una cola, y el cliente continúa su actividad sin esperar al servidor. El paso asíncrono de mensajes es similar al manejo de interrupciones.

Presupuestos de tiempo. Para ciertas aplicaciones críticas respecto al tiempo, es importante trazar escenarios en términos de tiempo exacto relativo al comienzo del escenario. Para designar tiempo relativo, se usan números de secuencia que denotan tiempo (en segundos), prefijados con el símbolo de sumar. Por ejemplo, en la Figura 5.29 se ve que el mensaje `activar()` es invocado por primera vez 5 segundos después del comienzo del escenario, seguido por el mensaje `preparado()` 6,5 segundos después del comienzo del escenario, y seguido entonces por el mensaje `encender()` después de 7 segundos.

Especificaciones

Al igual que para los diagramas de clases, cada entidad en un diagrama de objetos puede tener su especificación, la cual proporciona su definición completa.

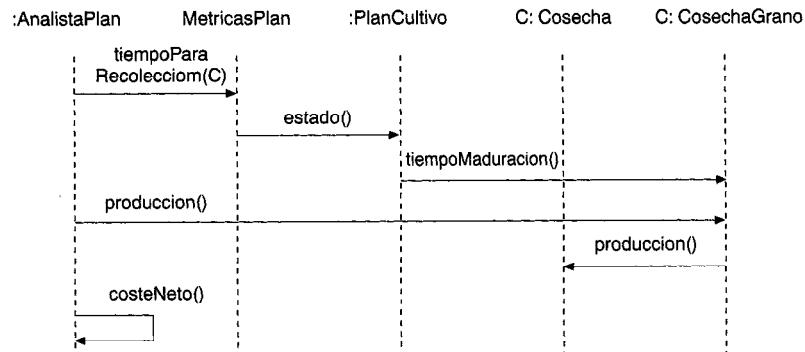


Figura 5.30. Diagrama de interacción del sistema de cultivo hidropónico.

Ya que las especificaciones para objetos y relaciones entre estos no añaden información sobre lo que ya se ha descrito en esta sección, no es necesario discutir aquí su especificación textual.

Por otra parte, las especificaciones para los diagramas de objetos en su conjunto sí que incorporan una parte significativa de información no gráfica que debe considerarse. Como se describió al comienzo de esta sección, todo diagrama de objetos debe designar un contexto. Se hace esto en la especificación del diagrama, como sigue:

Contexto: global | categoría | clase | operación

En particular, el ámbito de un diagrama de objetos puede ser *global*, o estar en el contexto de una *categoría* de clases determinada, en el de una *clase* o en el de una *operación* (incluyendo los métodos y los subprogramas libres).

5.5. Diagramas de interacción

Elementos esenciales: objetos e interacciones

Un *diagrama de interacciones* se usa para realizar una traza de la ejecución de un escenario en el mismo contexto que un diagrama de objetos¹⁷. Realmente, en gran parte un diagrama de interacción es simplemente otra forma de representar un diagrama de objetos. Por ejemplo, en la Figura 5.30, se proporciona un diagrama de interacción que duplica la mayor parte de la semántica del diagrama de objetos mostrado en la Figura 5.25. La ventaja de usar un diagrama de interacción es que resulta más fácil leer el paso de mensajes en orden rela-

¹⁷ Estos diagramas son generalizaciones de los diagramas de traza de eventos de Rumbaugh [14] y de los diagramas de interacción de Jacobson [15].

tivo. La ventaja de usar un diagrama de objetos reside en que soporta bien el aumento de escala para muchos objetos con llamadas complejas, y permite la inclusión de otra información, como enlaces, valores de atributos, papeles, flujo de datos y visibilidad. Puesto que cada diagrama aporta beneficios a tener en cuenta, se incluyen ambos en el método¹⁸.

Los diagramas de interacción no introducen conceptos o iconos nuevos; antes bien, toman la mayoría de los elementos esenciales de los diagramas de objetos y los reestructuran. Como indica la Figura 5.30, un diagrama de interacción aparece en forma tabular. Las entidades de interés (que son las mismas que para los diagramas de objetos) se escriben horizontalmente a lo largo de la franja superior del diagrama. Se dibuja una línea vertical discontinua bajo cada objeto. Los mensajes (que pueden denotar eventos o la invocación de operaciones) se muestran horizontalmente con la misma sintaxis y símbolos de sincronización que para los diagramas de objetos. Los extremos de los iconos de mensaje se conectan con las líneas verticales que se conectan a su vez con las entidades en la parte superior del diagrama, y se dibujan en sentido del cliente al servidor. El orden se indica mediante la posición en la vertical, siendo el primer mensaje el de la parte superior y el último el de la parte inferior. Como consecuencia, no son necesarios los números de secuencia.

Los diagramas de interacción son frecuentemente mejores que los diagramas de objetos para capturar la semántica de los escenarios en un momento temprano del ciclo de vida del desarrollo, antes de que se hayan identificado los protocolos de las clases individuales. Como se explica en el próximo capítulo, los diagramas de interacción iniciales tienden a centrarse en los eventos en oposición a las operaciones, porque los eventos ayudan a definir los límites de un sistema que se está desarrollando. A medida que avanza el desarrollo y se refina la estructura de clases del sistema, tiende a desplazarse el énfasis hacia los diagramas de objetos, cuya semántica es más expresiva.

Conceptos avanzados

Los diagramas de interacción son conceptualmente muy simples; sin embargo, existen dos elementos evidentes que pueden añadirse para hacerlos más expresivos en presencia de ciertos patrones de interacción complicados.

Guiones. Para escenarios complejos que involucran condiciones o iteraciones, los diagramas de interacción pueden mejorarse mediante el uso de guiones. Como se ve en el ejemplo de la Figura 5.31, un guión puede escribirse a la izquierda de un diagrama de interacción, alineándose los pasos del guión con las invocaciones de los mensajes. Los guiones pueden redactarse con una forma li-

¹⁸ Los diagramas de objetos y los diagramas de interacción están lo suficientemente cerca entre sí en términos semánticos como para que sea posible para las herramientas la generación de un diagrama a partir del otro, con pérdidas mínimas de información.

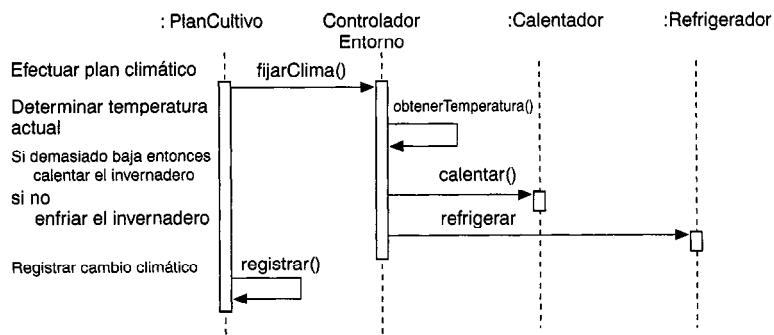


Figura 5.31. Guiones y centros de control.

bre de texto español¹⁹, o bien usando la sintaxis del lenguaje de implantación elegido.

Centro de control. Ni los diagramas de objetos ni los de interacción indican por sí solos el centro de control a medida que se pasan los mensajes. Por ejemplo, si el objeto A envía los mensajes x e y a otros objetos, no queda claro si x e y son mensajes independientes de A o si se los ha invocado como parte del mismo mensaje circundante z. Como se muestra en la Figura 5.31, se pueden adornar las líneas verticales que descienden de cada objeto de un diagrama de interacción con un rectángulo que representa el tiempo relativo durante el cual el flujo de control está centrado en ese objeto. Por ejemplo, se ve que la instancia anónima de `PlanCultivo` es el centro primario de control, y su comportamiento al llevar a efecto un plan climático incluye la invocación de otros métodos, que a su vez llaman a otros métodos que eventualmente devuelven el control al objeto `PlanCultivo`.

5.6. Diagramas de módulos

Elementos esenciales: los módulos y sus dependencias

Se utiliza un diagrama de módulos para mostrar la asignación de clases y objetos a módulos en el diseño físico de un sistema. Un solo diagrama de módulos representa una vista de la estructura de módulos de un sistema. Durante el desarrollo, se usan diagramas de módulos para indicar la disposición en capas y la partición física de la arquitectura.

Algunos lenguajes, entre los que destaca Smalltalk, carecen del concepto de

¹⁹ Texto inglés, en el original. (*N. del T.*)

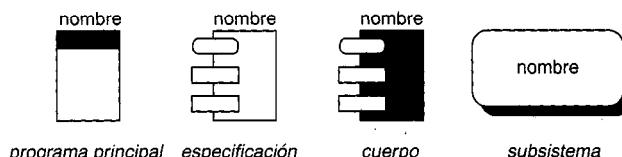


Figura 5.32. Iconos de módulo y subsistema.

arquitectura física formada por módulos; en estos casos, los diagramas de módulos no son necesarios.

Los dos elementos esenciales de un diagrama de módulos son los módulos y sus dependencias.

Módulos. La Figura 5.32 muestra los iconos que se usan para representar varios tipos de módulos. Los primeros tres iconos denotan archivos, distinguidos por su función. El ícono de programa principal denota un archivo que contiene la raíz del programa. En C++ por ejemplo, sería probablemente algún archivo .cpp que contiene la definición de la privilegiada función no miembro llamada `main`. Lo normal es que haya exactamente uno de tales módulos por programa. El ícono de especificación y el ícono de cuerpo denotan archivos que contienen, respectivamente, la declaración y la definición de las entidades. En C++, por ejemplo, los módulos de especificación denotan archivos .h, y los módulos de cuerpo denotan archivos .cpp.

Se explicará el significado del ícono de subsistema en una sección posterior.

Se requiere un nombre para cada módulo; este nombre suele denotar simplemente el nombre del archivo físico correspondiente en el directorio de desarrollo. Se escriben usualmente tales nombres sin sus sufijos, que serían redundantes cuando se los asociase con un ícono de módulo particular. Si el nombre es particularmente largo, este puede abreviarse o bien agrandarse el ícono. Todo nombre completo de archivo debe ser único en relación con el subsistema que lo contiene. Dependiendo de las necesidades de los entornos de desarrollo particulares, es posible que se impongan otras restricciones sobre los nombres, como la necesidad de prefijos distintivos o la necesidad de nombres únicos para el sistema completo.

Cada módulo engloba la declaración o definición de clases, objetos y otros detalles del lenguaje. Conceptualmente, se puede adoptar una visión más cercana de un módulo para ver el contenido físico del archivo que le corresponde.

Dependencias. La única relación que puede darse entre dos módulos es una dependencia de compilación, representada por una línea dirigida que apunta al módulo respecto al cual existe la dependencia. En C++ por ejemplo, se indica una dependencia de compilación mediante directivas `#include`. Análogamente en Ada, las dependencias de compilación se indican con cláusulas `with`. En general, no puede haber ciclos en un conjunto de dependencias de compilación. La realización de una ordenación topológica entre todas las dependen-

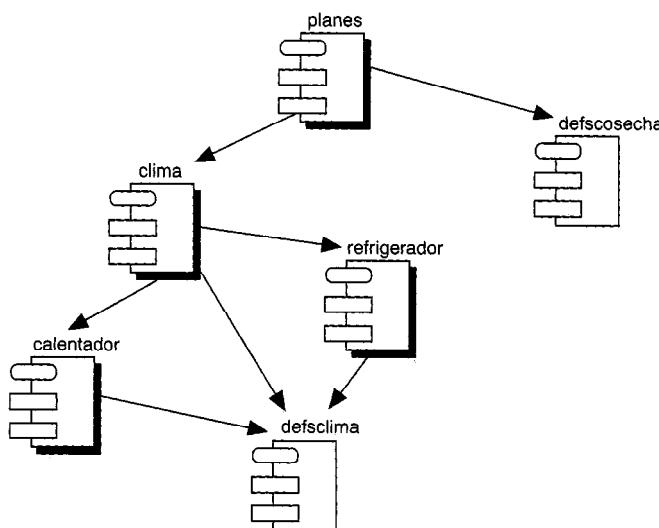


Figura 5.33. Diagrama de módulos del sistema de cultivo hidropónico.

cias de los módulos de la estructura de módulos de un sistema es suficiente para calcular un orden parcial de compilación.

Ejemplo. En la Figura 5.33 se proporciona un ejemplo de esta notación, obtenido de la arquitectura física del sistema de cultivo hidropónico. Se ven seis módulos. Dos de ellos, `defscosecha` y `defsclima`, son solamente especificaciones, y sirven para suministrar tipos y constantes comunes. Los otros cuatro módulos se muestran con sus especificaciones y cuerpos agrupados juntos: es un estilo típico en el dibujo de diagramas de módulos, al estar tan íntimamente relacionados la especificación y cuerpo de un módulo. Puesto que se han superpuesto ambas partes, la dependencia del cuerpo respecto a la especificación correspondiente está oculta, aunque de hecho existe. De este modo, está oculto el nombre del cuerpo, lo cual no es problema porque la convención adoptada es nombrar las especificaciones y los cuerpos con el mismo nombre excepto por un sufijo distintivo (como `.h` y `.cpp`, respectivamente).

Las dependencias de este diagrama sugieren un orden parcial de compilación. Por ejemplo, el cuerpo de `clima` depende de la especificación de `calentador`, que a su vez depende de la especificación de `defsclima`.

Elementos esenciales: subsistemas

Como se explicó en el Capítulo 2, un sistema grande puede descomponerse en muchos cientos, si no unos pocos miles, de módulos. Es imposible intentar comprender la arquitectura física de un sistema semejante sin trocearlo más. En

la práctica, los desarrolladores tienden a usar convenciones informales para recoger módulos relacionados en estructuras de directorios. Por razones similares, se introduce la noción de subsistema para los diagramas de módulos, que paralleiza el papel jugado por las categorías de clases en los diagramas de clases. En concreto, los subsistemas representan agrupaciones de módulos relacionados lógicamente.

Subsistemas. Los subsistemas sirven para particionar el modelo físico de un sistema. Un subsistema es un agregado que contiene otros módulos y otros subsistemas. Cada módulo del sistema debe habitar en un solo subsistema o en el nivel superior del sistema.

La Figura 5.32 muestra el ícono que se usa para representar un subsistema. Al igual que para un módulo, se requiere un nombre para cada subsistema. Las reglas de nomenclatura de los sistemas son iguales que para nombrar módulos individuales, aunque los nombres completos de subsistema no suelen incluir sufijos distintivos.

Algunos de los módulos englobados por un subsistema pueden ser públicos, lo que quiere decir que son exportados por el subsistema y por tanto utilizables fuera del mismo. Otros módulos pueden ser parte de la implantación del subsistema, lo que significa que no se consideran utilizables por ningún otro módulo de fuera del subsistema. Por convención, todo módulo de un subsistema se considera público, a menos que se defina explícitamente de otra forma. La restricción de acceso a módulos de implantación se consigue usando los mismos conceptos avanzados que para la restricción de acceso a las categorías de clases.

Un subsistema puede tener dependencias respecto a otros subsistemas o módulos, y un módulo puede tener dependencias respecto a un subsistema. Por consistencia, se aplica el mismo ícono de dependencia descrito anteriormente.

En la práctica, un sistema grande tiene un diagrama de módulos del nivel superior, que consta de los subsistemas al nivel más alto de abstracción. Mediante este diagrama un desarrollador llega a comprender la arquitectura física general de un sistema.

Ejemplo. La Figura 5.34 muestra un ejemplo de un diagrama de módulos del nivel superior para el sistema de cultivo hidropónico. Si se adopta una visión ampliada de cualquiera de los siete subsistemas que aparecen, se encontrarán todos sus módulos correspondientes.

Nótese cómo esta arquitectura física se corresponde con la arquitectura lógica del sistema de cultivo hidropónico mostrada en la Figura 5.7. Estas estructuras son isomorfas en gran medida, aunque hay pequeñas diferencias. En particular, se ha tomado la decisión de separar las clases de dispositivos de bajo nivel de las categorías de clases Clima y Nutrientes, y colocar sus módulos correspondientes en un subsistema llamado Dispositivos. También se ha dividido la categoría de clases Invernadero en dos subsistemas llamados ControlClima y Alimentador.

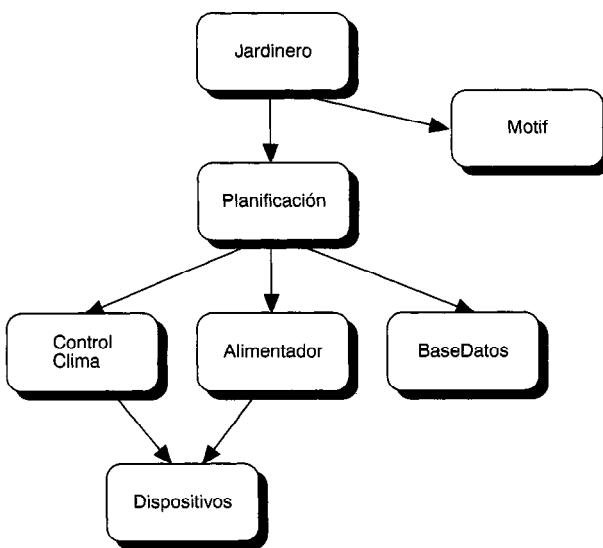


Figura 5.34. Diagrama de módulos de nivel superior para el sistema de cultivo hidropónico.

Conceptos avanzados

Adaptación a los lenguajes. Ciertos lenguajes, entre los que destaca Ada, definen otros tipos de módulos aparte de los normales que aparecen en la Figura 5.32. En particular, Ada define paquetes genéricos, subprogramas genéricos y tareas como unidades de compilación separadas. Es por tanto razonable aumentar los iconos esenciales de los diagramas de módulos para incluir iconos que representen tipos de módulo dependientes del lenguaje.

Segmentación. Especialmente para plataformas que tengan modelos de memoria severamente restringidos, es importante la decisión de generar código en segmentos diferentes, o incluso de producir un esquema de overlays*. Los diagramas de módulos pueden extenderse para ayudar a visualizar esta segmentación incluyendo adornos específicos del lenguaje que denoten el segmento de datos o código que corresponde a cada módulo del diagrama de módulos.

Especificaciones

Al igual que en los diagramas de clases y objetos, cada entidad de un diagrama de módulos puede tener una especificación, que proporciona su definición

* Literalmente, solapamientos, pero es de uso común el término en lengua inglesa. (*N. del T.*)

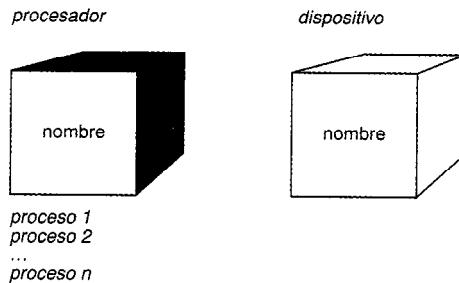


Figura 5.35. Iconos de procesador y dispositivo.

completa. Ya que las especificaciones para módulos y sus dependencias no añaden información a lo que ya se ha descrito en esta sección, no es necesario discutir aquí su especificación textual.

Si hay algún grado de integración entre las herramientas que soporten esta notación y las herramientas para entornos de programación, es razonable usar los diagramas de módulos como un medio para visualizar los módulos manejados por el entorno de programación. La ampliación de la visión de un módulo específico o de un subsistema en un diagrama de módulos es equivalente por tanto a un desplazamiento hacia su fichero o directorio físico correspondiente, y viceversa.

5.7. Diagramas de procesos

Elementos esenciales: procesadores, dispositivos y conexiones

Se usa un diagrama de procesos para mostrar la asignación de procesos a procesadores en el diseño físico de un sistema. Un solo diagrama de procesos representa una vista de la estructura de procesos de un sistema. Durante el desarrollo se usan diagramas de procesos para indicar la colección física de procesadores y dispositivos que sirven como plataforma de ejecución del sistema.

Los tres elementos esenciales de un diagrama de procesos son los procesadores, los dispositivos y sus conexiones.

Procesadores. La Figura 5.35 muestra el ícono que se utiliza para representar un procesador. Un procesador es un fragmento de hardware capaz de ejecutar programas. Se requiere un nombre para cada procesador, y no hay restricciones particulares en cuanto a esos nombres porque denotan entidades del hardware, y no del software.

Se puede añadir al ícono de procesador una lista de procesos. Cada proceso

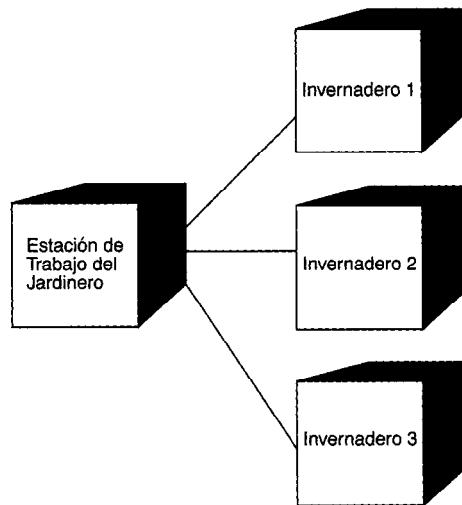


Figura 5.36. Diagrama de procesos del sistema de cultivo hidropónico.

de esta lista denota la raíz de un programa principal (de un diagrama de módulos) o el nombre de un objeto activo (de un diagrama de objetos).

Dispositivos. La Figura 5.35 muestra el icono que se utiliza para representar un dispositivo. Un dispositivo es un fragmento de hardware incapaz de ejecutar programas (como mínimo hasta el punto que interesa a nuestro modelo lógico). Al igual que para los procesadores, se requiere un nombre para cada dispositivo. No hay restricciones particulares sobre nombres de dispositivo, y de hecho esos nombres pueden ser bastante genéricos, como `modem` o `terminal`.

Conexiones. Los procesadores y dispositivos deben comunicarse con otros elementos semejantes. Mediante una línea no dirigida, se puede indicar la conexión entre un dispositivo y un procesador, un procesador y otro procesador, o un dispositivo y otro dispositivo. Una conexión suele representar algún acoplamiento directo del hardware, como un cable RS232, una conexión Ethernet o quizás incluso una vía hacia memoria compartida. Una conexión también puede representar acoplamientos menos directos, como comunicaciones entre un satélite y la tierra. Las conexiones se consideran habitualmente bidireccionales, aunque si una conexión concreta es unidireccional puede añadirse una punta de flecha para mostrar la dirección. Cada conexión puede incluir una etiqueta opcional que le da nombre.

Ejemplo. En la Figura 5.36, se ofrece un ejemplo de esta notación, obtenido a partir de la arquitectura física del sistema de cultivo hidropónico. Aquí se aprecia que los arquitectos del sistema han decidido descomponer el mismo

en una red de tres computadores, uno de ellos asignado como estación de trabajo del jardinero, y los demás asignados a invernaderos individuales. Los procesos que se ejecutan en los computadores del invernadero no pueden comunicarse directamente entre sí, aunque sí pueden comunicarse con procesos que se ejecutan en la estación de trabajo del jardinero. Por razones de simplicidad, se ha decidido no mostrar ningún dispositivo en este diagrama, aunque se espera que haya unos pocos actuadores y sensores en el sistema.

Conceptos avanzados

Adaptación. La Figura 5.35 muestra los iconos estándar que se utilizan para representar procesadores y dispositivos, pero es razonable (y de hecho es deseable) permitir representaciones alternativas. Por ejemplo, podrían definirse iconos específicos para representar gráficamente un microcomputador empotrado (un procesador), un disco, un terminal y un conversor analógico/digital (todos ellos dispositivos), y utilizar entonces en un diagrama de procesos estos iconos en vez de los iconos estándar. Haciendo esto, se ofrece una visualización de la plataforma física de la implantación que resulta directamente comprensible para los arquitectos de hardware y de sistemas, así como para los usuarios finales del sistema, que probablemente no serán expertos en desarrollo de software.

Anidamiento. La configuración hardware de un sistema es a veces muy complicada, y puede involucrar jerarquías complejas de procesadores y dispositivos. En algunas circunstancias, por tanto, es útil el disponer de la capacidad para representar grupos de procesadores, dispositivos y conexiones, al igual que las categorías de clases representan agrupaciones lógicas de clases y objetos. Se pueden indicar tales grupos de hardware con un ícono marcado con un nombre, con forma de rectángulo de líneas discontinuas redondeado y con sombra. Cada uno de estos íconos denota un grupo distinto de procesadores, dispositivos y conexiones, y de este modo la ampliación de un grupo revela estas entidades anidadas. Se pueden definir conexiones entre grupos, así como entre procesadores, dispositivos y grupos.

Planificación de procesos. Es necesario adoptar alguna política para planificar la ejecución de los procesos en un procesador. Básicamente existen cinco enfoques generales para la planificación, y se puede documentar cuál de ellos se utiliza mediante el adorno de cada ícono de procesador con uno de los nombres:

- Desplazante Los procesos de prioridad mayor que están listos para ejecutarse pueden desplazar a otros de menor prioridad que están en ejecución actualmente; típicamente, los procesos de igual prioridad reciben una fracción de tiempo para ejecutarse, así que los recursos computacionales están distribuidos equitativamente.
- No desplazante El proceso actual continúa ejecutándose hasta que cede el control.
- Cíclico El control pasa de un proceso a otro, y cada proceso recibe una cantidad fija de tiempo de proceso, que suele llamarse un *cuadro**; se puede asignar tiempo a un proceso por cuadros o por subcuadros.
- Ejecutivo Hay algún algoritmo que controla la planificación de procesos.
- Manual Los procesos son planificados por un usuario externo al sistema.

Para explicar mejor la planificación que utiliza un procesador específico, a veces es útil incluir un diagrama de objetos o un diagrama de interacción, especialmente si se utiliza planificación ejecutiva.

Especificaciones

Al igual que con todos los demás diagramas, cada procesador, dispositivo o conexión puede tener una especificación, que proporciona su definición completa. Ya que las especificaciones para esas entidades no añaden información a lo que ya se ha descrito anteriormente, no es necesario discutir aquí su especificación textual.

5.8. Aplicación de la notación

Los productos del diseño orientado a objetos

Típicamente, el análisis de un sistema incluirá conjuntos de diagramas de objetos (para expresar el comportamiento del sistema mediante escenarios), diagramas de clases (para expresar los papeles y responsabilidades de los agentes que proporcionan el comportamiento del sistema) y diagramas de transición de estados (para mostrar el comportamiento de esos agentes respecto al orden de los eventos). Análogamente, el diseño de un sistema, incluyendo su arquitectura

* *Frame* en la versión original en inglés. (*N. del T.*)

e implantación, incluirá conjuntos de diagramas de clases, diagramas de objetos, diagramas de módulos y diagramas de procesos, así como sus correspondientes vistas dinámicas.

Entre estos diagramas hay una conectividad entre extremos, que permite seguir la pista a los requerimientos desde la implantación hacia atrás, hasta la especificación. Comenzando por un diagrama de procesos, un procesador puede designar un programa principal, que estará definido en algún diagrama de módulos. Este diagrama de módulos puede contener la definición de una colección de clases y objetos, cuyas definiciones se encontrarán en los diagramas de clases o de objetos apropiados. Por último, las definiciones de clases individuales apuntan hacia los requisitos, porque estas clases, en general, reflejan directamente el vocabulario del espacio del problema.

La notación descrita en este capítulo puede utilizarse manualmente, aunque para aplicaciones más grandes pide a gritos una herramienta que la soporte de forma automática. Las herramientas pueden proporcionar comprobaciones de consistencia, comprobaciones de restricciones, comprobaciones de completud y análisis, y pueden ayudar a un desarrollador a hojear a través de los productos del análisis y diseño sin restricción alguna. Por ejemplo, mientras examina un diagrama de módulos, un desarrollador podría querer estudiar un mecanismo particular; puede utilizar una herramienta para localizar todas las clases asignadas a un módulo particular. Mientras examina un diagrama de objetos que describe un escenario que utiliza una de esas clases, el desarrollador podría querer ver su lugar en la trama de herencias. Por último, si ese escenario implicase a un objeto activo, el desarrollador podría usar una herramienta para encontrar el procesador al que se ha asignado ese hilo de control. El uso de herramientas de este modo libera a los desarrolladores del tedio de mantener consistentes todos los detalles del análisis y el diseño, permitiéndoles centrarse en los aspectos creativos del proceso de desarrollo.

Escalado ascendente y descendente

Se ha encontrado que esta notación con sus variantes es aplicable tanto a sistemas pequeños que sólo constan de aproximadamente una docena de clases, como a otros que se componen de varios miles de clases. Como se verá en los siguientes dos capítulos, esta notación es especialmente aplicable a un enfoque incremental e iterativo del desarrollo. Uno no crea un diagrama y luego se aleja de él, tratándolo como un artefacto sagrado e inmutable. Antes bien, estos diagramas evolucionan durante el proceso de diseño a medida que se toman nuevas decisiones de diseño y se establece un detalle mayor.

También se ha hallado que esta notación es altamente independiente del lenguaje. Es aplicable a cualquiera de un amplio espectro de lenguajes de programación orientados a objetos.

Este capítulo ha descrito los productos esenciales del desarrollo orientado a objetos, incluyendo su sintaxis y semántica. Los dos capítulos siguientes descri-

birán el proceso que conduce a esos productos. Los cinco capítulos restantes demuestran la aplicación práctica de esta notación y ese proceso a una amplia gama de problemas.

Resumen

- El diseño no es el acto de dibujar un diagrama; un diagrama se limita a plasmar un diseño.
- En el diseño de un sistema complejo es importante ver el diseño desde muchas perspectivas: a saber, su estructura física y lógica y su semántica estática y dinámica.
- La notación para el desarrollo orientado a objetos incluye cuatro diagramas básicos (diagramas de clases, diagramas de objetos, diagramas de módulos y diagramas de procesos) y dos diagramas suplementarios (diagramas de transición de estados y diagramas de interacción).
- Se usa un diagrama de clases para mostrar la existencia de clases y sus relaciones en el diseño lógico de un sistema. Un diagrama de clases simple representa una vista de la estructura de clases de un sistema.
- Se usa un diagrama de objetos para mostrar la existencia de objetos y sus relaciones en el diseño lógico de un sistema. Un diagrama de objetos simple suele usarse para representar un escenario.
- Se usa un diagrama de módulos para mostrar la asignación de clases y objetos a módulos en el diseño físico de un sistema. Un diagrama de módulos simple representa una vista de la arquitectura de módulos de un sistema.
- Se usa un diagrama de procesos para mostrar la asignación de procesos a procesadores en el diseño físico de un sistema. Un diagrama de procesos simple representa una vista de la arquitectura de procesos de un sistema.
- Se usa un diagrama de transición de estados para mostrar el espacio de estados de una instancia de una clase dada, los eventos que causan una transición de un estado a otro y las acciones que se derivan de un cambio de estado.
- Se usa un diagrama de interacción para seguir la pista a la ejecución de un escenario en el mismo contexto que un diagrama de objetos.

Lecturas recomendadas

Desde la publicación de la primera edición de este libro, he intentado de forma unilateral incorporar los mejores elementos notacionales de muchos otros diseñadores de metodologías, especialmente Rumbaugh y Jacobson, al método Booch, y he eliminado

o simplificado elementos de la notación original de Booch que demostraron ser tópicos, inconsistentes o de utilidad marginal, mientras me esforzaba por mantener una integridad conceptual en la notación. Este capítulo es la culminación de este esfuerzo de unificación.

Se ha escrito muchísimo sobre notaciones para análisis y diseño de software; el libro de Martin y McClure [H 1988] es una referencia general para muchos de los enfoques más tradicionales. Graham [F 1991] examina varias notaciones específicas de los métodos orientados a objetos.

Una primera forma de la notación descrita en este capítulo fue documentada primero por Booch [F 1981]. Esta notación evolucionó después para incorporar la potencia expresiva de las redes semánticas (Stillings et al. [A 1987] y Barr y Feigenbaum [J 1981]), los diagramas entidad—relación (Chen [E 1976]), modelos de entidad (Ross [F 1987]), redes de Petri (Peterson [J 1977], Sahraoui [F 1987] y Bruon y Balsamo [F 1986]), asociaciones (Rumbaugh [F 1991]) y diagramas de estados (Harel [F 1987]). El trabajo de Rumbaugh es especialmente interesante, porque como él dice, nuestros métodos son más parecidos que diferentes.

Los iconos que representan objetos y paquetes fueron inspirados por la iAPX 432 [D 1981]. La notación para diagramas de objetos deriva de Seidewitz [F 1985]. La notación para la semántica de la concurrencia está adaptada del trabajo de Buhr [F 1988, 1989].

Chang [G 1990] proporciona una buena inspección del tema más general de los lenguajes visuales.

Notas bibliográficas

- [1] Shear, D. December 8, 1988. CASE Shows Promise, but Confusion Still Exists. *EDN* vol. 33(25), p. 168.
- [2] Whitehead, A. 1958. *An Introduction to Mathematics*. New York, NY: Oxford University Press.
- [3] Defense Science Board. *Report of the Defense Science Board Task Force on Military Software*. September 1987. Washington, D.C.: Office of the Undersecretary of Defense for Acquisition, p. 8.
- [4] Kleyn, M. and Gingrich, P. September 1988. GraphTrace – Understanding Object-Oriented System Using Concurrently Animated Views. *SIGPLAN Notices* vol. 23(11), p. 192.
- [5] Weinberg, G. 1988. *Rethinking Systems Analysis and Design*. New York, NY: Dorset House, p. 157.
- [6] Intel. 1981. *iAPX 432 Object Primer*. Santa Clara, CA.
- [7] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall.
- [8] Stroustrup, B. 1991. *The C+ Programming Language*, Second Edition. Reading, Massachusetts: Addison-Wesley Publishing Company.
- [9] Kiczales, G., Rivieres, J., and Bobrow, D. 1991. *The Art of the Metaobject Protocol*. Cambridge, Massachusetts: The MIT Press.
- [10] Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1993. *A Catalog of Object-Oriented Design Patterns*. Cupertino, California: Taligent.

- [11] Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* vol. 8.
- [12] Rumbaugh, *Object-Oriented Modeling and Design*.
- [13] Bear, S., Allen, P., Coleman, D., and Hayes, F. Graphical Specification of Object-Oriented System. *Object-Oriented Programming Systems, Languages, and Applications*. Ottawa, Canada: OOPSLA'90.
- [14] Rumbaugh, *Object-Oriented Modeling and Design*.
- [15] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. 1992. *Object-Oriented Software Engineering*. Workingham, England: Addison-Wesley Publishing Company.

El proceso

El ingeniero del software aficionado siempre busca la magia, algún método o herramienta sensacional cuya aplicación asegure que el desarrollo de software será trivial. Un rasgo distintivo del ingeniero del software profesional es que sabe que no existe tal panacea. Los aficionados quieren frecuentemente seguir instrucciones en forma de receta; los profesionales saben que los enfoques directos del desarrollo suelen conducir a productos de diseño ineptos, nacidos de una sucesión de mentiras, y detrás de los cuales los desarrolladores pueden escudarse para evitar aceptar responsabilidades de decisiones iniciales erróneas. El ingeniero del software aficionado, o bien ignora la documentación por completo o bien sigue un proceso dirigido por la documentación, preocupándose más sobre el aspecto que presentan de cara al cliente esos productos de papel que sobre la sustancia que contienen. El profesional reconoce la importancia que tiene crear ciertos documentos, pero nunca lo hace a expensas de realizar sensibles innovaciones arquitectónicas.

El proceso del diseño orientado a objetos no puede describirse mediante recetas, aunque está lo bastante bien definido como para ofrecer un proceso predecible y repetible para la organización de desarrollo de software madura. En este capítulo se examina este proceso incremental e iterativo en detalle, y se considera el propósito, productos, actividades y medidas de sus diversas fases.

6.1. Principios iniciales

Rasgos de los proyectos con éxito

Un proyecto de software con éxito es aquel en el que lo que se entrega satisface y posiblemente excede las expectativas del cliente, se ha desarrollado de forma económica y ajustada en el tiempo, y es elástico al cambio y la adaptación. Bajo estos supuestos, hemos observado que hay dos rasgos comunes a prácticamente

todos los sistemas con éxito orientados a objetos que hemos encontrado, y que brillaban por su ausencia en los que hemos considerado fracasados.

- La existencia de una fuerte visión arquitectónica.
- La aplicación de un ciclo de vida del desarrollo bien dirigido, iterativo e incremental.

Visión arquitectónica. Un sistema que tiene una buena arquitectura es aquel que tiene integridad conceptual y, como establece Brooks con firmeza, «la integridad conceptual es la consideración más importante en el diseño de un sistema» [1]. Como se describió en los Capítulos 1 y 5, la arquitectura de un sistema orientado a objetos abarca su estructura de clases y objetos, organizada en términos de diferentes capas y particiones. En cierto modo, la arquitectura de un sistema es en gran medida irrelevante para los usuarios finales. Sin embargo, como apunta Stroustrup, tener una «estructura interna limpia» es esencial para construir un sistema comprensible, que pueda extenderse y reorganizarse, y que sea mantenible y pueda probarse [2]. Es más, sólo teniendo una clara interpretación de la arquitectura de un sistema se hace posible el descubrir abstracciones y mecanismos comunes. La explotación de estos aspectos comunes lleva en última instancia a la construcción de sistemas más simples y, por tanto, más pequeños y fiables.

Al igual que no existe una forma «correcta» de clasificar abstracciones, no existe una forma «correcta» para idear la arquitectura de un sistema dado. Para cualquier dominio de aplicación, existen desde luego algunas vías profundamente estúpidas, y ocasionalmente algunas vías muy elegantes, para diseñar la arquitectura de una solución. ¿Cómo se distingue entonces una arquitectura buena de una mala?

Fundamentalmente, las arquitecturas buenas tienden a ser orientadas a objetos. Esto no quiere decir que todas las arquitecturas orientadas a objetos sean buenas, ni que sólo ellas lo sean. Sin embargo, como se dijo en los Capítulos 1 y 2, puede verse que la aplicación de los principios subyacentes a la descomposición orientada a objetos tiende a producir arquitecturas que exhiben las propiedades deseables de la complejidad organizada.

Las buenas arquitecturas software tienden a tener varios atributos en común:

- Están construidas en capas de abstracción bien definidas, representando cada capa una abstracción coherente y ofreciéndose a través de un interfaz controlado y bien definido, y estando construida sobre facilidades igualmente bien definidas y controladas a niveles más bajos de abstracción.
- Existe una clara separación de intereses entre el interfaz y la implantación de cada capa, haciendo posible el cambio de la implantación de una capa sin violar las suposiciones realizadas por sus clientes.
- La arquitectura es simple: el comportamiento común se consigue mediante abstracciones comunes y mecanismos comunes.

Hay que distinguir entre decisiones arquitectónicas estratégicas y tácticas. Una *decisión estratégica* es aquella que tiene vastas implicaciones arquitectónicas, e involucra así a la organización de las estructuras de la arquitectura al nivel más alto. Mecanismos para detección y recuperación de errores, paradigmas de interfaces de usuario, políticas de manejo de memoria y persistencia de objetos, enfoques de la sincronización de procesos en aplicaciones en tiempo real..., son todas ellas decisiones arquitectónicas estratégicas. En contraste, una *decisión táctica* tiene sólo implicaciones arquitectónicas locales, y por tanto normalmente sólo involucra los detalles del interfaz e implantación de una abstracción. El protocolo de una clase, la signatura de un método o la elección de un algoritmo particular para implantar un método son todas decisiones tácticas.

Un aspecto fundamental al adoptar una visión arquitectónica fuerte es el mantenimiento de un equilibrio entre estas decisiones estratégicas y tácticas. En ausencia de buenas decisiones estratégicas, incluso la clase diseñada con más astucia no será nunca lo bastante apta. Todo un conjunto de decisiones estratégicas meditadas en profundidad será arruinado si no se presta una atención cuidadosa al diseño de las clases individuales. En ambos casos, el olvidarse de una visión arquitectónica no deja más que el equivalente en software del barro.

Ciclo de vida incremental e iterativo. Considérense dos extremos: una organización que no tiene bien definido un ciclo de vida del desarrollo, y una que tiene políticas muy rígidas e impuestas de forma estricta que dictan todos y cada uno de los aspectos del desarrollo. En el primer caso, lo que hay es anarquía: mediante un trabajo duro y las contribuciones individuales de unos pocos desarrolladores, el equipo puede eventualmente producir algo de valor, pero no se puede predecir nunca nada con fiabilidad: ni progreso hasta la fecha, ni trabajo restante, y por supuesto tampoco calidad. El equipo probablemente será muy ineficiente y, en casos extremos, puede que nunca llegue a rematar y por tanto a entregar un producto de software que satisfaga las expectativas actuales o futuras de su cliente. Éste es un ejemplo de proyecto en caída libre¹. En el segundo caso, existe una dictadura, en la que se castiga la creatividad, se desalienta la experimentación que podría producir una arquitectura más elegante, y las expectativas reales del cliente nunca se comunican correctamente al desarrollador de abajo, oculto tras un verdadero muro de papel erigido por la burocracia de la organización.

Los proyectos con éxito orientados a objetos que hemos encontrado no siguen ni los ciclos de desarrollo anárquicos ni los draconianos. Antes bien, se encuentra que el proceso que conduce a la construcción próspera de arquitecturas orientadas a objetos tiende a ser iterativo e incremental. Ese proceso es iterativo en el sentido de que conlleva el refinamiento sucesivo de una arquitectura orientada a objetos, por el cual se aplica la experiencia y resultados de cada versión a la siguiente iteración del análisis y el diseño. El proceso es incremental

¹ Existe una oportunidad extrema de que un proyecto en caída libre aterrice eventualmente intacto, pero uno no querría apostar el futuro de su compañía sobre esa oportunidad.

en el sentido de que cada pasada por un ciclo análisis/diseño/evolución lleva a refinar gradualmente las decisiones estratégicas y tácticas, convergiendo en última instancia hacia una solución que se encuentra con los requerimientos reales (y habitualmente no expresados) del usuario final, y que además es simple, fiable y adaptable.

Un ciclo de vida del desarrollo iterativo e incremental es la antítesis del ciclo de vida tradicional en cascada, y así no representa ni un proceso descendente ni ascendente en sentido estricto. Es tranquilizador el darse cuenta de que hay precedentes para este estilo de desarrollo en las comunidades del hardware y el software [3, 4]. Por ejemplo, supóngase que hay que enfrentarse al problema de distribuir el personal de una organización para diseñar e implantar un dispositivo multiplaca muy complejo o algún chip VLSI de encargo. Se podría usar la organización horizontal tradicional, en la que se tiene una progresión en cascada de los productos, con los arquitectos de sistemas alimentando a los diseñadores lógicos que alimentan a los diseñadores de circuitos. Éste es un ejemplo de diseño descendente, y requiera que los diseñadores sean «hombres altos y enjutos» a causa de las habilidades especializadas y profundas que cada uno debe poseer [5]. Alternativamente, se podría usar una disposición vertical, en la que se tienen buenos diseñadores para todo que toman rebanadas del proyecto entero, desde la concepción arquitectónica hasta el diseño de circuitos. Este estilo de desarrollo es mucho más iterativo e incremental, y las habilidades que deben tener estos diseñadores nos llevan a llamarlos «hombres bajos y gordos», a causa de la amplia visión arquitectónica que cada uno debe poseer.

Nuestra experiencia indica que el desarrollo orientado a objetos no es ni estrictamente descendente ni estrictamente ascendente. En vez de eso, como sugiere Druke, los sistemas complejos de software bien estructurados se crean mejor mediante el uso del «diseño global circular». Este estilo de diseño enfatiza el desarrollo incremental e iterativo de un sistema mediante el refinamiento de vistas lógicas y físicas diferentes, aunque consistentes del sistema como un todo. El diseño global circular es el fundamento del proceso del diseño orientado a objetos.

Para unos pocos dominios de aplicación, el problema que se está resolviendo puede estar ya bien definido, y con muchas implantaciones diferentes funcionando actualmente. Aquí es posible codificar casi completamente el proceso de desarrollo: los diseñadores de un sistema nuevo en tal dominio ya comprenden cuáles son las abstracciones importantes; ya saben qué mecanismos habría que emplear, y suelen conocer ya el rango de comportamiento que se espera de un sistema semejante. La creatividad sigue siendo importante en un proceso de este tipo, pero el problema está lo suficientemente restringido como para afrontar inmediatamente la mayoría de las decisiones estratégicas del sistema. En estas circunstancias se conseguirán posiblemente cotas de productividad radicalmente altas, porque la mayor parte del riesgo de desarrolllo se ha eliminado [6]. Cuanto más se sabe sobre el problema que se va a resolver, más fácil es resolverlo.

La mayoría de los problemas del software de dimensión industrial no son

así: la mayoría implica el equilibrio de un solo conjunto de requerimientos funcionales y de ejecución, y esta tarea reclama toda la energía creativa del equipo de desarrollo.

Además, cualquier actividad humana que requiera creatividad e innovación demanda un proceso incremental e iterativo que se apoya en la experiencia, inteligencia y talento de todos los miembros del equipo². Es por tanto imposible proporcionar ninguna receta.

Hacia un proceso de diseño racional

Sin embargo, claramente deseamos ser preceptivos; de otro modo, nunca asegurariamos un proceso de desarrollo maduro y repetible para ninguna organización. Es por esta razón por lo que se habló anteriormente de tener un ciclo de vida del desarrollo incremental e iterativo y bien dirigido: bien dirigido en el sentido de que el proceso pueda controlarse y medirse, aunque no tan rígido como para que fracase otorgando suficientes grados de libertad para promover la creatividad y la innovación.

Es fundamental para la madurez de una organización de software el tener un proceso preceptivo. Como describe Humphrey, existen cinco niveles distintos de madurez de un proceso [7]:

- Inicial El proceso de desarrollo es *ad hoc* y a menudo caótico. Las organizaciones pueden progresar introduciendo controles de proyecto básicos.
- Repetible La organización tiene control razonable sobre sus planes y compromisos. Las organizaciones pueden progresar institucionalizando un proceso bien definido.
- Definido El proceso de desarrollo está razonablemente bien definido, comprendido y practicado; sirve como fundamento estable para calibrar al equipo y predecir el progreso. Las organizaciones pueden progresar instrumentando sus prácticas de desarrollo.
- Dirigido La organización tiene medidas cuantitativas de su proceso. Las organizaciones pueden progresar disminuyendo el coste de la recogida de estos datos, e instituyendo prácticas que permitan a estos datos influir en el proceso.

² Los experimentos de Curtis y sus colegas refuerzan estas observaciones. Curtis estudió el trabajo de desarrolladores profesionales de software filmándolos en acción y analizando después las diferentes actividades que emprendían (análisis, diseño, implantación, etc.) y cuándo las aplicaban. Partiendo de estos estudios, llegó a la conclusión de que «el diseño del software parece ser una colección de procesos intercalados, iterativos y poco ordenados bajo un control oportunista... El desarrollo descendente equilibrado parece ser un caso especial que se da cuando se dispone de un esquema relevante de diseño o el problema es pequeño... Los buenos diseñadores trabajan a múltiples niveles de abstracción y detalle simultáneamente» [8].

- Optimo La organización tiene establecido un proceso bien sintonizado que brinda consistentemente productos de alta calidad de modo predecible, ajustado en el tiempo y efectivo en cuanto a costes.

Desgraciadamente, como observan Parnas y Clements, «nunca encontraremos un proceso que nos permita diseñar software de forma perfectamente racional», a causa de la necesidad de creatividad e innovación durante el proceso de desarrollo. Sin embargo, como continúan diciendo, «la buena noticia es que se puede fingir...[Puesto que] los diseñadores necesitan una guía, se llegará más cerca de lo que es un proceso racional si se intenta seguir el proceso en vez de proceder sobre unas bases *ad hoc*. Cuando una organización emprende muchos proyectos de software, hay ventajas en disponer de un procedimiento estándar... Si hay acuerdo en un proceso ideal, se hace mucho más fácil medir el progreso que experimenta el proyecto» [9].

A medida que se mueven las organizaciones de desarrollo hacia niveles más altos de madurez, ¿cómo se reconcilia la necesidad de creatividad e innovación con el requerimiento de prácticas de gestión más controladas? La respuesta parece residir en la distinción de los micro y macroelementos del proceso de desarrollo. El microproceso está más estrechamente relacionado con el modelo espiral de desarrollo de Boehm, y sirve como marco de referencia para un enfoque iterativo e incremental del desarrollo [10]. El macroproceso tiene más que ver con el ciclo de vida tradicional en cascada, y sirve como el marco de referencia controlador para el microproceso. Reconciliando estos dos procesos dispares, se acaba por «fingir» un proceso de desarrollo plenamente racional, y así se dispone de los fundamentos para el nivel definido de madurez del proceso del software.

Hay que hacer hincapié en que cada proyecto es único, y de aquí que los desarrolladores deban hacer un balance entre la informalidad del microproceso y la formalidad del macroproceso. En aplicaciones preliminares, desarrolladas por un equipo estrechamente unido de desarrolladores con amplia experiencia, demasiada formalidad ahogaría la innovación; en proyectos muy complejos, desarrollados por un gran equipo de personas que probablemente estarán distribuidas geográficamente y en el tiempo, una formalidad demasiado pequeña conducirá hacia el caos.

El resto de este capítulo proporciona una visión general y después una descripción detallada del propósito, productos, actividades y medidas que constituyen los microprocesos y macroprocesos del desarrollo. En el siguiente capítulo se examinan las implicaciones prácticas de este proceso, principalmente desde la perspectiva de los gestores que deben supervisar proyectos orientados a objetos.

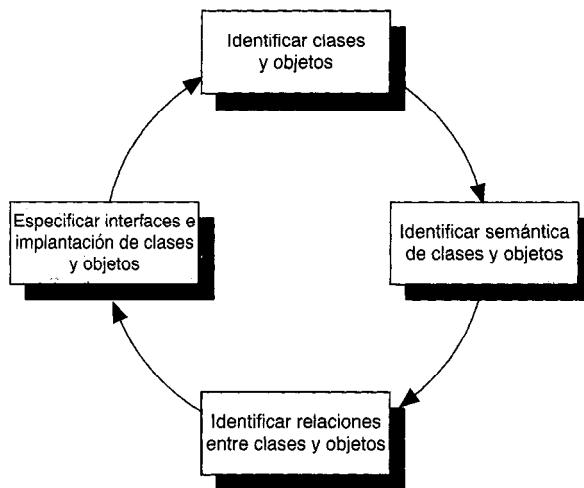


Figura 6.1. El microproceso de desarrollo.

6.2. El microproceso del desarrollo

Visión general

El microproceso del desarrollo orientado a objetos está dirigido en gran parte por la corriente de escenarios y productos arquitectónicos que emergen del macroproceso y son refinados sucesivas veces por él. En gran medida, el microproceso representa las actividades diarias del desarrollador individual o de un equipo pequeño de ellos.

El microproceso se aplica igualmente al ingeniero del software y al arquitecto del software. Desde la perspectiva del ingeniero, el microproceso ofrece una guía a la hora de tomar la mirada de decisiones tácticas que forman parte de la fabricación y adaptación diarias de la arquitectura; desde la perspectiva del arquitecto, el microproceso ofrece un marco de referencia para desplegar la arquitectura y explorar diseños alternativos.

En el microproceso, las fases tradicionales del análisis y diseño son intencionalmente borrosas, y el proceso está bajo un control oportunista. Como observa Stroustrup, «no hay métodos “de recetario” que puedan reemplazar a la inteligencia, la experiencia y el buen gusto en el diseño y la programación... Las diferentes fases de un proyecto de software, tales como diseño, programación y pruebas, no pueden separarse estrictamente» [11].

Como ilustra la Figura 6.1, el microproceso tiende a seguir las siguientes actividades:

- Identificar las clases y objetos a un nivel dado de abstracción.
- Identificar la semántica de estas clases y objetos.

- Identificar las relaciones entre estas clases y objetos.
- Especificar el interfaz y después la implementación de estas clases y objetos.

A continuación se examinará cada una de esas actividades en detalle.

Identificación de clases y objetos

Propósito. El propósito de identificar las clases y objetos es establecer los límites del problema que se maneja. Adicionalmente, esta actividad es el primer paso al proyectar una descomposición orientada a objetos del sistema que se desarrolla.

Como parte del análisis, se aplica este paso para descubrir aquellas abstracciones que forman el vocabulario del dominio del problema, y haciendo esto, se comienza a restringir el problema decidiendo qué es de interés y qué no lo es. Como parte del diseño, se aplica este paso para inventar nuevas abstracciones que forman elementos de la solución. A medida que avanza la implantación, se aplica este paso con el fin de inventar abstracciones de nivel inferior que se pueden usar para construir otras de nivel superior, y para descubrir aspectos comunes entre abstracciones existentes, que se pueden explotar entonces con objeto de simplificar la arquitectura del sistema.

Productos. El producto central de este paso es un diccionario de datos que se actualiza a medida que avanza el desarrollo. Inicialmente, puede ser suficiente acumular una «lista de cosas» que contenga todas las clases y objetos importantes, utilizando nombres significativos que impliquen su semántica [12]. Con el avance del desarrollo, y especialmente cuando va creciendo el diccionario, se hace necesario formalizar el depósito de información (*repository*), quizás mediante el uso de una simple base de datos *ad hoc* que maneje la lista, o una herramienta más específica que soporte el método directamente³. En sus variantes más formales, un diccionario de datos sirve como un índice hacia todos los demás productos del proceso de desarrollo, incluyendo los diversos diagramas y especificaciones de la notación del desarrollo orientado a objetos.

El diccionario de datos sirve así como un depósito central para las abstracciones relevantes del sistema. Inicialmente, es permisible mantener el diccionario abierto: algunas cosas podrían pasar a ser clases, otros objetos, y otras simplemente atributos de o sinónimos para otras abstracciones. A lo largo del tiempo, este diccionario se refinará añadiendo nuevas abstracciones, eliminando abstracciones irrelevantes, y consolidando las similares en una.

Hay tres beneficios esenciales que se derivan de la creación de un diccionario de datos como parte de esta actividad. Primero, el mantenimiento de un diccionario ayuda a establecer un vocabulario común y consistente que puede uti-

³ Formalmente, un diccionario de datos para desarrollo orientado a objetos contiene la especificación de cada elemento de la arquitectura.

lizarse a lo largo del proyecto. Segundo, un diccionario puede servir como un vehículo eficiente para hojear a través de todos los elementos de un proyecto de manera arbitraria. Esta característica es especialmente útil si se añaden miembros nuevos al equipo de desarrollo, que tienen que orientarse rápidamente de cara a la solución que ya se está desarrollando. Tercero, un diccionario de datos permite a los arquitectos adoptar una visión global del proyecto, que puede llevar al descubrimiento de aspectos comunes que de otra forma se soslayarían.

Actividades. Como se describió en el Capítulo 4, la identificación de clases y objetos implica dos actividades: descubrimiento e invención.

No todos los desarrolladores tienen que ser expertos en estas actividades. Los analistas, habitualmente trabajando en conjunción con expertos del dominio, deben ser buenos descubriendo abstracciones, capaces de examinar el dominio del problema y encontrar clases y objetos significativos. Análogamente, los arquitectos y los desarrolladores más veteranos deben ser expertos ideando clases y objetos que se derivan del dominio de la solución. Se discutirá la naturaleza de esta división del trabajo con más detalle en el siguiente capítulo.

En ambos casos, se efectúan estas actividades aplicando cualquiera de las diversas aproximaciones a la clasificación descritas en el Capítulo 4. El siguiente orden de los sucesos podría ser típico:

- Aplicar el enfoque clásico del análisis orientado a objetos (página 178) para generar un conjunto de clases y objetos candidatos. En los momentos iniciales del ciclo de vida, las cosas tangibles y los papeles que juegan son buenos puntos de partida. Más adelante, los eventos resultantes producirán otras abstracciones de primer y segundo orden: para cada evento, debe haber algún objeto que en última instancia es responsable de detectar ese evento y/o reaccionar ante él.
- Aplicar las técnicas de análisis del comportamiento (página 179) para identificar abstracciones que están directamente relacionadas con puntos funcionales del sistema. Los puntos funcionales del sistema, como se verá más adelante en este capítulo, provienen del ámbito del macroproceso, y representan comportamientos explícitos, visibles exteriormente y que pueden probarse. Al igual que con los eventos, para cada comportamiento debe haber entidades que inicien y participen en tal comportamiento.
- Partiendo de los escenarios relevantes generados como parte del macroproceso, aplicar las técnicas de análisis de casos de uso (página 181). En estadios tempranos del ciclo de vida se siguen escenarios iniciales que describen comportamientos amplios del sistema. A medida que avanza el desarrollo, se examinan escenarios más detallados así como escenarios periféricos con el fin de explorar los rincones oscuros del comportamiento deseado del sistema.

Para cada uno de estos enfoques, el uso de fichas CRC es un catalizador

efectivo para el proceso de tormenta de ideas, y tiene la ventaja añadida de ayudar a cuajar al equipo empujándolo a comunicarse⁴.

Algunos de los objetos y clases que se identifican inicialmente en el ciclo de vida estarán equivocados, pero eso no tiene por qué ser malo. Muchas de las cosas tangibles y los papeles que se encuentran tempranamente en el ciclo de vida se arrastrarán todo el camino hasta la implantación, al ser tan fundamentales en nuestro modelo conceptual del problema. A medida que se aprende más sobre el problema, se cambiarán probablemente los límites de ciertas abstracciones reasignando responsabilidades, combinando abstracciones similares y, bastante a menudo, dividiendo abstracciones más grandes en grupos de abstracciones que colaboran, formando así algunos de los mecanismos de la solución.

Hitos y medidas. Se completa con éxito esta fase cuando se dispone de un diccionario de datos razonablemente estable. A causa de la naturaleza incremental e iterativa del microproceso, no se espera completar o congelar este diccionario hasta muy tarde en el proceso de desarrollo. Antes bien, es suficiente tener un diccionario que contenga un conjunto amplio de abstracciones, con nombres consistentes y con una separación clara de responsabilidades.

Una medida de bondad, por tanto, es que el diccionario no sufra cambios salvajes cada vez que se itera a través del microproceso. Un diccionario que cambia rápidamente es un signo de que o bien el equipo de desarrollo no ha conseguido centrarse todavía, o bien la arquitectura es defectuosa en algún sentido. Cuando avanza el desarrollo, se puede seguir la estabilidad en las partes de nivel inferior de la arquitectura siguiendo los cambios locales en las abstracciones que colaboran.

Identificación de la semántica de las clases y objetos

Propósito. El propósito de identificar la semántica de clases y objetos es establecer el comportamiento y atributos de cada abstracción que se identifica en la fase previa. Aquí se refinan las abstracciones candidatas mediante una distribución inteligente y medible de responsabilidades.

Como parte del análisis, se aplica este paso para asignar las responsabilidades para diferentes comportamientos del sistema. Como parte del diseño, se aplica este paso para conseguir una clara separación de intereses entre las partes de la solución. A medida que avanza la implantación, se pasa de descripciones libres de los papeles y responsabilidades a la especificación de un protocolo concreto para cada abstracción, culminando eventualmente en una firma precisa para cada operación.

Productos. Hay varios productos que emergen de esta etapa. El primero es

⁴ Es un estereotipo terrible, pero algunos desarrolladores de software no son conocidos especialmente por ser grandes comunicadores.

un refinamiento del diccionario de datos, por el cual se asignan inicialmente responsabilidades a cada abstracción. Con el avance del desarrollo, se pueden crear especificaciones para cada abstracción (como se describió en el Capítulo 5), que declaran las operaciones (con su nombre) que forman el protocolo de cada clase. Tan pronto como sea posible, se deseará capturar formalmente estas decisiones escribiendo el interfaz de cada clase en el lenguaje de implantación particular. Para C++, esto significa suministrar archivos *.h*; para Ada, esto significa suministrar las especificaciones del paquete (*package*); para CLOS, esto significa escribir las funciones genéricas de cada clase; para Smalltalk, declarar pero no implementar los métodos de cada clase. Si se está tratando con los elementos de base de datos del problema, y especialmente si se está usando una base de datos orientada a objetos, se pueden producir los rudimentos del esquema.

Además de estos productos, que son por naturaleza más tácticos, se pueden producir también diagramas de objetos y diagramas de interacción que comienzan a capturar la semántica de los escenarios que se derivan del macroproceso. Estos diagramas sirven para capturar formalmente la narración de cada escenario, y reflejar así una distribución explícita de responsabilidades entre objetos que colaboran. En este punto se puede empezar también a introducir máquinas de estados finitos para ciertas abstracciones.

Al igual que en el paso anterior, se puede usar una base de datos *ad hoc* o una herramienta específica del método para llevar cuenta de las responsabilidades de cada abstracción, de forma que el equipo pueda desarrollar un lenguaje de expresión consistente. Una vez que se producen interfaces formales para las clases, se puede empezar a utilizar las herramientas de programación para probar y reforzar las decisiones de diseño.

El beneficio primario de los productos más formales de esta etapa es que fuerzan al desarrollador a considerar los aspectos pragmáticos del protocolo de cada abstracción. La incapacidad para especificar una semántica clara es un signo de que las propias abstracciones son defectuosas.

Actividades. Existen tres actividades asociadas con este paso: narración de sucesos (*storyboarding*)⁵, diseño de clases aisladas y depuración de patrones.

Los escenarios primarios y periféricos generados por el macroproceso son los conductores principales de la narración de sucesos. Esta actividad representa una identificación descendente de la semántica y, donde concierne a los puntos funcionales del sistema, afronta cuestiones estratégicas. El siguiente podría ser un orden típico de los eventos:

- Seleccionar un escenario o conjunto de ellos relacionado con un solo punto

⁵ El término inglés *storyboarding* se refiere a una actividad, propia del mundo del cine, la televisión y la publicidad, en la que se construye una narración gráfica (generalmente en forma de viñetas) que sirve de borrador para la acción que se filmará posteriormente. Desgraciadamente, en su ámbito de origen se utiliza este vocablo en inglés, por lo que no se puede acudir a equivalencias existentes; aquí el autor lo utiliza en un sentido amplio, y basándose en su significado informático, se ha traducido como «narración de sucesos». También se utiliza el término «presentación». (N. del T.)

funcional; identificar las abstracciones (de la etapa anterior) relevantes al escenario.

- Pasearse por la actividad del escenario, asignando responsabilidades suficientes a cada abstracción para cumplir el comportamiento deseado. Según se necesite, asignar atributos que representen elementos estructurales que se requieran para llevar a cabo ciertas responsabilidades.
- Al ir avanzando, reasignar responsabilidades para distribuir el comportamiento con un equilibrio razonable. Si se puede, reusar o adaptar responsabilidades existentes. Es muy habitual partir responsabilidades grandes en varias pequeñas; unir otras triviales en un comportamiento mayor es menos frecuente, pero tampoco es raro.

Informalmente, se pueden usar fichas CRC para la narración de sucesos. Más formalmente, el equipo de desarrollo puede escribir diagramas de objetos o diagramas de interacción. Durante el análisis, esta narración de sucesos la realiza típicamente un equipo que incluye al analista, el experto del dominio, el arquitecto y alguien de control de calidad (pero no se limita necesariamente a ellos). Durante el diseño y después en la implantación, la narración la llevan a cabo los arquitectos y los desarrolladores veteranos para refinar decisiones estratégicas, y los desarrolladores individuales para refinar decisiones tácticas. La participación de miembros adicionales del equipo en la narración de sucesos es un modo muy efectivo de formar a los desarrolladores jóvenes y de comunicar la visión arquitectónica.

En los inicios del proceso de desarrollo, se puede especificar la semántica de las clases y objetos redactando las responsabilidades para cada abstracción en un texto de formato libre. Normalmente basta una frase o una sola sentencia; cualquier cosa mayor sugiere que determinada responsabilidad es demasiado compleja y debería dividirse en otras más pequeñas. En fases más avanzadas de ese proceso, cuando se empieza a refinar el protocolo de las abstracciones individuales, se puede dar nombre a operaciones específicas, ignorando sus firmas completas. Tan pronto como resulte práctico, se pueden asignar firmas completas a cada operación. De este modo, la cosa es manejable: una responsabilidad concreta la satisface un conjunto de operaciones cooperativas, y cada operación contribuye de alguna forma a las responsabilidades de una abstracción. En este punto, se pueden introducir máquinas de estados finitos para ciertas clases, especialmente aquellas cuyas responsabilidades implican comportamiento dirigido por eventos u ordenado en estados, con el fin de capturar la semántica dinámica de sus protocolos⁶.

En esta fase es importante centrarse en el comportamiento, no en la estructura. Los atributos representan elementos estructurales, y así existe el peligro, especialmente en fases tempranas del análisis, de vincular decisiones de implantación demasiado pronto requiriendo la presencia de ciertos atributos. Los atri-

⁶ Como se describió en el Capítulo 3, un protocolo específica que ciertas operaciones van a invocarse en un orden específico. Para cualquier clase que no sea trivial, las operaciones rara vez están solas; cada una tiene precondiciones que hay que satisfacer, muchas veces invocando otras operaciones.

butos deberían identificarse en este momento sólo en la medida en que sean esenciales para construir un modelo conceptual del escenario.

El diseño de clases aisladas representa una identificación ascendente de la semántica. Aquí se centra la atención en una sola abstracción y, aplicando los heurísticos para diseño de clases descritos en el Capítulo 3, se consideran las operaciones que completan la abstracción. Esta actividad es de naturaleza más táctica, porque el interés se dirige hacia un buen diseño de clases, no hacia el diseño arquitectónico. Un orden típico de los eventos podría ser el siguiente:

- Seleccionar una abstracción y enumerar sus papeles y responsabilidades.
- Idear un conjunto suficiente de operaciones que satisfagan estas responsabilidades. Donde se pueda, intentar reusar operaciones para papeles y responsabilidades conceptualmente similares.
- Considerar cada operación por turno y asegurarse de que es primitiva. Si no lo es, aislar y exponer sus operaciones más primitivas. Las operaciones compuestas pueden retenerse en la propia clase (si es lo bastante común, o por razones de eficiencia) o migrar a una utilidad de clase (sobre todo si es probable que cambie a menudo). Donde sea posible, considerar un conjunto mínimo de operaciones primitivas.
- Especialmente más tarde en el ciclo del desarrollo, considerar las necesidades de construcción, copia y destrucción [13]. Es mejor tener una política estratégica común para estos comportamientos, en vez de permitir que las clases individuales sigan sus propias pautas, a menos que haya razones de peso para hacerlo así.
- Considerar la necesidad de completud: añadir otras operaciones primitivas que no son necesariamente exigidas por los clientes inmediatos, pero cuya presencia rodea a la abstracción, y por tanto serían usadas probablemente por futuros clientes. Asumiendo que es imposible tener una completud perfecta, inclinarse más hacia la simplicidad que hacia la complejidad.

Es importante evitar buscar relaciones de herencia demasiado pronto: la introducción de la herencia prematuramente suele llevar a pérdidas de integridad de tipos.

En las primeras etapas del desarrollo, el diseño de clases está en realidad aislado. Sin embargo, una vez que se tienen entramados de herencias, esta fase debe afrontar la ubicación de las operaciones en la jerarquía. A medida que se consideran las operaciones asociadas con una abstracción dada, hay que decidir entonces a qué nivel de abstracción están mejor situadas. Las operaciones que pueden utilizarse por parte de un conjunto de clases hermanas deberían migrar hacia una superclase común, posiblemente introduciendo una nueva clase abstracta intermedia. Las operaciones que pueden ser utilizadas por un conjunto disjunto de clases deberían encapsularse en una clase aditiva.

La tercera actividad, depuración de patrones, reconoce la importancia de los aspectos comunes. A medida que se identifica la semántica de las clases y ob-

jetos, hay que ser sensible a los patrones de comportamiento, que representan oportunidades de reuso. Un orden típico de los eventos podría ser el siguiente:

- Dado el conjunto completo de escenarios a este nivel de abstracción, buscar patrones de interacción entre abstracciones. Tales colaboraciones pueden representar mecanismos o pautas implícitos, que deberían ser examinados para asegurar que no hay diferencias gratuitas entre cada invocación. Los patrones de colaboración que no son triviales deberían documentarse explícitamente como decisiones estratégicas, de forma que puedan reutilizarse en lugar de reinventarse. Esta actividad preserva la integridad de la visión arquitectónica.
- Dado el conjunto de responsabilidades generado a este nivel de abstracción, buscar patrones de comportamiento. Los papeles y responsabilidades comunes deberían unificarse en forma de clases base, abstractas o aditivas comunes.
- Especialmente en fases tardías del ciclo de vida, cuando se están especificando operaciones concretas, buscar patrones en las signaturas de las operaciones. Eliminar cualquier diferencia gratuita e introducir clases aditivas o clases de utilidades cuando se halle que tales signaturas son repetitivas.

Nótese que las actividades de identificar y especificar la semántica de clases y objetos se aplican tanto a clases individuales como a categorías de clases. La semántica de una clase, al igual que la de una categoría de clases, abarca sus papeles y sus responsabilidades, así como sus operaciones. En el caso de una clase individual, estas operaciones pueden expresarse eventualmente como funciones miembro concretas; en el caso de una categoría de clases, estas operaciones representan los servicios exportados por la categoría, y son proporcionados en última instancia por un conjunto de clases que colaboran, no sólo por una simple clase. De este modo, las actividades descritas arriba se aplican igual de bien al diseño de clases y al diseño arquitectónico.

Hitos y medidas. Se completa con éxito esta fase cuando para cada abstracción se tiene un conjunto de responsabilidades y/o operaciones razonablemente suficiente, primitivo y completo. En etapas tempranas del proceso de desarrollo, es suficiente tener una declaración informal de responsabilidades. A medida que avanza el desarrollo, hay que tener una semántica establecida con mayor precisión.

Las medidas de bondad incluyen todos los heurísticos para clases descritos en el Capítulo 3. Las responsabilidades y operaciones que no son ni simples ni claras sugieren que la abstracción dada aún no está bien definida. La incapacidad para expresar un fichero de cabecera concreto u otros tipos de interfaces de clase formales sugiere también que la abstracción está mal formada, o que la abstracción la está haciendo la persona equivocada⁷.

⁷ Guárdese el lector de los analistas o arquitectos que no están dispuestos o son incapaces de expresar concretamente la semántica de sus abstracciones: esto es signo de arrogancia o ineptitud.

Durante los recorridos de cada escenario, es de esperar que haya vivos debates. Tales actividades ayudan a comunicar la visión arquitectónica, y ayudan a desarrollar la destreza en la abstracción. La abstracción que no se ha examinado no está dignamente escrita.

Identificación de las relaciones entre clases y objetos

Propósito. El propósito de identificar las relaciones entre clases y objetos es consolidar las fronteras y reconocer los colaboradores de cada abstracción que se identificó previamente en el microproceso. Esta actividad formaliza la separación tanto conceptual como física de intereses entre abstracciones que se había iniciado en la etapa anterior.

Como parte del análisis, se aplica este paso para especificar las asociaciones entre clases y objetos (incluyendo ciertas relaciones importantes de herencia y agregación). La expresión de la existencia de una asociación identifica alguna dependencia semántica entre dos abstracciones, así como alguna posibilidad de navegación desde una entidad a otra. Como parte del diseño, se aplica este paso para especificar las colaboraciones que forman los mecanismos de la arquitectura, así como el agrupamiento (de nivel superior) de las clases en categorías y de los módulos en subsistemas. Con el avance de la implantación, se refinan las relaciones similares a las asociaciones en relaciones más orientadas a la implantación, incluyendo instanciación y uso.

Productos. Los diagramas de clases, diagramas de objetos y diagramas de módulos son los productos principales de este paso. Aunque finalmente hay que expresar de una forma concreta las decisiones de análisis y diseño que conciernen a las relaciones (a saber, a través de los lenguajes de programación), los diagramas ofrecen una visión más amplia de la arquitectura, y además permiten expresar relaciones que no se ven apoyadas por la lingüística de los sistemas de programación.

Durante el análisis, se producen diagramas de clases que establecen las asociaciones entre abstracciones, y añaden detalles del estadio anterior (las operaciones y atributos de ciertas abstracciones) según sea necesario para capturar las utilidades importantes de las decisiones. Durante el diseño se refinan estos diagramas para mostrar las decisiones tácticas que se han tomado sobre herencia, agregación, *instanciación* y uso.

No es deseable (ni siquiera posible) producir un vasto conjunto de diagramas que expresen todas las visiones concebibles de las relaciones entre las abstracciones. En lugar de eso hay que centrarse en las visiones «interesantes», donde la medida de *interés* abarca cualquier conjunto de abstracciones relacionadas cuyas relaciones son expresión de alguna decisión arquitectónica fundamental, o que expresan un detalle necesario para completar un plano para la implantación.

A medida que avanza el diseño arquitectónico, también se generan diagra-

mas de clases que contienen categorías de clases que identifican el agrupamiento de abstracciones en capas y particiones. Estos productos sirven para documentar el marco de referencia arquitectónico.

Durante el análisis, se producen también diagramas de objetos que completan el recorrido de escenarios que había comenzado en la etapa anterior. Lo que es diferente aquí es que ahora puede considerarse la interacción entre clases y objetos, y por tanto pueden descubrirse patrones de interacción que estaban ocultos y que se buscaría explotar. Esto suele llevar a un «pellizco» local del entramado de herencias. Durante el diseño se usan diagramas de objetos junto con máquinas de estados finitos más detalladas para mostrar la acción dinámica de los mecanismos. En realidad, un producto explícito de este paso es un conjunto de diagramas que identifican las colaboraciones que sirven como mecanismos del diseño.

A medida que avanza la implantación, hay que tomar decisiones sobre el empaquetamiento físico del sistema en módulos, y la asignación de procesos a procesadores. Son ambas decisiones de relaciones, que pueden expresarse en diagramas de módulos y de procesos.

El diccionario de datos se actualiza también como parte de esta etapa para reflejar la asignación de clases y objetos a categorías y de módulos a subsistemas.

La utilidad principal de estos productos es que ayudan a visualizar y razonar sobre relaciones que pueden atravesar entidades conceptual y físicamente distintas.

Actividades. Existen tres actividades asociadas con este paso: la especificación de asociaciones, la identificación de varias colaboraciones y el refinamiento de las asociaciones.

La identificación de asociaciones es principalmente una actividad de análisis y de diseño inicial. Como se explicó en el Capítulo 3, las asociaciones son semánticamente débiles: sólo representan alguna suerte de dependencia semántica, el papel y cardinalidad de cada participante en la relación, y posiblemente una declaración de navegabilidad. Sin embargo, durante el análisis y el diseño inicial, esto es con frecuencia suficiente, porque captura suficientes detalles interesantes sobre la relación entre dos abstracciones, aunque nos previene de realizar declaraciones prematuras de diseño detallado. Un orden típico de los eventos para esta actividad podría ser el siguiente:

- Recoger un conjunto de clases a un nivel dado de abstracción, o asociado con una familia concreta de escenarios; poblar el diagrama con las operaciones y atributos importantes de cada abstracción según se necesite para ilustrar las propiedades significativas del problema que se modela.
- Considerar la presencia de una dependencia semántica entre dos clases cualesquiera, y establecer una asociación si existe tal dependencia. La necesidad de navegación de un objeto a otro y la necesidad de deducir algún comportamiento de un objeto son causas para introducir asociaciones. Las

dependencias indirectas son causa para introducir nuevas abstracciones que sirven como agentes o intermediarios. Algunas asociaciones pueden identificarse inmediatamente como relaciones de agregación o de especialización/generalización.

- Para cada asociación, especificar el papel de cada participante, así como cualquier cardinalidad relevante u otro tipo de restricción.
- Validar estas decisiones recorriendo los escenarios y asegurándose de que esas asociaciones están situadas de forma necesaria y suficiente para proporcionar la navegación y comportamiento entre abstracciones que requiere cada escenario.

Los diagramas de clases son el modelo principal que genera esta actividad.

La identificación de colaboraciones es más que nada una actividad de diseño, y es también en gran parte un problema de clasificación, como se describió en el Capítulo 4. Como tal, esta etapa requiere creatividad y perspicacia. Dependiendo de en qué punto del macroproceso se está, hay una serie de tipos de colaboración diferentes que hay que considerar:

- Como parte de la formulación de las decisiones estratégicas, hay que especificar los mecanismos identificados en el paso previo mediante la producción de un diagrama de objetos para cada uno, ilustrando su semántica dinámica. Validar cada mecanismo recorriendo los escenarios principales y periféricos. Donde hay oportunidades para la concurrencia, especificar los actores, agentes y servidores, y los medios de sincronización entre ellos. Sobre la marcha, puede descubrirse la necesidad de introducir nuevas vías entre objetos, así como de eliminar o consolidar otras redundantes o poco utilizadas.
- Según se encuentren aspectos comunes entre clases, deben ubicarse esas clases en una jerarquía de generalización/especialización. Como se describió en el Capítulo 3, normalmente es mejor crear bosques de clases en vez de un solo árbol de clases. Partiendo del paso previo, ya se habrán identificado las candidatas a clase base, abstracta o aditiva, que pueden colocarse ahora en un grafo de herencias. Añadir clases concretas significativas al diagrama de clases resultante, y revisarlo para considerar su bondad, de acuerdo con los heurísticos del Capítulo 3. En particular, hay que tener cuidado con el equilibrio (el grafo no debería ser demasiado alto ni demasiado corto, y tampoco demasiado ancho ni demasiado delgado). Donde aparezcan patrones de estructura o comportamiento entre clases, reorganizar la trama para maximizar los elementos comunes (pero no a expensas de la simplicidad).
- Como parte del diseño arquitectónico, hay que considerar el agrupamiento de clases en categorías y la organización de los módulos en subsistemas. Estas decisiones tienen implicaciones estratégicas. Los arquitectos pueden usar diagramas de clases para especificar la jerarquía de categorías de clases que forman las capas y divisiones del sistema que se desarrolla. Típicamente, esto se hace en sentido descendente, adoptando

una visión global del sistema y particionándolo en abstracciones que denoten servicios generales del sistema que sean lógicamente cohesivos y/o cambien probablemente de forma independiente. Esta arquitectura puede refinarse también en sentido ascendente, a medida que se identifican grupos de clases semánticamente cercanas en cada pasada del microproceso. Con el avance del desarrollo, hay que tomar también decisiones sobre la asignación de cada clase a una categoría. Cuando las categorías existentes se van hinchaendo, o cuando se hacen evidentes nuevos agrupamientos, se puede decidir introducir nuevas categorías de clases o reorganizar la asignación de otras existentes. La organización de los módulos en subsistemas sigue un conjunto de actividades parecido, excepto en que aquí el centro de atención son los elementos del modelo físico, y así se generan diagramas de módulos para capturar las decisiones.

- La asignación de clases y objetos a módulos es en cierta manera una decisión local, y la mayoría de las veces refleja las relaciones de visibilidad entre abstracciones. Como se describió en el Capítulo 5, la correspondencia entre el modelo lógico y el físico da al desarrollador la oportunidad de abrir o restringir el acceso a cada abstracción, así como empaquetar abstracciones relacionadas lógicamente que es de esperar que cambien juntas. Como se discutirá en el siguiente capítulo, la estructura de división de trabajo en el equipo de desarrollo también da color a esa acción de correspondencia de lo lógico con lo físico. En cualquier caso, se pueden capturar las decisiones en forma de diagramas de módulos.

La tercera actividad de esta fase del microproceso, el refinamiento de las asociaciones, es una actividad de análisis y de diseño. Durante el análisis, se pueden desplegar ciertas asociaciones en otras relaciones más precisas semánticamente para reflejar la comprensión creciente que se tiene del dominio del problema. Durante el diseño, se transforman análogamente asociaciones así como se añaden nuevas relaciones concretas con el fin de proporcionar un plano para la implantación.

La herencia, contención, *instanciación* y uso son los tipos principales de relaciones de interés, junto con otras propiedades como etiquetas, papeles, cardinalidad, etc. Un orden típico de los eventos para esta actividad podría ser el siguiente:

- Dada una colección de clases ya agrupadas por algún conjunto de asociaciones, buscar patrones de comportamiento que representen oportunidades para la generalización/especialización. Situar las clases en el contexto de una trama (enrejado) de herencias existente, o fabricar una trama si no hay ninguna apropiada.
- Si hay patrones de estructura, considerar la creación de nuevas clases que capturen esta estructura común, e introducirlas bien sea mediante herencia (como clases asociativas) o mediante agregación.
- Buscar clases similares en su comportamiento que o bien sean hermanos disjuntos en un grafo de herencias o bien que no sean aún parte de un

grafo de herencias, y considerar la posibilidad de introducir clases parametrizadas comunes.

- Considerar la navegabilidad de las asociaciones existentes, y restringirlas como sea posible. Reemplazarlas por relaciones de uso simples si no se desea que se dé la propiedad de navegación bidireccional.
- A medida que avanza el desarrollo, introducir detalles tácticos como declaraciones de papeles, claves, cardinalidad, amistad, posturas, etc. No es deseable establecer todos los detalles: incluir sólo información que represente una posición importante de análisis o diseño, o que sea necesaria para la implantación.

Hitos y medidas. Se completa con éxito esta fase cuando se ha especificado la semántica y las relaciones entre ciertas abstracciones interesantes de forma suficiente como para servir como planos para su implantación.

Las medidas de bondad comprenden la cohesión, el acoplamiento y la completitud (estado completo). Al revisar las relaciones que se descubren o inventan durante esta fase, se busca tener abstracciones lógicamente cohesivas y débilmente acopladas. Además, se busca identificar todas las relaciones importantes a un nivel dado de abstracción, de forma que la implantación no obligue a introducir nuevas relaciones significativas o realizar acciones poco naturales para utilizar las que ya se han especificado. En el paso siguiente, si se encuentra que las abstracciones son difíciles de implantar, esto será síntoma de que aún no se ha ideado un conjunto significativo de relaciones entre ellas.

Implementación de clases y objetos

Propósito. Durante el análisis, el propósito de implantar clases y objetos es proporcionar un refinamiento de las abstracciones existentes suficiente para descubrir nuevas clases y objetos del nivel siguiente de abstracción, que se introducen entonces en la siguiente iteración del microproceso. Durante el diseño, el propósito de esta actividad es crear representaciones tangibles de las abstracciones en apoyo del refinamiento sucesivo de las versiones ejecutables en el macroproceso.

La ordenación de esta etapa es intencionada: el microproceso se centra primero en el comportamiento, y aplaza las decisiones sobre la representación hasta el momento más tardío posible. Esta estrategia evita decisiones de implantación prematuras que pueden arruinar oportunidades para arquitecturas más pequeñas o simples, y también ofrece la libertad de cambiar las representaciones según se necesite por razones de eficiencia, mientras se limita el rompimiento de la arquitectura existente.

Productos. Las decisiones sobre la representación de cada abstracción y la correspondencia entre estas representaciones y el modelo físico dirigen los pro-

ductos de esta etapa. En momentos iniciales del proceso de desarrollo se pueden capturar estas decisiones tácticas de representación en forma de especificaciones de clase refinadas. Donde estas decisiones sean de interés general o representen oportunidades de reutilización, se las documenta también en diagramas de clases (mostrando su semántica estática) y máquinas de estados finitos o diagramas de interacción (mostrando su semántica dinámica). Según avanza el desarrollo, y según se llevan a cabo más ligaduras con el lenguaje de implantación, se comienza a entregar pseudocódigo o código ejecutable.

Para mostrar las ligaduras que existen en la implementación entre lo lógico y lo físico, se entregan también diagramas de módulos, que pueden utilizarse entonces para visualizar la correspondencia de la arquitectura con su realización en forma de código. Con el avance del desarrollo, se pueden utilizar herramientas específicas del método que generan código automáticamente partiendo de estos diagramas (ingeniería directa), o bien realizan ingeniería inversa generando los diagramas a partir de la implementación.

Como parte de esta etapa, también se actualiza el diccionario de datos, incluyendo las nuevas clases y objetos que se han descubierto o inventado al formular la implementación de abstracciones existentes. Estas nuevas abstracciones son parte de las entradas de la siguiente ronda del micropoproceso.

Actividades. Existe una actividad primordial asociada a esta etapa: la selección de las estructuras y algoritmos que suministran la semántica de las abstracciones que se identificaron previamente en el micropoproceso. Mientras las primeras tres fases del micropoproceso se centran en la vista externa de las abstracciones, este paso se centra en su vista interna.

Durante el análisis, los resultados de esta actividad son relativamente abstractos: el interés no está tanto en tomar decisiones de representación, sino en descubrir las nuevas abstracciones en las que se puede delegar responsabilidad. Durante el diseño, y especialmente en fases avanzadas del diseño de clases, hay que tomar cada vez más decisiones concretas.

Un orden típico de los eventos para esta actividad podría ser el siguiente:

- Para cada clase, considerar de nuevo su protocolo. Identificar los patrones de uso entre clientes, con el fin de determinar qué operaciones son centrales, y por tanto cuáles habría que optimizar. Según avanza la implantación, desarrollar signaturas precisas para todas las operaciones significativas.
- Antes de elegir una representación de cero, considerar el uso de herencia private o protected para la implantación, o el uso de clases parametrizadas. Seleccionar las clases abstractas o aditivas apropiadas (o crear otras nuevas, si el problema es lo bastante general), y ajustar el entramado de herencias como sea necesario.
- Considerar los objetos en los que se podría delegar responsabilidad. Para un ajuste óptimo, esto puede requerir un pequeño reajuste de las responsabilidades y/o protocolo de la abstracción de nivel inferior.

- Si la semántica de la abstracción no puede suministrarse mediante herencia, instanciación o delegación, considerar una representación conveniente partiendo de primitivas del lenguaje. Tener presente la importancia de las operaciones desde la perspectiva de los clientes de la abstracción, y seleccionar una representación que sea óptima respecto a los patrones de uso esperados. Recordar que no se puede optimizar, sin embargo, respecto a todos los usos. A medida que se gana información empírica de versiones sucesivas, se puede identificar qué abstracciones no son eficientes en cuanto a tiempo y/o espacio, y alterar su implantación localmente, con cierta preocupación de no violar las suposiciones que los clientes hagan sobre la abstracción.
- Seleccionar un algoritmo conveniente para cada operación. Introducir operaciones auxiliares para dividir los algoritmos complejos en partes reusables y menos complicadas. Considerar los pros y contras entre almacenar o calcular ciertos estados de una abstracción.

Hitos y medidas. Durante el análisis se completa esta fase con éxito una vez que se han identificado todas las abstracciones interesantes necesarias para satisfacer las responsabilidades de abstracciones de nivel superior identificadas durante esta pasada del microproceso. Durante el diseño, se completa esta fase cuando se tiene un modelo ejecutable o casi ejecutable de las abstracciones.

La medida de bondad principal para esta fase es la simplicidad. Las implantaciones complejas, difíciles o ineficientes son una indicación de que la propia abstracción tiene carencias, o de que se ha elegido una representación pobre.

6.3. El macroproceso del desarrollo

Visión general

El macroproceso sirve como el marco de referencia para controlar al microproceso. Este procedimiento más amplio dicta una serie de productos y actividades medibles que permiten al equipo de desarrollo tasar el riesgo de forma significativa y realizar correcciones iniciales al microproceso, de forma que se centren mejor las actividades de análisis y diseño del equipo. El macroproceso representa las actividades del equipo de desarrollo completo en una escala de semanas o meses de cada vez.

Muchos elementos del macroproceso son simplemente buenas prácticas de gestión del software, y por tanto se aplican igualmente a sistemas orientados a objetos y no orientados a objetos. Éstas incluyen prácticas básicas como gestión de configuraciones, control de calidad, recorridos del código y documentación. En el siguiente capítulo se tratarán varias de estas cuestiones pragmáticas en el contexto del desarrollo de software orientado a objetos. El centro de interés en

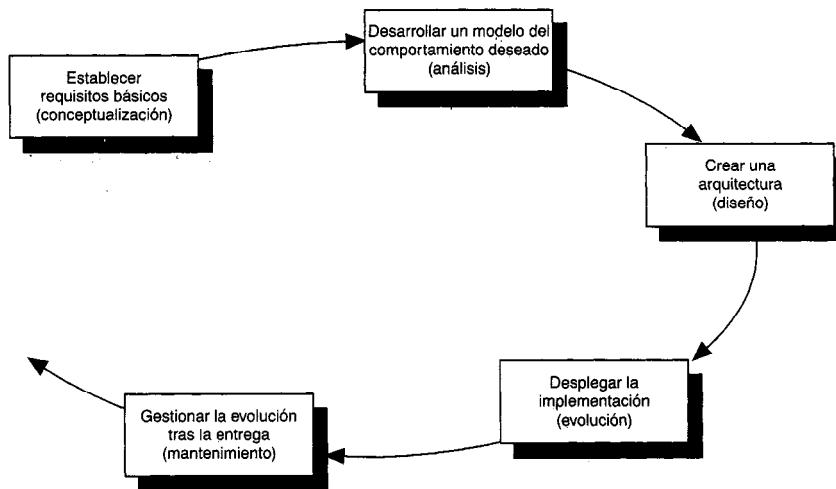


Figura 6.2. El macroproceso del desarrollo.

este capítulo será describir un proceso esencial que se orienta a la construcción de sistemas orientados a objetos. En palabras de Parnas, ésta es la forma en la que se fingirá un proceso racional de diseño para construir sistemas orientados a objetos.

El macroproceso es principalmente del interés de la dirección técnica del equipo de desarrollo, cuya preocupación principal es sutilmente diferente de la del desarrollador individual. Ambos están interesados en entregar software de calidad que satisfaga las necesidades del cliente⁸. Sin embargo, a los usuarios finales generalmente podría importarles menos el hecho de que los desarrolladores usen clases parametrizadas y funciones polimórficas con habilidad; los clientes están mucho más interesados en fechas, calidad y completud, y están en su derecho. Por esta razón, el macroproceso se centra en el riesgo y la visión arquitectónica, los dos elementos gestionables que tienen el mayor impacto en las fechas, calidad y completud (estado completo).

En el macroproceso, las fases tradicionales del análisis y el diseño se retienen hasta un gran alcance, y el proceso está razonablemente bien ordenado. Como ilustra la Figura 6.2, el macroproceso tiende a atravesar las siguientes actividades:

- Establecer los requisitos centrales para el software (conceptualización).
- Desarrollar un modelo del comportamiento deseado del sistema (análisis).
- Crear una arquitectura para la implementación (diseño).

⁸ Bueno, la mayoría de ellos. Desgraciadamente, algunos directores están más interesados en construir imperios que en construir software. Véase también la nota al pie de la página 276: supongo que Dante podría proponer un lugar para ambos grupos de personas.

- Transformar la implementación mediante refinamiento sucesivo (evolución).
- Gestionar la evolución postventa o postentrega (mantenimiento).

Para todo el software de interés, el macroproceso se repite tras las versiones principales del producto. Esto es especialmente cierto en organizaciones que se concentran en la fabricación de familias de programas, que frecuentemente representan una inversión de capital significativa.

La filosofía básica del macroproceso es la del desarrollo incremental. Como lo define Vonk, «en el caso del desarrollo incremental, el sistema en su conjunto se construye paso a paso, y cada versión sucesiva consta de la versión anterior sin cambios más una serie de funciones nuevas» [14]. Este enfoque es extremadamente apropiado para el paradigma orientado a objetos, y ofrece una serie de beneficios en relación a la gestión del riesgo. Como dice Gilb con toda corrección, «la entrega evolucionaria se ideó para darnos señales de alarma temprana para verdades inminenteamente desagradables» [15].

Examinemos con detalle cada una de las actividades del macroproceso. Por supuesto, una de las características de una organización de desarrollo madura es el conocimiento de cuándo hay que romper estas reglas, y por tanto se apuntarán estos grados de libertad sobre la marcha.

Conceptualización

Propósito. La conceptualización persigue establecer los requisitos esenciales para el sistema. Para cualquier elemento de software verdaderamente nuevo, o incluso para la adaptación nueva de un sistema existente, hay algún momento en el tiempo en el cual, en la mente del desarrollador, el arquitecto, el analista o el usuario final, salta una idea para alguna aplicación. Esta idea puede representar una nueva aventura de negocios, un nuevo producto complementario en una línea de productos existente, o quizás un nuevo conjunto de características para un sistema de software existente. No es el propósito de la conceptualización definir completamente estas ideas, sino que el propósito de la conceptualización es establecer la visión de la idea y validar sus suposiciones.

Productos. Los prototipos son los productos primarios de la conceptualización. Específicamente, para todo sistema nuevo de relevancia, debería haber alguna prueba del concepto, que se manifestase en forma de un prototipo rápido-y-sucio. Tales prototipos son por su propia naturaleza incompletos, y sólo están diseñados de manera marginal. Sin embargo, conservando los prototipos interesantes (aunque quizás rechazados), la organización mantiene una memoria corporativa de sus visiones originales, y conserva así las suposiciones que se habían hecho cuando las aplicaciones se concibieron por primera vez. A medida que avanza el desarrollo de sistemas de producción, este repositorio pro-

porciona un lugar para experimentar sin restricciones, al cual los analistas y los arquitectos pueden volver para intentar nuevas ideas.

Obviamente, para aplicaciones a escala masiva (como aquellas de relevancia nacional, o las que tienen implicaciones multinacionales), el propio esfuerzo de construcción de prototipos puede ser una gran empresa. Esto es de esperar, y de hecho se promueve. Es mucho mejor descubrir que las suposiciones sobre funcionalidad, eficacia, tamaño o complejidad eran erróneas durante la prueba del concepto, en vez de hacerlo más tarde, cuando el abandono de la vía de desarrollo actual demostraría ser financiera o socialmente desastrosa.

Hay que hacer hincapié en que todos estos prototipos están destinados a ser desechados. No debería permitirse que los prototipos evolucionasen directamente al interior del sistema en producción, a menos que haya una razón absolutamente poderosa. La conveniencia con motivo de encontrar un atajo en la planificación no es en absoluto una razón de peso: esta decisión representa un ahorro falso que optimiza respecto a la rapidez del desarrollo, e ignora el coste de posesión del software.

Actividades. La conceptualización es, por su propia naturaleza, una actividad intensamente creativa, y por tanto no debería verse encadenada por reglas de desarrollo rígidas. Lo que quizás sea más importante para la organización de desarrollo es habilitar una estructura que proporcione suficientes recursos para explorar nuevas ideas⁹. Las ideas nuevas pueden surgir prácticamente de cualquier fuente: usuarios finales, grupos de usuarios, desarrolladores, analistas, el equipo de mercadotecnia, etc. Es una sabia conducta por parte de la dirección el mantener un registro de estas ideas nuevas, de forma que pueda dárseles prioridad, y asignárselas de forma inteligente los escasos recursos para explorar las más prometedoras. Una vez que se ha seleccionado una vía particular para la exploración, un orden típico de los eventos es el siguiente:

- Establecer un conjunto de objetivos para la prueba del concepto, incluyendo criterios para saber cuándo finaliza este esfuerzo.
- Reunir un equipo apropiado para desarrollar el prototipo. Frecuentemente puede ser un equipo de una sola persona (habitualmente el visionario original). Lo mejor que puede hacer la organización de desarrollo para facilitar el trabajo del equipo es apartarse de su camino.
- Evaluar el prototipo resultante, y tomar una decisión explícita sobre el desarrollo del producto o exploraciones posteriores. Las decisiones para desarrollar un producto deberían tomarse con una evaluación razonable de los riesgos potenciales, los cuales podría no haber previsto la prueba del concepto.

En la conceptualización no hay nada inherentemente orientado a objetos.

⁹ Si la propia organización no lo hace, los desarrolladores individuales lo harán de todas formas, con despecho hacia la compañía para la que trabajan. Ésta es la forma en que nacen nuevas compañías de software. Esto es frecuentemente muy bueno para la industria en su conjunto, pero suele representar una pérdida neta para la organización.

Todos y cada uno de los paradigmas de programación deberían permitir el desarrollo de pruebas de concepto. Sin embargo, se da con frecuencia el caso de que, en presencia de un marco de referencia de aplicaciones orientado a objetos razonablemente rico, el desarrollo de prototipos suele ser más rápido que en las otras alternativas.

No es extraño ver pruebas de concepto desarrolladas en un lenguaje (como Smalltalk, por ejemplo) mientras que el desarrollo del producto se realiza en otro distinto (como C++).

Hitos y medidas. Es importante que se establezcan criterios explícitos para la terminación de un prototipo. Las pruebas de concepto están dirigidas frecuentemente por razones de planificación (lo que significa que el prototipo debe entregarse en determinada fecha) en vez de por razones de características. Esto no tiene por qué ser malo, porque limita artificialmente el esfuerzo de construcción de prototipos, y desalienta la tendencia a entregar prematuramente un sistema de producción.

La dirección superior puede medir con frecuencia la salud de la organización de desarrollo de software midiendo su respuesta ante nuevas ideas. Cualquier organización que no produce ideas nuevas por sí misma está muerta, o en un negocio moribundo. La acción más prudente suele ser diversificar o abandonar el negocio. En contraste, cualquier organización inundada por ideas nuevas, pero que es aún incapaz de darles prioridad de forma inteligente, está fuera de control. Tales organizaciones suelen derrochar recursos de desarrollo significativos por saltar al desarrollo de los productos con demasiada precipitación, sin explorar los riesgos del esfuerzo a través de una prueba del concepto. La acción más prudente aquí es formalizar el proceso de producción, y hacer explícito el salto de concepto a producto.

Análisis

Propósito. Como dice Mellor, «el propósito del análisis es proporcionar una descripción de un problema. La descripción debe ser completa, consistente, legible y revisable por las diversas partes interesadas, [y] verificable frente a la realidad» [16]. En nuestras propias palabras, el propósito del análisis es proporcionar un modelo del comportamiento del sistema.

Hay que hacer hincapié en que el análisis se centra en el comportamiento, no en la forma. No es apropiado perseguir cuestiones de diseño de clases, representación u otras decisiones tácticas durante esta fase. Antes bien, el análisis debe dar lugar a una declaración de lo que hace el sistema, no de cómo lo hace. Cualquier afirmación intencionada sobre el «cómo» durante el análisis debería entenderse como útil solamente a efectos de exponer el comportamiento del sistema, y no como requisitos verificables del diseño.

A este respecto, los propósitos del análisis y el diseño son bastante diferentes. En el análisis, se busca modelar el mundo identificando las clases y los ob-

jetos (y sus papeles, responsabilidades y colaboraciones) que forman el vocabulario del dominio del problema. En el diseño, se inventan los artefactos que proporcionan el comportamiento que requiere el modelo del análisis. En este sentido, el análisis es la fase que primero reúne a los usuarios y los desarrolladores de un sistema, uniéndolos con un vocabulario común extraído del dominio del problema.

Centrándose en el comportamiento, se llega a identificar los puntos funcionales de un sistema. Los puntos funcionales, descritos en primer lugar por Allan Albrecht, denotan los comportamientos de un sistema observables exteriormente y comprobables [17]. Desde la perspectiva del usuario final, un punto funcional representa alguna actividad primaria de un sistema en respuesta a algún evento¹⁰. Los puntos funcionales denotan frecuentemente (pero no siempre) la correspondencia entre entradas y salidas, y representan así las transformaciones que el sistema realiza con su entorno. Desde la perspectiva del analista, un punto funcional representa un quantum preciso de comportamiento. En realidad, los puntos funcionales proporcionan una medida de la complejidad: cuanto mayor sea el número de puntos funcionales, más complejo es el sistema. Durante el análisis, se captura la semántica de los puntos funcionales de un sistema mediante los escenarios.

El análisis nunca está aislado. Durante esta fase no se espera proyectar una comprensión exhaustiva del comportamiento del sistema. Realmente, afirmamos que no es ni posible ni deseable llevar a cabo un análisis completo antes de permitir que el diseño comience. El propio acto de construir un sistema plantea cuestiones de comportamiento que ninguna cantidad razonable de análisis puede desvelar con eficiencia. Es suficiente completar un análisis de todos los comportamientos principales del sistema, con unos pocos comportamientos secundarios considerados también para asegurar que no se omite ningún patrón esencial de comportamiento.

Un análisis razonablemente completo y formal es esencial para servir a las necesidades de trazabilidad. La trazabilidad es en gran medida un problema de responsabilidad, a través de la cual se asegura que no se ha olvidado ningún punto funcional. La trazabilidad es esencial también para la gestión del riesgo. Según avanza el desarrollo en cualquier sistema no trivial, la administración del sistema tendrá que realizar difíciles negociaciones en la asignación de recursos, o en la resolución de alguna cuestión táctica desagradable. Si se tiene trazabilidad desde los puntos funcionales hasta la implantación, es mucho más fácil evaluar el impacto que tiene la alteración de la arquitectura cuando se presentan problemas tan enredados como éste.

Productos. DeChampeaux sugiere que la salida de un análisis es una descripción de la función del sistema, junto con declaraciones sobre la eficiencia y los recursos requeridos [19]. En desarrollo orientado a objetos se capturan esas

¹⁰ En el dominio de la gestión de sistemas de información, Dreger hace notar que un punto funcional representa una función de los asuntos del usuario final [18].

descripciones mediante escenarios, donde cada escenario denota algún punto funcional particular. Se usan escenarios principales para ilustrar los comportamientos clave, y los escenarios secundarios para mostrar comportamiento que se da bajo condiciones excepcionales.

Como se describió en capítulos anteriores, se usan técnicas de fichas CRC para realizar narraciones de sucesos en los escenarios, y se usan entonces diagramas de objetos para ilustrar la semántica de cada escenario con mayor precisión. Tales diagramas deben mostrar los objetos que colaboran para conseguir la función, así como el proceso de colaboración (es decir, la forma bien ordenada en la que los objetos interactúan pasándose mensajes). Además de los diagramas de objetos, se incluirán también diagramas de clases (para mostrar las asociaciones entre las clases del objeto) y máquinas de estados finitos (para mostrar el ciclo de vida de ciertos objetos importantes).

Con frecuencia, estos productos del análisis se reunirán en un documento formal de análisis de requisitos, que establece los requisitos del comportamiento del sistema tal como los ilustran los diagramas, más un análisis de todos los aspectos no relacionados con el comportamiento, como la eficiencia, fiabilidad, seguridad y transportabilidad [20].

Un producto secundario del análisis es una evaluación de los riesgos que identifica las áreas conocidas de riesgo técnico que pueden impactar en el proceso de diseño. El enfrentarse pronto a la presencia de riesgos en el proceso de desarrollo hace mucho más fácil el realizar concesiones arquitectónicas pragmáticas en fases posteriores de ese proceso de desarrollo.

Actividades. Hay dos actividades principales asociadas al análisis: análisis del dominio y planificación del escenario.

Como se describió en el Capítulo 4, el análisis de dominios busca identificar las clases y objetos que son comunes a un dominio de problema particular. Antes de ponerse a implantar un sistema completamente nuevo, frecuentemente es una sabia medida el estudiar otros existentes. De este modo se obtienen beneficios de la experiencia de otros proyectos en los que hubo que tomar decisiones de desarrollo parecidas. En el mejor caso, los resultados de un análisis de dominios pueden llevar a descubrir que no se necesita desarrollar ningún software nuevo, sino que se pueden reutilizar o adaptar marcos de referencia existentes.

La planificación de escenarios es la actividad central del análisis. Es interesante el hecho de que parece darse una confluencia de pensamiento sobre esta actividad entre otros diseñadores de metodologías, especialmente Rubin y Goldberg, Adams, Wirfs-Brock, Coad y Jacobson. Se refleja a continuación un orden típico de los eventos para esta actividad:

- Identificar todos los puntos funcionales principales del sistema y, si es posible, reunirlos en grupos de comportamientos relacionados funcionalmente. Considerar también la agrupación de acuerdo con jerarquías de

funciones, en las que ciertas funciones de alto nivel se construyen sobre otras más primitivas.

- Para cada conjunto interesante de puntos funcionales, realizar una narración de sucesos de un escenario, usando las técnicas de casos de uso y análisis de comportamientos descritas en el Capítulo 4¹¹. Las técnicas de fichas CRC son efectivas al realizar discusiones sobre cada escenario. Cuando se va aclarando la semántica de cada escenario, documentar con diagramas de objetos que ilustren los objetos que inician o contribuyen al comportamiento, y que colaboran para llevar a cabo las actividades del escenario. Incluir un guión que muestre los eventos que disparan el escenario y el orden resultante de las acciones. Además, documentar cualquier cuestión de suposiciones, restricciones o eficiencia para cada escenario [21].
- Según se necesite, generar escenarios secundarios que ilustren comportamiento bajo condiciones excepcionales.
- Donde el ciclo de vida de ciertos objetos sea importante o esencial para un escenario, desarrollar una máquina de estados finitos para la clase de los objetos.
- Recolectar patrones entre escenarios, y expresar estos patrones en términos de escenarios más abstractos y generalizados, o en términos de diagramas de clases que muestren las asociaciones entre abstracciones clave.
- Actualizar el diccionario de datos en evolución para incluir las nuevas clases y objetos identificados para cada escenario, junto con sus papeles y responsabilidades.

Como se describe posteriormente en el siguiente capítulo, la planificación de escenarios la realizan los analistas, en conjunción con el experto del dominio y el arquitecto. Adicionalmente, debería participar en la planificación de escenarios el personal de control de calidad, ya que los escenarios representan comportamientos que pueden someterse a pruebas. El involucrar pronto en el proceso a personal de control de calidad ayuda a institucionalizar un compromiso con esta. El involucrar a otros miembros del equipo de desarrollo durante la planificación de escenarios es también una vía efectiva para que se interesen en el proceso de desarrollo, y para alentar su comprensión de la visión del sistema.

Hitos y medidas. Se completa con éxito esta fase cuando se han desarrollado y rubricado escenarios para todos los comportamientos fundamentales del sistema. Con *rubricado* quiere decirse que los productos resultantes del análisis han sido validados por el experto del dominio, usuario final, analista y arquitecto; con *fundamental* se hace referencia a comportamientos que son centrales al propósito de la aplicación. Una vez más, ni se espera ni se desea un análisis completo. Es suficiente que se consideren sólo los comportamientos primarios y algunos secundarios.

¹¹ Jacobson [22] y Rubin y Goldberg [23] ofrecen amplios tratamientos de este tema.

Las medidas de bondad incluyen completud y simplicidad. Un buen análisis cubrirá todas las actividades primordiales y un conjunto estadísticamente interesante de las secundarias. Un buen análisis realizará también recorridos de todos los escenarios importantes estratégicamente, de forma que ayude a comunicar una visión del sistema a todo el equipo de desarrollo. Además, un buen análisis descubrirá también patrones de comportamiento, produciendo una estructura de clases simple que explote todo lo que es común entre escenarios diferentes.

Otro hito importante del análisis es la entrega de una estimación del riesgo, que ayuda al equipo a manejar futuros intercambios entre estrategia y táctica.

Diseño

Propósito. El propósito del diseño es crear una arquitectura para la implantación que va a desplegarse, y establecer las políticas tácticas comunes que deben utilizarse por parte de elementos dispares del sistema. Se comienza el proceso de diseño tan pronto como se tenga un modelo razonablemente completo del comportamiento del sistema. Es importante evitar diseños prematuros, en los que el desarrollo comienza antes de que el análisis alcance su término. Es igualmente importante evitar un diseño retrasado, en el que la organización se afana en intentar completar un modelo de análisis perfecto (y por tanto inalcanzable)¹².

Productos. Hay dos productos principales del diseño: una descripción de la arquitectura y descripciones de políticas tácticas comunes.

Se puede describir una arquitectura mediante diagramas, así como mediante versiones arquitectónicas del sistema. Como se describió en capítulos anteriores, la arquitectura de un sistema orientado a objetos abarca su estructura de clases y objetos, y así pueden usarse diagramas de clases y objetos para mostrar estas organizaciones estratégicas. A nivel arquitectónico, es más importante mostrar el agrupamiento de clases en categorías de clases (para la arquitectura lógica) y el agrupamiento de módulos en subsistemas (para la arquitectura física). Pueden entregarse estos diagramas como parte de un documento de arquitectura formal, que debería revisarse con el equipo completo y actualizarse a medida que evoluciona la arquitectura.

Se usan versiones arquitectónicas como manifestaciones tangibles del propio diseño de la arquitectura. Una versión arquitectónica denota una sección vertical a través de la arquitectura completa, capturando la semántica importante (aunque incompleta) de todas las categorías y subsistemas importantes. Una versión arquitectónica debería ser ejecutable, permitiendo así que la arquitectura se instrumentase, estudiase y evaluase con precisión. Como se discutirá

¹² Esta situación se diagnostica habitualmente como parálisis del análisis.

en la próxima sección, estas versiones arquitectónicas llegan a ser el fundamento del sistema de producción que se despliega.

Las políticas tácticas comunes incluyen a los mecanismos localizados que aparecen por todo el sistema. Abarcan artefactos de diseño como políticas de detección y gestión de errores, gestión de memoria, gestión de almacenamiento de datos y enfoques generalizados del control. Es importante diseñar explícitamente estas políticas; de otro modo se verá cómo los desarrolladores inventan soluciones *ad hoc* para problemas comunes, arruinando así la arquitectura estratégica por una «putrefacción» del software.

Se capturan las descripciones de políticas comunes mediante escenarios y versiones ejecutables de cada mecanismo.

Actividades. Existen tres actividades asociadas con el diseño: planificación arquitectónica, diseño táctico y planificación de versiones.

La planificación arquitectónica conlleva el proyectar las capas y particiones del sistema completo. Abarca una descomposición lógica, que representa un agrupamiento de clases, así como una descomposición física, que representa un agrupamiento de módulos y la asignación de funciones a diferentes procesadores. Un orden típico de los eventos para esta actividad es como sigue:

- Considerar el agrupamiento de puntos funcionales partiendo de los productos del análisis, y asignar éstos a capas y particiones de la arquitectura. Las funciones que se construyen sobre otras deberían estar en capas diferentes; las funciones que colaboran para producir comportamientos de un nivel de abstracción similar deberían estar en particiones que representan servicios análogos.
- Validar la arquitectura creando una versión ejecutable que satisfaga parcialmente la semántica de unos pocos escenarios interesantes del sistema, tal como se derivan del análisis.
- Instrumentar esa arquitectura y evaluar sus puntos débiles y fuertes. Identificar el riesgo de cada interfaz arquitectónico clave de modo que los recursos puedan asignarse significativamente cuando comience la evolución.

El centro de atención de la planificación arquitectónica es crear en momentos muy tempranos del ciclo de vida un marco de referencia de la aplicación específico del dominio que se pueda refinar sucesivamente.

El diseño táctico conlleva la toma de decisiones sobre la gran cantidad de políticas comunes. Como se describe más arriba, un diseño táctico pobre puede arruinar incluso la arquitectura más profunda, y así se mitiga este riesgo identificando explícitamente políticas tácticas y estableciendo incentivos para la adhesión a estas políticas. Un orden típico de los eventos para esta actividad es como sigue:

- En relación al dominio de aplicación, enumerar las políticas comunes que deben seguir elementos dispares de la arquitectura. Algunas de tales polí-

ticas son fundamentales, lo que significa que afrontan cuestiones independientes del dominio, como gestión de memoria, manejo de errores, etc. Otras políticas son específicas del dominio, e incluyen pautas y mecanismos que son afines a ese dominio, como políticas de control en sistemas en tiempo real, o gestión de transacciones y bases de datos en sistemas de información.

- Para cada política común, desarrollar un escenario que describa la semántica de esa política. Además, capturar su semántica en forma de un prototipo ejecutable que pueda instrumentarse y refinarse.
- Documentar cada política y efectuar un recorrido parejo, para difundir su visión arquitectónica.

La planificación de versiones fija el teatro en el que se desenvuelve la evolución arquitectónica. Tomando los puntos funcionales requeridos y la evaluación de riesgos generados durante el análisis, la planificación de versiones sirve para identificar una serie controlada de versiones arquitectónicas, creciendo cada una en su funcionalidad, abarcando finalmente los requerimientos del sistema de producción completo. Un orden típico de los eventos sería:

- Dados los escenarios identificados durante el análisis, organizarlos en orden de comportamientos fundamentales a periféricos. Dando prioridad a los escenarios que mejor puedan completarse con un equipo que incluya personal experto del dominio, personal de análisis, de arquitectura y de control de calidad.
- Asignar los puntos funcionales mencionados a una serie de versiones arquitectónicas cuyo producto final representa el sistema de producción.
- Ajustar las metas y planes de esta corriente de versiones de forma que las fechas de entrega estén lo bastante separadas como para permitir un tiempo de desarrollo adecuado, y como para que las versiones estén sincronizadas con otras actividades de desarrollo, como documentación y pruebas de campo.
- Comenzar la planificación de tareas, en la que se identifica una estructura de división de trabajo, y se identifican los recursos de desarrollo que son necesarios para conseguir cada versión arquitectónica.

Una planificación de versiones normal por productos es un plan de desarrollo formal, que identifica la sucesión de versiones arquitectónicas, las tareas de equipo y las tasas de riesgo.

Hitos y medidas. Se completa esta fase con éxito cuando se ha validado la arquitectura mediante un prototipo y mediante revisión formal. Además, hay que haber aprobado el diseño de todas las políticas tácticas importantes, y hay que tener un plan para sucesivas versiones.

La medida principal de bondad es la simplicidad. Una buena arquitectura es aquella que incorpora las características de los sistemas complejos organizados, como se describió en el Capítulo 1.

Los principales beneficios de esta actividad es la identificación temprana de defectos de arquitectura y el establecimiento de políticas comunes que produzcan una arquitectura simple.

Evolución

Propósito. El propósito de la fase evolutiva es aumentar y cambiar la implantación mediante refino sucesivo, lo que conduce en última instancia al sistema en producción.

La evolución de una arquitectura es, en gran parte, cuestión de intentar satisfacer una serie de restricciones que compiten, incluyendo funcionalidad, tiempo y espacio: siempre se está limitado por la restricción mayor. Por ejemplo, si el peso del computador es un factor crítico (como en el diseño de vehículos espaciales), entonces hay que considerar el peso de los chips individuales de memoria, y la cantidad de memoria que permiten los límites admisibles de peso limita a su vez el tamaño del programa que puede cargarse. Si se relaja una restricción concreta, pasarán a ser posibles otras alternativas de diseño; refúrcese cualquier restricción, y ciertos diseños se vuelven intratables. Desplegando la implantación de un sistema de software en lugar de adoptar un enfoque más monolítico para el desarrollo, puede identificarse qué restricciones son realmente importantes y cuáles son ilusiones. Por esta razón, el desarrollo evolutivo se centra en diseñar primero por funcionalidad y después por eficacia local. En fases iniciales del diseño, es habitual que no se sepa suficiente para comprender en qué parte del sistema surgirán cuellos de botella para el rendimiento. Analizando el comportamiento de versiones incrementales mediante histogramas u otras técnicas similares, el equipo de desarrollo puede comprender mejor cómo sintonizar el sistema a lo largo del tiempo.

La evolución es por tanto esencialmente el proceso de desarrollo de productos. Como observa Andert, el diseño «es un momento de innovación, mejora y libertad sin restricciones para modificar código con el fin de conseguir los objetivos del producto. La producción es un proceso metodológico controlado de elevar la calidad del producto hasta el punto en que este producto pueda expeditirse» [24].

Pages-Jones sugiere una serie de ventajas para este tipo de desarrollo incremental:

- «Se proporciona una importante realimentación hacia los usuarios cuando esta es más necesaria, más útil y más significativa.
- Los usuarios pueden utilizar varias versiones-esqueleto del sistema que les permitan realizar una transición suave de su viejo sistema a su nuevo sistema.
- Es menos probable que el proyecto sea cortado si cae tras las fechas previstas.
- Los interfaces principales del sistema se prueban primero y con más frecuencia.

- Los recursos de pruebas se distribuyen de manera más homogénea.
- Los implantadores pueden ver pronto resultados de un sistema en funcionamiento, con lo que su moral aumenta.
- Si hay poco tiempo, la codificación y la prueba pueden comenzar antes de que finalice el diseño» [25].

Productos. El producto principal de la evolución es una corriente de versiones ejecutables que representan sucesivos refinamientos a la versión inicial de la arquitectura. Los productos secundarios incluyen prototipos de comportamiento que se usan para explorar diseños alternativos o para analizar mejor los rincones oscuros de la funcionalidad del sistema.

Estas versiones ejecutables siguen el plan establecido en la actividad anterior de planificación de versiones. Para un proyecto de tamaño modesto que implique de 12 a 18 meses de tiempo total de desarrollo, esto podría significar una versión cada dos o tres meses. Para proyectos más complejos que requieran un esfuerzo de desarrollo mucho mayor, esto podría significar una versión cada seis meses, aproximadamente. Las previsiones de versiones más extendidas son sospechosas, porque no fuerzan a rematar el microproceso, y pueden ocultar áreas de riesgo que se están ignorando, ya sea intencionadamente o no.

¿A quién se le entrega una versión ejecutable? En momentos iniciales del proceso de desarrollo, las versiones ejecutables principales se remiten del equipo de desarrollo al personal de control de calidad, que puede comenzar a probar la versión frente a los escenarios establecidos durante el análisis, y a recoger información sobre la completitud (plenitud), corrección y robustez de esa versión. Esta recolección precoz de datos ayuda a identificar problemas de calidad, que se solucionan más fácilmente durante la evolución de la versión siguiente. En momentos posteriores del proceso de desarrollo, las versiones ejecutables se remiten a usuarios finales selectos (los clientes alfa y beta) de forma controlada. Por controlada se entiende que el equipo de desarrollo fija cuidadosamente las expectativas para cada versión, e identifica aspectos que desea evaluar.

Las necesidades del microproceso dictan que se completarán muchas más versiones internas al equipo de desarrollo, de las que sólo se remiten unas pocas versiones ejecutables a los agentes externos. Estas versiones internas representan una especie de integración continua del sistema, y existen para obligar al microproceso a rematar y finalizar sus actividades.

Entre cada sucesiva versión externa, el equipo de desarrollo también puede producir prototipos de comportamiento. Un prototipo de comportamiento sirve para explorar algún elemento aislado del sistema, tal como un nuevo algoritmo, un modelo de interfaz de usuario o un esquema de base de datos. Su propósito es la exploración rápida de alternativas de diseño, de forma que se puedan resolver pronto áreas de riesgo sin poner en peligro las versiones de producción. Los prototipos de comportamiento son verdaderamente prototipos, y por tanto están destinados a que se los deseche después de que han servido a sus propósitos. Típicamente, un equipo utilizará prototipos de comportamiento para realizar narraciones sobre la semántica de interfaces de usuario y presentárselos a

los usuarios finales para una realimentación temprana, o para sopesar la eficacia de cara a la implantación de políticas tácticas.

Por implicación, la documentación del sistema evoluciona junto con las versiones arquitectónicas. En vez de tratar la producción de documentación como un hito fundamental, suele ser mejor considerarla un artefacto del proceso evolucionario, natural y generado semiautomáticamente.

Actividades. Hay dos actividades que se asocian con la evolución: la aplicación del microproceso y la gestión de cambios.

El trabajo desempeñado entre las versiones ejecutables representa un proceso de desarrollo comprimido, y por tanto es esencialmente una iteración¹³ del microproceso. Esta actividad comienza con un análisis de los requisitos para la siguiente versión, procede al diseño de una arquitectura y continúa con la invención de las clases y objetos necesarios para implantar este diseño. Un orden típico de los eventos para esta actividad es como sigue:

- Identificar los puntos funcionales que debe satisfacer esta versión ejecutable, así como las áreas de mayor riesgo, especialmente aquellas identificadas mediante la evaluación de la versión anterior.
- Asignar tareas al equipo para llevar a cabo esta versión, e iniciar una iteración del microproceso. Supervisar el microproceso estableciendo revisiones apropiadas del diseño, y dirigiendo respecto a hitos intermedios cuyo cumplimiento lleve del orden de pocos días o una semana.
- Según se necesite para comprender la semántica del comportamiento deseado del sistema, asignar desarrolladores para producir prototipos de comportamiento. Establecer criterios claros para los objetivos y finalización de cada prototipo. Sobre la finalización, decidir sobre un enfoque para integrar los resultados del esfuerzo de construcción de prototipos en las versiones subsiguientes.
- Forzar la finalización del microproceso integrando y liberando la versión ejecutable.

Tras cada versión, es importante revisar el plan original de versiones, y ajustar los requisitos y previsiones para versiones posteriores según sea necesario. Frecuentemente, esto implica pequeños ajustes de fechas, o migraciones de funcionalidad de una versión a otra.

La gestión de cambios existe en reconocimiento a la naturaleza incremental e iterativa de los sistemas orientados a objetos. Es tentador el permitir cambios indisciplinados en las jerarquías de clases, protocolos de las clases o mecanismos, pero el cambio sin restricciones tiende a corromper la arquitectura estratégica y conduce al desgranamiento del equipo de desarrollo.

En la práctica, se encuentra que se esperan los siguientes tipos de cambio durante la evolución de un sistema:

¹³ El autor utiliza el término *spin*, cuyo significado es «vuelta», para referirse a un periodo completo de actividad del microproceso. En castellano se ha juzgado más explícito un término como iteración, que él mismo utiliza en otras ocasiones. (N. del T.)

- Añadir una nueva clase o una nueva colaboración entre clases.
- Cambiar la implantación de una clase.
- Cambiar la representación de una clase.
- Reorganizar la estructura de clases.
- Cambiar el interfaz de una clase.

Cada tipo de cambio se produce por razones diferentes, y cada uno tiene un coste distinto.

Un desarrollador añadirá nuevas clases cuando se descubran nuevas abstracciones clave o se inventen nuevos mecanismos. El coste de realizar tales cambios no suele tener consecuencias en términos de recursos de computación y sobrecarga en la gestión. Cuando se añade una nueva clase, hay que considerar en qué lugar encaja en la estructura de clases existente. Cuando se inventa una nueva colaboración debería efectuarse un pequeño análisis del dominio para ver si es en realidad una colaboración de un patrón de ellas.

El cambio de implantación de una clase generalmente tampoco resulta costoso. En el diseño orientado a objetos, se suele crear primero el interfaz de una clase y después se aborda su implantación (es decir, la implantación de sus funciones miembro). Una vez que el interfaz se estabiliza en un grado razonable, se puede elegir una representación para esa clase y completar la implantación de sus métodos. La implantación de un método particular puede cambiarse más tarde, normalmente para localizar un error o mejorar su eficacia. También se podría cambiar la implantación de un método para aprovechar métodos nuevos definidos en una superclase existente o añadida. En cualquier caso, el cambio de la implantación de un método no suele ser costoso, especialmente si se ha encapsulado previamente la implantación de la clase.

En una línea similar, se podría alterar la representación de una clase (en C++, los miembros `protected` y `private` de una clase). Normalmente, esto se hace para que las instancias de la clase sean más eficientes en cuanto a espacio o para crear métodos más eficientes respecto al tiempo. Si la representación de la clase está encapsulada, cosa posible en Smalltalk, C++, CLOS y Ada, un cambio en la representación no romperá lógicamente el modo en que los clientes interaccionan con las instancias de esa clase (a menos, por supuesto, que esta nueva representación no ofrezca el comportamiento que se espera de la clase). Por otra parte, si la representación de la clase no está encapsulada, lo que también es posible en cualquier lenguaje, un cambio en la representación es mucho más peligroso, porque los clientes pueden haber sido escritos dependiendo de una representación particular. Esto es especialmente cierto en el caso de las subclases: el cambio en la representación de una superclase afecta a la representación de todas sus subclases. En cualquier caso, el cambio de la representación de una clase incurre en un coste: hay que recomilar su interfaz, su implantación y todos sus clientes (es decir, sus subclases e instancias), todos los clientes de sus clientes, y así sucesivamente.

Es habitual reorganizar la estructura de clases de un sistema, aunque menos frecuente que los otros tipos de cambio que se han mencionado. Como obser-

van Stefk y Bobrow, «los programadores crean a menudo nuevas clases y reorganizan sus clases a medida que comprenden las oportunidades de factorizar partes de sus programas» [26]. La reorganización de una estructura de clases suele tomar la forma de cambios en las relaciones de herencia, añadido de nuevas clases abstractas y desplazamiento de las responsabilidades e implantación de métodos comunes a clases más altas en la estructura de clases. En la práctica, la reorganización de la estructura de clases de un sistema suele suceder con frecuencia al principio, y después se estabiliza con el tiempo a medida que sus desarrolladores comprenden mejor cómo trabajan juntas todas las abstracciones clave. Debe promoverse la reorganización de la estructura de clases en etapas iniciales del diseño porque puede resultar en una gran economía de expresión, lo que significa que se tendrán implantaciones más pequeñas y menos clases para comprender y mantener. Sin embargo, la reorganización de la estructura de clases no se produce sin ningún coste. Típicamente, el cambio de la ubicación de una clase alta en la jerarquía deja obsoletas todas las clases de debajo y exige su recompilación (y por tanto la recompilación de las clases que dependen de estas, y así sucesivamente).

Un tipo de cambio igualmente importante que ocurre durante la evolución de un sistema es un cambio del interfaz de una clase. Un desarrollador suele cambiar el interfaz de una clase ya sea para añadir comportamiento nuevo, para satisfacer la semántica de algún nuevo papel para sus objetos, o para añadir una operación que siempre formó parte de la abstracción pero que inicialmente no se exportó y que ahora necesita algún cliente. En la práctica, el uso de los heurísticos para construir clases de calidad que se discutieron en el Capítulo 3 (específicamente los conceptos de construir interfaces primitivos, suficientes y completos) reduce la probabilidad de tales cambios. Sin embargo, nuestra experiencia es que esos cambios son inevitables. Nunca hemos escrito una clase no trivial cuyo interfaz fuese completamente correcto a la primera.

Es raro pero no impensable el eliminar un método existente; normalmente sólo se hace para encapsular mejor una abstracción. Con más frecuencia, se añade un nuevo método o se redefine* un método definido en alguna superclase. En los tres casos, el cambio es costoso, porque afecta lógicamente a todos los clientes, haciéndolos obsoletos y forzando a su recompilación. Afortunadamente, estos últimos tipos de cambio —añadir y redefinir métodos— son compatibles en sentido creciente. De hecho, se encuentra en la práctica que la mayoría de los cambios en los interfaces que se realizan sobre clases bien definidas durante la evolución de un sistema son compatibles en sentido creciente. Esto hace posible aplicar sofisticadas tecnologías de compiladores, tales como la compilación incremental, para reducir el impacto de estos cambios. La compilación incremental permite recompilar declaraciones y sentencias individuales de una en una, en vez de hacerlo con módulos enteros, lo que significa que la recompilación de la mayoría de los clientes puede optimizarse.

* Se ha decidido traducir «override» por *redefinir*. También se utilizan los términos «sustituir», «anular», «reemplazar». (N. del T.)

¿Por qué el coste de recompilación llega a ser un problema? Para sistemas pequeños, no hay tal problema, porque recompilar un programa completo podría llevar sólo unos pocos minutos. Sin embargo, para grandes sistemas, la cosa es muy distinta. Recompilar un programa de cientos de miles de líneas podría llevar incluso medio día de tiempo de computador. ¿Puede el lector imaginarse el realizar un cambio en el software del sistema de computadores de un barco y decir después al capitán que no puede hacerse a la mar porque uno no ha terminado todavía de recompilar? En el caso extremo, los costes de recompilación pueden ser tan altos como para que los desarrolladores se inhiban de hacer cambios que darían lugar a mejoras razonables. La recompilación es una cuestión especialmente importante en lenguajes de programación orientados a objetos, porque la herencia introduce dependencias de compilación [27]. Para lenguajes de programación orientados a objetos con comprobación estricta de tipos, los costes de recompilación pueden ser aún mayores; en tales lenguajes, se intercambia tiempo de compilación por seguridad.

Los tipos de cambio que se han discutido hasta aquí son los más fáciles: el mayor riesgo es un cambio arquitectónico drástico, que puede hundir un proyecto. Frecuentemente, este cambio es el resultado de ingenieros brillantes con demasiadas buenas ideas [28].

Hitos y medidas. Se completa con éxito esta fase cuando la funcionalidad y calidad de las versiones son suficientes para expedir el producto. Las versiones de formas intermedias ejecutables son los hitos principales que se usan para gestionar el desarrollo del producto final. La medida principal de bondad es por tanto hasta qué grado se satisfacen los puntos funcionales asignados a cada versión intermedia, y en qué medida se ha confluído con las previsiones temporales establecidas durante la planificación de versiones.

Otras dos medidas de adecuación esenciales son el seguimiento de tasas de descubrimiento de defectos, y la medida de la tasa de cambio de interfaces arquitectónicas y políticas tácticas claves.

Brevemente, la tasa de descubrimiento de defectos es una medida de la rapidez con la que se detectan nuevos errores [29]. Invirtiendo pronto en control de calidad en el proceso de desarrollo, es posible establecer medidas de la calidad para cada versión, que el equipo directivo puede usar para identificar áreas de riesgo y también para calibrar al equipo de desarrollo. Tras cada versión, la tasa de descubrimiento de errores suele ondular. Una tasa de descubrimiento de defectos estancada suele ser síntoma de errores sin descubrir. Una tasa de descubrimiento de defectos desproporcionada es una indicación de que la arquitectura aún no se ha estabilizado, o de que hay nuevos elementos en una versión dada que se han diseñado o implantado incorrectamente. Estas medidas se usan para ajustar los centros de interés de las versiones subsiguientes.

La medición de la tasa de cambios de interfaces arquitectónicas y políticas tácticas es la medida principal de la estabilidad de la arquitectura [30]. Durante la evolución es de esperar que se den cambios localizados, pero si las tramas de herencias o los límites entre categorías de clases o subsistemas se están cam-

biando con frecuencia, esto es síntoma de problemas arquitectónicos, y por tanto debería reconocerse como un área de riesgo cuando se planifique la siguiente versión.

Mantenimiento

Productos. El mantenimiento es la actividad de gestionar la evolución postventa. Esta fase es en gran medida una continuación de la fase anterior, excepto en que la innovación arquitectónica es menos preocupante. En lugar de eso, se realizan cambios más localizados al sistema a medida que se añaden nuevos requisitos y se eliminan errores persistentes.

Lehman y Belady han realizado una serie de convincentes observaciones respecto a la maduración de un sistema de software ya desplegado:

- Un programa que se usa en un entorno del mundo real necesariamente ha de cambiar o bien volverse menos y menos útil en ese entorno (ley del cambio continuo).
- A medida que cambia un programa en evolución, su estructura se vuelve más compleja a menos que se realicen esfuerzos activos para evitar este fenómeno (ley de la complejidad creciente)» [31].

Se distingue la conservación de un sistema de software de su mantenimiento. Durante el mantenimiento, a los desarrolladores se les pedirá realizar continuas mejoras a un sistema existente; estos desarrolladores a menudo no son los desarrolladores originales, sino un grupo diferente de personas. La conservación, por contra, conlleva el uso de recursos de desarrollo excesivos para apuntalar un sistema que envejece y que con frecuencia tiene una arquitectura diseñada deficientemente, y es por tanto difícil de comprender y modificar. Hay que tomar una decisión económica: si el coste de posesión de este software es mayor que el coste de desarrollar un sistema nuevo, la línea de actuación más piadosa es poner a ese sistema anciano metafóricamente «a pastar» o, si las condiciones lo dictan así, abandonarlo o tirarlo.

Productos. Ya que el mantenimiento es en cierto sentido la evolución continua de un sistema, sus productos son similares a los de la fase previa. Además, el mantenimiento implica gestionar una lista-guía de nuevas tareas. Inmediatamente, tras la entrega del sistema en producción, sus desarrolladores y usuarios finales ya tendrán probablemente una serie de mejoras o modificaciones que les gustaría realizar en las versiones de producción siguientes, y que no se efectuaron en el producto inicial por razones económicas. Además, a medida que hay más usuarios rodando el sistema, se descubrirán nuevos errores y patrones de uso que el control de calidad no pudo anticipar¹⁴. Una lista-guía sirve

¹⁴ Los usuarios son asombrosamente creativos cuando se trata de emplear un sistema de formas inesperadas.

como vehículo para recoger errores y requisitos de mejora, de forma que pueda dárseles prioridad para futuras versiones.

Actividades. El mantenimiento involucra actividades que son ligeramente diferentes de las requeridas durante la evolución de un sistema. Especialmente si se ha hecho un buen trabajo en la arquitectura original, el añadido de funcionalidad nueva o la modificación de algún comportamiento existente llegará de forma natural.

Además de las actividades usuales de la evolución, el mantenimiento implica una actividad de planificación que da prioridad a las tareas de la lista-guion. Un orden típico de los eventos para esta actividad es el siguiente:

- Asignar prioridad a las peticiones de mejoras básicas o informes de errores que denotan problemas del sistema, y estimar el coste de volver a desarrollarlo.
- Establecer una colección significativa de estos cambios y tratarlos como puntos funcionales para la siguiente evolución.
- Si los recursos lo permiten, añadir mejoras menos intensas y más localizadas (las llamadas «frutas que cuelgan bajo») a la siguiente versión.
- Gestionar la siguiente versión evolucionaria.

Hitos y medidas. Los hitos del mantenimiento implican versiones de producción continuadas, así como versiones intermedias de depuración de errores.

Se sabe que aún se está efectuando el mantenimiento de un sistema si la arquitectura sigue siendo flexible al cambio; se sabe que se ha entrado en la etapa de conservación cuando la respuesta a nuevas mejoras comienza a requerir recursos de desarrollo excesivos.

Resumen

- Los proyectos de éxito suelen caracterizarse por una visión arquitectónica fuerte y por un ciclo de vida del desarrollo bien dirigido, iterativo e incremental.
- No es posible un proceso de desarrollo completamente racional, pero puede emularse reconciliando el micro y el macroproceso del desarrollo.
- El microproceso del desarrollo orientado a objetos está dirigido por la corriente de escenarios y productos arquitectónicos que emergen del macroproceso; el microproceso representa actividades diarias del equipo de desarrollo.
- El primer paso del microproceso implica la identificación de clases y objetos a un nivel dado de abstracción; las actividades principales incluyen descubrimiento e invención.

- El segundo paso del microproceso implica la identificación de la semántica de estas clases y objetos; las actividades primarias incluyen la escenificación de narraciones de sucesos, diseño de clases aisladas y recolección de patrones.
- El tercer paso del microproceso implica la identificación de las relaciones entre estas clases y objetos; las actividades principales incluyen la especificación de asociaciones, la identificación de colaboraciones y el refinamiento de asociaciones.
- El cuarto paso del microproceso implica la implantación de estas clases y objetos; la actividad principal es la selección de estructuras de datos y algoritmos.
- El macroproceso del desarrollo orientado a objetos sirve como el marco de referencia que controla el microproceso y define una serie de productos medibles y actividades para gestionar el riesgo.
- El primer paso del macroproceso es la conceptualización, que establece los requisitos esenciales del sistema; su actividad sirve como una prueba de concepto, y por tanto es bastante incontrolada, de forma que permita innovar sin restricciones.
- El segundo paso del macroproceso es el diseño, que crea una arquitectura para la implantación y establece políticas tácticas comunes; las actividades principales incluyen planificación arquitectónica, diseño táctico y planificación de versiones.
- El cuarto paso del macroproceso es la evolución, que utiliza refinamiento sucesivo para llevar finalmente al sistema en producción; las actividades principales incluyen la aplicación del microproceso y la gestión de cambios.
- El quinto paso en el macroproceso es el mantenimiento, que es esencialmente la gestión de la evolución postventa (postentrega); las actividades principales son similares a las del cuarto paso, con la adición de la gestión de una lista-guión.

Lecturas recomendadas

Una forma inicial del proceso descrito en este capítulo fue documentada por primera vez por Booch [F 1982]. Berard trabajó después sobre esta documentación en [F 1986]. Enfoques relacionados son GOOD (General Object-Oriented Design) de Seidewitz y Stark [F 1985, 1986, 1987], SOOD (Structured Object-Oriented Design) de Lockheed [C 1988], MOOD (Multiple-view Object Oriented Design) de Kerth [F 1988] y HOOD (Hierarchical Object-Oriented Design) de CISI Ingenierie y Matra para la European Space Station [F 1987]. Otros trabajos relacionados más recientes son Stroustrup [G 1991] y Microsoft [G 1992], que sugieren procesos sustancialmente similares.

Además de los trabajos citados en las lecturas recomendadas del Capítulo 2, otra serie de diseñadores de metodologías han propuesto procesos específicos del desarrollo ori-

tado a objetos, para los cuales la bibliografía proporciona un extenso conjunto de referencias. Algunas de las contribuciones más interesantes provienen de Alabios [F 1988], Boyd [F 1987], Buhr [F 1984], Cherry [F 1987, 1990], deChampeaux [F 1992], Felsinger [F 1987], Firesmith [F 1986, 1993], Hines y Unger [G 1986], Jacobson [F 1985], Jamsa [F 1984], Kadie [F 1986], Masiero y Germano [F 1988], Nielsen [F 1988], Nies [F 1986], Rajlich y Silva [F 1987] y Shumate [F 1987].

Pueden encontrarse comparaciones de diversos procesos de desarrollo orientado a objetos en Arnold [F 1991], Boehm—Davis y Ross [H 1984], deChampeaux [B 1991], Cribbs, Moon y Roe [F 1992], Fowler [F 1992], Kelly [F 1986], Mannino [F 1987], Song [F 1992] y Webster [F 1988]. Brookman [F 1991] y Fichman [F 1992] ofrecen una comparación de métodos estructurados y orientados a objetos.

Pueden encontrarse estudios empíricos de procesos de software en Curtis [H 1992] así como en el Software Process Workshop [H 1988]. Otra referencia interesante es Guindon [H 1987], que estudia los procesos preliminares utilizados por los desarrolladores en fases tempranas del proceso de desarrollo. Rechtin [H 1992] ofrece una guía práctica para el arquitecto de software que debe dirigir el proceso de desarrollo. Humphrey [H 1989] es la referencia seminal sobre madurez del proceso de software. Parnas [H 1986] es la referencia clásica sobre cómo emular tal proceso maduro.

Notas bibliográficas

- [1] Brooks, F. 1975. *The Mythical Man-Month*. Reading, MA: Addison-Wesley, p. 42.
- [2] Stroustrup, B. 1991. *The C+ Programming Language*, Second Edition. Reading, MA: Addison-Wesley.
- [3] Maccoby, M. December 1991. The Innovative Mind at Work. *IEEE Spectrum*, vol. 28(12).
- [4] Lammers, S. 1986. *Programmers at Work*. Redmond, WA: Microsoft Press.
- [5] Druke, M. 1989. Comunicación privada.
- [6] Jones, C. September 1984. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering*, vol. SE-10(5).
- [7] Humphrey, W. 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley, p. 5.
- [8] Curtis, B May 17, 1989 ... *But You Have to Understand, This Isn't the Way We Develop Software at Our Company*. MCC Technical Report Number STP-203-89. Austin, TX: Microelectronics and Computer Technology Corporation, p. x.
- [9] Parnas, D. and Clements, P. 1986. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering* vol. SE-12(2).
- [10] Boehm, B. August 1986. A Spiral Model of Software Development and Enhancement. *Software Engineering Notes* vol. 11(4), p. 22.
- [11] Stroustrup, B. 1991 *The C+ Programming Language*, Second Edition. Reading, MA: Addison-Wesley, p. 362.
- [12] Brownsword, L. 1989. Comunicación privada.
- [13] Stroustrup, p. 373.
- [14] Vonk, R. 1990. *Prototyping*. Englewood Cliffs, NJ: Prentice-Hall, p. 31.
- [15] Gilb, T. 1988. *Principles of Software Engineering Management*. Reading, Massachusetts Addison-Wesley, p. 92.

- [16] Mellor, S., Hecht, A., Tryon, D., and Hywari, W. September 1988. Object-Oriented Analysis: Theory and Practice, Course Notes, in *Object-Oriented Programming Systems, Languages, and Applications*. San Diego, CA: OOPSLA'88, p. 13.
- [17] Symons, C. 1988. Function Point Analysis: Difficulties and Improvements. *IEEE Transactions on Software Engineering* vol. (14)1.
- [18] Dreger, B. 1989. *Function Point Analysis*. Englewood Cliffs, New Jersey: Prentice Hall, p. 5.
- [19] DeChampeaux, D., Balzer, B., Bulman, D., Culver-Lozo, K., Jacobson, I., Mellor, S. *The Object-Oriented Software Development Process*. Vancouver, Canada: OOPSLA'92.
- [20] Davis, A. 1990. *Software Requirements: Analysis and Specification*. Englewood Cliffs, New Jersey: Prentice-Hall.
- [21] Rubin, K. 1993. Comunicación privada.
- [22] Jacobson, I., Christerson, M., Johsson, P., and Overgaard, G. 1992. *Object-Oriented Software Engineering*. Workingham, England: Addison-Wesley Publishing Company.
- [23] Rubin, K. and Goldberg, A. September 1992. *Object Behavior Analysis*. Communications of the ACM vol. 35(9).
- [24] Andert, G. 1992. Comunicación privada.
- [25] Page-Jones, M. 1988. *The Practical Guide to Structured System Design*. Englewood Cliffs, NJ: Yourdon Press, pp. 261-265.
- [26] Stefik, M. and Bobrow, D. Winter 1986. Object-Oriented Programming: Themes and Variations., *AI Magazine* vol. 6(4), p. 41.
- [27] Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall, p. 340.
- [28] Andert, G. 1993. Comunicación privada.
- [29] Walsh, J. *Preliminary Defect Data from the Iterative Development of a Large C++ Program*. Vancouver, Canada: OOPSLA'92.
- [30] Chmura, L., Norcio, A., and Wicinski, T. July 1990. Evaluating Software Design Processes by Analyzing Change Date Over Time. *IEEE Transactions on Software Engineering* vol. 16(7).
- [31] As quoted in Sommerville, I. 1989. *Software Engineering*. Third Edition Wokingham, England: Addison-Wesley, p. 546.

/

Aspectos pragmáticos

En la actualidad, el desarrollo de software sigue siendo una actividad enormemente laboriosa; en gran medida, la mejor manera de caracterizarlo todavía es como una industria de pequeños artesanos [1]. Un informe de Kishida, Teramoto, Torri y Urano pone de relieve que, incluso en Japón, la industria de software «aún confía principalmente en el enfoque informal de papel—y—lápiz en las primeras fases del desarrollo» [2].

Para agravar la situación, se da el hecho de que el diseño no es una ciencia exacta. Considérese el diseño de una base de datos compleja usando un modelo entidad-relación, uno de los fundamentos del diseño orientado a objetos. Como observa Hawryszkiewycz, «aunque esto suena como muy directo, en realidad conlleva una cierta cantidad de percepción personal de la importancia de diversos objetos de la empresa. El resultado es que el proceso de diseño no es determinista: diferentes diseñadores pueden producir diferentes modelos de empresa para la misma empresa» [3].

Se puede concluir razonablemente que no importa lo sofisticado que sea el método de desarrollo, y no importa lo bien fundamentadas que estén sus bases teóricas: no pueden ignorarse los aspectos prácticos del diseño de sistemas para el mundo real. Esto significa que hay que considerar buenas prácticas de gestión por lo que se refiere a temas como administración de personal, gestión de versiones y control de calidad. Para el técnico, éstos son temas extremadamente aburridos; para el ingeniero del software profesional, son realidades a las que hay que enfrentarse si uno quiere tener éxito al construir sistemas complejos de software. Así, este capítulo se centra en los aspectos prácticos del desarrollo orientado a objetos, y examina el impacto del modelo de objetos en diversas prácticas de gestión.

7.1. Gestión y planificación

En presencia de un ciclo de vida incremental e iterativo, es de importancia capital tener un liderazgo fuerte en el proyecto que gestione y dirija activamente las actividades del mismo. Son demasiados los proyectos que se descarrían por una falta de objetivos claros, y la presencia de un equipo gestor fuerte mitiga este problema.

Gestión del riesgo

En última instancia, la responsabilidad del director de desarrollo del software es gestionar riesgo tanto técnico como no técnico. Los riesgos técnicos en sistemas orientados a objetos incluyen problemas como la selección de tramas de herencias que ofrezcan el mejor compromiso entre su uso y su flexibilidad, o la elección de mecanismos que produzcan una eficacia aceptable a la vez que simplifican la arquitectura del sistema. Los riesgos no técnicos abarcan cuestiones como supervisar la entrega puntual de software por parte de una tercera persona que ejerce de vendedor, o gestionar la relación entre el cliente y el equipo de desarrollo, de forma que se facilite el descubrimiento de los requisitos reales del sistema durante el análisis.

Como se describió en el capítulo anterior, el microproceso del desarrollo orientado a objetos es inestable de forma innata, y requiere una dirección activa para forzar su conclusión. Afortunadamente, el macroproceso del desarrollo orientado a objetos se ha diseñado para llevar a esa finalización proporcionando una serie de productos tangibles que la dirección puede estudiar para averiguar la salud del proyecto, junto con controles que permiten a la dirección redirigir los recursos del equipo como sea necesario. El enfoque evolutivo que el macroproceso adopta para el desarrollo significa que hay oportunidades para identificar problemas en momentos tempranos del ciclo de vida y responder convenientemente a esos riesgos antes de que comprometan el éxito del proyecto.

Muchas de las prácticas básicas de la gestión del desarrollo del software, como la planificación de tareas y los recorridos de revisión, no se ven afectadas por la tecnología orientada a objetos. Lo que es diferente en la gestión de un proyecto orientado a objetos, sin embargo, es que las tareas que se planifican y los productos que se revisan son sutilmente diferentes que en los sistemas no orientados a objetos.

Planificación de tareas

En cualquier proyecto de tamaño modesto a grande, es razonable tener reuniones semanales del equipo para discutir el trabajo que se ha completado y las actividades para la semana siguiente. Es necesaria una frecuencia mínima de reunión para favorecer la comunicación entre los miembros del equipo; dema-

siadas reuniones destruyen la productividad, y de hecho son un signo de que el proyecto ha perdido su camino. El desarrollo de software orientado a objetos requiere que los desarrolladores individuales tengan «masas críticas» de tiempo no sujeto a planes para que puedan pensar, innovar y desarrollar, así como tener encuentros informales con otros miembros del equipo, según sea necesario, para discutir cuestiones técnicas detalladas. El equipo de gestión debe prever este tiempo no estructurado.

Tales reuniones proporcionan un vehículo simple pero efectivo para hacer planes bien ajustados en el microproceso, así como para conseguir preparación respecto a riesgos que aparecen en el horizonte. Estas reuniones pueden resultar en pequeños ajustes de las asignaciones de trabajo, de forma que se asegure un progreso firme: ningún proyecto puede permitirse que ninguno de sus desarrolladores esté sentado sin hacer nada mientras espera a que otros miembros del equipo estabilicen su parte de la arquitectura. Esto es especialmente cierto para sistemas orientados a objetos, en los que el diseño de clases y mecanismos impregna la arquitectura. El desarrollo puede llegar a detenerse si ciertas clases clave no están consolidadas.

A una escala mayor, la planificación de tareas implica planificar las entregas del macroproceso. Entre versiones evolucionarias, el equipo de dirección debe evaluar los riesgos inminentes del proyecto, orientar recursos de desarrollo como sea necesario para atacar estos riesgos¹, y después gestionar la siguiente iteración del microproceso que produce un sistema estable que satisface los escenarios que los planes requerían para esa versión. Si la planificación de tareas a este nivel falla, la mayoría de las veces es a causa de previsiones demasiado optimistas [4]. El desarrollo que se había considerado «simple cuestión de programación» se extiende a varios días de trabajo; los planes se van por la ventana cuando desarrolladores que trabajan en una parte del sistema asumen ciertos protocolos de otras partes, pero se han visto equivocados por la entrega de clases fabricadas incompleta o incorrectamente. Hay traiciones aún mayores: los planes pueden verse heridos de muerte por la aparición de problemas de funcionamiento o errores del compilador, que constituyen obstáculos que hay que rodear, muchas veces corrompiendo ciertas decisiones tácticas de diseño.

La clave para no caer en el arbitrio de planificar con demasiado optimismo es la calibración del equipo de desarrollo y sus herramientas. Típicamente, la planificación de tareas funciona como sigue. Primero, el equipo de gestión dirige las energías de un desarrollador a una parte específica del sistema, digamos, por ejemplo, el diseño de un conjunto de clases para servir de interfaz con una base de datos relacional. El desarrollador considera el alcance del esfuerzo, y vuelve con una estimación del tiempo que lleva completarlo, estimación en la que la dirección confía para planificar las actividades de otros desarrolladores. El problema es que estas estimaciones no siempre son fiables, porque suelen representar condiciones que se dan en el mejor caso posible. Un desarrollador podría declarar una semana de esfuerzo para una tarea, mientras otro podría pre-

¹ Gilb observa que «si uno no ataca activamente los riesgos, ellos atacarán activamente a uno» [5].

ver un mes. Cuando se hace de verdad el trabajo, podría llevarles a ambos tres semanas: el primero habría subestimado el esfuerzo (el problema habitual en muchos desarrolladores), y el segundo habría realizado estimaciones mucho más realistas (normalmente porque ha comprendido la diferencia entre tiempo real de trabajo y tiempo de calendario, que a menudo se llena con multitud de actividades no funcionales). Con el fin de desarrollar previsiones en las que el equipo pueda confiar, es por tanto necesario que el equipo de gestión idee factores multiplicativos para las estimaciones de cada desarrollador. Esto no significa que la dirección no confie en sus desarrolladores; es una simple aceptación del hecho de que la mayoría de los desarrolladores están centrados en cuestiones técnicas, no en cuestiones de planificación. La dirección debe ayudar a sus desarrolladores a aprender a planificar correctamente, una habilidad que sólo se adquiere mediante experiencia en el campo de batalla.

El proceso del desarrollo orientado a objetos ayuda explícitamente a desarrollar estos factores de calibración. Su ciclo de vida incremental e iterativo implica que hay muchos hitos intermedios que se establecen pronto en el proyecto, y que los gestores pueden usar para recoger datos sobre la forma en que cada desarrollador fija y cumple previsiones temporales. Esto significa que, a medida que avanza el desarrollo evolutivo, la dirección conseguirá con el tiempo una mejor comprensión de la productividad real de cada uno de sus desarrolladores, y los desarrolladores individuales pueden ganar experiencia en la estimación de su propio trabajo con mayor exactitud. La misma lección se aplica a las herramientas: con el énfasis puesto en la entrega pronta de versiones arquitectónicas, el proceso de desarrollo orientado a objetos incita al uso temprano de herramientas, lo que conduce a la identificación de sus limitaciones antes de que sea demasiado tarde para cambiar de rumbo.

Recorridos de inspección

Los recorridos de inspección son otra práctica bien establecida que todo equipo de desarrollo debería emplear. Al igual que con la planificación de tareas, el transcurso de las revisiones de software apenas se ve afectado por la tecnología orientada a objetos. Sin embargo, en relación a los sistemas no orientados a objetos, lo que se revisa es otra cosa.

La dirección debe intentar encontrar un equilibrio entre demasiados recorridos de inspección y demasiado pocos. En todos los sistemas, excepto en los que involucran vidas humanas, simplemente no es económico revisar cada línea de código. Por tanto, la dirección debe orientar los escasos recursos de su equipo para revisar aquellos aspectos del sistema que representen cuestiones de desarrollo estratégicas. Para sistemas orientados a objetos, esto sugiere el conducir las revisiones formales sobre los escenarios así como sobre la arquitectura del sistema, con revisiones mucho más informales centradas en cuestiones tácticas menores.

Como se describió en el capítulo anterior, los escenarios son un producto

primario de la fase de análisis del desarrollo orientado a objetos, y sirven para captar el comportamiento que se desea del sistema en términos de sus puntos funcionales. Las revisiones formales de escenarios las dirigen los analistas del equipo, junto con expertos del dominio u otros usuarios finales, y son presentadas por otros desarrolladores. Es mejor hacer tales revisiones a lo largo de la fase de análisis, en vez de esperar a efectuar una revisión masiva al final del mismo, cuando ya es demasiado tarde para hacer nada útil que permita enderezar el esfuerzo de análisis. La experiencia con el método muestra que incluso las personas que no son programadores pueden entender los escenarios presentados mediante guiones o los formalismos de los diagramas de objetos². En última instancia, tales revisiones ayudan a establecer un vocabulario común entre los desarrolladores de un sistema y sus usuarios. El permitir a otros miembros del equipo de desarrollo presenciar estas revisiones les enfrenta a los requerimientos reales del sistema en fases tempranas del proceso de desarrollo.

Las revisiones arquitectónicas deberían centrarse en la estructura general del sistema, incluyendo su estructura de clases y sus mecanismos. Al igual que en las revisiones de escenarios, las arquitectónicas deberían hacerse a lo largo del proyecto, dirigidas por el arquitecto del proyecto u otros diseñadores. Las revisiones iniciales harán hincapié en cuestiones generales de la arquitectura, mientras que las posteriores pueden centrarse en cierta categoría de clases o en mecanismos específicos muy utilizados. El propósito principal de tales revisiones es validar los diseños al principio del ciclo de vida. Haciéndolo así, también se ayuda a comunicar la visión de la arquitectura. Un propósito secundario de tales revisiones es incrementar la visibilidad de la arquitectura de forma que se creen oportunidades para descubrir patrones de clases o colaboraciones de objetos, que pueden explotarse entonces con el tiempo para simplificar la arquitectura.

Las revisiones informales deberían efectuarse semanalmente, y generalmente conllevan la revisión análoga de ciertos agrupamientos de clases de mecanismos de nivel inferior. El propósito de tales revisiones es validar estas decisiones tácticas; su propósito secundario es proporcionar un vehículo para que los desarrolladores más veteranos instruyan a los miembros jóvenes del equipo.

7.2. Administración de personal

Asignación de recursos

Uno de los aspectos más atractivos de la gestión de proyectos orientados a objetos es que, en el estado estable, suele darse una reducción en la cantidad total de recursos que se necesitan y un desplazamiento en el ritmo de su despliegue

² Se ha encontrado uso de la notación en revisiones que implicaban grupos de no-programadores tan diversos como astrónomos, biólogos, meteorólogos, físicos y banqueros.

respecto a métodos más tradicionales. La frase operativa aquí es «en el estado estable». Hablando en general, el primer proyecto orientado a objetos que aborde una organización requerirá algunos recursos más que para los métodos no orientados a objetos, más que nada a causa de la curva de aprendizaje inherente a la adopción de toda nueva tecnología. Los beneficios esenciales en cuanto a recursos del modelo de objetos no se manifestarán hasta el segundo o tercer proyecto, momento en el cual el equipo de desarrollo tiene más destreza en el diseño de clases y en la recolección de abstracciones y mecanismos comunes, y el equipo de gestión se encuentra más cómodo dirigiendo el proceso de desarrollo incremental e iterativo.

Para el análisis, los requisitos de recursos no suelen cambiar mucho cuando se emplean métodos orientados a objetos. Sin embargo, puesto que el proceso orientado a objetos pone el énfasis en el diseño arquitectónico, se tiende a acelerar el despliegue de los arquitectos y otros diseñadores demasiado pronto en el proceso de desarrollo, dedicándolos a veces durante fases posteriores del análisis a comenzar la exploración arquitectónica. Durante la evolución, suelen requerirse menos recursos, principalmente porque el trabajo en curso tiende a usar como trampolín las abstracciones y mecanismos comunes inventados anteriormente durante el diseño arquitectónico o versiones evolucionadas anteriormente. La prueba puede requerir también menos recursos, principalmente porque el añadir una nueva funcionalidad a una clase o mecanismo se consigue sobre todo modificando una estructura que se sabe que se comportaba correctamente antes del cambio. Así, la prueba tiende a comenzar antes en el ciclo de vida, y se manifiesta como una actividad acumulativa en vez de monolítica. La integración requiere usualmente muchos menos recursos en comparación con los métodos tradicionales, y la razón principal es que la integración sucede incrementalmente a lo largo de todo el ciclo de vida del desarrollo, en vez de ocurrir en un evento explosivo. Así, en una situación de estabilidad, la suma neta de todos los recursos humanos requeridos para el desarrollo orientado a objetos es normalmente menor que la de los requeridos en enfoques tradicionales. Es más, cuando se considera el coste de posesión de software orientado a objetos, los costes del ciclo de vida total son con frecuencia menores, porque el producto resultante tiende a ser de mucha mejor calidad, y por eso es mucho más flexible ante el cambio.

Papeles del equipo de desarrollo

Es importante recordar que el desarrollo de software es al fin y al cabo un esfuerzo humano. Los desarrolladores no son partes intercambiables y el despliegue con éxito de cualquier sistema complejo requiere las habilidades únicas y variadas de un equipo bien centrado de personas.

La experiencia sugiere que el proceso de desarrollo orientado a objetos requiere una partición sutilmente distinta de cometidos en comparación con los

métodos tradicionales. Se ha encontrado que los siguientes tres papeles son centrales en un proyecto orientado a objetos:

- Arquitecto del proyecto.
- Jefe del subsistema.
- Programador de aplicación.

El arquitecto del proyecto es el visionario, y es responsable de la evolución y mantenimiento de desarrollar la arquitectura del sistema. Para sistemas de tamaño pequeño a medio, el diseño arquitectónico suele ser responsabilidad de una o dos personas especialmente perspicaces. Para proyectos mayores, puede haber una responsabilidad compartida por un equipo más grande. El arquitecto del proyecto no tiene por qué ser el desarrollador más veterano, sino que más bien es el más cualificado para tomar decisiones estratégicas, usualmente como resultado de su extensa experiencia en la construcción de tipos similares de sistema. Gracias a esta experiencia, esos desarrolladores conocen intuitivamente los patrones arquitectónicos comunes que interesan en un dominio dado, y saben qué problemas de funcionamiento se aplican a ciertas variantes de la arquitectura. Los arquitectos no son necesariamente los mejores programadores de todos, aunque deberían tener una habilidad adecuada para programar. Del mismo modo que un arquitecto de edificios debería ser diestro en aspectos de construcción, en general no resulta acertado emplear a un arquitecto de software que no sea también un programador razonablemente decente. Los arquitectos del proyecto también deberían estar versados en la notación y el proceso del desarrollo orientado a objetos, porque al fin y al cabo deben expresar su visión arquitectónica en términos de agrupaciones de clases y colaboraciones de objetos.

En general es mala práctica contratar a un arquitecto externo que, hablando de forma alegórica, irrumpa en un caballo blanco, proclama cierta visión arquitectónica, y a continuación se esfuma mientras otros sufren las consecuencias de esas decisiones. Es mucho mejor ocupar activamente a un arquitecto durante el proceso de análisis y después retener a ese arquitecto durante la mayor parte de la evolución del sistema (si no toda). Así, este arquitecto se familiarizará más con las necesidades reales del sistema, y con el tiempo estará sujeto a las implicaciones de sus decisiones arquitectónicas. Además, manteniendo la responsabilidad de la integridad arquitectónica en manos de una sola persona o de un equipo pequeño, se aumentan las posibilidades de desarrollar una arquitectura pequeña y más flexible.

Los jefes de subsistema son los principales generadores de abstracciones del proyecto. Un jefe de subsistema es responsable del diseño de una categoría de clases completa o subsistema. En conjunción con el arquitecto del proyecto, cada jefe debe idear, defender y negociar el interfaz de una categoría de clases concreta o subsistema, y dirigir después su implantación. Un jefe de subsistema es por tanto el propietario último de una agrupación de clases y sus mecanismos asociados, y es también responsable de sus pruebas y versiones durante la evolución del sistema.

Los jefes de subsistema deben estar versados en la notación y el proceso del desarrollo orientado a objetos. Generalmente son mejores y más rápidos programadores que el arquitecto del proyecto, pero les falta la amplia experiencia del arquitecto. Por término medio, los jefes de subsistema constituyen más o menos la tercera parte o la mitad del equipo de desarrollo.

Los ingenieros de aplicación son los desarrolladores menos veteranos en un proyecto, y asumen una de dos responsabilidades. Ciertos ingenieros de aplicación son responsables de la implantación de una categoría o subsistema, bajo la supervisión de su jefe de subsistema. Esta actividad puede implicar el diseño de alguna clase, pero en general conlleva la implantación y la posterior prueba unitaria de las clases y mecanismos inventados por otros diseñadores del equipo. Otros ingenieros de aplicación son responsables de tomar las clases diseñadas por el arquitecto y los jefes de subsistema y ensamblarlas para llevar a cabo los puntos funcionales del sistema. En cierto sentido, estos ingenieros son responsables de escribir pequeños programas en el lenguaje específico del dominio definido por las clases y mecanismos de la arquitectura.

Los ingenieros de aplicación están familiarizados con la notación y el proceso del desarrollo orientado a objetos, pero no son necesariamente expertos; sin embargo, los ingenieros de aplicación son muy buenos programadores y comprenden las pautas e idiosincrasia de los lenguajes de programación dados. Por término medio, la mitad o algo más del equipo de desarrollo consiste en ingenieros de aplicación.

Esta división de cometidos soluciona el problema de personal al que se enfrenta la mayoría de las organizaciones de desarrollo de software, que normalmente sólo tienen un puñado de diseñadores realmente buenos y muchos otros con menos experiencia. El beneficio social de este enfoque a la división de personal es que ofrece una carrera a las personas más jóvenes del equipo: en concreto, los desarrolladores jóvenes trabajan guiados por desarrolladores más veteranos en una relación maestro/aprendiz. A medida que ganan experiencia en el uso de clases bien diseñadas, con el tiempo aprenden a diseñar sus propias clases de calidad. El corolario de esta disposición es que no todos los desarrolladores necesitan ser expertos en generar abstracciones, pero su habilidad en ese terreno puede crecer con el tiempo.

En proyectos mayores, puede haber otra serie de papeles de desarrollo distintos que se necesitan para efectuar el trabajo del proyecto. La mayoría de estos papeles (como el responsable de herramientas) son indiferentes al uso de tecnología orientada a objetos, aunque algunos de ellos están especialmente relacionados con el modelo de objetos (como el ingeniero de reutilización):

- Jefe de proyecto

Responsable de la gestión activa de los productos a entregar, tareas, recursos y planes del proyecto.

• Analista	Responsable de desarrollar e interpretar los requisitos del usuario final; debe ser un experto en el dominio del problema, aunque no debe estar aislado del resto del equipo de desarrollo.
• Ingeniero de reutilización	Responsable de gestionar el almacén de componentes y diseños del proyecto; mediante su participación en revisiones y otras actividades, busca activamente oportunidades para aprovechar elementos comunes y promueve su explotación; adquiere, produce y adapta componentes para uso general en el proyecto o la organización completa.
• Control de calidad	Responsable de medir los productos del proceso de desarrollo; generalmente dirige pruebas a nivel de sistema de todos los prototipos y versiones de producción.
• Jefe de integración	Responsable de ensamblar versiones compatibles de categorías y subsistemas ya completados con el fin de formar una versión que se pueda entregar; responsable de mantener las configuraciones de los productos que se generan.
• Documentalista	Responsable de producir documentación del producto y su arquitectura para el usuario final.
• Responsable de herramientas	Responsable de crear y adaptar herramientas software que faciliten la producción de los elementos del proyecto que van a entregarse, especialmente respecto al código generado
• Administrador del sistema	Responsable de gestionar los recursos físicos de computación utilizados por el proyecto.

Por supuesto, no todos los proyectos requieren todos esos papeles. Para proyectos pequeños, muchas de estas responsabilidades deben ser compartidas por la misma persona; para proyectos mayores, cada papel puede representar una organización entera.

La experiencia indica que el desarrollo orientado a objetos hace posible utilizar equipos de desarrollo más pequeños en comparación con los métodos tra-

dicionales. En realidad, no es extraño que un equipo de alrededor de 30-40 desarrolladores produzca más de un millón de líneas de código de calidad final en un solo año. Sin embargo, estamos de acuerdo con Boehm, que observa que «los mejores resultados se dan con menos y mejores personas» [6]. Desgraciadamente, intentar asignar a un proyecto menos personal del que el folklore tradicional sugiere que se necesita puede provocar resistencias. Como se sugirió en el capítulo anterior, tal enfoque va en contra de los intentos de algunos gestores para construir imperios. A otros gestores les gusta ocultarse tras grandes números de empleados, porque más gente representa más poder. Además, si un proyecto falla, hay más subordinados a quien echar la culpa.

Sólo por el hecho de que un proyecto aplique el método de diseño más sofisticado o la herramienta más reciente y fantástica no puede decirse que un jefe tenga derecho a eludir su responsabilidad por contratar diseñadores que pueden pensar o dejar que el proyecto vuele con piloto automático [7].

7.3. Gestión de versiones

Integración

Los proyectos de dimensión industrial requieren el desarrollo de familias de programas. En cualquier momento dado del proceso de desarrollo, habrá múltiples prototipos y versiones de producción, así como andamiajes de desarrollo y pruebas. La mayoría de las veces, cada desarrollador tendrá su propia versión ejecutable del sistema que se desarrolla.

Como se explicó en el capítulo anterior, la naturaleza del proceso incremental e iterativo del desarrollo orientado a objetos significa que debería darse raramente, si es que se da, un solo evento «explosivo» de integración. En vez de eso, generalmente habrá muchos eventos de integración más pequeños, cada uno de los cuales marcará la creación de otro prototipo o versión arquitectónica. Cada una de tales versiones suele ser incremental por naturaleza, y habrá evolucionado de una versión estable anterior. Como Davis *et al.*, observan, «cuando se usa desarrollo incremental, el software se construye deliberadamente para satisfacer inicialmente menos requisitos, pero se construye de forma que se facilite la incorporación de requisitos nuevos y se logre así una adaptabilidad mayor» [8]. Desde la perspectiva del usuario último del sistema, el macroproceso genera una serie de versiones ejecutables, cada una con funcionalidad creciente, que a la larga evolucionan hasta el sistema en producción final. Desde la perspectiva de los que están en la organización, se construyen en realidad muchas más versiones, y sólo algunas se congelan y se convierten en línea base para estabilizar interfaces importantes del sistema. Esta estrategia tiende a reducir el riesgo de desarrollo, porque acelera el descubrimiento de problemas de arquitectura y eficiencia en fases tempranas del proceso de desarrollo.

Para proyectos mayores, una organización puede producir una versión interna del sistema cada pocas semanas y suministrar después cada pocos meses una versión en funcionamiento a sus clientes para que la revisen, de acuerdo con las necesidades del proyecto. En una situación estable, una versión consta de un conjunto de subsistemas compatibles junto con su documentación asociada. La construcción de una versión es posible siempre y cuando los subsistemas principales del proyecto sean lo bastante estables y funcionen juntos lo suficientemente bien como para proporcionar algún nivel nuevo de funcionalidad.

Gestión de configuraciones y control de versiones

Considérese esta corriente de versiones desde la perspectiva de un desarrollador individual, que podría ser responsable de implantar un subsistema particular. Debe tener una versión de trabajo de ese subsistema, es decir, una versión en desarrollo. Con el fin de proceder con el desarrollo posterior, como mínimo deben estar disponibles los interfaces de todos los subsistemas importados. A medida que esta versión de trabajo se estabiliza, se entrega a un equipo de integración, responsable de recoger un conjunto de subsistemas compatibles para todo el sistema. Finalmente, esta colección de subsistemas es fija y tiene una línea de base, y forman parte de una versión interna. Esta versión interna se convierte así en la versión operativa actual, visible para todos los desarrolladores activos que necesiten refinar más su parte particular de la implantación. Mientras tanto, el desarrollador individual puede trabajar en una nueva versión de su subsistema. Así, el desarrollo puede avanzar en paralelo, siendo posible la estabilidad gracias a que los interfaces de los subsistemas están bien definidos y bien vigilados.

En este modelo está implícita la idea de que la unidad de control de versiones es un agrupamiento de clases, no una clase individual. La experiencia sugiere que la gestión de versiones de clases es de una granularidad demasiado fina, ya que ninguna clase tiende a estar sola. En lugar de eso, es mejor hacer versiones de grupos relacionados de clases. Hablando en términos prácticos, esto significa hacer versiones de subsistemas, ya que los grupos de clases (que forman categorías de clases en la vista lógica del sistema) se corresponden con subsistemas (en la vista física del sistema).

En cualquier momento dado de la evolución de un sistema pueden existir múltiples versiones de un subsistema particular: podría haber una versión del subsistema para la versión del sistema que se está desarrollando, otra para la versión interna actual, y otra para la última versión del sistema que se le ha entregado al usuario. Esto intensifica la necesidad de herramientas razonablemente potentes de gestión de configuraciones y control de versiones.

El código fuente no es el único producto del desarrollo que debería ponerse bajo la gestión de configuraciones. Los mismos conceptos se aplican a todos los demás productos del desarrollo orientado a objetos, como los requisitos, diagramas de clases, diagramas de objetos, diagramas de módulos y diagramas de procesos.

Prueba

El principio de integración continua se aplica también a la prueba del software, que también debería ser una actividad continua durante el proceso de desarrollo. En el contexto de arquitecturas orientadas a objetos, la prueba debe abarcar al menos tres dimensiones:

- Prueba unitaria Implica la prueba de clases y mecanismos individuales; es responsabilidad del ingeniero de aplicación que ha implantado la estructura.
- Prueba de subsistema Implica probar una categoría completa o subsistema; es responsabilidad del jefe de subsistema; las pruebas de subsistema pueden utilizarse como pruebas de regresión para cada versión del subsistema que se completa nuevamente.
- Prueba del sistema Implica probar el sistema como un todo; es responsabilidad del equipo de control de calidad; las pruebas del sistema también se usan típicamente como pruebas de regresión por parte del equipo de integración cuando se ensamblan nuevas versiones.

La prueba debería centrarse en el comportamiento externo del sistema; un propósito secundario de la prueba es tantejar los límites del sistema con el fin de comprender cómo falla bajo ciertas condiciones.

7.4. Reutilización

Elementos de la reutilización

Cualquier artefacto del desarrollo de software puede reutilizarse, incluyendo código, diseños, escenarios y documentación. Como se puso de relieve en el Capítulo 3, en lenguajes de programación orientados a objetos, las clases sirven como el vehículo lingüístico primario para la reutilización: de las clases pueden derivarse subclases para especializar o extender la clase base. Asimismo, como se explicó en el Capítulo 4, se pueden reutilizar patrones de clases, objetos y diseños en forma de modismos, mecanismos y marcos de referencia. La reutilización de patrones está a un nivel de abstracción más alto que la reutilización de clases individuales, y por tanto proporciona un impulso mayor (pero es más difícil de conseguir).

Es peligroso y engañoso acotar cifras para niveles de reutilización [9]. En

proyectos de éxito, hemos encontrado factores de reutilización tan altos como el 70 % (lo que significa que casi tres cuartas partes del software del sistema se tomó intacto de alguna otra fuente) y tan bajos como el 0 %. El grado de reutilización no debería verse como una cota a alcanzar, porque la reutilización potencial parece variar brutalmente con el dominio y está afectada por muchos factores no técnicos, como presiones de la planificación, naturaleza de las relaciones con los subarrendatarios y consideraciones de seguridad.

En última instancia, cualquier cantidad de reutilización es mejor que no reutilizar nada, porque la reutilización representa un ahorro de recursos que de otro modo se emplearían en reinventar algún problema que ya se había solucionado en abstracción.

Institucionalizar la reutilización

La reutilización dentro de un proyecto o incluso de una organización completa no ocurre sin más, debe institucionalizarse. Esto significa que las oportunidades para reutilizar deben buscarse y recompensarse. En realidad, esta es la razón por la que se incluye la recolección de patrones como una actividad explícita del macroproceso.

Un programa de reutilización efectiva se alcanza mejor responsabilizando a personas concretas de esa actividad. Como se describió en el capítulo anterior, esta actividad conlleva la identificación de oportunidades para aprovechar aspectos comunes, descubiertas usualmente mediante revisiones arquitectónicas, y la explotación de esas oportunidades, casi siempre produciendo nuevos componentes o adaptando otros existentes, y defendiendo su reutilización entre los desarrolladores. Este enfoque requiere que se premie explícitamente la reutilización. Incluso las recompensas simples son muy efectivas al promover la reutilización: por ejemplo, el que se reconozca igualmente al autor que a quien reutiliza suele ser útil. Para proponer algo más tangible, puede ser efectivo ofrecer una cena gratis o un fin de semana lejos para el desarrollador (y un acompañante) cuyo código haya sido reutilizado con más frecuencia, o para quien reutilizó más código en cierto período de tiempo³.

Al final, la reutilización cuesta recursos a corto plazo, pero los ahorra a largo plazo. Una actividad de reutilización sólo tendrá éxito en una organización que adopte una visión a largo plazo del desarrollo de software y optimice los recursos para algo más que para el proyecto actual.

³ Este es un premio frecuentemente bien recibido por la pareja del desarrollador, que probablemente no lo ha visto mucho durante las angustias finales del desarrollo del software.

7.5. Control de calidad y métricas

Calidad del software

Schulmeyer y McManus definen la calidad del software como «la aptitud de uso del producto completo de software» [10]. La calidad del software no sucede sin más: debe introducirse en el sistema. Realmente, el uso de tecnología orientada a objetos no lleva automáticamente a software de calidad: sigue siendo posible escribir software muy malo usando lenguajes de programación orientados a objetos.

Esta es la razón por la que se pone tanto énfasis en la arquitectura del software en el proceso del desarrollo orientado a objetos. Una arquitectura simple y adaptable es fundamental para cualquier calidad del software; su calidad se completa adoptando decisiones tácticas simples y consistentes.

El control de calidad del software involucra «las actividades sistemáticas que proporcionan evidencia de la aptitud de uso del producto completo de software» [11]. El control de calidad busca ofrecer medidas cuantificables de bondad para la calidad de un sistema de software. Muchas de tales medidas tradicionales son aplicables directamente a los sistemas orientados a objetos.

Como se describió antes, los recorridos de revisión y otros tipos de inspecciones son prácticas importantes, incluso en sistemas orientados a objetos, y ofrecen un conocimiento mayor sobre la calidad del software. Puede que la medida cuantificable de bondad más importante sea la tasa de descubrimiento de defectos. Durante la evolución del sistema, se rastrean los defectos del software de acuerdo con su importancia y localización. La tasa de descubrimiento de defectos es en relación con esto una medida de la rapidez con la que se están descubriendo defectos, medida que se representa gráficamente respecto al tiempo. Como observa Dobbins, «el número real de defectos es menos importante que la inclinación de la línea» [12]. Un proyecto que está bajo control tendrá una curva que alcanza forma de campana, con la tasa de descubrimiento de defectos que alcanza un máximo hacia la mitad del período de prueba y que cae después hacia cero. Un proyecto fuera de control tendrá una curva que disminuye muy lentamente, o no disminuye en absoluto.

Una de las razones por las que el macroproceso del desarrollo orientado a objetos funciona tan bien es que desde muy pronto permite una recolección de datos continua sobre la tasa de descubrimiento de defectos. Para cada versión incremental se puede realizar una prueba del sistema y representar la tasa de descubrimiento de defectos frente al tiempo. Incluso aunque las versiones iniciales tengan menos funcionalidad, se sigue esperando ver una curva en forma de campana para cualquier versión en un proyecto saludable.

La densidad de defectos es otra medida interesante de calidad. El enfoque tradicional es la medida de los defectos por cada mil líneas de código (KSLOC)⁴

⁴ KSLOC: En inglés, *Kilo Source Lines Of Code* (Mil líneas de código fuente). (N. del T.)

y en general sigue siendo aplicable a sistemas orientados a objetos. En proyectos saludables, la densidad de defectos tiende a «alcanzar un valor estable después de que se han inspeccionado aproximadamente 10.000 líneas de código y a partir de ahí permanecerá casi constante cualquiera que sea el tamaño del volumen de código restante» [13].

En los sistemas orientados a objetos se ha encontrado que también es útil medir la densidad de defectos según los números de defectos por categoría de clases o por clase. Con esta medida, parece aplicarse la regla 80/20: el 80 % de los defectos del software aparecerá en el 20 % de las clases del sistema [14].

Además de los enfoques más formales para la recogida de información sobre defectos a través de la prueba del sistema, también se ha comprobado que es útil instituir (a nivel de proyecto o compañía completa) «cazas de errores» durante las cuales todo el mundo puede ejercitarse una versión durante un período limitado de tiempo. Se ofrecen entonces premios a la persona que encuentre más defectos, así como a la persona que encuentre el defecto más oscuro. No hacen falta premios extravagantes: tazas para el café, vales para cenas o películas, o incluso camisetas, son recompensas adecuadas para el cazador de errores más intrépido.

Métricas orientadas a objetos

Puede que la forma más espantosa para que un jefe mida el progreso sea la medición de las líneas de código que se producen. El número de caracteres de fin de línea en un fragmento de código fuente no tiene la más mínima correlación con su perfección o complejidad. Contribuye a las carencias de este enfoque propio de Neanderthal la facilidad con la que se puede jugar con los números, lo que resulta en cifras de productividad que pueden diferir entre sí incluso en dos órdenes de magnitud. Por ejemplo, ¿qué es exactamente una línea de código (especialmente en Smalltalk)? ¿Se cuentan líneas físicas, o caracteres de punto y coma? ¿Qué pasa al contar múltiples sentencias que aparecen en una línea o sentencias que ocupan más de una línea? Análogamente, ¿cómo se mide el trabajo que conllevan? ¿Se mide a todo el personal, o quizás sólo a los programadores? El día de trabajo, ¿se mide como un día de ocho horas, o se cuenta también el tiempo que un programador emplea en trabajar hasta altas horas de la madrugada? Las medidas tradicionales de la complejidad, más adecuadas para lenguajes de programación de generaciones iniciales, también tienen mínima correlación con la perfección y complejidad en sistemas orientados a objetos, y por tanto son prácticamente inútiles cuando se aplican al sistema en su conjunto.

Por ejemplo, la métrica Ciclomática de McCabe, cuando se aplica a un sistema orientado a objetos en su conjunto, no da una medida muy significativa de la complejidad, porque está ciega ante los mecanismos y la estructura de clases. Sin embargo, se ha encontrado útil generar una métrica ciclomática por clase. Esto da alguna indicación de la complejidad relativa de las clases individuales,

y puede usarse luego para dirigir inspecciones a las clases más complejas, que probablemente tengan la mayoría de los defectos.

Se tiende a medir el progreso contando las clases en el diseño lógico, o los módulos en el diseño físico, que están completados y en desarrollo. Como se describió en el capítulo anterior, otra medida de progreso es la estabilidad de los interfaces clave (es decir, con qué frecuencia cambian). Al principio, los interfaces de todas las abstracciones clave cambiarán a diario, si no a cada hora. Con el tiempo, los interfaces más importantes se estabilizarán primero, los siguientes más importantes se estabilizarán a continuación, y así sucesivamente. Hacia el final del ciclo de vida del desarrollo, sólo habrá que cambiar unos pocos interfaces insignificantes, ya que el mayor énfasis estará en conseguir que las clases y módulos ya diseñados funcionen juntos. Ocasionalmente, pueden necesitarse unos pocos cambios en un interfaz crítico, pero tales cambios suelen ser compatibles en sentido creciente. Incluso siendo así, tales cambios se hacen sólo tras una evaluación cuidadosa de su impacto. Estos cambios pueden introducirse entonces incrementalmente en el sistema en producción como parte del ciclo habitual de versiones.

Chidamber y Kemerer sugieren una serie de métricas que son aplicables directamente a sistemas orientados a objetos [15]:

- Métodos ponderados por clase.
- Profundidad del árbol de herencias.
- Número de hijos.
- Acoplamiento entre objetos.
- Respuesta para una clase.
- Falta de cohesión en los métodos.

Los métodos ponderados por clase dan una medida relativa de la complejidad de una clase individual; si se considera que todos los métodos son igual de complejos, se convierte en una medida del número de métodos por clase. En general una clase con un número significativamente mayor de métodos que sus hermanas es más compleja, tiende a ser más específica de la aplicación, y muchas veces es huésped de un mayor número de defectos.

La profundidad del árbol de herencias y el número de hijos son medidas de la forma y tamaño de la estructura de clases. Como se describió en el Capítulo 3, los sistemas orientados a objetos bien estructurados tienden a tener una arquitectura de bosques de clases, en lugar de una sola trama de herencias muy grande. Como regla general, se tiende a construir tramas que estén equilibradas y que generalmente no sean más profundas de 7 ± 2 clases ni más anchas de 7 ± 2 clases.

El acoplamiento entre objetos es una medida de su posibilidad de conexión con otros objetos, y por tanto es una medida de la sobrecarga a que está sometida su clase. Al igual que en medidas tradicionales del acoplamiento, se busca diseñar objetos débilmente acoplados, que tienen un gran potencial de reutilización.

La respuesta para una clase es una medida de los métodos que sus instancias

pueden llamar; la cohesión en los métodos es una medida de la unidad de la abstracción de la clase. En general, una clase que puede invocar un número significativamente mayor de métodos que sus hermanas es más compleja. Una clase con baja cohesión entre sus métodos sugiere una abstracción accidental o inapropiada: tal clase debería, en general, reabstraerse en más de una clase, o sus responsabilidades deberían delegarse a otras clases existentes.

7.6. Documentación

El legado del desarrollo

El desarrollo de un sistema de software conlleva mucho más que la pura escritura del código fuente. Ciertos productos del desarrollo ofrecen vías para que su equipo directivo y sus usuarios consigan mejor conocimiento sobre el progreso del proyecto. También se busca dejar constancia de un legado de decisiones de análisis y diseño para posibles mantenedores del sistema. Como se puso de relieve en el Capítulo 5, los productos del desarrollo orientado a objetos incluyen en general conjuntos de diagramas de clases, diagramas de objetos, diagramas de módulos y diagramas de procesos. En conjunto, estos diagramas ofrecen una vía para remontarse a los requisitos del sistema. Los diagramas de procesos denotan programas, que son los módulos básicos que se encuentran en los diagramas de módulos. Cada módulo representa la implantación de alguna combinación de clases y objetos, que a su vez aparecen en los diagramas de clases y diagramas de objetos, respectivamente. Por último, los diagramas de objetos denotan escenarios especificados por los requisitos, y los diagramas de clases representan abstracciones clave que forman el vocabulario del dominio del problema.

Contenidos de la documentación

La documentación de la arquitectura e implantación de un sistema es importante, pero la producción de esos documentos nunca debería dirigir el proceso de desarrollo: la documentación es un producto del proceso de desarrollo, esencial aunque secundario. También es importante recordar que los documentos son productos vivos a los que debería permitirse evolucionar al lado de la evolución incremental e iterativa de las versiones del proyecto. Junto con el código generado, los documentos producidos sirven como la base de la mayoría de las revisiones formales e informales.

¿Qué debe documentarse? Obviamente, hay que producir documentación para el usuario final, instruyéndolo sobre la operación e instalación de cada

versión⁵. Además, hay que producir documentación de análisis para capturar la semántica de los puntos funcionales del sistema tal como se la ve mediante escenarios. Hay que generar también documentación de arquitectura e implantación, para comunicar la visión y los detalles de la arquitectura al equipo de desarrollo y para conservar información sobre todas las decisiones estratégicas relevantes, de forma que el sistema esté dispuesto para adaptarse y evolucionar en el tiempo con facilidad.

En general, la documentación esencial de la arquitectura e implantación de un sistema debería incluir lo siguiente:

- Documentación de la arquitectura de alto nivel del sistema.
- Documentación de las abstracciones y mecanismos clave de la arquitectura.
- Documentación de los escenarios que ilustran el comportamiento práctico de aspectos clave del sistema.

La peor documentación que se puede crear para un sistema orientado a objetos es una descripción separada de la semántica de cada método clase por clase. Este enfoque tiende a producir un montón de documentación inútil que no hay quien lea y en la que nadie puede confiar, y no documenta bien las cuestiones arquitectónicas más importantes que trascienden las clases individuales, es decir, las colaboraciones entre clases y objetos. Es mucho mejor documentar estas estructuras de nivel más alto, que pueden expresarse en diagramas de la notación, pero no tienen expresión lingüística directa en el lenguaje de programación, y dar a los programadores referencias hacia los interfaces de ciertas clases importantes para los detalles tácticos.

7.7. Herramientas

Con los lenguajes primitivos, a un equipo de desarrollo le bastaba tener un conjunto mínimo de herramientas: un editor, un compilador, un montador de enlaces (*linker*) y un cargador eran frecuentemente todo lo que se necesitaba (y muchas veces todo lo que existía). Si el equipo tenía mucha suerte, podía incluso conseguir un depurador del código fuente. Los sistemas complejos cambian por completo el panorama: intentar construir un sistema grande de software con un conjunto mínimo de herramientas es equivalente a construir un edificio de varios pisos con herramientas manuales de piedra.

Las prácticas del desarrollo orientado a objetos cambian también el panorama. Las herramientas de desarrollo de software tradicionales sólo incorporan conocimiento sobre el código fuente, pero ya que el análisis y diseño orientados

⁵ Hay una regla no escrita que dice que para el software de productividad personal, un sistema que obligue al usuario a mirar constantemente el manual es hostil a ese usuario. Los interfaces de usuario orientados a objetos, concretamente, deberían diseñarse para que su uso fuese intuitivo y autoconsistente, con el fin de minimizar o eliminar las necesidades de documentación para el usuario final.

a objetos ponen el énfasis en abstracciones y mecanismos clave, se necesitan herramientas que puedan concentrarse en semánticas más ricas. Además, el rápido desarrollo de versiones definido por el macroproceso del desarrollo orientado a objetos requiere herramientas que ofrezcan una rápida alternancia, especialmente para el ciclo edición/compilación/ejecución/depuración.

Es importante elegir herramientas que soporten bien el cambio de escala. Una herramienta que funciona para un desarrollador que escribe una pequeña aplicación independiente no se adaptará necesariamente para versiones de producción de aplicaciones más complejas. Realmente, para cada herramienta, habrá un umbral a partir del cual se excede su capacidad, haciendo que sus ventajas se vean claramente sobrepasadas por sus riesgos y sus chapuzas.

Tipos de herramientas

Hemos identificado al menos siete tipos diferentes de herramientas aplicables al desarrollo orientado a objetos. La primera es un sistema basado en gráficos que soporta la notación orientada a objetos presentada en el Capítulo 5. Tal herramienta puede utilizarse durante el análisis para capturar la semántica de los escenarios, así como en etapas iniciales del proceso de desarrollo para capturar decisiones estratégicas y tácticas, mantener control sobre los productos del diseño y coordinar las actividades de diseño de un equipo de desarrolladores. En realidad, tal herramienta puede usarse por todo el ciclo de vida, a medida que el diseño evoluciona hacia una implantación en producción. Esas herramientas también son útiles durante el mantenimiento del sistema. En concreto, se ha encontrado que es posible realizar ingeniería inversa sobre muchos de los aspectos interesantes de un sistema orientado a objetos, produciendo al menos la estructura de clases y la arquitectura de módulos del sistema tal como se ha construido. Esta característica es bastante importante: con herramientas CASE tradicionales, los desarrolladores pueden generar maravillosos dibujos, sólo para darse cuenta de que esos dibujos están obsoletos una vez que la implantación avanza, porque los programadores juegan con la implantación sin actualizar el diseño. La ingeniería inversa hace menos probable que la documentación de diseño esté alguna vez desfasada respecto a la implantación real.

La siguiente herramienta que se ha juzgado importante para el desarrollo orientado a objetos es un hojeador (browser) que conoce la estructura de clases y la arquitectura de módulos de un sistema⁶. Las jerarquías de clases pueden llegar a ser tan complejas que incluso es difícil encontrar todas las abstracciones que forman parte del diseño o son candidatas para la reutilización [16]. Mientras se examina un fragmento de programa, un desarrollador puede querer ver la definición de la clase de algún objeto. Tras encontrar la clase, podría desear visitar algunas de sus superclases. Mientras está viendo una superclase concreta, podría querer hojear todos los usos de esa clase antes de instalar un cambio en

⁶ Integrando el primer tipo de herramienta con el entorno de desarrollo de software del computador, se hace posible hojear entre el diseño y su implantación.

su interfaz. Este tipo de examen es especialmente aburrido si uno tiene que preocuparse de ficheros, que son un artefacto de las decisiones de diseño físico, no lógico. Por esta razón, los hojeadores son una herramienta importante para el análisis y diseño orientados a objetos. Por ejemplo, el entorno estándar de Smalltalk permite examinar todas las clases de un sistema en las formas que se han descrito. Existen utilidades similares en entornos de otros lenguajes de programación orientados a objetos, aunque con diferentes grados de sofisticación.

Otra herramienta que puede considerarse importante, si no absolutamente esencial, es un compilador incremental. El tipo de desarrollo evolutivo que se da en el desarrollo orientado a objetos pide a gritos un compilador incremental que pueda compilar declaraciones y sentencias individuales. Meyrowitz hace notar que «UNIX, tal como viene, con su orientación hacia la compilación por lotes de grandes ficheros de programas en bibliotecas que se enlazan después con otros fragmentos de código, no proporciona el soporte necesario para la programación orientada a objetos. Es claramente inaceptable el que se necesite un ciclo de compilación y enlazado de diez minutos ¡simplemente para cambiar la implantación de un método y que se necesite un ciclo de compilación y enlace de una hora simplemente para añadir un campo a una superclase de alto nivel! Los métodos y las ... definiciones de campos compilados incrementalmente son una necesidad para una depuración rápida» [17]. Existen compiladores incrementales para muchos de los lenguajes descritos en el apéndice; desgraciadamente, la mayoría de las implantaciones consisten en compiladores tradicionales, orientados a lotes.

Lo siguiente que se ha encontrado es que los proyectos significantes necesitan depuradores que sepan cosas sobre la semántica de las clases y objetos. Cuando se depura un programa, muchas veces es necesario examinar las variables de instancia y las variables de clase asociadas con un objeto. Los depuradores tradicionales para programación no orientada a objetos no incorporan conocimiento sobre clases y objetos. Así, si se intenta usar un depurador estándar de C para programas en C++, si es que se puede, no permitirá al desarrollador encontrar la información realmente importante que necesita para depurar un programa orientado a objetos. La situación es especialmente crítica con lenguajes de programación orientados a objetos que soportan múltiples hilos de control. En cualquier momento durante la ejecución de un programa semejante, puede haber varios procesos activos. Estas circunstancias requieren un depurador que permita al desarrollador ejercer un control sobre todos los hilos de control individuales, normalmente objeto a objeto.

En esa misma categoría de herramientas de depuración se incluyen herramientas como analizadores de rendimiento, que fuerzan la capacidad del software, normalmente en términos de utilización de recursos, y herramientas de análisis de memoria, que identifican violaciones de acceso a memoria, tales como escribir en memoria dinámica liberada, leer de memoria sin inicializar o leer y escribir fuera de los límites de una matriz.

A continuación, especialmente para proyectos mayores, hay que tener herramientas de gestión de configuraciones y control de versiones. Como se men-

ciónó anteriormente, la categoría o subsistema es la mejor unidad para la gestión de configuraciones.

Otra herramienta que se ha considerado importante en el desarrollo orientado a objetos es un bibliotecario de clases. La mayoría de los lenguajes mencionados en este libro tienen bibliotecas de clases predefinidas, o bibliotecas de clases disponibles comercialmente. A medida que un proyecto madura, esta biblioteca crece a medida que con el tiempo se van añadiendo componentes software reusables específicos del dominio. Una biblioteca de este tipo no tarda mucho en alcanzar proporciones enormes, lo que hace difícil a un desarrollador encontrar una clase o módulo que satisfaga sus necesidades. Una razón por la que una biblioteca puede llegar a tal tamaño es que normalmente una clase dada tiene muchas implantaciones, cada una de las cuales tiene diferente semántica de espacio y tiempo. Si el coste estimado para encontrar cierto componente (coste que normalmente se exagera) es mayor que el coste estimado para crearlo de cero (coste normalmente subestimado), toda esperanza de reutilización se habrá perdido. Por esta razón, es importante tener al menos alguna herramienta mínima de bibliotecario que permita a los desarrolladores localizar clases y módulos de acuerdo con diferentes criterios y añadir a la biblioteca clases y módulos a medida que se desarrollan.

El último tipo de herramienta que se ha encontrado útil para ciertos sistemas orientados a objetos es un constructor de interfaces gráficos de usuario (IGU). Para sistemas que impliquen gran cantidad de interacción con el usuario, es mucho mejor usar tal herramienta para crear de forma interactiva diálogos y otras ventanas que crear estos artefactos desde el nivel inferior del código. El código generado por estas herramientas puede conectarse entonces al resto del sistema orientado a objetos y, donde sea necesario, puede ajustarse a mano.

Implicaciones en la organización

Esta necesidad de herramientas potentes crea una demanda para dos papeles específicos en la organización de desarrollo: un ingeniero de reutilización y un responsable de herramientas. Entre otras cosas, los deberes del ingeniero de reutilización son mantener la biblioteca de clases de un proyecto. Sin un esfuerzo activo, tal biblioteca puede convertirse en un vasto y desierto país de clases-basura a través del cual ningún desarrollador querría caminar jamás. Además, también es necesario muchas veces adoptar una posición activa para alentar la reutilización, y el ingeniero de reutilización puede facilitar este proceso recolectando los productos de los esfuerzos de diseño del momento. Los deberes de un responsable de herramientas son crear herramientas específicas del dominio y adaptar otras existentes a las necesidades de un proyecto. Por ejemplo, un proyecto podría necesitar un andamiaje de pruebas común para probar ciertos aspectos de un interfaz de usuario, o podría necesitar un hojeador de clases a medida. Un responsable de herramientas está en la mejor posición para inventar

esas herramientas, normalmente a partir de componentes que ya están en la biblioteca de clases. Tales herramientas también pueden usarse para esfuerzos posteriores de desarrollo.

Un jefe que ya se haya enfrentado con recursos humanos escasos puede lamentarse de que las herramientas potentes, así como los responsables de herramientas e ingenieros de reutilización designados, son un lujo que no puede permitirse. No negamos este hecho en algunos proyectos de recursos restringidos. Sin embargo, en muchos otros proyectos, hemos encontrado que estas actividades se realizan de todas formas, normalmente de manera *ad hoc*. Abogamos por las inversiones explícitas en herramientas y personas que hagan estas actividades *ad hoc* más precisas y eficientes, lo que añade un valor real a todo el desarrollo.

7.8. Temas especiales

Cuestiones específicas del dominio

Se ha apreciado que ciertos dominios de aplicaciones justifican una consideración arquitectónica especial.

El diseño de un interfaz de usuario efectivo sigue teniendo mucho más de arte que de ciencia. Para este dominio, el uso de prototipos es completamente esencial. Debe haber una realimentación pronta y frecuente de los usuarios finales, con el fin de evaluar los gestos, comportamiento ante errores, y otros paradigmas de interacción del usuario. La generación de escenarios es muy efectiva para dirigir el análisis del interfaz de usuario.

Algunas aplicaciones involucran un componente importante de base de datos; otras aplicaciones pueden requerir integración con bases de datos cuyos esquemas no pueden cambiarse, normalmente porque ya están pobladas con grandes cantidades de datos (el problema del legado de los datos). Para tales dominios, el principio de separación de intereses es aplicable directamente: es mejor encapsular el acceso a todas esas bases de datos dentro de los confines de interfaces de clases bien definidas. Este principio es particularmente importante cuando se mezcla descomposición orientada a objetos con tecnología de bases de datos. En presencia de una base de datos orientada a objetos, el interfaz entre la base de datos y el resto de la aplicación puede tener muchas menos «costuras», pero hay que recordar que las bases de datos orientadas a objetos son más efectivas para la persistencia de objetos y menos para almacenamiento masivo de datos.

Considérense también los sistemas en tiempo real. *Tiempo real* significa cosas distintas en contextos distintos: el tiempo real podría denotar respuestas por debajo del segundo en sistemas centrados en el usuario, y respuestas por debajo del microsegundo en aplicaciones de adquisición y control de datos. Es impor-

tante darse cuenta de que incluso en los sistemas en tiempo real muy exigente, no todos los componentes del sistema tienen que (o pueden) optimizarse. En realidad, para la mayoría de los sistemas complejos, el riesgo más grande es si el sistema puede completarse o no, no si funcionará dentro de sus requisitos de eficacia. Por esta razón, hay que hacer una advertencia contra la optimización prematura. Hay que centrarse en producir arquitecturas simples, y la generación evolutiva de versiones iluminará los cuellos de botella que el sistema presenta respecto a la eficiencia lo bastante pronto como para tomar una acción correctiva.

El término *sistemas legados* se refiere a aplicaciones para las cuales existe una gran inversión de capital en software que no puede abandonarse por razones de economía o seguridad. Sin embargo, tales sistemas pueden tener costes de mantenimiento intolerables, que requieran su reemplazo gradual. Afortunadamente, combatir con sistemas legados es muy parecido a combatir con bases de datos: se encapsula el acceso a las funciones del sistema legado dentro del contexto de interfaces de clases bien definidas y, con el tiempo, se desplaza la cubierta de la arquitectura orientada a objetos para sustituir cierta funcionalidad que actualmente proporciona el sistema legado. Por supuesto, es esencial comenzar con una visión arquitectónica del aspecto que tendrá el sistema final, de forma que la sustitución incremental del sistema legado no acabará convirtiéndose en un mosaico de parches inconsistentes de software.

Transferencia tecnológica

Como dice Kempf, «El aprendizaje de la programación orientada a objetos puede ser perfectamente una tarea más difícil que la de aprender 'sólo' otro lenguaje de programación. Puede que se deba a que se involucra un estilo diferente de programación en vez de una sintaxis diferente dentro del mismo marco de referencia. Esto significa que no estamos hablando de un nuevo lenguaje sino de una nueva forma de pensar» [18].

¿Cómo desarrollar esta mentalidad orientada a objetos? Se recomienda lo siguiente:

- Proporcionar entrenamiento formal sobre los elementos del modelo de objetos tanto a los desarrolladores como a los gestores.
- Usar primero el desarrollo orientado a objetos en un proyecto de bajo riesgo, y dejar que el equipo cometa errores; utilizar estos miembros del equipo como semilla de otro proyecto y para que actúen como mentores del enfoque orientado a objetos.
- Exponer a los desarrolladores y gestores a ejemplos de sistemas orientados a objetos bien estructurados.

Son buenos proyectos candidatos las herramientas de desarrollo de software o las bibliotecas de clases específicas del dominio, que pueden usarse entonces como recursos en proyectos posteriores.

Según nuestra experiencia, bastan unas pocas semanas para que un desarrollador profesional domine la sintaxis y semántica de un nuevo lenguaje de programación. Pueden hacer falta varias semanas más para que el mismo desarrollador comience a apreciar la importancia y potencia de las clases y objetos. Finalmente, puede llevar tanto como seis meses de experiencia el que ese desarrollador madure como un diseñador de clases competente. Esto no tiene por qué ser malo, porque en cualquier disciplina lleva tiempo dominar el arte.

Se ha encontrado que el aprendizaje por ejemplos es con frecuencia un enfoque eficiente y efectivo. Una vez que una organización ha acumulado una masa crítica de aplicaciones escritas en un estilo orientado a objetos, el introducir a nuevos desarrolladores y gestores en el desarrollo orientado a objetos es mucho más fácil. Los desarrolladores comienzan como programadores de aplicaciones, utilizando las abstracciones bien estructuradas que ya existen. Con el tiempo, los desarrolladores que han estudiado y utilizado estos componentes bajo la supervisión de una persona más experimentada ganan suficiente experiencia para desarrollar un marco de referencia conceptual significativo del modelo de objetos y llegan a ser diseñadores de clases efectivos.

7.9. Las ventajas y los riesgos del desarrollo orientado a objetos

Las ventajas del desarrollo orientado a objetos

Quienes adoptan la tecnología orientada a objetos suelen abrazar estas prácticas por una de dos razones. Primero, buscan una ventaja en la competitividad, como un tiempo reducido para la comercialización, mayor flexibilidad de los productos o fiabilidad en la planificación. Segundo, pueden tener problemas que sean tan complejos que parezcan no tener ninguna otra solución.

En el Capítulo 2 se sugirió que el uso del modelo de objetos conduce a construir sistemas que incorporan los cinco atributos de los sistemas complejos bien estructurados. El modelo de objetos forma el marco de referencia conceptual para la notación y el proceso del desarrollo orientado a objetos, y así el propio método se beneficia de estas ventajas. En ese capítulo también se apuntaron los beneficios que se derivan de las siguientes características del modelo de objetos (y, por tanto, del desarrollo orientado a objetos):

- Explota la potencia expresiva de todos los lenguajes de programación orientados a objetos.
- Alienta la reutilización de componentes del software.
- Lleva a sistemas más flexibles al cambio.
- Reduce el riesgo de desarrollo.
- Resulta atractivo al funcionamiento de la mente humana.

Hay una serie de casos de estudio que apoyan estos hallazgos; en particular, apuntan que el enfoque orientado a objetos puede reducir el tiempo de desarrollo y el tamaño del código fuente resultante [19, 20, 21].

Los riesgos del desarrollo orientado a objetos

En el lado más oscuro del diseño orientado a objetos, se encuentran dos áreas de riesgo que deben ser consideradas: eficacia y costes de puesta en marcha.

En relación con los lenguajes procedimentales, definitivamente existe un coste de eficacia por enviar un mensaje de un objeto a otro en un lenguaje de programación orientado a objetos. Como se apuntó en el Capítulo 3, para invocaciones de métodos que no pueden resolverse estáticamente, una implantación debe realizar una búsqueda dinámica con el fin de encontrar el método definido por la clase del objeto receptor. Los estudios indican que, en el peor caso, una invocación de método puede llevar de 1,75 a 2,5 veces el tiempo de una simple llamada a subprograma [22, 23]. En el lado positivo, centrémonos en la frase operativa, «no puede resolverse estáticamente». La experiencia indica que se necesita realmente búsqueda dinámica sólo en aproximadamente el 20 por ciento de la mayoría de las invocaciones a métodos. Con un lenguaje con comprobación estricta de tipos, un compilador puede frecuentemente determinar qué invocaciones pueden resolverse estáticamente y generar código para llamadas a subprograma en vez de para búsquedas de método.

Otra fuente de sobrecarga para la ejecución proviene no tanto de la naturaleza de los lenguajes de programación orientados a objetos, sino de la forma en que se los utiliza en conjunción con el desarrollo orientado a objetos. Como ya se ha dicho muchas veces, el desarrollo orientado a objetos lleva a crear sistemas cuyos componentes están construidos en capas de abstracción. Una implicación de esta división en capas es que los métodos individuales son generalmente muy pequeños, ya que se construyen sobre otros de nivel inferior. Otra implicación es que a veces hay que escribir métodos para conseguir un acceso protegido a los campos (de otro modo encapsulados) de un objeto. Este montón de métodos significa que se puede acabar con una saturación de invocaciones de métodos. La invocación de un método a un nivel alto de abstracción suele resultar en una cascada de llamadas a métodos; los métodos de alto nivel suelen llamar a otros de nivel inferior, y así sucesivamente. Para aplicaciones en las que el tiempo es un recurso limitado, tantas invocaciones a métodos pueden ser inaceptables. De nuevo desde el lado positivo, tal división en capas es esencial para la comprensión de un sistema; puede que no llegue nunca a ser posible tener un sistema complejo funcionando si no se empieza por un diseño en capas. Se recomienda diseñar primero por funcionalidad, e instrumentar después el sistema en funcionamiento para determinar dónde hay realmente cuellos de botella en la velocidad. Estos cuellos de botella pueden eliminarse muchas veces declarando los métodos apropiados como *inline* (intercambiando por tanto espacio por tiempo),

reduciendo la jerarquía de clases o rompiendo el encapsulamiento de los atributos de una clase.

Un riesgo de eficacia relacionado se deriva de la sobrecarga de las clases: una clase profunda en una trama de herencias puede tener muchas superclases, cuyo código hay que incluir en la clase más específica al montar los enlaces (*linking*). Para aplicaciones orientadas a objetos pequeñas, esto puede significar en la práctica que las jerarquías de clases profundas deben evitarse, porque requieren demasiado código objeto. Este problema puede mitigarse un poco utilizando un compilador y un montador de enlaces experto que puedan eliminar todo el código muerto.

Hay aún otra fuente de cuellos de botella para la eficacia en el contexto de los lenguajes de programación orientados a objetos, que se deriva del comportamiento paginado de las aplicaciones en ejecución. La mayoría de los compiladores ubican el código objeto en segmentos, con el código de cada unidad de compilación (con frecuencia un solo archivo) situado en uno o más segmentos. Este modelo supone una alta localidad de la referencia; los subprogramas dentro de un segmento llaman a subprogramas del mismo segmento. Sin embargo, en sistemas orientados a objetos, raramente se da esa localidad de la referencia. Para sistemas mayores, las clases suelen declararse en archivos separados, y puesto que los métodos de una clase se construyen usualmente sobre los de otras clases, una sola invocación de método pueden involucrar código de muchos segmentos diferentes. Esto viola las suposiciones que la mayoría de los computadores realizan sobre el comportamiento de los programas en tiempo de ejecución, particularmente los computadores con CPUs en tubería y sistemas de memoria por paginación. Volviendo al lado positivo, ésta es la razón por la que se separan las decisiones de diseño lógicas y físicas. Si un sistema en funcionamiento se ralentiza durante la ejecución debido a un intercambio de páginas excesivo, el acotar el problema es en gran medida cuestión de cambiar la asignación física de las clases a módulos. Esta es una decisión de diseño del modelo físico del sistema, que no tiene efecto sobre su diseño lógico.

Otro riesgo de eficacia en sistemas orientados a objetos proviene de la asignación y destrucción dinámica de objetos. Asignar memoria a un objeto en el *heap* (montón) es una acción dinámica, en oposición con la ubicación estática de un objeto, ya sea globalmente o en un marco de pila, y la ubicación en el *heap* suele costar más recursos de computación. En muchos tipos de sistema, esta propiedad no causa ningún problema real, pero en aplicaciones críticas respecto al tiempo, no pueden admitirse los ciclos que se necesitan para completar una asignación del *heap*. Hay soluciones simples para este problema: o bien preasignar espacio a tales objetos durante la elaboración del programa, en vez de asignarlo durante la ejecución de algún algoritmo crítico respecto al tiempo, o bien reemplazar el gestor de memoria del sistema por defecto por uno afinado para el comportamiento de ese sistema específico.

Otra nota positiva: ciertas propiedades de los sistemas orientados a objetos a menudo ensombrecen todas estas fuentes de problemas de eficacia. Por ejemplo, Russo y Kaplan relatan que el tiempo de ejecución de un programa en C++

es en muchas ocasiones más rápido que el de su equivalente funcional en C [24]. Ellos atribuyen esta diferencia al uso de funciones virtuales, que eliminan la necesidad de algunas formas de comprobación explícita de tipos y de algunas estructuras de control. En realidad, en nuestra experiencia, los tamaños del código de sistemas orientados a objetos suelen ser menores que sus implantaciones no orientadas a objetos funcionalmente equivalentes.

En algunos proyectos, los costes de puesta en marcha asociados con el desarrollo orientado a objetos pueden manifestarse como una barrera muy real para la adopción de esta tecnología. El uso de cualquier tecnología tan novedosa como ésta requiere la capitalización de las herramientas de desarrollo del software. Además, si una organización de desarrollo está utilizando un lenguaje de programación orientada a objetos por primera vez, en general no tendrá una base establecida de software específico de la aplicación para reutilizar. En resumen, tienen que partir desde cero o al menos pensar cómo conectar sus aplicaciones orientadas a objetos con otras no orientadas a objetos ya existentes. Por último, un primer intento de usar desarrollo orientado a objetos seguramente fallará sin el entrenamiento apropiado. Un lenguaje de programación orientado a objetos no es «sólo otro lenguaje de programación» que puede aprenderse en un curso de tres días o leyendo un libro. Como se ha puesto de relieve, lleva un tiempo desarrollar la mentalidad adecuada para el diseño orientado a objetos, y esta nueva forma de pensar deben abrazarla tanto los desarrolladores como sus gestores.

Resumen

- El desarrollo y despliegue con éxito de un sistema de software complejo implica mucho más que la simple generación de código.
- Muchas de las prácticas básicas de la gestión del desarrollo de software, como los recorridos de inspección, no se ven afectadas por la tecnología orientada a objetos.
- En un estado estable, los proyectos orientados a objetos requieren típicamente una reducción de recursos durante el desarrollo; los papeles que se requieren de esos recursos son sutilmente diferentes que para sistemas no orientados a objetos.
- En el análisis y diseño orientados a objetos, nunca hay un solo evento explosivo de integración; la unidad de gestión de configuraciones para las versiones debería ser la categoría o el subsistema, no el archivo individual de una clase.
- La reutilización debe estar institucionalizada para tener éxito.
- La tasa de descubrimiento de defectos y la densidad de defectos son medidas útiles para la calidad de un sistema orientado a objetos. Otras medidas útiles incluyen varias métricas orientadas a clases.
- La documentación nunca debería dirigir el proceso de desarrollo.

- El desarrollo orientado a objetos requiere herramientas sutilmente diferentes que los sistemas no orientados a objetos.
- La transición de una organización hacia el uso del modelo de objetos requiere un cambio de mentalidad; el aprender un lenguaje de programación orientado a objetos es algo más que aprender «sólo otro lenguaje de programación».
- Existen muchas ventajas en la tecnología orientada a objetos, así como algunos riesgos; la experiencia indica que los beneficios sobrepasan con mucho a los riesgos.

Lecturas recomendadas

Van Genuchten [H 1991] y Jones [H 1992] examinan riesgos del software habituales.

Para comprender la mente del programador individual, véase Weinberg [J 1971, H 1988]. Abdel-Hamid y Madnick [H 1991] estudian la dinámica de los equipos de desarrollo.

Gilb [H 1988] y Charette [H 1989] son referencias principales para las prácticas de gestión de la ingeniería del software. El trabajo de Aron [H 1974] ofrece una mirada comprensiva a la gestión del programador individual y de equipos de programadores. Para un estudio realista de lo que de veras sucede durante el desarrollo, cuando la práctica echa a la teoría por la ventana, véanse los trabajos de Glass [G 1982], Lammers [H 1986] y Humphrey [H 1989]. DeMarco y Lister [H 1987], Yourdon [H 1989], Rettig [H 1990] y Thomsett [H 1990] ofrecen una serie de recomendaciones para el jefe de desarrollo.

Pueden encontrarse detalles sobre cómo conducir recorridos de inspección del software en Weinberg y Freedman [H 1990] y Yourdon [H 1989a].

Schulmeyer y McManus [H 1992] ofrecen una excelente referencia general sobre garantía de calidad del software. Chidamber y Kemerer [H 1991] y Walsh [H 1992, 1993] estudian el control de calidad y las métricas en el contexto de los sistemas orientados a objetos.

Se describen sugerencias sobre cómo pueden mudarse individuos y organizaciones al modelo de objetos en Goldberg [C 1978], Goldberg y Kay [G 1977] y Kempf [G 1987].

Notas bibliográficas

- [1] Dijkstra, E. May 1968. The Structure of the «THE» Multiprogramming System. *Communications of the ACM* vol. 11(5), p. 341.
- [2] Kishida, K., Teramoto, M., Torri, K., and Urano, Y. September 1988. Quality Assurance Technology in Japan. *IEEE Software* vol. 4(5), p. 13.
- [3] Hawryszkiewycz, I. 1984. *Database Analysis and Design*. Chicago, IL: Science Research Associates, p. 115.
- [4] Van Genuchten, M. June 1991. Why is Software Late? An Empirical Study of Rea-

- sons for Delay in Software Development. *IEEE Transactions on Software Engineering* vol. 17(6), p. 589.
- [5] Gilb, T. 1988. *Principles of Software Engineering Management*. Reading, Massachusetts Addison-Wesley Publishing Company, p. 73.
- [6] As quoted in Zelkowitz, M. June 1978. Perspectives on Software Engineering. *ACM Computing Surveys* vol. 10(2), p. 204.
- [7] Showalter, J. 1989. Comunicación privada.
- [8] Davis, A., Bersoff, E., and Comer, E. October 1988. A Strategy for Comparing Alternative Software Development Life Cycle Models. *IEEE Transactions on Software Engineering* vol. 14(10), p. 1456.
- [9] Goldberg, A. 1993. Comunicación privada.
- [10] Schulmeyer, G. and McManus, J. 1992. *Handbook of Software Quality Assurance*, Second Edition. New York, New York: Van Nostrand Reinhold, p. 5.
- [11] Schulmeyer, p. 7.
- [12] Schulmeyer, p. 184.
- [13] Schulmeyer, p. 169.
- [14] Walsh, J. *Preliminary Defect Data from the Iterative Development of a Large C++ Program*. Vancouver, Canada: OOPSLA'92.
- [15] Chidamber, S. and Kemerer, C. 1993. *A Metrics Suite for Object-Oriented Design*. Cambridge, Massachusetts: MIT Sloan School of Management.
- [16] Lang, K. and Peralmutter, B. November 1986. Oaklisp: an Object-Oriented Scheme with First-Class Types. *SIGPLAN Notices* vol. 21(11), p. 34.
- [17] Meyrowitz, N. November 1986. Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework. *SIGPLAN Notices* vol. 21(11), p. 200.
- [18] Kempf, R. October 1987. Teaching Object-Oriented Programming with the KEE System. *SIGPLAN Notices* vol. 22(12), p. 11.
- [19] Schmucker, K. 1986. *Object-Oriented Programming for the Macintosh*. Hasbrouk Heights, NJ: Hayden, p. 11.
- [20] Taylor, D. 1992. *Object-Oriented Information Systems*. New York, New York John Wiley and Sons.
- [21] Pinson, L. and Wiener, R. 1990. *Applications of Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley Publishing Company.
- [22] Simonian, R. and Crone, M. November/December 1988. InnovAda: True Object-Oriented Programming in Ada. *Journal of Object-Oriented Programming* vol. 1(4), p. 19.
- [23] Pascoe, G. August 1986. Elements of Object-Oriented Programming. *Byte* vol. 11(8), p. 144.
- [24] Russo, V. and Kaplan, S. 1988. A C++ Interpreter for Scheme. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association, p. 106.



Aplicaciones

Para construir una teoría, hay que saber un montón sobre los fenómenos básicos de la materia en cuestión. En teoría de la computación, simplemente no sabemos suficiente sobre esos fenómenos como para dar lecciones sobre ello de forma muy abstracta. En vez de eso, hay que enseñar más sobre los ejemplos particulares que comprendemos ahora a la perfección, y confiar en que seremos capaces de adivinar y probar principios más generales.

MARVIN MINSKY*
Form and Content in Computer Science

* Minsky, M. April, 1970. Form and Content in Computer Science. *Journal of the Association for Computing Machinery*. Vol. 17 (2), p. 197.

Adquisición de datos: Estación de monitorización del clima

Los métodos son algo maravilloso, pero desde la perspectiva del ingeniero en ejercicio, la notación o proceso más elegante que se haya ideado jamás no sirve para nada si no ayuda a construir sistemas para el mundo real. Los últimos siete capítulos no han sido más que un preludio para esta sección del libro, en la que se aplica análisis y diseño orientados a objetos a la construcción práctica de sistemas de software. En este capítulo y los cuatro siguientes, se comienza con un conjunto de requisitos del sistema y se usa entonces la notación y proceso del desarrollo orientado a objetos para conducirnos hacia una implantación. Se ha elegido un conjunto de aplicaciones de dominios muy variados, abarcando adquisición de datos, marcos de referencia, sistemas de gestión de información, inteligencia artificial y gobierno y control, cada uno de los cuales implica su propio conjunto único de problemas. Puesto que el interés está en el análisis y el diseño en vez de en la programación, no se presenta la implantación completa de todos los problemas, aunque se ofrecerán suficientes detalles para mostrar la correspondencia desde el análisis a través del diseño hasta la implantación, y para subrayar aspectos particularmente interesantes de la arquitectura del sistema.

8.1. Análisis

Definición de los límites del problema

El recuadro proporciona los requisitos para un sistema de monitorización del clima. Esta es una aplicación simple, que abarca sólo un puñado de clases. En

realidad, en un primer vistazo, el novato en el paradigma orientado a objetos puede verse tentado a emprender este problema de forma inherentemente no orientada a objetos, considerando el flujo de los datos y las diversas correspondencias entrada/salida implicadas. Sin embargo, como se verá, incluso un sistema tan pequeño como éste se presta bien a una arquitectura orientada a objetos, y el hacerlo así muestra algunos de los principios básicos del proceso de desarrollo orientado a objetos.

Requisitos de la estación de monitorización del clima

Este sistema proporcionará monitorización automática de varias condiciones climatológicas. Específicamente, debe medir:

- Velocidad y dirección del viento.
- Temperatura.
- Presión atmosférica.
- Humedad.

El sistema proporcionará también las siguientes medidas derivadas:

- Factor de enfriamiento por el viento (*wind chill*).
- Temperatura del punto de rocío (*dew point*).
- Tendencia de la temperatura.
- Tendencia de la presión atmosférica.

El sistema tendrá una forma de determinar la fecha y hora actuales, de forma que pueda informar de los valores máximos y mínimos de cualquiera de las cuatro medidas principales durante las últimas 24 horas.

El sistema tendrá una pantalla que indicará continuamente todas las medidas principales y derivadas, así como la fecha y la hora. Mediante el uso de un teclado, el usuario puede dirigir al sistema para que muestre el máximo o el mínimo en las últimas 24 horas de cualquier medida principal, junto con la hora en que se dio tal valor.

El sistema permitirá al usuario calibrar sus sensores respecto a valores conocidos, y fijar la hora y fecha actuales.

Se comienza el análisis considerando el hardware en el que debe ejecutarse el software. Este es un problema inherente al análisis de sistemas, que involucra problemas de manufacturabilidad y costes que quedan muy lejos del ámbito de este texto. Para confinar el problema y permitir así la exposición de las cuestiones del análisis y diseño del software, se harán las siguientes suposiciones estratégicas:

- Se usará un computador monotarjeta (SBC)¹ con un procesador de tipo 486².

¹ SBC: De Single Board Computer. (*N. del T.*)

² Esto puede parecer algo exagerado, pero la economía de escala es tal que un SBC de tipo 486 es sólo un poco más caro que un computador basado en un procesador de una generación anterior. El especificar hardware con

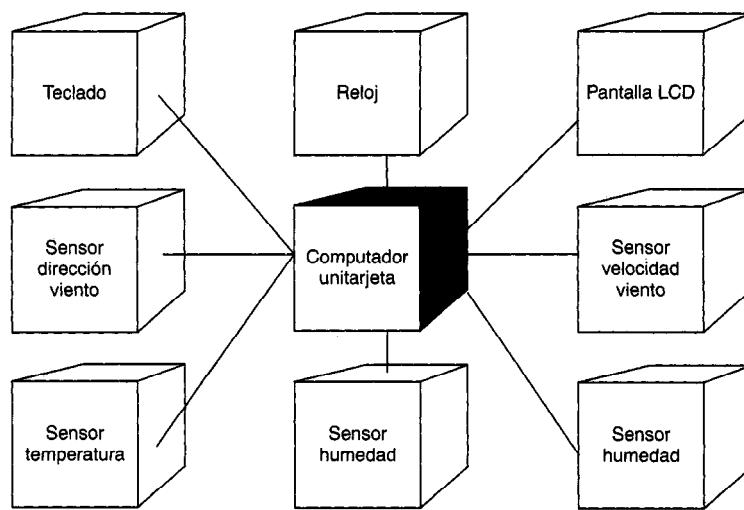


Figura 8.1. Hardware del sistema de monitorización del clima.

- La fecha y hora son suministradas por un reloj en tarjeta, accesible mediante entrada/salida por correspondencia de memoria.
- La dirección y velocidad del viento se miden con un armazón que incluye una veleta (capaz de apreciar viento de 16 direcciones distintas) y un anemómetro de cazoletas (que avanza un contador con cada revolución).
- La entrada del usuario se proporciona mediante un moderno teclado telefónico, gestionado por un circuito en tarjeta que suministra una señal sonora para cada pulsación de tecla. La última entrada del usuario está accesible mediante E/S por correspondencia de memoria.
- La pantalla es un moderno dispositivo gráfico de cristal líquido, manejado por un circuito en tarjeta capaz de procesar un conjunto simple de primitivas gráficas, como mensajes para dibujar líneas y arcos, llenar regiones y visualizar texto.
- Un temporizador en tarjeta interrumpe al computador cada 1/60 de segundo.

La Figura 8.1 proporciona un diagrama de procesos que ilustra esta plataforma hardware.

Se ha elegido poner un poco de hardware en este problema, de forma que podamos centrarnos mejor en el software del sistema. Obviamente, podría necesitarse más del software si se hiciera menos mediante el hardware (por ejemplo, eliminando algún hardware del dispositivo de entrada del usuario y del dispositivo LCD), pero ya se sabe que el cambio de la frontera hardware/software influye muy poco en nuestra arquitectura orientada a objetos. En realidad, una

capacidad excesiva significa que se puede fabricar una familia de sistemas que usan el mismo hardware, y cuyos miembros se distinguen más que nada por su software.

de las características de un sistema orientado a objetos es que tiende a hablar con el vocabulario del espacio de su problema, y representa así una máquina virtual que refleja la abstracción que se hace de las entidades clave del problema. El cambio de los detalles del hardware del sistema sólo impacta en la abstracción de sus capas inferiores.

El uso de E/S por correspondencia de memoria es bastante común para sistemas empotrados como éste, pero obviamente habría sido preferible ocultar los secretos de esta decisión particular, porque estos detalles son muy dependientes de la implantación y por tanto están sujetos a cambios. Se pueden aislar fácilmente las abstracciones software de estos detalles retorcidos envolviendo una clase alrededor de cada uno de estos interfaces. Por ejemplo, se podría idear una clase simple para acceder a la fecha y hora: se comienza por hacer un pequeño análisis de la clase aislada, en el que se considera qué papeles y responsabilidades debería abarcar esta abstracción³. Así, se podría decidir que esta clase es responsable de llevar cuenta de la hora actual en horas, minutos y segundos, así como del mes, día y año. El análisis podría decidir volcar esas responsabilidades en dos servicios, denotados por las operaciones `horaActual` y `fechaActual`, respectivamente. La operación `horaActual` devuelve una cadena de caracteres con el siguiente formato:

13 : 56 : 42

que muestra la hora, minuto y segundo actuales. La operación `fechaActual` devuelve una cadena con el siguiente formato:

16 - 6 - 94

que muestra el día, mes y año.

Un análisis posterior sugiere que una abstracción más completa permitiría a un cliente elegir un formato de 12 o de 24 horas para la hora, que puede proporcionarse en forma de un modificador adicional llamado `fijarFormato`.

Especificando el comportamiento de esta abstracción desde la perspectiva de sus clientes public, se ha establecido una separación clara entre su interfaz y su implantación. La idea básica aquí es construir la vista externa de cada clase como si se tuviese un control completo sobre su plataforma subyacente, y después implantar la clase como un puente hacia su vista interna real. Así, la implantación de una clase en la frontera hardware/software del sistema sirve para atornillar la vista externa de la abstracción a su plataforma subyacente, que muchas veces está restringida por decisiones del sistema que no está en la mano del ingeniero del software adoptar. Por supuesto, el hueco entre las vistas externa e interna de

³ En realidad, en vez de empezar por diseñar una clase nueva desde cero, se debería comenzar buscando una clase existente que ya satisficiera nuestras necesidades. Una clase de fecha y hora es ciertamente un buen candidato para la reutilización: la abstracción es tan común que es probable que alguien haya desarrollado y probado ya una clase semejante. Para el propósito de este capítulo, se asumirá que no se pudo encontrar esa clase.

una abstracción no debe ser tan ancho como para requerir una implantación densa e ineficiente para conectar ambas vistas.

El modelo de E/S por correspondencia de memoria podría proporcionar solamente acceso a la fecha y hora un entero de 16 bits, que representase el número de segundos desde que se encendió el sistema⁴. Una responsabilidad de la clase de fecha y hora debe ser por tanto el traducir estos datos en bruto a algún valor significativo. El llevar a cabo esta responsabilidad requiere un nuevo conjunto de servicios para fijar la fecha y hora, que se proporcionan mediante las operaciones `fijarHora`, `fijarMinuto`, `fijarSegundo`, `fijarDía`, `fijarMes` y `fijarAnio`⁵.

Se puede resumir la abstracción de una clase fecha/hora como sigue:

```

Nombre: FechaHora
Responsabilidades:
    Llevar cuenta de la hora y fecha actuales.
Operaciones:
    horaActual
    fechaActual
    fijarFormato
    fijarHora
    fijarMinuto
    fijarSegundo
    fijarDía
    fijarMes
    fijarAnio
Atributos:
    hora
    fecha

```

Las instancias de esta clase tienen un ciclo de vida dinámico, que puede expresarse en el diagrama de transición de estados que aparece en la Figura 8.2. Aquí se ve que en la inicialización, una instancia de esta clase reinicializa sus atributos de fecha y hora, y entra entonces incondicionalmente en el estado `Funcionando`, en el que comienza en el modo 24 horas. Una vez que está en el estado `Funcionando`, la recepción de la operación `fijarFormato` podría cambiar el estado del objeto entre los modos de 12 y 24 horas. Sin embargo, con indiferencia del estado anidado, el fijar la fecha u hora hace que el objeto renormalice sus atributos. Análogamente, el solicitar su hora o fecha hace que el objeto calcule un nuevo valor de cadena de caracteres.

Se ha especificado el comportamiento de esta abstracción con suficiente de-

⁴ Una implantación simple podría usar un temporizador hardware que avanzase un contador a cada segundo. Una implantación más sofisticada podría usar un chip de fecha y hora con una batería. En ambos casos, la vista externa de la clase presenta el mismo contrato a sus clientes. La implantación es responsable entonces de ligar este contrato al hardware.

⁵ Como se indicó en un capítulo anterior, se evita el uso de caracteres ASCII expandidos (como la letra ñ) en el código. (*N. del T.*)

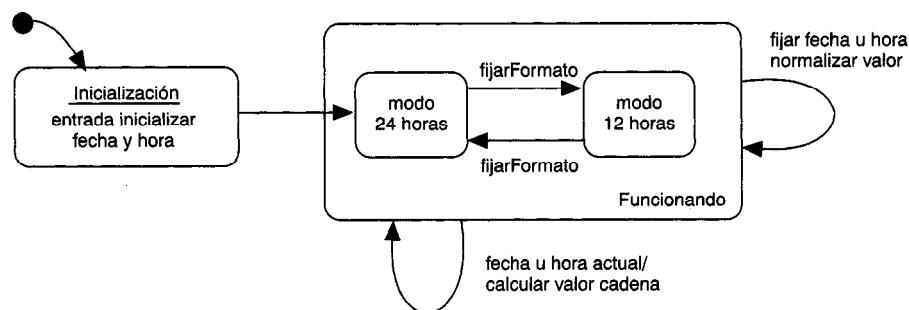


Figura 8.2. Ciclo de vida de FechaHora.

talle para ofrecer su uso en escenarios con otros clientes que puedan descubrirse durante el análisis. Antes de considerar estos escenarios, pasamos a especificar el comportamiento de los otros objetos tangibles del sistema.

La clase `SensorTemperatura` sirve como una analogía de los sensores de temperatura hardware del sistema. Un análisis de la clase aislada produce el siguiente primer grabado de la vista externa de la abstracción:

Nombre:
`SensorTemperatura`

Responsabilidades:
 Llevar cuenta de la temperatura actual.

Operaciones:
`temperaturaActual`
`fijarTemperaturaBaja`
`fijarTemperaturaAlta`

Atributos:
`temperatura`

La operación `temperaturaActual` se explica por sí misma. Las otras dos operaciones se derivan directamente de los requisitos, que obligan a proporcionar un mecanismo para calibrar cada sensor. Por el momento, se asumirá que cada valor de un sensor de temperatura se representa por un número de punto fijo, cuyos puntos bajo y alto pueden calibrarse para adecuarse a valores reales conocidos. Se traducen los números intermedios a sus temperaturas reales por interpolación lineal simple entre esos dos puntos, como se ilustra en la Figura 8.3.

El lector cuidadoso puede preguntarse por qué se ha propuesto una clase para esta abstracción, cuando los requisitos implican que hay exactamente un sensor de temperatura en el sistema. Esto no deja de ser cierto, pero en la sospecha de que pueda reutilizarse esta abstracción, se decide reflejarla como una clase, con lo que se separa de los aspectos concretos de este sistema. De hecho, el número de sensores de temperatura monitorizados por un sistema particular es bastante intrascendente para nuestra arquitectura, y diseñando una clase, se simplifica

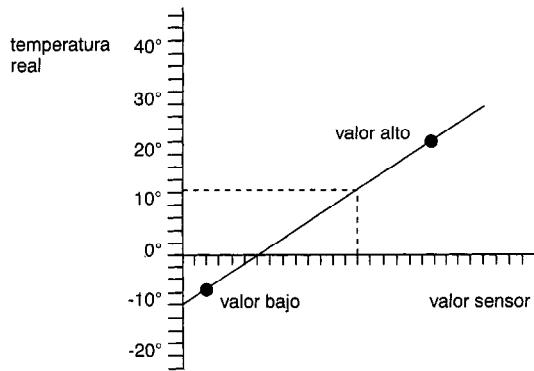


Figura 8.3. Calibración SensorTemperatura.

para otros programas de esta familia la manipulación de cualquier número de sensores.

Se puede expresar la abstracción del sensor de presión atmosférica en la siguiente especificación:

```

Nombre:
  SensorPresion
Responsabilidades:
  Llevar cuenta de la presion atmosferica actual.
Operaciones:
  presionActual
  fijarPresionBaja
  fijarPresionAlta
Atributos:
  presion

```

Una revisión de los requisitos del sistema revela que puede que se haya omitido un comportamiento importante para esta clase y la anterior, `SensorTemperatura`. En concreto, los requisitos obligan a proporcionar un medio para informar de la tendencia de la temperatura y la presión. Por ahora (ya que se está haciendo análisis, no diseño) bastará con centrarse en la naturaleza de este comportamiento y, lo que es más importante, decidir a qué abstracción habría que hacer responsable de él.

Tanto para el `SensorTemperatura` como para el `SensorPresion`, se pueden expresar las tendencias como números en punto flotante entre -1 y 1, que representan la pendiente de una línea que se adapte a una serie de valores a lo largo de algún intervalo de tiempo⁶. Así, se puede añadir la siguiente responsabilidad y su operación correspondiente a ambas clases:

⁶ Un valor de 0 significa que la temperatura o presión es estable. Un valor de 0,1 denota un crecimiento modesto; un valor de -0,3 denota valores en rápido descenso. Un valor cercano a -1 o 1 sugiere un cataclismo ambiental, que está fuera del ámbito de escenarios que el sistema se espera que maneje de forma correcta.

Responsabilidades:

Informar de la tendencia de la presión o temperatura como pendiente de una línea que se adapta a los valores pasados en un intervalo dado.

Operaciones:

tendencia

Ya que este comportamiento es común a las clases de sensores de presión y temperatura, el análisis sugiere la invención de una superclase común, que se llamará **SensorTendencia**, responsable de proporcionar este comportamiento común.

Para completar habría que apuntar que hay una visión alternativa del mundo que podría haberse elegido en este análisis. La decisión fue hacer que este comportamiento común fuese responsabilidad de la propia clase de sensor. Se podría haber decidido hacer que este comportamiento fuese parte de algún agente externo que periódicamente interrogase al sensor concreto y calculase su tendencia, pero se rechazó este enfoque, porque era innecesariamente complejo. La especificación original de las clases de sensor de presión y temperatura sugería que cada abstracción tenía suficiente conocimiento para llevar a cabo este comportamiento de información de la tendencia, y combinando responsabilidades (aunque en forma de una superclase), se acaba teniendo una abstracción simple y conceptualmente cohesiva.

La abstracción del sensor de humedad puede expresarse en la especificación siguiente:

Nombre:

SensorHumedad

Responsabilidades:

Llevar cuenta de la humedad actual, expresada como un porcentaje de saturación de 0% a 100%

Operaciones:

humedadActual

fijarHumedadBaja

fijarHumedadAlta

Atributos:

presión

El **SensorHumedad** no tiene responsabilidades de cálculo de la tendencia y, por tanto, no es una subclase de **SensorTendencia**.

Una revisión de los requisitos del sistema sugiere algún comportamiento común a las clases **SensorTemperatura**, **SensorPresion** y **SensorHumedad**. En concreto, los requisitos obligan a proporcionar un medio para informar de los valores máximos y mínimos de cada uno de estos sensores durante un período de 24 horas. Se podría capturar este comportamiento en la especificación siguiente, común a las tres clases:

Responsabilidades:

Informar de los valores máximo y mínimo en un período de 24 horas.

Operaciones:

valorMaximo

valorMinimo

horaValorMaximo

horaValorMinimo

Se aplaza la decisión sobre cómo llevar a cabo esta responsabilidad, porque es una cuestión de diseño, no de análisis. Sin embargo, ya que este comportamiento es común a las tres clases de sensor, el análisis sugiere la invención de una superclase común, que se llama `SensorHistorico`, responsable de proporcionar este comportamiento común. `SensorHumedad` es una subclase directa de `SensorHistorico`, al igual que `SensorTendencia`, que sirve como una clase abstracta intermedia, estableciendo una conexión entre las abstracciones `SensorHistorico` y las clases concretas `SensorTemperatura` y `SensorPresion`.

La abstracción del sensor de velocidad del viento puede expresarse en la especificación siguiente:

Nombre:

`SensorVelocidadViento`

Responsabilidades:

Llevar cuenta de la velocidad actual del viento.

Operaciones:

`velocidadActual`

`fijarVelocidadBaja`

`fijarVelocidadAlta`

Atributos:

`velocidad`

Los requisitos sugieren que no se puede detectar directamente la velocidad actual del viento; en vez de eso, hay que calcular su valor tomando el número de revoluciones de las cazoletas del armazón, dividiendo por el intervalo en el que se contaron esas revoluciones, y aplicando entonces un valor de escala apropiado para el armazón concreto. Ni que decir tiene que este cálculo es uno de los secretos de esta clase; a los clientes no debería preocuparles cómo se calcula `velocidadActual`, siempre y cuando esta operación satisfaga su contrato y proporcione valores significativos.

Un análisis rápido del dominio de las últimas cuatro clases concretas (`SensorTemperatura`, `SensorPresion`, `SensorHumedad` y `SensorVelocidadViento`) revela otro comportamiento común más: todas estas clases deben saber cómo calibrarse a sí mismas proporcionando una interpolación lineal entre dos puntos de datos conocidos. En vez de repetir este comportamiento en las cuatro clases, se decide que sea responsabilidad de una superclase de nivel aún más alto, que se llama `SensorCalibrado`, cuya especificación incluye lo siguiente:

Responsabilidades:

Proporcionar una interpolación lineal de valores, dados dos puntos de datos conocidos.

Operaciones:

- valorActual
- fijarValorAlto
- fijarValorBajo

SensorCalibrado es una superclase inmediata de SensorHistorico⁷.

El sensor final concreto para la dirección del viento es un poco distinto, porque no requiere ni calibración ni historia. Se puede expresar la abstracción de esta entidad en la especificación siguiente:

Nombre:

SensorDireccionViento

Responsabilidades:

Llevar cuenta de la dirección actual del viento, en términos de puntos de una rosa de los vientos.

Operaciones:

- direccionActual

Atributos:

- direccion

Para unificar las abstracciones de los sensores, se generan la clase base abstracta Sensor, que sirve como la superclase inmediata para las clases Sensor-DireccionViento y SensorCalibrado. La Figura 8.4 ilustra esta jerarquía completa.

Aunque no forma parte de la jerarquía de sensores, la abstracción del teclado para la entrada de usuario tiene una especificación simple:

Nombre:

Teclado

Responsabilidades:

Llevar cuenta de la última entrada de usuario.

Operaciones:

- ultimaTeclaPulsada

Atributos:

- clave

Nótese que esta clase no tiene conocimiento sobre lo que significa una tecla particular: las instancias de esta clase sólo saben que se pulsaron una o varias teclas. Se delega la responsabilidad de interpretar lo que significan esas teclas en una clase diferente, que se identificará cuando se apliquen esas clases límite concretas a los escenarios.

La abstracción de una clase DispositivoLCD sirve para aislar el software del

⁷ Esta jerarquía admite la piedra de toque para la herencia: un SensorTemperatura es un tipo de Sensor-Tendencia, que es también un tipo de SensorHistorico, que a su vez es un tipo de SensorCalibrado.

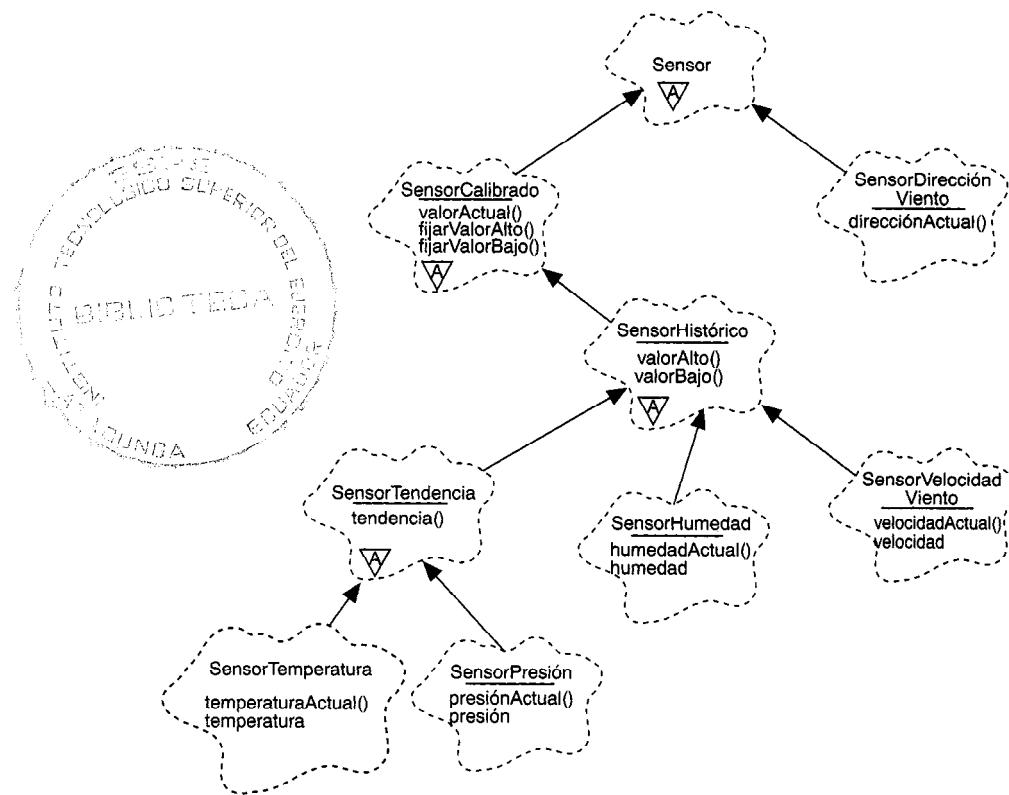


Figura 8.4. Jerarquía de clases de Sensor.

hardware particular que pueda utilizarse. Para estaciones de trabajo y computadores personales, han surgido estándares ampliamente difundidos (aunque en conflicto) para la programación de gráficos, como los interfaces de programación para Motif y para Windows de Microsoft. Desgraciadamente, para controladores empotrados no prevalecen tales estándares comunes. Para desconectar el software del hardware particular de gráficos que podría usarse, el análisis nos lleva a realizar prototipos de algunas visualizaciones habituales del sistema de monitorización del clima, y determinar entonces las necesidades del interfaz.

La Figura 8.5 ofrece tal prototipo. Aquí se ha omitido la visualización del factor de enfriamiento por el viento y temperatura de rocío como demandan los requisitos, así como detalles como el modo en que se visualiza el valor máximo y mínimo en 24 horas de las medidas principales. Sin embargo, aparecen algunos patrones: sólo se necesita visualizar texto (en dos tamaños diferentes y dos estilos diferentes), círculos y líneas (de diversos grosos). Además, se ve que algunos elementos de la pantalla son estáticos (como la etiqueta TEMPERATURA), mientras otros son dinámicos (como la dirección del viento). Se decide repre-

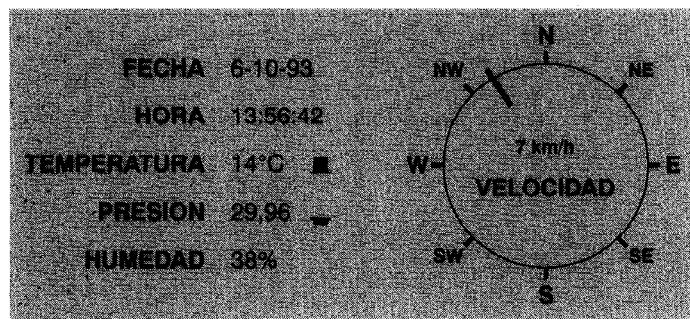


Figura 8.5. Pantalla del sistema de monitorización del clima.

sentar ambos elementos (estáticos y dinámicos) por medio del software. De este modo, se alivia la carga del hardware eliminando la necesidad de etiquetas especiales en el propio LCD, pero se exige un poco más al software.

Se pueden traducir estos requisitos en la siguiente especificación de clase:

```

Nombre:
  DispositivoLCD
Responsabilidades:
  Manejar el dispositivo LCD y ofrecer servicios para visualizar
  ciertos elementos gráficos.
Operaciones:
  dibujarTexto
  dibujarLinea
  dibujarCirculo
  fijarTamanioTexto
  fijarEstiloTexto
  fijarTamanioLapiz

```

Al igual que la clase Teclado, la clase DispositivoLCD no tiene conocimiento de lo que significan los elementos que manipula. Las instancias de esta clase sólo saben cómo mostrar texto y líneas, pero no saben lo que representan esas figuras. Esta separación de intereses da lugar a abstracciones débilmente acopladas (que es lo que se desea), pero requiere que se encuentre algún agente responsable de la mediación entre los sensores descarnados y la pantalla. Se aplaza la invención de esta nueva abstracción hasta que se estudien algunos escenarios aplicables a este sistema.

La última frontera que hay que considerar es la del temporizador. Se va a realizar la suposición simplificada de que hay exactamente un temporizador por sistema, cuyo comportamiento es interrumpir al computador cada 1/60 segundo invocando a una rutina de servicio de la interrupción. Ahora, este es un detalle particularmente específico, y sería mejor si se pudiera ocultar este detalle de implantación del resto de las abstracciones del sistema. Se puede conseguir

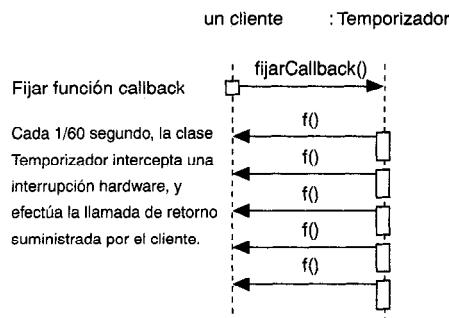


Figura 8.6. Diagrama de interacción del temporizador.

creando una clase que usa una función callback y que exporta sólo miembros static (de forma que se restringe al sistema a tener exactamente un temporizador).

La Figura 8.6 proporciona un diagrama de interacción que ilustra un caso de uso para esta abstracción. Aquí se ve cómo colaboran el temporizador y su cliente: el cliente comienza por suministrar una función callback, y cada 0,1 segundos el temporizador llama a esa función. De este modo, se libera al cliente del conocimiento sobre cómo interceptar eventos temporizados, y se libera al temporizador del conocimiento sobre qué hacer cuando sucede un evento de ese tipo. La responsabilidad principal que este protocolo otorga al cliente es simplemente que la ejecución de esta función callback debe ocupar siempre menos de 0,1 segundos, porque de otro modo el temporizador se saltará un evento.

Mediante la intercepción de eventos temporales, la clase Temporizador sirve como una abstracción activa, lo que significa que está en la raíz de un hilo de control. Se puede expresar la abstracción de esta clase en la especificación siguiente:

```

Nombre: Temporizador
Responsabilidades: Interceptar todos los eventos temporizados y despachar la función callback correspondiente.
Operaciones: fijarCallback()
  
```

Escenarios

Ahora que se han establecido las abstracciones que están en las fronteras del sistema, se continúa el análisis estudiando varios escenarios de su utilización. Se comienza enumerando una serie de casos de uso primarios, desde la perspectiva de los clientes de este sistema:

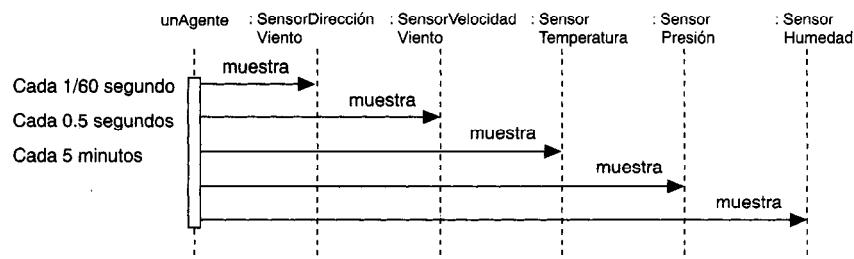


Figura 8.7. Escenario para monitorizar mediciones básicas.

- Monitorizar medidas básicas del clima incluyendo la velocidad y dirección del viento, temperatura, presión atmosférica y humedad.
- Monitorizar medidas derivadas incluyendo factor de enfriamiento por el viento, temperatura de rocío, tendencia de la temperatura y tendencia de la presión atmosférica.
- Visualizar los valores máximo y mínimo de una medida determinada.
- Fijar la fecha y hora.
- Calibrar un sensor seleccionado.
- Encender el sistema.

Se añaden a esta lista dos casos de uso secundarios:

- Fallo de energía.
- Fallo de sensor.

Examinemos varios de estos escenarios con el fin de iluminar el comportamiento —pero no el diseño— del sistema.

La monitorización de las medidas básicas del clima es el punto funcional principal del sistema de monitorización del clima. Una de las restricciones del sistema es que no se pueden tomar medidas a mayor velocidad de 60 veces por segundo. Afortunadamente, la mayoría de las características más interesantes del tiempo meteorológico cambian mucho más lentamente. El análisis sugiere que son suficientes las siguientes frecuencias de muestreo para capturar condiciones cambiantes:

- Cada 0,1 segundos dirección del viento.
- Cada 0,5 segundos velocidad del viento.
- Cada 5 minutos temperatura, presión atmosférica y humedad.

Anteriormente se decidió que las clases que representan a cada sensor principal no deberían tener la responsabilidad de tratar con eventos temporizados. El análisis requiere, por tanto, que se invente un agente externo que colabore con estos sensores para constituir este escenario. Por ahora, se aplazará la especificación del comportamiento de este agente (cómo sabe cuándo iniciar un muestreo es una cuestión de diseño, no de análisis). El diagrama de interacción de la Figura 8.7 ilustra este escenario. En él se ve que cuando el agente comienza a

muestrear, consulta por turno a cada sensor, pero se salta intencionadamente algunos de ellos con el fin de muestreárselos a una frecuencia menor. Consultando a cada sensor en vez de permitiendo que cada sensor actúe como un hilo de control, la ejecución del sistema se hace más predecible, porque el agente puede controlar el flujo de los eventos. Ya que este nombre refleja su lugar en el comportamiento del sistema, se hará que este agente sea una instancia de la clase Muestreador.

Hay que continuar este escenario preguntándose cuál de los objetos del diagrama de interacción es entonces responsable de representar los valores muestreados en la única instancia de la clase DispositivoLCD. Al final, hay que elegir entre dos opciones: se puede hacer que cada sensor sea responsable de representarse a sí mismo (el patrón habitual utilizado en arquitecturas de tipo MVC) o se puede hacer que un objeto separado se responsabilice de este comportamiento. Para este problema particular, se elige la segunda opción, porque permite encapsular en una clase todas las decisiones de diseño sobre la distribución de la pantalla⁸. Así, se añade la siguiente especificación de clase a los productos del análisis:

```
Nombre: GestorPantalla
Responsabilidades:
    Gestionar la distribución de los elementos en el dispositivo LCD.
Operaciones:
    dibujarElementosEstaticos
    dibujarHora
    dibujarFecha
    dibujarTemperatura
    dibujarHumedad
    dibujarPresion
    dibujarEnfriamientoViento
    dibujarPuntoRocio
    dibujarVelocidadViento
    dibujarDireccionViento
    dibujarMaximoMinimo
```

La operación dibujarElementosEstaticos sirve para dibujar las partes de la pantalla que no se modifican, como la rosa de los vientos que se usa para indicar la dirección del viento. Se supondrá también que las operaciones dibujarTemperatura y dibujarPresion son responsables de mostrar sus tendencias correspondientes (por tanto, a medida que se avance en la implantación habrá que proporcionar una firma adecuada para estas operaciones).

⁸ El problema dominante aquí es *dónde* se visualiza cada elemento, no *cómo* aparece a la vista. Ya que ésta es una decisión susceptible de ser modificada (y no se asume la existencia de ningún mecanismo general de gestión de recursos como suele ser habitual en IGUs como Motif y Windows), es mejor encapsular en una clase todo el conocimiento sobre dónde representar cada elemento en el dispositivo LCD. Un cambio en las suposiciones que se hacen acerca de la distribución en el panel requerirá, por tanto, que intervenga una sola clase en lugar de muchas.

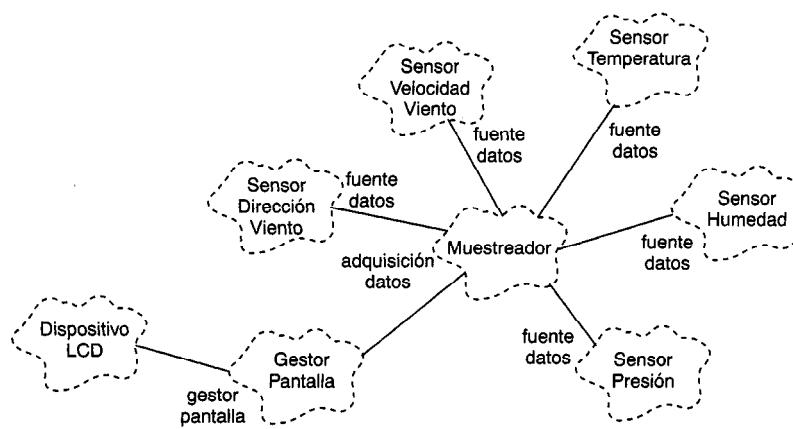


Figura 8.8. Clases de muestreo y visualización.

La Figura 8.8 proporciona un diagrama de clases que ilustra las abstracciones que deben colaborar para llevar a cabo este escenario. Nótese que también se indica el papel que juega cada abstracción en su asociación con otras clases.

Hay un importante efecto lateral de la decisión de incluir la clase `GestorPantalla`⁹. En concreto, la internacionalización del software, es decir, su adaptación a diferentes países y lenguajes, se facilita mucho con esta decisión de diseño, porque el conocimiento sobre cómo se llaman los elementos (como `TEMPERATURA` o `VELOCIDAD`) es parte de los secretos de esta clase.

La internacionalización nos lleva a considerar una cuestión sobre la que los requisitos no dicen nada: ¿debería el sistema mostrar la temperatura en grados Centígrados o Fahrenheit? Análogamente, ¿debería mostrar la velocidad del viento en kilómetros por hora (KPH) o en millas por hora (MPH)? Al fin y al cabo, el software no debería restringirnos. Ya que se busca flexibilidad para el usuario final, hay que añadir una operación `fijarModo` a ambas clases `SensorTemperatura` y `SensorVelocidadViento`. También hay que añadir una nueva responsabilidad a cada una de esas clases, que haga que sus instancias se construyan en un estado estable conocido. Por último, hay que modificar la presentación de la operación `GestorPantalla::dibujarElementosEstaticos` en consonancia, de forma que cuando se cambien las unidades de medida el gestor de pantalla pueda actualizar el panel de visualización si es necesario.

Este descubrimiento nos lleva a añadir un escenario más para someterlo a consideración del análisis, a saber:

- Fijar la unidad de medida para la temperatura y la velocidad del viento.

⁹ ¿Es ésta una decisión de análisis o una decisión de diseño? La cuestión puede argumentarse en ambas direcciones, aunque tales argumentos son considerablemente académicos cuando se trata de la necesidad de entregar software de producción. Si una decisión hace avanzar nuestra comprensión sobre el comportamiento que se desea del sistema y además nos lleva a una arquitectura elegante, no importa realmente cómo se la denomine.

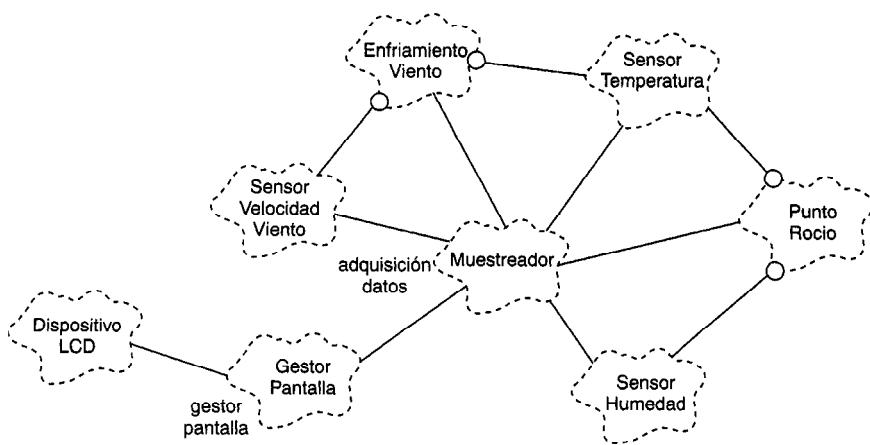


Figura 8.9. Mediciones derivadas.

Se aplazará la consideración de este escenario hasta que se estudien los otros casos de uso que tienen que ver con la interacción con el usuario.

La monitorización de las medidas derivadas de las tendencias de temperatura y presión pueden conseguirse mediante el protocolo que ya se estableció para las clases `SensorTemperatura` y `SensorPresion`. Sin embargo, para completar este escenario para todas las medidas derivadas, nos vemos en la obligación de descubrir dos nuevas clases, que se bautizan como `EnfriamientoViento` y `PuntoRocio`, responsables de calcular sus valores respectivos. Ninguna de estas abstracciones representa a un sensor, porque no denotan ningún dispositivo tangible del sistema. En lugar de eso, cada una actúa como un agente que colabora con otras dos clases para llevar a cabo sus responsabilidades. En concreto, la clase `EnfriamientoViento` conspira con las clases `SensorTemperatura` y `SensorVelocidadViento`, y la clase `PuntoRocio` conspira con las clases `SensorTemperatura` y `SensorHumedad`. A su vez, las clases `EnfriamientoViento` y `PuntoRocio` colaboran con la clase `Muestreador`, utilizando el mismo mecanismo que utiliza `Muestreador` para monitorizar todas las medidas primarias del clima. La Figura 8.9 ilustra las clases implicadas en este escenario; básicamente, este diagrama de clases es una visión del sistema sólo un poco distinta que la que se mostró en la Figura 8.8.

¿Por qué se definen `EnfriamientoViento` y `PuntoRocio` como clases, en vez de realizar sin más su cálculo mediante una simple función no miembro? La respuesta es que esta situación admite la piedra de toque para las abstracciones orientadas a objetos: las instancias de `EnfriamientoViento` y `PuntoRocio` proporcionan algún comportamiento (a saber, el cálculo de sus respectivos valores), encapsulan algún estado (cada una debe mantener una asociación con una instancia particular de dos sensores concretos diferentes), y cada una de ellas tiene una identidad única (cada asociación particular entre un sensor de velocidad del viento y un sensor de temperatura debe tener su propio objeto).

`EnfriamientoViento`). «Objetivando» estas abstracciones aparentemente algorítmicas, también se acaba teniendo una arquitectura más reutilizable: tanto `EnfriamientoViento` como `PuntoRocio` pueden despegarse de su aplicación particular, porque cada una representa un claro contrato con sus clientes, y cada una ofrece una clara separación de intereses en relación a todas las demás abstracciones.

A continuación se consideran los diversos escenarios que se relacionan con la interacción del usuario con el sistema de monitorización del clima. La decisión sobre los gestos correctos del usuario al interactuar con un controlador empotrado como éste sigue siendo un arte, como lo es el diseñar un interfaz gráfico de usuario. Un tratado completo sobre cómo diseñar tales interfaces está fuera del alcance de este texto, pero el mensaje básico para el analista del software es que la utilización de prototipos funciona, y es verdaderamente fundamental al ayudar a mitigar los riesgos que conlleva el diseño de interfaces de usuario. Además, mediante la implantación de las decisiones en términos de una arquitectura orientada a objetos, se hace relativamente fácil cambiar estas decisiones sobre interfaces de usuario sin hacer pedazos el edificio del diseño.

Considérense algunos guiones posibles que narran los escenarios de la interacción de usuario:

Visualización de los valores máximo y mínimo de una medida seleccionada.

1. El usuario pulsa la tecla SELECCIONAR.
2. El sistema muestra el mensaje SELECCIONANDO.
3. El usuario pulsa cualquiera de las teclas VELOCIDAD VIENTO, TEMPERATURA, PRESIÓN o HUMEDAD; cualquier otra pulsación (excepto EJECUTAR) se ignora.
4. El sistema hace parpadear la etiqueta correspondiente.
5. El usuario pulsa la tecla ARRIBA o ABAJO para seleccionar la visión del valor máximo o mínimo en 24 horas, respectivamente; cualquier otra pulsación de tecla (excepto EJECUTAR) se ignora.
6. El sistema muestra el valor seleccionado, junto con la hora de ocurrencia.
7. El control vuelve al paso 3 ó 5.

Nota: el usuario puede pulsar la tecla EJECUTAR para realizar o abandonar la operación, momento en el que el mensaje parpadeante, el valor seleccionado y el mensaje SELECCIONANDO se eliminan.

Este escenario nos lleva a mejorar la clase GestorPantalla añadiendo las operaciones parpadearEtiqueta (que hace que la etiqueta elegida parpadee o deje de parpadear, de acuerdo con un argumento) y mostrarModo (que muestra un mensaje de texto en la pantalla LCD).

La fijación de la fecha y hora siguen un escenario similar:

Fijar la hora y la fecha.

1. El usuario pulsa la tecla SELECCIONAR.
2. El sistema muestra el mensaje SELECCIONANDO.
3. El usuario pulsa cualquiera de las teclas HORA o FECHA; cualquier otra pulsación (excepto EJECUTAR y las teclas listadas en el paso 3 del escenario anterior) se ignora.
4. El sistema parpadea la etiqueta correspondiente; también el primer campo del elemento seleccionado (las horas para la hora y el mes para la fecha).
5. El usuario pulsa las teclas IZQUIERDA o DERECHA para seleccionar otro campo (la selección es cíclica); y pulsa ARRIBA o ABAJO para elevar o disminuir el valor del campo seleccionado.
6. El control vuelve al paso 3 ó 5.

Nota: el usuario puede pulsar la tecla EJECUTAR para realizar o abandonar la operación, momento en el que el mensaje parpadeante y el mensaje SELECCIONANDO se eliminan, y la fecha u hora se actualizan.

La calibración de un sensor particular sigue un patrón de gestos del usuario:

Calibrar un sensor.

1. El usuario pulsa la tecla CALIBRAR.
2. El sistema muestra CALIBRANDO.
3. El usuario pulsa cualquiera de las teclas VELOCIDAD VIENTO, TEMPERATURA, PRESIÓN o HUMEDAD; cualquier otra pulsación (excepto EJECUTAR) se ignora.
4. El sistema hace parpadear la etiqueta correspondiente.
5. El usuario pulsa las teclas ARRIBA o ABAJO para seleccionar el punto alto o bajo de calibración.
6. La pantalla hace parpadear el valor correspondiente.
7. El usuario pulsa las teclas ARRIBA o ABAJO para ajustar el valor seleccionado.
8. El control vuelve al paso 3 o 8.

Nota: el usuario puede pulsar la tecla EJECUTAR para realizar o abandonar la operación, momento en el que el mensaje parpadeante y el mensaje CALIBRANDO se eliminan, y se actualiza la calibración.

Mientras se calibra, hay que avisar a las instancias de la clase Muestreador para que no muestreen el elemento seleccionado, de otra forma el usuario recibiría información errónea. Este escenario requiere, por tanto, la introducción de dos nuevas operaciones para la clase Muestreador, a saber, inhibirMuestreo y reanudarMuestreo, teniendo ambas una presentación que especifica una medida particular.

El primer escenario principal que implica al interfaz de usuario concierne al establecimiento de unidades de medida:

Establecer la unidad de medida para temperatura y velocidad del viento.

Temp		Presión
<		>
Humedad		Viento
Hora		Fecha
Seleccionar	Calibrar	Modo

Figura 8.10. Teclado de usuario del sistema de monitorización del clima.

1. El usuario pulsa la tecla MODO.
 2. El sistema muestra MODO.
 3. El usuario pulsa cualquiera de las teclas VELOCIDAD VIENTO o TEMPERATURA; cualquier otra pulsación (excepto EJECUTAR) se ignora.
 4. El sistema parpadea la etiqueta correspondiente.
 5. El usuario pulsa ARRIBA o ABAJO para cambiar la unidad actual de medida.
 6. El sistema actualiza la unidad de medida para el elemento seleccionado.
 7. El control vuelve a los pasos 3 ó 5.
- Nota: el usuario puede pulsar la tecla EJECUTAR para realizar o abandonar la operación, momento en el que el mensaje parpadeante y el mensaje MODE se eliminan, y se establece la unidad de medida para ese elemento.

Un estudio de estos escenarios lleva a decidir sobre una disposición de los botones en el teclado (una decisión del sistema), que se ilustra en la Figura 8.10.

Cada uno de estos escenarios de interfaz de usuario conlleva alguna forma de comportamiento ordenado respecto a eventos, y por tanto es adecuado para su expresión mediante el uso de diagramas de transición de estados. Puesto que estos escenarios están tan estrechamente relacionados, se decide idear una nueva clase, GestorEntrada, responsable de efectuar la siguiente especificación contractual:

```

Nombre:
GestorEntrada
Responsabilidades:
Gestionar y despachar la entrada del usuario.
Operaciones:
procesarPulsacionTecla

```

Esa única operación, procesarPulsacionTecla, da vida a la máquina de estados que actúa tras las instancias de esta clase.

Como se ve en la Figura 8.11, el diagrama de transición de estados más externo para esta clase abarca cuatro estados: Funcionando, Calibrando, Seleccionando y Modo. Estos estados se corresponden directamente con los escena-

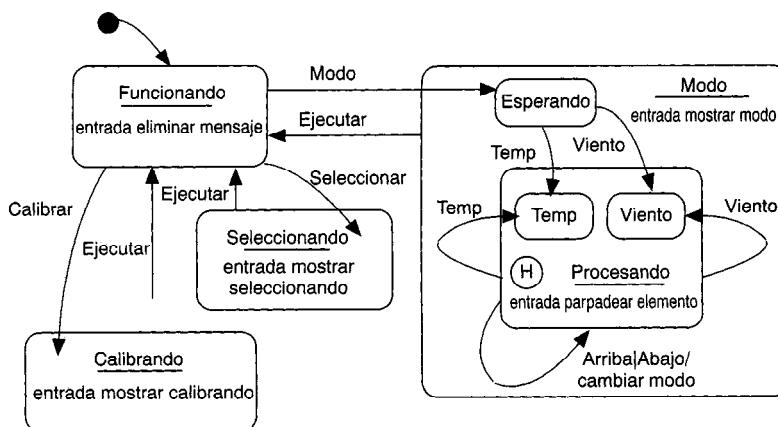


Figura 8.11. Diagrama de transición de estados del GestorEntrada.

rios anteriores. Se pasa a los respectivos estados según la primera pulsación de tecla interceptada mientras se está **Funcionando**, y se vuelve al estado **Funcionando** cuando la última pulsación de tecla vuelve a ser **Ejecutar**. Cada vez que se entra en **Funcionando**, se borra el mensaje de la pantalla.

Se ha expandido el estado **Modo** para mostrar cómo se podría expresar más formalmente la semántica dinámica del escenario. Cuando se entra por primera vez en este estado, la acción de entrada es mostrar un mensaje apropiado en la pantalla. Se comienza en el estado **Esperando**, y se sale de él si se intercepta una pulsación del usuario en las teclas **Temperatura** o **Velocidad Viento**, lo que causa la entrada en un estado anidado de **Procesando**, o una pulsación de la tecla **Ejecutar**, lo que devuelve el control al estado más externo **Funcionando**. Cada vez que se entra en **Procesando**, se hace parpadear el elemento apropiado; en entradas subsiguientes a este estado, se entra en el mismo estado anidado al que se había accedido previamente, **Temp** o **Viento**.

Mientras se está en el estado **Temp** o **Viento**, se puede interceptar una de cinco pulsaciones de tecla: **Arriba** o **Abajo** (que cambia el modo correspondiente), **Temp** o **Viento** (que hacen que se vaya al estado anidado apropiado) o **Ejecutar** (que nos saca del estado exterior **Modo**).

Los estados **Seleccionando** y **Calibrando** se expanden análogamente para revelar más estados anidados. No se mostrarán aquí sus diagramas de transición de estados expandidos, porque su presentación no revela nada particularmente interesante sobre el problema que se maneja¹⁰.

El último escenario primario involucra al encendido del sistema, que requiere que se creen todos sus objetos de forma ordenada, asegurando que cada

¹⁰ Por supuesto, para un sistema en producción, un análisis amplio completaría la exposición de este diagrama de transición de estados. Se puede aplazar esta tarea aquí, porque es más bien aburrida que otra cosa, y de hecho no revela nada que no se sepa ya sobre el sistema que se está construyendo.

uno de ellos comienza en un estado inicial estable. Se puede escribir un guión para el análisis de este escenario como sigue:

Encender el sistema

1. Se suministra energía.
2. Se construye cada sensor; los sensores históricos borran su historia, y los de tendencia preparan sus algoritmos de cálculo de pendiente.
3. El buffer de entrada de usuario se inicializa, haciendo que se descarten las pulsaciones falsas (debidas al ruido de encendido).
4. Se dibujan los elementos estáticos de la pantalla.
5. Se inicia el proceso de muestreo.

Postcondiciones: Los valores pasados máximo y mínimo de las medidas primarias se establecen con el valor y fecha de su primera muestra.

Se allanan las tendencias de presión y temperatura.

El GestorEntrada está en el estado Funcionando.

Nótese el uso de las postcondiciones en el guión para especificar el estado esperado del sistema después de que se completa este escenario. Como se verá, no hay un agente en el sistema que lleve a cabo este escenario; en vez de eso, este comportamiento resulta de la colaboración de una serie de objetos, cada uno de los cuales tiene la responsabilidad de ponerse a sí mismo en un estado inicial estable.

Esto completa el estudio de los escenarios primarios del sistema de monitorización del clima. Para que estuviese completo del todo, se podría desear hacer un recorrido a través de los diversos escenarios secundarios. En este punto, sin embargo, se ha expuesto un número suficiente de puntos funcionales del sistema, y se va a proceder con un diseño arquitectónico, de forma que puedan comenzar a validarse las decisiones estratégicas.

8.2. Diseño

Marco de referencia arquitectónica

Todo sistema de software necesita tener una filosofía organizativa simple pero potente (piénsese en ello como el equivalente software de una frase pegadiza que describe la arquitectura del sistema), y el sistema de monitorización del clima no es una excepción. El siguiente paso en el proceso de desarrollo es articular este marco de referencia arquitectónica, de forma que se pueda tener una base estable sobre la que desplegar los puntos funcionales del sistema.

En dominios de la adquisición de datos y el control de procesos, hay mu-

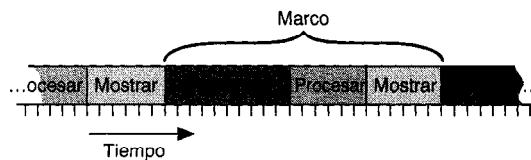


Figura 8.12. Procesamiento por marcos temporales.

chos patrones arquitectónicos posibles que se pueden seguir, pero las dos alternativas más comunes implican o bien la sincronización de actores autónomos o el procesamiento basado en marcos de tiempo.

En el primer modelo, la arquitectura abarca una serie de objetos relativamente independientes, cada uno de los cuales sirve como un hilo de control. Por ejemplo, se podrían inventar varios objetos sensor nuevos que se construyesen sobre abstracciones hardware o software más primitivas, siendo cada uno de tales objetos responsable de tomar su propia muestra y enviarla a algún agente central que procesase esas muestras. Esta arquitectura tiene sus ventajas; es casi el único marco de referencia significativo si se tiene un sistema distribuido en el que se deban recoger muestras de muchas posiciones remotas. Esta arquitectura también se presta a una mayor optimización local del proceso de muestreo (cada actor muestreador tiene el conocimiento para ajustarse a sí mismo a condiciones cambiantes, quizás incrementando o decrementando su frecuencia de muestreo según lo justifiquen las condiciones).

Sin embargo, este modelo arquitectónico no suele ser el más indicado para sistemas en tiempo real riguroso, que deben ser completamente predecibles respecto al momento en el que tienen lugar los eventos. El sistema de monitorización del clima no es de tiempo real riguroso, pero requiere una pizca de comportamiento predecible y ordenado. Por esta razón, se adopta un modelo alternativo, el del procesamiento por marcos de tiempo.

Como se ilustra en la Figura 8.12, este modelo toma el tiempo y lo divide en varios marcos (normalmente de longitud fija), que se dividirán posteriormente en submarcos, cada uno de los cuales contiene algún comportamiento funcional. La actividad puede ser distinta de un marco a otro. Por ejemplo, se podría muestrear la dirección del viento cada 10 marcos, pero muestrear la velocidad del viento sólo cada 30 marcos¹¹. La ventaja principal de este patrón arquitectónico es que se puede controlar más rigurosamente el orden de los eventos.

La Figura 8.13 ofrece un diagrama de clases que expresa esta arquitectura para el sistema de monitorización del clima. Aquí se ve la mayoría de las clases que se descubrieron antes durante el análisis, siendo la diferencia principal aquí que ahora se muestra cómo colaboran todas las abstracciones clave con las demás. Siguiendo la pauta típica en los diagramas de clases para sistemas de producción, no se muestran (y no se puede hacer) todas las clases y todas las rela-

¹¹ Por ejemplo, si a cada marco se le ha asignado 1/60 segundo, 30 marcos representan 0,5 segundos.

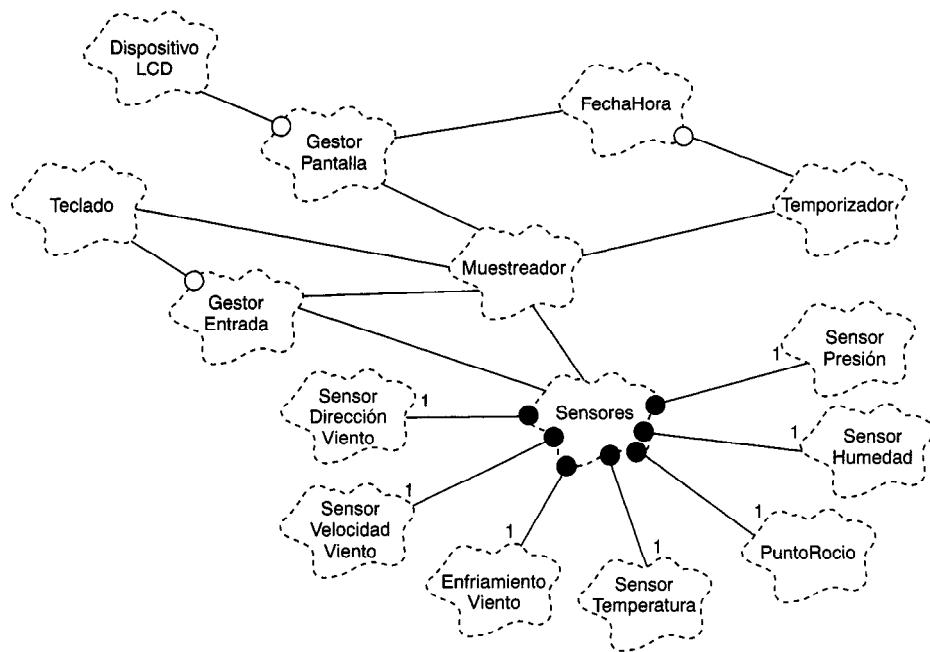


Figura 8.13. Arquitectura del sistema de monitorización del clima.

ciones. Por ejemplo, se ha omitido la jerarquía de clases que se refiere a todos los sensores.

Se ha inventado una nueva clase en esta arquitectura, que es la clase `Sensores`, cuya responsabilidad es servir como la colección de todos los sensores físicos del sistema. Puesto que al menos otros dos agentes del sistema (`Muestreador` y `GestorEntrada`) deben asociarse con la colección completa de los sensores, el empaquetarlos todos juntos en una clase contenedor permite tratar a los sensores del sistema como un todo lógico.

Mecanismo de marcos

El comportamiento central de esta arquitectura se efectúa por una colaboración de las clases `Muestreador` y `Temporizador`, y sería acertado hacer durante el diseño arquitectónico un prototipo concreto de estas clases, de forma que se puedan validar las suposiciones hechas.

Se comienza por refinar el interfaz de la clase `Temporizador` que invoca una función callback. Se pueden expresar estas decisiones de diseño en las siguientes declaraciones de clase en C++. Primero, se introduce un `typedef` que permite llamar a los tics de reloj en el vocabulario del espacio del problema:

```
// Tics de reloj, medidos en 1/60 de segundo
typedef unsigned int Tic;
```

A continuación, se presenta la clase Temporizador:

```
class Temporizador {
public:

    static fijarCallback(void (*)(Tic));
    static comenzarTemporizacion();

    static Tic numeroDeTics();

private:
    ...
};
```

Es una clase poco usual, pero recuérdese que contiene algunos secretos poco usuales. Se usa la primera función miembro `fijarCallback` para asignar una función callback al temporizador. Se lanza el comportamiento del temporizador invocando `comenzarTemporizacion`, tras lo cual la única entidad `Temporizador` llama a la función callback cada 1/60 de segundo. Nótese que se introduce una operación explícita de comienzo, porque no se puede confiar en ningún ordenamiento dependiente de la implantación en la elaboración de las declaraciones.

Antes de volverse hacia la clase `Muestreador`, se introduce una nueva declaración que sirve para nombrar los diversos sensores de este sistema particular:

```
// Enumeracion de los nombres de sensor
enum NombreSensor {Direccion, Velocidad, EnfriamientoViento,
                    Temperatura, PuntoRocio, Humedad, Presion};
```

Se puede expresar el interfaz de la clase `Muestreador` como sigue:

```
class Muestreador {
public:

    Muestreador();
    ~Muestreador();

    void fijarFrecMuestreo(NombreSensor, Tic);
    void muestrear(tic);

    Tic frecMuestreo() const;

protected:
    ...
```

```
};
```

Se ha introducido el modificador `fijarFrecMuestreo` y su selector `frecMuestreo` para que los clientes puedan alterar dinámicamente el comportamiento de los objetos muestreadores.

Para conectar las clases `Temporizador` y `Muestreador`, sólo se necesita un poco de código de pegamento. Primero se declara una instancia de `Muestreador` y una función no miembro:

```
Muestreador muestreador;

void adquirir(Tic t)
{
    muestreador.muestrear(t);
}
```

Y ahora se puede escribir un fragmento de la función principal, que simplemente asigna la función callback al temporizador y comienza el proceso de muestreo:

```
main() {
    Temporizador::fijarCallback(adquirir);
    Temporizador::comenzarTemporizacion();

    while(1) {
        ;
    }

    return 0;
}
```

Este es un programa principal claramente típico en sistemas orientados a objetos: es corto (porque el trabajo real se delega en objetos clave del sistema), y lleva un bucle de despacho (que en este caso no hace nada, porque no se tiene procesamiento de fondo que completar)¹².

Para continuar este camino de la arquitectura del sistema, se proporciona a continuación un interfaz para la clase `Sensores`. Por ahora, se supone la existencia de las diversas clases de sensores concretos:

```
class Sensores : protected Coleccion {
public:
```

¹² Este otro patrón arquitectónico más: los bucles de despacho aparecen en la mayoría de los sistemas IGU, en los que el bucle sirve para interceptar eventos externos o internos y despacharlos mediante los agentes apropiados.

```

Sensores();
virtual ~Sensores();

void aniadirSensor(const Sensor& NombreSensor, unsigned
int id=0);

unsigned int numeroDeSensores() const;
unsigned int numeroDeSensores(NombreSensor);
Sensor& sensor(NombreSensor, unsigned int id=0);

protected:
...
};

```

Esta es básicamente una clase colección, y por esta razón se hace que `Sensores` sea una subclase de la clase fundamental `Coleccion`¹³. Se hace a `Coleccion` una superclase `protected`, porque no se quiere exponer la mayoría de sus operaciones a los clientes de la clase `Sensores`. La declaración de `Sensores` proporciona sólo un escaso conjunto de operaciones, porque el problema está lo bastante restringido para saber que los sensores sólo se añaden y nunca se eliminan de la colección.

Se ha inventado una clase de colección generalizada de sensores que puede contener múltiples instancias del mismo tipo de sensor, distinguiéndose cada instancia dentro de su clase por un identificador único, que se numera empezando por cero.

Hay que revisar la especificación de la clase `Muestreador` con el fin de efectuar su asociación con las clases `Sensores` y `GestorPantalla`:

```

class Muestreador {
public:

    Muestreador(Sensor&, GestorPantalla&);
    ...
protected:
    Sensor& repSensores
    GestorPantalla& repGestorPantalla;
};

```

También hay que revisar la declaración de la única instancia de la clase `Muestreador`:

```

Sensores sensores;
GestorPantalla pantalla;

Muestreador muestreador(sensores, pantalla);

```

¹³ Las clases básicas se discuten con detalle en el siguiente capítulo.

La construcción del objeto `Muestreador` conecta este agente con la colección específica de sensores y el gestor de pantalla particular que se usan en el sistema.

Ahora se puede implantar la operación clave de la clase `Muestreador`, `Muestrear`:

```
void Muestreador::muestrear(Tic t)
{
    for (NombreSensor nombre = Direccion; nombre
         <= Presion; nombre++)
        for (unsigned int id = 0; id
             < repSensores.numeroDeSensores(nombre); id++)
            if (!(t % freqMuestreo(nombre)))
                repGestorPantalla.dibujar(repSensores.sensor
                                              (nombre,id).valorActual(),
                                              nombre, id);
}
```

La acción de esta función miembro es iterar a través de cada tipo de sensor y, a su vez, cada sensor concreto de ese tipo en la colección. Para cada sensor que encuentra, `muestrear` comprueba si es el momento de muestrear su valor, y si es así referencia al sensor de la colección, toma su valor actual y entrega este valor al gestor de pantalla asociado con la instancia `Muestreador`¹⁴.

La semántica de esta operación se apoya en el comportamiento polimórfico de una operación, a saber:

```
virtual float valorActual();
```

definida para la clase base `Sensor`. Esta operación también se basa en la operación siguiente:

```
void dibujar(float, NombreSensor, unsigned int id = 0);
```

definida para la clase `GestorPantalla`.

Ahora que se ha refinado este elemento de la arquitectura, se presenta un nuevo diagrama de clases en la Figura 8.14 que subraya este mecanismo de marcos.

8.3. Evolución

Planificación de versiones

Ahora que se ha validado la arquitectura pasando a través de varios escenarios, se puede proceder con el desarrollo incremental de los puntos funcionales del

¹⁴ Un enfoque alternativo sería que cada sensor proporcionase una función miembro que devolviese su frecuencia de muestreo y otra función miembro que dibujase el sensor en el LCD. Este diseño haría la implantación de la clase `Muestreador` más simple y extensible, aunque desplazaría más responsabilidades hacia las clases `Sensor`.

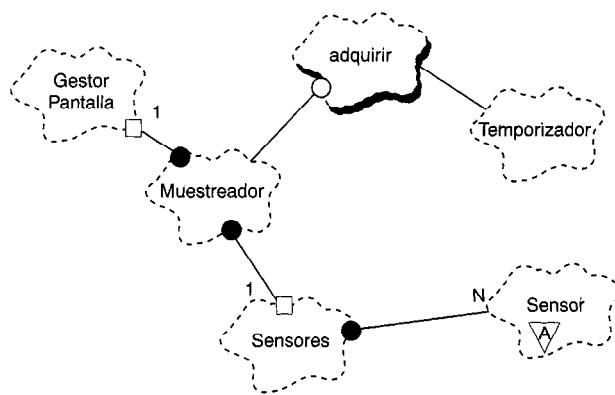


Figura 8.14. Mecanismos de marcos.

sistema. Se comienza este proceso proponiendo una secuencia de versiones, cada una de las cuales se construye sobre la anterior:

- Desarrollar una versión con funcionalidad mínima, que monitorice sólo un sensor.
- Completar la jerarquía de sensores.
- Completar las clases responsables de gestionar la pantalla.
- Completar las clases responsables de gestionar el interfaz de usuario.

Se podría haber ordenado estas versiones de muchísimas formas, pero se elige esta en orden de mayor a menor riesgo, forzando así al proceso de desarrollo a atacar directamente los problemas difíciles en primer lugar.

El desarrollo de la versión de funcionalidad mínima fuerza a tomar un corte vertical a través de la arquitectura, y a implantar partes pequeñas de prácticamente todas las abstracciones clave. Esta actividad se enfrenta al riesgo más alto del proyecto, a saber, si se tienen o no las abstracciones correctas con los papeles y responsabilidades correctos. Esta actividad proporciona también una realimentación temprana, porque ahora se puede jugar con un sistema ejecutable. En realidad, como se discutió en el Capítulo 7, el forzar a una terminación temprana como esta tiene una serie de ventajas técnicas y sociales. Por parte técnica, obliga a comenzar a atornillar entre sí las partes hardware y software del sistema, identificando así pronto cualquier desadaptación de impedancias. En el aspecto social, permite obtener una realimentación temprana sobre el aspecto y sensación del sistema, desde la perspectiva de los usuarios reales.

Puesto que el completar esta versión es en gran medida cuestión de implantación táctica (el llamado bloqueo y forcejeo que todo equipo de desarrollo debe hacer), no vamos a molestar más con la exposición de cualquier otra cosa sobre su estructura. Sin embargo, vamos a centrarnos ahora en los elementos de versiones posteriores, porque revelan algunas interioridades interesantes sobre el proceso de desarrollo.

Mecanismo de los sensores

Al inventar la arquitectura para este sistema, ya se ha visto cómo ha habido que desplegar iterativa e incrementalmente la abstracción de las clases de sensor, despliegue que había comenzado durante el análisis. En esta versión evolutiva, se espera trabajar sobre lo que previamente se había completado sobre un sistema funcional mínimo, y dar término a los detalles de esta jerarquía de clases.

En este punto del ciclo de desarrollo, la jerarquía de clases que se presentó por primera vez en la Figura 8.4 permanece estable aunque (y esto no es sorprendente) hubo que ajustar la ubicación de ciertas operaciones polimórficas, con el fin de extraer más aspectos comunes. En concreto, en una sección anterior se advirtió de la necesidad de la operación `valorActual`, declarada en la clase base abstracta `Sensor`. Se puede completar el diseño de esta clase escribiendo la siguiente declaración en C++:

```
class Sensor {
public:

    Sensor(NombreSensor, unsigned int id = 0);
    virtual ~Sensor();

    virtual float valorActual() = 0;
    virtual float valorBruto() = 0;

    NombreSensor nombre() const
    unsigned int id() const;

protected:
    ...
};
```

Esta es una clase abstracta porque incluye funciones miembro virtuales puras.

Nótese que mediante el constructor de la clase, se dio a las instancias de esta clase un conocimiento sobre su nombre e identificación. Esta es esencialmente una forma de identificación de tipos en tiempo de ejecución, pero aquí es inevitable proporcionar esta información, porque los requisitos exigen que cada instancia de sensor tenga asignada una dirección de E/S por correspondencia de memoria. Se pueden ocultar los secretos de esta correspondencia haciendo que esta dirección sea función del nombre e identificación del sensor.

Ahora que se ha añadido esta nueva responsabilidad, se puede volver atrás y simplificar la presentación de `GestorPantalla::dibujar` para tomar sólo un argumento simple, a saber, una referencia a un objeto `sensor`. Se pueden eliminar los otros argumentos de esta función miembro, porque el gestor de pantalla puede preguntar ahora al objeto sensor su nombre e identificación.

Es aconsejable hacer este cambio, porque simplifica ciertos interfaces entre clases. En realidad, si no se consigue sostener la arquitectura con cambios pe-

queños que se propaguen como éste, esa arquitectura sufrirá eventualmente una corrupción del software, en la que los protocolos entre clases que colaboran llegan a aplicarse inconsistentemente.

La declaración de la subclase inmediata `SensorCalibrado` se construye sobre esta clase base:

```
class SensorCalibrado : public Sensor {
public:

    SensorCalibrado(NombreSensor, unsigned int id = 0);
    virtual ~SensorCalibrado();

    void fijarValorAlto(float, float);
    void fijarValorBajo(float, float);

    virtual float valorActual();
    virtual float valorBruto() = 0;

protected:
    ...
};
```

Esta clase introduce dos operaciones nuevas (`fijarValorAlto` y `fijarValorBajo`), e implanta la función pura declarada previamente, `valorActual`.

A continuación, considérese la declaración de la subclase `SensorHistorico`, que se construye sobre la clase `SensorCalibrado`:

```
class SensorHistorico : public SensorCalibrado {
public:

    SensorHistorico(NombreSensor, unsigned int id = 0);
    virtual ~SensorHistorico();

    float valorMaximo() const;
    float valorMinimo() const;
    const char* horaDeValorMaximo() const;
    const char* horaDeValorMinimo() const;

protected:
    ...
};
```

Esta clase introduce cuatro operaciones nuevas, cuya implantación requiere colaboración con la clase `FechaHora`. Nótese que `SensorHistorico` sigue siendo una clase abstracta, porque aún no se ha completado la definición de la función virtual pura `valorBruto`, que se aplaza como responsabilidad de una subclase concreta.

La clase `SensorTendencia` hereda de `SensorHistorico`, y añade una responsabilidad nueva:

```
class SensorTendencia : public SensorHistorico {
public:

    SensorTendencia(NombreSensor, unsigned int id = 0);
    virtual ~SensorTendencia();

    float tendencia() const;

protected:
    ...
};
```

Esta clase introduce una nueva función miembro. Al igual que con algunas de las demás operaciones nuevas que han añadido ciertas clases intermedias, se declara `tendencia` como no virtual, porque no se desea que las subclases cambien su comportamiento.

Finalmente, se alcanzan subclases concretas como `SensorTemperatura`:

```
class SensorTemperatura : public SensorTemperatura {
public:

    SensorTemperatura(unsigned int id = 0);
    virtual ~SensorTemperatura();

    virtual float valorBruto();
    float temperaturaActual();

protected:
    ...
};
```

Nótese que la signatura del constructor de esta clase es ligeramente diferente de la de sus superclases, simplemente porque a este nivel de abstracción se conoce el nombre específico de la clase. Nótese también que se ha introducido la operación `temperaturaActual`, que se sigue del análisis anterior. Esta operación es semánticamente la misma que la función polimórfica `valorActual`, pero se decide incluir ambas porque la operación `temperaturaActual` es un poco más segura respecto al tipo.

Una vez que se ha completado con éxito la implantación de todas las clases de esta jerarquía y se las ha integrado con la versión previa, se puede pasar al siguiente nivel de la funcionalidad del sistema.

Mecanismo de visualización

La implantación de la siguiente versión, que completa la funcionalidad de las clases `GestorPantalla` y `DispositivoLCD`, prácticamente no requiere nuevo trabajo de diseño, sólo algunas decisiones tácticas sobre la presentación y semántica de ciertas funciones miembro. Combinando las decisiones que se tomaron durante el análisis del primer prototipo arquitectónico, en el que se adoptaron algunas posiciones importantes respecto al protocolo para visualizar valores de los sensores, se deriva el siguiente interfaz concreto en C++:

```
class GestorPantalla {
public:

    GestorPantalla();
    ~GestorPantalla();

    void borrar();
    void refrescar();
    void dibujar(Sensor&);
    void dibujarElementosEstaticos(EscalaTemperatura,
        EscalaVelocidad);
    void dibujarHora(const char*);
    void dibujarFecha(const char*);
    void dibujarTemperatura(float, unsigned int id = 0);
    void dibujarHumedad(float, unsigned int id = 0);
    void dibujarPresion(float, unsigned int id = 0);
    void dibujarEnfriamientoViento(float, unsigned int id = 0);
    void dibujarPuntoRocio(float, unsigned int id = 0);
    void dibujarVelocidadViento(float, unsigned int id = 0);
    void dibujarDireccionViento(unsigned int, unsigned int id = 0);
    void dibujarMaximoMinimo(float, const
        char*, NombreSensor, unsigned int id=0);
    void fijarEscalaTemperatura(EscalaTemperatura);
    void fijarEscalaVelocidad(EscalaVelocidad);

protected:
    // ...
};
```

Ninguna de estas operaciones es virtual, porque ni se esperan ni se desean subclases.

Nótese que esta clase exporta varias operaciones primitivas (como `dibujarHora` y `refrescar`), pero también ofrece la operación compuesta `dibujar`, cuya presencia simplifica en gran medida la acción de los clientes que deban interactuar con instancias del `GestorPantalla`.

El `GestorPantalla` en última instancia utiliza los recursos de la clase `DispositivoLCD`, que como se describió anteriormente sirve como una piel so-

bre el hardware subyacente. De este modo, el GestorPantalla eleva el nivel de abstracción proporcionando un protocolo que se dirige directamente a la naturaleza del espacio del problema.

Mecanismo de interfaz de usuario

El objetivo de la última versión principal es el diseño táctico e implantación de las clases Teclado y GestorEntrada. Al igual que con la clase DispositivoLCD, la clase Teclado sirve como una piel sobre el hardware subyacente, que releva por tanto al GestorEntradas de los detalles sucios del diálogo directo con el hardware. El desgajar esas dos abstracciones también facilita el reemplazar el dispositivo físico de entrada sin desestabilizar la arquitectura.

Se comienza con una declaración que da nombre a las teclas físicas en el vocabulario del espacio del problema:

```
enum Tecla {tEjecutar, tSeleccionar, tCalibrar, tModo, tArriba,
            tAbajo, tIzquierda, tDerecha, tTemperatura, tPresion,
            tHumedad, tViento, tHora, tFecha, tNoasignada};
```

Se usa el prefijo `t` para evitar conflictos de nombres con literales definidos en `NombreSensor`.

A continuación, se puede capturar la abstracción de la clase Teclado como sigue:

```
class Teclado {
public:

    Teclado();
    ~Teclado();

    int entradaPendiente() const;
    Tecla ultimaPulsacionTecla() const;

protected:
    ...
};
```

El protocolo de esta clase se deriva del análisis anterior. Se ha añadido la operación `entradaPendiente` para que los clientes puedan consultar si existe una entrada del usuario que aún no se ha procesado.

La clase GestorEntrada tiene un interfaz igualmente poco denso:

```
class GestorEntrada {
public:

    GestorEntrada(Teclado&);
```

```

~GestorEntrada();

void procesarPulsacionTecla();

protected:
    Teclado& repTeclado;
};

```

Como se verá, la mayoría del trabajo interesante de esta clase se realiza en la implantación de su máquina de estados finitos.

Como se ilustró en el Figura 8.13, las instancias de la clase `Muestreador`, `GestorEntrada` y `Teclado` colaboran para responder a la entrada del usuario. Para integrar estas tres abstracciones, hay que modificar sutilmente el interfaz de la clase `Muestreador` para incluir un nuevo objeto miembro, `repGestorEntrada`:

```

class Muestreador {
public:

    Muestreador(Sensor &, GestorPantalla&, GestorEntrada&);

    ...

protected:
    Sensores& repSensores;
    GestorPantalla& repGestorPantalla;
    GestorEntrada& repGestorEntrada;
};

```

Mediante esta decisión de diseño se establece una asociación entre instancias de las clases `Sensores`, `GestorPantalla` y `GestorEntrada` en el momento en que se construye una instancia de `Muestreador`. Usando referencias, se establece que las instancias de `Muestreador` siempre deben tener una colección de sensores, un gestor de pantalla y un gestor de entrada. Una representación alternativa que utilizase punteros proporcionaría una asociación más débil, permitiendo a `Muestreador` omitir uno o más de sus componentes.

También hay que modificar incrementalmente la implantación de la función miembro clave `Muestreador::muestrear`

```

void Muestreador::muestrear(Tic t)
{
    repGestorEntrada.procesarPulsacionTecla();
    for (NombreSensor nombre=Direccion; nombre <= Presion;
        nombre++)
        for (unsigned int id = 0; id <
            repSensores.numeroDeSensores(nombre); id++)
            if (! (t % frecMuestreo(nombre)))

```

```

        repGestorPantalla.dibujar(repSensores.sensor
        (nombre, id));
    }
}

```

Aquí se ha añadido una llamada a `procesarPulsacionTecla` al comienzo de cada marco de tiempo.

La operación `procesarPulsacionTecla` es el punto de entrada a la máquina de estados finitos que controla las instancias de esta clase. En última instancia, pueden adoptarse dos aproximaciones para implantar ésta o cualquiera otra máquina de estados finitos: se puede representar explícitamente a los estados como objetos (y, por tanto, depender de su comportamiento polimórfico) o se pueden usar literales de enumeración para denotar cada estado distinto.

Para máquinas de estados finitos de tamaño modesto como la que incorpora la clase `GestorEntrada`, es suficiente utilizar el segundo enfoque. Así, podrían introducirse primero los nombres de los estados más exteriores de la clase:

```

enum EstadoEntrada { Funcionando, Seleccionando,
                     Calibrando, Modo};

```

A continuación, se introducen algunas funciones de apoyo `protected`:

```

class GestorEntrada {
public:
    ...
protected:
    Teclado& repTeclado;
    EstadoEntrada repEstado;

    void entrarSeleccionando();
    void entrarCalibrando();
    void entrarModo();
};

```

Finalmente, se pueden comenzar a implantar las transiciones de estado que se introdujeron en la Figura 8.11:

```

void GestorEntrada::procesarPulsacionTecla()
{
    if (repTeclado.entradaPendiente()) {
        Tecla tecla = repTeclado.ultimaPulsacionTecla();
        switch (repEstado) {
            case Funcionando:
                if (tecla == tSeleccionar)
                    entrarSeleccionando();
                else if (tecla == tCalibrar)
                    entrarCalibrando();
                else if (tecla == tModo)

```

```

        entrarModo();
        break;
    case Seleccionando:
        ...
        break;
    case Calibrando:
        ...
        break;
    case Modo:
        ...
        break;
    {
}
{
}

```

La implantación de esta función miembro y sus funciones de apoyo asociadas reproduce así el diagrama de transición de estados de la Figura 8.11.

8.4. Mantenimiento

La implantación completa de este sistema básico de monitorización del clima es de tamaño modesto, y abarca sólo alrededor de 20 clases. Sin embargo, para cualquier pieza de software verdaderamente útil, el cambio es inevitable. Considerese el impacto de dos mejoras a la arquitectura de este sistema.

El sistema proporciona hasta ahora la monitorización de muchas condiciones climatológicas interesantes, pero se puede descubrir pronto que los usuarios quieren medir también las precipitaciones. ¿Cuál es el impacto de añadir un pluviómetro?

Felizmente, no hay que alterar radicalmente la arquitectura; simplemente hay que aumentarla. Utilizando como línea base la visión arquitectónica del sistema que aparecía en la Figura 8.13, para implantar esta característica hay que:

- Crear una nueva clase `SensorPluviosidad` e insertarlo en el lugar adecuado en la jerarquía de clases de sensor (un `SensorPluviosidad` es un tipo de `SensorHistorico`).
- Actualizar la enumeración `Nombresensor`.
- Actualizar el `GestorPantalla` para que sepa cómo mostrar valores de este sensor.
- Actualizar el `GestorEntrada` para que sepa cómo evaluar la nueva tecla `Pluviosidad`.
- Añadir correctamente instancias de esta clase a la colección `sensores` del sistema.

Es necesario tratar algunas otras pequeñas cuestiones tácticas para insertar esta

nueva abstracción, pero al fin y al cabo no se necesita romper la arquitectura del sistema ni sus mecanismos clave.

Considérese un tipo de funcionalidad totalmente diferente; supóngase que se desea la capacidad de remitir un registro diario de las condiciones climatológicas a un computador remoto. Para implantar esta característica, hay que realizar los siguientes cambios:

- Crear una nueva clase `PuertoSerie`, responsable de manejar un puerto RS232 utilizado para la comunicación en serie.
- Inventar una nueva clase `GestorInforme` responsable de recoger la información necesaria para el envío. Básicamente, esta clase debe utilizar los recursos de la clase de colección `Sensores` junto con sus sensores concretos asociados.
- Modificar la implantación de `Muestreador:muestrear` para atender periódicamente al puerto serie.

Es propio de los sistemas orientados a objetos bien construidos que al hacer un cambio como este no se haga pedazos la arquitectura existente, sino que se reutilicen y aumenten sus mecanismos existentes.

Lecturas recomendadas

Los problemas de sincronización de procesos, interbloqueo, bloqueo activo y condiciones de carrera se discuten en detalle en Hansen [H 1977], Ben-Ari [H 1982] y Holt *et al.* [H 1978]. Mellichamp [H 1983], Glass [H 1983] y Foster [H 1981] ofrecen referencias generales sobre los problemas de desarrollar aplicaciones en tiempo real. Pueden encontrarse textos sobre la concurrencia vista como la interacción entre hardware y software en Lorin [H 1972].

Marcos de referencia: Biblioteca básica de clases

Un beneficio importante de los lenguajes de programación orientados a objetos como C++ y Smalltalk es el grado de reutilización que puede conseguirse en los sistemas bien diseñados. Un alto grado de reutilización significa que hay que escribir mucho menos código para cada nueva aplicación; consecuentemente, hay mucho menos código que mantener.

En última instancia, la reutilización del software puede adoptar muchas formas: se pueden reutilizar líneas individuales de código, clases específicas o sociedades de clases relacionadas lógicamente. La reutilización de líneas de código individuales es la forma más simple (¿qué programador no ha usado un editor para copiar la implantación de algún algoritmo y pegarla en otra aplicación?) pero ofrece los menores beneficios (porque hay que repetir el código en varias aplicaciones). Se puede hacer mucho mejor si se usan lenguajes orientados a objetos y se toman clases existentes, especializándolas o aumentándolas mediante herencia. Se puede conseguir un aprovechamiento aún mayor reusando conjuntos enteros de clases organizadas en un marco de referencia. Como se discutió en el Capítulo 4, un marco de referencia es una colección de clases que proporcionan un conjunto de servicios para un dominio particular; exporta así una serie de mecanismos y clases individuales que los clientes pueden usar o adaptar.

Los marcos de referencia pueden, en realidad, ser neutrales respecto al dominio, lo que significa que pueden aplicarse a una amplia gama de aplicaciones. Las bibliotecas básicas generales, bibliotecas matemáticas y bibliotecas para interfaces gráficos de usuario entran en esta categoría. Los marcos de referencia pueden también ser específicos para un dominio de aplicación vertical concreto, como para registros de pacientes de un hospital, comercio de valores y obligaciones, gestión general de negocios y sistemas de comutación telefónica. Donde exista una familia de programas que resuelvan problemas sustancialmente similares, hay una oportunidad para un marco de referencia de aplicaciones.

En este capítulo, se aplica tecnología orientada a objetos a la creación de una biblioteca básica de clases¹. En el capítulo anterior, el corazón del problema lo constituyeron los problemas del control en tiempo real y la distribución inteligente del comportamiento entre entre varios objetos autónomos y relativamente estáticos. En el problema actual, dominan dos problemas muy diferentes: el deseo de una arquitectura adaptable que ofrezca un rango de alternativas de espacio y de tiempo, y la necesidad de mecanismos generales para gestión de almacenamiento y sincronización.

9.1. Análisis

Definición de los límites del problema

El recuadro de texto proporciona los requisitos detallados para esta biblioteca básica de clases. Desgraciadamente, estos requisitos son más bien abiertos: una biblioteca que proporcionase abstracciones para todas las clases fundamentales necesarias en todas las aplicaciones posibles sería enorme. La tarea del analista, por tanto, requiere una poda juiciosa del espacio del problema, de forma que el problema que quede sea abordable. Un problema como éste podría sufrir con facilidad la parálisis del análisis, y por eso hay que centrarse en proporcionar abstracciones y servicios de biblioteca que sean del uso más general, en vez de intentar hacer que este marco de referencia sea todo para todo el mundo (seguramente acabaría sin proporcionar nada útil a nadie). Se comienza con un análisis del dominio, examinando primero la teoría de algoritmos y estructuras de datos, y recogiendo entonces abstracciones que se hayan encontrado en programas en producción.

Para buscar sus apoyos teóricos, se puede buscar el conocimiento de expertos del dominio, como el que se refleja en el trabajo seminal de Knuth [2], así como de otros maestros de este campo, entre los que destacan Aho, Hopcroft y Ullman [3], Kernighan y Plauger [4], Sedgewick [5], Stubbs y Webre [6], Tenenbaum y Augenstein [7] y Wirth [8]. A medida que se continúe el estudio, se pueden recoger instancias específicas de abstracciones fundamentales, como colas, pilas y grafos, así como algoritmos para ordenación rápida, emparejamiento de patrones por expresiones regulares y búsqueda en orden en árboles.

Una cosa que se descubre en este análisis es la clara separación de las abstracciones estructurales (como colas, pilas y grafos) en contraposición a las abstracciones algorítmicas (como ordenación, emparejamiento de patrones y búsqueda). Las entidades de la primera categoría son candidatas obvias a ser clases. Las entidades de la segunda categoría pueden no parecer tratables mediante una descomposición orientada a objetos, en una primera mirada. Sin embargo, con

¹ La arquitectura de marco de referencia descrita en este capítulo es la de los Booch Components en C++ [1].

Requisitos de la biblioteca de clases básicas

Esta biblioteca de clases debe proporcionar una colección de estructuras de datos independientes del dominio suficientes para cubrir las necesidades de la mayoría de las aplicaciones C++ con calidad de producción. Además, la biblioteca debe ser:

- Completa La biblioteca debe proporcionar una familia de clases, unidas por un interfaz compartido pero empleando cada una representación diferente, de forma que los desarrolladores puedan seleccionar las que tengan la semántica de tiempo y espacio más apropiada para la aplicación que se esté tratando.
- Adaptable Todos los aspectos específicos de la plataforma deben estar claramente identificados y aislados, de forma que puedan realizarse sustituciones locales. En particular, los desarrolladores deben tener control sobre políticas de gestión del almacenamiento, así como sobre la semántica de la sincronización de procesos.
- Eficiente Los componentes deben ser fáciles de ensamblar (eficientes en términos de recursos de compilación), imponer la menor sobrecarga en memoria y tiempo de ejecución (eficiente en recursos de ejecución) y ser más fiable que los mecanismos creados a mano (eficiente en recursos del desarrollador).
- Segura Cada abstracción debe ser segura respecto al tipo, de forma que las suposiciones estáticas sobre el comportamiento de una clase puedan reforzarse por el sistema de compilación. Deberían usarse excepciones para identificar condiciones bajo las cuales se viola la semántica dinámica de una clase; el elevar una excepción no debe corromper el estado del objeto que lo hizo.
- Simple La biblioteca debe usar una organización clara y consistente que facilite la identificación y selección de clases concretas apropiadas.
- Extensible Los desarrolladores deben ser capaces de añadir nuevas clases independientemente, conservándose al mismo tiempo la integridad arquitectónica del marco de referencia.

Esta biblioteca debe ser también pequeña; en condiciones de igualdad, es mucho más probable que los programadores construyan su propia clase en lugar de reutilizar una que sea difícil de entender.

Se asume la existencia de compiladores de C++ que soporten clases parametrizadas y excepciones. Por razones de portabilidad, esta biblioteca no debe depender de ningún servicio del sistema operativo.

la mentalidad correcta, se pueden «objetificar» esos algoritmos: van a definirse clases cuyas instancias son agentes responsables de llevar a cabo estas acciones. Como se discute después en este capítulo, al objetificar estas abstracciones algorítmicas se pueden aprovechar los beneficios de los aspectos comunes formando una jerarquía de generalización/especialización.

Como primera decisión de análisis, por tanto, se elige delimitar el problema organizando las abstracciones en una de dos categorías principales:

- | | |
|----------------|---|
| • Estructuras | Contiene todas las abstracciones estructurales. |
| • Herramientas | Contiene todas las abstracciones algorítmicas. |

Como se verá muy pronto, hay una relación «de uso» entre estas dos categorías: ciertas herramientas se construyen sobre los servicios más primitivos proporcionados por algunas de las estructuras.

En la segunda fase del análisis del dominio se estudian las clases básicas utilizadas por los sistemas de producción en diversas áreas de aplicación (cuanto más amplio sea el espectro, mejor). En el camino, pueden descubrirse abstracciones comunes que se solapen con las que se habían encontrado en la primera fase del análisis: es una buena indicación de que verdaderamente se han descubierto abstracciones generales, así que esas se mantendrán indefinidamente dentro de la frontera del problema. También pueden encontrarse ciertas abstracciones sesgadas hacia el dominio, como moneda corriente, coordenadas astronómicas y medidas de masa y tamaño. Se decide rechazar esas abstracciones para la biblioteca, porque son o difíciles de generalizar (como la moneda corriente), o altamente específicas del dominio (como las coordenadas astronómicas) o tan primitivas que es difícil encontrar una buena razón para convertirlas en ciudadanos de primera clase (como las medidas de masa y tamaño).

Sobre la base de este análisis, vamos a determinar los siguientes tipos de estructura:

- | | |
|---------------|--|
| • Bolsas | Colección de elementos (posiblemente duplicados). |
| • Colecciones | Colección indexable de elementos. |
| • Cola doble | Secuencia de elementos en la que esos elementos pueden añadirse y eliminarse por cualquier extremo. |
| • Grafos | Colección de nodos y arcos sin raíz, que puede tener ciclos y referencias cruzadas; se permite compartición estructural. |
| • Listas | Secuencia de elementos con raíz; se permite compartición estructural. |
| • Mapas | Diccionario de pares elemento/valor. |
| • Colas | Secuencia de elementos en la que esos elementos pueden añadirse por un extremo y eliminarse por el extremo opuesto. |
| • Anillos | Secuencia de elementos en la que esos elementos pueden añadirse y eliminarse por el extremo superior de una estructura circular. |

• Conjuntos	Colección de elementos (no duplicados).
• Pilas	Secuencia de elementos en la que esos elementos pueden añadirse y eliminarse por el mismo extremo.
• Cadenas	Secuencia indexable de elementos, con comportamientos que implican la manipulación de subcadenas.
• Árboles	Colección (con raíz) de nodos y arcos, que no pueden contener ciclos ni referencias cruzadas; se permite compartición estructural.

Como se discutió en el Capítulo 4, la organización de las abstracciones presentadas en esta lista es un problema de clasificación. Se elige esta organización concreta porque ofrece una clara separación de comportamientos entre cada categoría de abstracciones.

Nótense los patrones de comportamiento que se encuentran recorriendo esta descomposición: algunas estructuras se comportan como colecciones (por ejemplo, las bolsas y conjuntos), mientras otras se comportan como secuencias (como colas dobles y pilas). Además, algunas estructuras permiten compartición estructural (como grafos, listas y árboles) mientras que otras son más monolíticas, y no permiten la compartición estructural de sus partes. Como se verá, se pueden aprovechar estos patrones con el fin de formar una arquitectura más simple durante el diseño.

El análisis revela también algunas variaciones funcionales deseables para algunas de estas clases. En particular, se considera necesario tener colecciones, colas dobles y colas ordenadas (las últimas se llaman a menudo *colas con prioridad*)². Además, se puede distinguir entre grafos dirigidos y no dirigidos, listas simplemente y doblemente enlazadas, así como entre árboles binarios, multcamino y AVL. Estas abstracciones especializadas son lo bastante parecidas entre sí como para decidir llevar a cabo más refinamientos a la categorización que se listó más arriba, en vez de hacer categorías separadas de abstracciones.

Aunque se han descubierto patrones significativos de comportamientos comunes, se decide explícitamente que en este momento no se van a organizar estas clases en una trama de herencias. Durante el análisis es suficiente articular los papeles de cada una de estas abstracciones diferentes; decidir sobre relaciones de herencia en este punto sería prematuro, así que se aplaza esta cuestión hasta el diseño arquitectónico.

También se pueden determinar los siguientes tipos de herramientas, basadas en el análisis del dominio que se ha hecho:

• Fecha/Hora	Operaciones para manipular fecha y hora.
• Filtros	Transformaciones de entrada, proceso

- | | |
|--|---|
| <ul style="list-style-type: none"> • Emparejamiento de patrones • Búsqueda • Ordenación • Utilidades | <ul style="list-style-type: none"> Operaciones para buscar secuencias dentro de otras secuencias. Operaciones para buscar elementos en estructuras. Operaciones para ordenar estructuras. Operaciones compuestas habituales que se construyen sobre operaciones estructurales más primitivas. |
|--|---|

Existen variaciones funcionales obvias para muchas de estas abstracciones. Por ejemplo, se puede distinguir entre muchos tipos diferentes de agentes de ordenación (tales como agentes responsables de ordenación quicksort, ordenación de burbuja, ordenación por montículo, etc.), así como entre diversos tipos de agentes de búsqueda (como agentes responsables de búsqueda secuencial, búsqueda binaria, y búsqueda pre, in y postorden en árboles). Como antes, se decide aplazar las decisiones sobre tramas de herencias entre estas abstracciones.

Patrones

Se han identificado ahora los elementos funcionales principales de esta biblioteca, pero un montón de abstracciones aisladas no constituye un marco de referencia. Como sugiere Wirfs-Brock, «un marco de referencia proporciona un modelo de interacción entre varios objetos que pertenecen a clases definidas por el marco de referencia... Para usar un marco de referencia, se estudian primero las colaboraciones y responsabilidades de varias clases» [9]. Ésta es entonces la piedra de toque para distinguir los marcos de referencia de simples tramas de herencias: un marco de referencia consta de una colección de clases junto con una serie de patrones de colaboración entre instancias de estas clases.

El análisis revela que hay una serie de patrones importantes esenciales para esta biblioteca de clases básicas, abarcando las siguientes cuestiones:

- Semántica de tiempo y espacio.
- Políticas de gestión del almacenamiento.
- Respuesta a condiciones excepcionales.
- Pautas para la iteración.
- Sincronización en presencia de múltiples hilos de control.

Como sugiere esta lista, el diseño de esta biblioteca de clases básicas exige un delicado equilibrio entre requisitos técnicos que compiten³. Si se intenta abordar estos problemas de forma completamente aislada respecto al resto, seguramente se acabará con poca compartición de protocolos, políticas o implanta-

³ En realidad, como observa Stroustrup, «el diseñar una biblioteca general es mucho más difícil que diseñar un programa ordinario» [10].

ción. Un enfoque tan ingenuo llevará de hecho a una abundancia de conceptos que intimidará a los clientes eventuales de esta biblioteca, y por tanto inhibirá su reutilización.

Considérese la perspectiva del desarrollador que debe usar esta biblioteca. ¿Qué representan sus clases? ¿Cómo trabajan juntas? ¿Cómo pueden adaptarse para satisfacer necesidades específicas del dominio? ¿Qué clases son realmente importantes, y cuáles pueden ignorarse? Estas son las cuestiones que hay que responder antes de poder esperar que los desarrolladores usen esta biblioteca para cualquier aplicación no trivial. Afortunadamente, no es necesario que el desarrollador comprenda todos los aspectos sutiles de una biblioteca tan grande como ésta, del mismo modo que no es necesario comprender cómo funciona un microprocesador para programar un computador en un lenguaje de alto nivel. En ambos casos, sin embargo, si es necesario puede exponerse la potencia pura de la implantación subyacente, pero sólo si el desarrollador está dispuesto a absorber la complejidad adicional.

Considérese el protocolo de cada abstracción de esta biblioteca desde la perspectiva de sus dos tipos de clientes: los clientes que usan una abstracción declarando instancias suyas y manipulando después estas instancias, y los clientes que crean subclases de una abstracción para especializar o aumentar su comportamiento. Diseñar en favor del primer cliente lleva a ocultar los detalles de implantación y centrarse en las responsabilidades de la abstracción en el mundo real; diseñar en favor del segundo cliente exige exponer ciertos detalles de implantación, pero no tantos como para permitir que se viole la semántica fundamental de la abstracción. Esto representa una tensión muy real de requisitos competidores en el diseño de tal biblioteca.

La parte verdaderamente difícil de la convivencia con cualquier biblioteca de clases grande e integrada es aprender qué mecanismos incorpora. Los patrones enumerados arriba sirven como el alma de la arquitectura de esta biblioteca; cuanto más sepa uno sobre estos mecanismos, más fácil es descubrir formas innovadoras de utilizar componentes existentes en vez de fabricar otros nuevos desde cero. En la práctica, se observa que los desarrolladores suelen comenzar utilizando las clases más obvias de una biblioteca. A medida que ganan confianza en ciertas abstracciones, van pasando al uso de clases más sofisticadas. Eventualmente, los desarrolladores pueden descubrir un patrón en su propia adaptación de una clase predefinida, y añadirla así a la biblioteca como una abstracción primitiva. Análogamente, un equipo de desarrolladores puede darse cuenta de que ciertas clases específicas del dominio aparecen una y otra vez en diversos sistemas; se introducen también en la biblioteca. Esta es precisamente la forma en que las bibliotecas de clases crecen con el tiempo: no de un día para otro, sino desde formas intermedias estables y más pequeñas.

En realidad, éste es precisamente el modo en que se va a expandir esta biblioteca: primero se va a inventar una arquitectura que afronte cada uno de los cinco patrones descritos arriba, y entonces se poblará la biblioteca desplegando su implantación.

9.2. Diseño

Cuestiones tácticas

La Ley de Coggins de la Ingeniería del Software establece que «la práctica debe tener precedencia sobre la elegancia, porque no se puede impresionar a la Naturaleza» [11]. Un corolario de esta ley es que el diseño nunca puede ser totalmente independiente del lenguaje. Las características y semántica particulares de un lenguaje dado influyen en las decisiones arquitectónicas, e ignorar esas influencias nos dejaría con abstracciones que no aprovechan las facilidades únicas del lenguaje, o con mecanismos que no pueden implantarse eficientemente en ningún lenguaje.

Como se discutió en el Capítulo 3, los lenguajes de programación orientados a objetos ofrecen tres facilidades básicas para organizar una colección rica de clases: herencia, agregación y parametrización. La herencia es ciertamente el aspecto más visible (y más popular) de la tecnología orientada a objetos; sin embargo, no es el único principio estructurador que se debería considerar. En realidad, como se verá, la parametrización combinada con la herencia y la agregación puede llevar a una arquitectura pequeña pero muy potente.

Considérese esta declaración abreviada de una clase de cola específica del dominio en C++:

```
class EventoRed...

class ColaEventos {
public:

    ColaEventos();
    virtual ~ColaEventos();

    virtual void borrar();
    virtual void anadir(const EventoRed&);
    virtual void extraer();

    virtual const EventoRed& cabecera() const;
    ...
};
```

Aquí se tiene la realización concreta de una cola de eventos: una estructura en la que se pueden añadir objetos evento a la cola, y eliminarlos de la cabecera de la misma. C++ apoya la abstracción permitiendo establecer el comportamiento público que se pretende de la cola (expresado mediante las operaciones `borrar`, `anadir`, `extraer` y `cabecera`), mientras se oculta su representación exacta.

Ciertos usos de esta abstracción pueden demandar una semántica ligeramente diferente; en concreto, se puede necesitar una cola con prioridad, en la

que los eventos se añaden a la cola en un orden determinado por su urgencia. Se puede aprovechar el trabajo que se ha realizado ya, obteniendo subclases de la clase base cola y especializando su comportamiento:

```
class ColaEventosPrioridad : public ColaEventos {
public:

    ColaEventosPrioridad();
    virtual ColaEventosPrioridad();

    virtual void anadir(const EventoRed&);
    ...
};
```

Las funciones virtuales apoyan la abstracción permitiendo redefinir la semántica de operaciones concretas (como `anadir`) de otra abstracción más generalizada.

En combinación con las clases parametrizadas, se pueden hacer abstracciones aún más generales. La semántica de las colas es la misma, no importa si se tiene una cola de berzas o una cola de reyes. Usando clases modelo, se puede redeclarar la clase base original como sigue:

```
template<class Elemento>
class Cola {
public:

    Cola();
    virtual ~Cola();

    virtual void borrar();
    virtual void anadir(const Elemento&);
    virtual void extraer();

    virtual const Elemento& cabecera() const;

    ...
};
```

Esta es una estrategia muy común cuando se aplican clases parametrizadas: tomar una clase concreta existente, identificar las formas en que su semántica es invariantes de acuerdo con los elementos que manipula, y extraer esos elementos como argumentos del modelo.

Nótese que se pueden combinar la herencia y la parametrización de algunas formas muy potentes. Por ejemplo, se puede redeclarar la subclase original como sigue:

```
template<class Elemento>
```

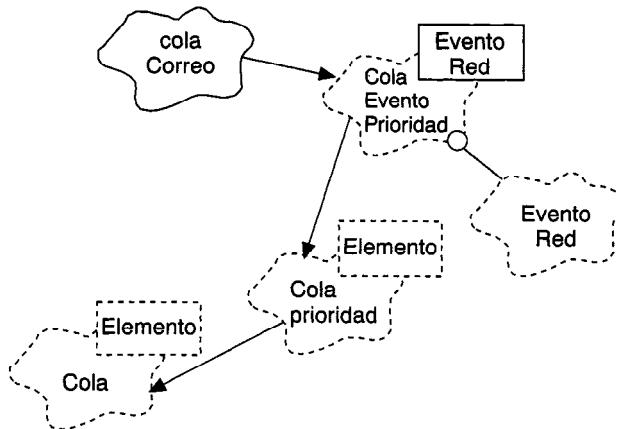


Figura 9.1. Herencia y parametrización.

```

class ColaPrioridad : public Cola<Elemento> {
public:

    ColaPrioridad();
    virtual ~ColaPrioridad();

    virtual void anadir(const Elemento&);
    ...
};
  
```

La seguridad respecto a los tipos es la ventaja clave que ofrece este enfoque. Se puede instanciar cualquier número de clases de cola concretas, como los siguientes:

```

Cola<char> colaCaracter;
typedef Cola<EventoRed> ColaEventos;
typedef ColaPrioridad<EventoRed> ColaEventosPrioridad;
  
```

El lenguaje apoyará las abstracciones, de forma que no podrán añadirse eventos a la cola de caracteres, ni valores en punto flotante a la cola de eventos.

La Figura 9.1 ilustra este diseño mostrando las relaciones entre una clase parametrizada (`Cola`), su subclase (`ColaPrioridad`), una de sus instancias (`ColaEventosPrioridad`) y una de sus instancias (`colaCorreos`).

Este ejemplo nos lleva a establecer el primer principio arquitectónico para esta biblioteca: excepto en unos pocos casos, las clases que se proporcionen deberían ser parametrizadas. Esta decisión apoya los requisitos de la biblioteca respecto a la seguridad.

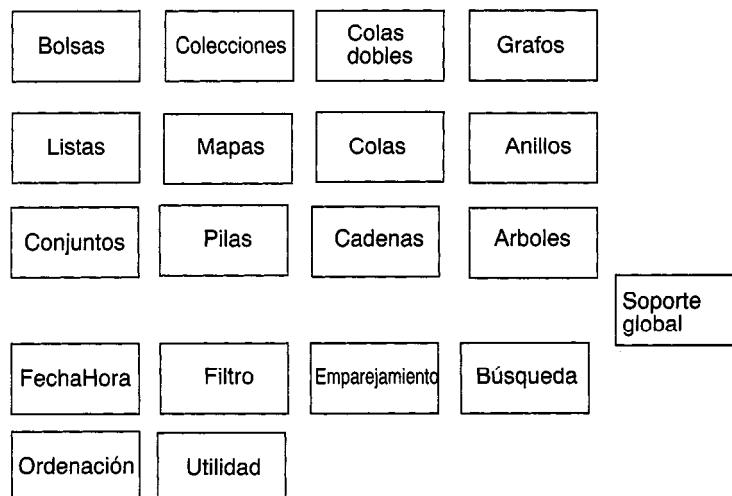


Figura 9.2. Categorías de clases de la biblioteca de clases básicas.

Macroorganización

Como se discutió en capítulos anteriores, la clase es un vehículo necesario pero no suficiente para la descomposición. Esta observación se aplica, ciertamente, a esta biblioteca de clases. Una de las peores organizaciones que se podrían idear sería formar una mera colección de clases, a través de la cual tendrían que navegar los desarrolladores para encontrar las clases que necesitasen. Puede hacerse mucho mejor ubicando cada agrupamiento de clases en su propia categoría, como se muestra en la Figura 9.2. Esta decisión ayuda a satisfacer los requisitos de la biblioteca respecto a la simplicidad.

Un rápido análisis del dominio sugiere que existe una oportunidad para explotar las representaciones comunes entre las clases de esta biblioteca. Por esta razón, se establece la existencia de la categoría accesible globalmente llamada *soporte*, cuyo propósito es organizar estas abstracciones de nivel inferior. También se usará esta categoría para recoger las clases necesarias en el soporte de los mecanismos comunes de la biblioteca.

Esto nos lleva a establecer el segundo principio arquitectónico para esta biblioteca: se decide hacer una distinción clara entre política e implementación. En cierto sentido, las abstracciones como colas, conjuntos y anillos representan políticas particulares para usar estructuras de nivel inferior como listas enlazadas o matrices. Por ejemplo, una cola define la política por la que los elementos sólo pueden añadirse por un extremo de la estructura, y eliminarse por el otro. Un conjunto, por otro lado, no fuerza tal política que requiera una ordenación de elementos. Un anillo obliga a una ordenación, pero fija la política de que la cabecera y la cola de sus elementos están conectadas. Se utilizará por tanto la

categoría de soporte para esas abstracciones más primitivas sobre las que pueden formularse diferentes políticas.

Exponiendo esta categoría a los constructores de la biblioteca, se apoya el requerimiento respecto a la extensibilidad. En general, los desarrolladores de aplicaciones sólo necesitan interesarse por las clases que se encuentran en las categorías de estructuras y herramientas. Los desarrolladores de la biblioteca y los usuarios avanzados, sin embargo, pueden querer hacer uso de las abstracciones más primitivas que se encuentran en *Soporte*, a partir de las cuales pueden construirse nuevas clases, o a través de las cuales puede modificarse el comportamiento de clases existentes.

Como sugiere la Figura 9.2, se organiza esta biblioteca como un bosque de clases, en vez de como un árbol; no hay una sola clase base, como ocurriría en lenguajes como Smalltalk.

Aunque no aparecen en esta figura, las clases de las categorías *Grafos*, *Listas* y *Arboles* son sutilmente distintas de las otras clases estructurales. Anteriormente, se hizo notar que las abstracciones como *deques* y *pilas* son monolíticas. Una estructura *monolítica* es aquella que se trata siempre como una unidad simple: no hay componentes distintos e identificables, y por tanto está garantizada la integridad referencial. Alternativamente, una estructura *polilítica* (como un grafo) es aquella en la que se permite compartición estructural. Por ejemplo, se pueden tener objetos que denotan una sublista de una lista mayor, una rama de un árbol mayor o vértices y arcos individuales de un grafo. La distinción fundamental entre estructuras monolíticas y polilíticas es que, en las estructuras monolíticas, las semánticas de copia, asignación e igualdad son profundas, mientras que en estructuras polilíticas la copia, la asignación y la igualdad son todas operaciones superficiales (lo que significa que los alias pueden compartir una referencia a una parte de una estructura mayor).

Familias de clases

Un tercer principio central al diseño de esta biblioteca es el concepto de construcción de familias de clases, relacionadas por líneas de herencia. Para cada tipo de estructura, se proporcionarán varias clases diferentes, unidas por un interfaz compartido (como la clase base abstracta *cola*), pero con varias subclases concretas, cada una de las cuales tiene una representación ligeramente diferente, y por lo tanto tiene una semántica distinta de espacio y tiempo. De este modo se apoya el requisito de la biblioteca respecto a la compleción. Un desarrollador puede seleccionar la clase concreta cuya semántica de espacio y tiempo se adapte mejor a las necesidades de una aplicación dada, aunque siga confiando en que, sea cual sea la clase concreta seleccionada, será funcionalmente la misma que cualquier otra clase concreta de la familia. Esta separación de intereses clara e intencionada entre una clase base abstracta y sus clases concretas permite a un desarrollador seleccionar inicialmente una clase concreta y posteriormente, a medida que se ajusta la aplicación, reemplazarla por una clase concreta her-

mana con el mínimo esfuerzo (el único coste real es la recompilación de todos los usos de la nueva clase). El desarrollador puede confiar en que la aplicación seguirá funcionando, porque todas las clases concretas hermanas comparten el mismo interfaz y el mismo comportamiento central. Otra implicación de esta organización es que hace posible copiar, asignar y comprobar la igualdad entre objetos de la misma familia de clases, incluso si cada objeto tiene una representación radicalmente diferente.

En un sentido muy simple, una clase base abstracta sirve así para capturar todas las decisiones de diseño público relevantes sobre la abstracción. Otro uso importante de las clases base abstractas es almacenar estado común que de otro modo podría ser muy caro calcular. Esto puede convertir un cálculo $O(n)$ en una recuperación $O(1)$. El coste de este estilo es la cooperación que se requiere entre la clase base abstracta y sus subclases, para mantener actualizado el resultado almacenado.

Los diversos miembros concretos de una familia de clases representan las *formas* de una abstracción. Según nuestra experiencia hay dos formas fundamentales para la mayoría de las abstracciones que todo desarrollador debe considerar cuando construye una aplicación seria. La primera de ellas es la forma de representación, que establece la implantación concreta de una clase base abstracta. En última instancia, sólo hay dos elecciones significativas para estructuras en memoria: la estructura se almacena en la pila, o se almacena en el heap. Se llama a estas variaciones las formas *limitada* y *no limitada* de una abstracción, respectivamente.

- Limitada La estructura se almacena en la pila, y por tanto tiene un tamaño estático en el momento en que el objeto se construye.
- No limitada La estructura se almacena en el **heap** (montículo), y por tanto puede crecer hasta los límites de la memoria disponible.

Puesto que las formas limitada y no limitada de una abstracción comparten un interfaz y comportamiento comunes, se decide hacerlas subclases directas de la clase base abstracta para cada estructura. Se discutirán con más detalle estas y otras variaciones en secciones posteriores.

La segunda variación importante concierne a la sincronización. Como se discutió en el Capítulo 2, muchas aplicaciones útiles implican a un único proceso. Se les llama sistemas *secuenciales*, porque sólo involucran un hilo de control. Ciertas aplicaciones, especialmente las que conllevan control en tiempo real, pueden requerir la sincronización de varios hilos de control simultáneos dentro del mismo sistema. Se llaman a tales sistemas *concurrentes*. La sincronización de múltiples hilos de control es importante a causa de los problemas de exclusión mutua. Dicho sencillamente, es incorrecto permitir a dos o más hilos de control actuar directamente sobre el mismo objeto al mismo tiempo, porque pueden interferir con el estado del objeto, y en última instancia corromper ese estado. Por ejemplo, considérense dos objetos activos que intenten añadir un elemento

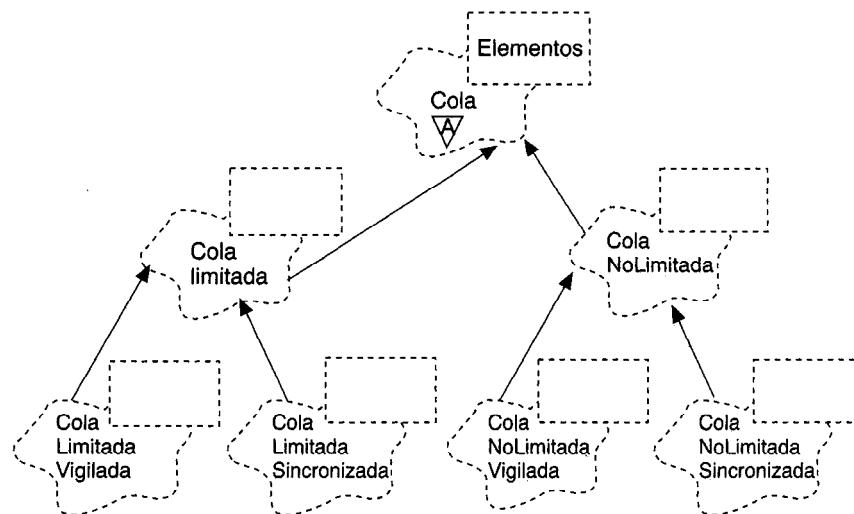


Figura 9.3. Familias de clases.

al mismo objeto **Cola**. El primer agente podría comenzar a añadir el nuevo elemento, ser desplazado por el planificador y dejar por tanto al objeto en un estado inconsistente para el segundo agente.

Como se describió en el Capítulo 3, sólo hay fundamentalmente tres alternativas de diseño posibles, que requieren diferentes grados de cooperación entre los agentes que interactúan con un objeto compartido:

- Secuencial.
- Vigilada.
- Síncrona.

Se discutirán estas variaciones con más detalle en una sección posterior.

Las interacciones entre la clase base abstracta, las formas de representación y las formas de sincronización producen la misma familia de clases para cada estructura como se muestra en la Figura 9.3. Esta arquitectura explica por qué se ha decidido organizar la biblioteca como una familia de clases en vez de como un árbol con una sola raíz:

- Refleja con exactitud la estructura regular de las diversas formas componentes.
- Conlleva menos complejidad y sobrecarga al seleccionar un componente de la biblioteca.
- Evita los interminables debates ontológicos engendrados por una aproximación «orientada a objetos pura».
- Simplifica la integración de la biblioteca con otras bibliotecas.

Microorganización

En apoyo del requisito de la biblioteca respecto a la simplicidad, se decide seguir un estilo consistente para todas las estructuras y herramientas de la biblioteca:

```
template<...>
class Nombre : public Superclase {
public:

    // constructores
    // virtual destructor

    // operadores

    // modificadores

    // selectores

protected:

    // objetos miembro

    // funciones auxiliares

private:
    // amigos
};
```

Por ejemplo, la definición de la clase base abstracta `Cola` comienza como sigue:

```
template<class Elemento>
class Cola {
```

La presentación de la plantilla* sirve para establecer los argumentos por los que puede parametrizarse la clase. Nótese que en C++, las plantillas están deliberadamente subespecificadas, lo que deja un grado de flexibilidad (y responsabilidad) en manos de los desarrolladores que instancian esos modelos.

A continuación, se proporciona el conjunto habitual de constructores y destructores:

```
Cola();
Cola(const Cola<Elemento>&);
virtual ~Cola();
```

* Se ha decidido traducir `template` como plantilla. Otros términos son patrón y modelo. (*N. del T.*)

Nótese que se ha declarado el destructor como virtual, ya que se desea un comportamiento polimórfico cuando se destruye un objeto de esta clase. Acto seguido, se tiene la declaración de todos los operadores:

```
virtual Cola<Elemento>& operator=(const Cola<Elemento>&);  
virtual int operator==(const Cola<Elemento>&) const;  
int operator!=(const Cola<Elemento>&) const;
```

Se definen `operator=` (asignación) y `operator==` (prueba de igualdad) como `virtual` por razones de seguridad respecto a los tipos. Es responsabilidad de las subclases sobrecargar estas dos funciones miembro, usando funciones cuya presentación toma un argumento de su propia clase especializada. De este modo, las subclases pueden aprovechar su conocimiento sobre la representación de sus propias instancias para proporcionar una implantación muy eficiente. Cuando no se conoce la subclase exacta o concreta de una cola (como cuando se pasa un objeto por referencia a la clase base), se invocan las operaciones de la clase base, utilizando algoritmos ligeramente menos eficientes pero más generales. Esta pauta tiene el efecto lateral de permitir que se asignen y se pruebe la igualdad de objetos cola con diferentes representaciones sin una colisión de tipos.

Si se desease restringir la copia, asignación o prueba de igualdad de ciertos objetos, se pueden declarar estas operaciones como `protected` o `private`.

A continuación se proporcionan todos los modificadores, que son operaciones que pueden alterar el estado del objeto:

```
virtual void borrar() = 0;  
virtual void anadir(const Elemento&) = 0;  
virtual void extraer() = 0;  
virtual void eliminar(unsigned int en) = 0;
```

Se declaran estas operaciones como virtuales puras, lo que significa que es responsabilidad de las subclases proporcionar su implantación real. En virtud de estas funciones virtuales puras, la clase `Cola` se define como abstracta.

Se usa el calificador `const` para indicar (y dejar que el lenguaje lo apoye) el uso de funciones selectoras que observan, pero no modifican, el estado de un objeto.

```
virtual unsigned int longitud() const = 0;  
virtual int estaVacia() const = 0;  
virtual const Elemento& cabecera() const = 0;  
virtual int posicion(const Elemento&) const = 0;
```

Estas operaciones también se declaran como virtuales puras, porque la clase `Cola` no tiene suficiente autoridad para llevar a cabo estas responsabilidades particulares.

En nuestro estilo, la parte `protected` de todas las clases comienza con aquellos objetos miembro que forman su representación y que se desea que sean ac-

cesibles a las subclases⁴. La clase base abstracta `Cola` no tiene tales miembros, aunque sus clases concretas si los tienen, como se verá en una sección posterior.

A tales objetos miembro les siguen las funciones auxiliares que requiere la clase base, implantadas polimórficamente por todas las subclases concretas. La clase `Cola` proporciona un conjunto típico de estas funciones miembro:

```
virtual void purgar() = 0;
virtual void anadir(const Elemento&) = 0;
virtual unsigned int cardinalidad() const = 0;
virtual const Elemento& elementoEn(unsigned int) const = 0;

virtual void bloquear();
virtual void desbloquear();
```

La razón por la que se suministran esas funciones auxiliares en particular se aclarará en una sección posterior.

Por último, se proporciona una parte `private`, que suele contener sólo declaraciones de amigas y la declaración de aquellos objetos miembro que se desea que sean inaccesibles a las subclases. En el caso de la clase `Cola`, sólo hay declaraciones `friend`:

```
friend class ColaIteradorActivo<Elemento>;
friend class ColaIteradorPasivo<Elemento>;
```

Como se describirá en una sección posterior, estas declaraciones `friend` son necesarias en apoyo de los modismos de iterador.

Semántica de tiempo y espacio

De los cinco patrones que impregnan la arquitectura de este marco de referencia, quizás el más importante sea el mecanismo que proporciona al cliente semánticas de tiempo y espacio alternativas dentro de cada familia de clases.

Considérese el rango de semánticas que debe cubrir una biblioteca general como esta. En una estación de trabajo que ofrece un gran espacio de direcciones virtuales, los clientes sacrificarán frecuentemente el espacio frente a la rapidez de las abstracciones. Por contra, en ciertos sistemas empotados, como satélites del espacio exterior o motores de automóvil, los recursos de memoria suelen ser preciosos, y por eso los clientes deben elegir abstracciones que conserven esos recursos escasos (por ejemplo, utilizando representaciones basadas en la pila y no en el heap). Anteriormente se distinguieron esas dos alternativas como limitada y no limitada, respectivamente.

⁴ A menos que haya razones de peso para hacer lo contrario, normalmente se declaran todos los objetos miembro como `private`. Aquí, sin embargo, hay una razón poderosa para hacerlos `protected`: las subclases necesitan acceder a esos miembros.

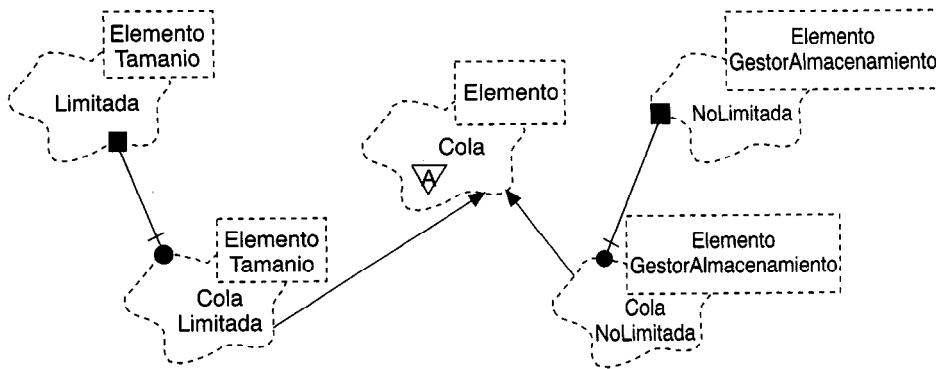


Figura 9.4. Formas limitadas y no limitadas.

Las formas no limitadas son aplicables en aquellos casos en los que el tamaño final de la estructura no puede predecirse, y en los que el reservar y liberar espacio del heap no es ni demasiado costoso ni inseguro (como puede ocurrir en ciertas aplicaciones críticas respecto al tiempo)⁵. Alternativamente, las formas limitadas son más apropiadas para estructuras más pequeñas, cuyos tamaños medios y máximos son predecibles, y donde el uso del heap (montículo) se considera poco seguro.

Todas las estructuras de esta biblioteca requieren este rango de alternativas, y por esta razón se inventan dos clases de soporte de nivel inferior, *NoLimitada* y *Limitada*, para proporcionar este comportamiento. La responsabilidad de la clase *NoLimitada* es proporcionar una estructura de lista enlazada muy eficiente que utiliza elementos situados en el heap; esta representación es eficiente respecto al tiempo, pero menos eficiente respecto al espacio, porque para cada elemento hay que tener también espacio para un puntero al siguiente elemento. La responsabilidad de la clase *Limitada* es proporcionar una clase base matriz-matriz muy eficiente y empaquetada óptimamente; esta representación es eficiente respecto al espacio, pero menos eficiente respecto al tiempo, porque cuando se añaden elementos nuevos en el medio del contenedor hay que desplazar mediante copia los elementos de una de las dos mitades.

Como se muestra en la Figura 9.4, se usa la agregación para ubicar estas clases de nivel inferior en la familia de clases. En concreto, el diagrama muestra que se usa contención física por valor con acceso protegido, lo que quiere decir que esta representación de nivel inferior sólo es accesible a las subclases y a sus amigas. En un diseño anterior, se intentó un estilo aditivo, por el que las clases *NoLimitada* y *Limitada* se introdujeron como superclases *protected*. Al final se

⁵ Ciertos requisitos críticos pueden proscribir por completo el uso de almacenamiento basado en el heap. Considerese el software de un marcapasos, y los resultados potencialmente fatales si tiene lugar una recolección de basura en un momento inoportuno. Considerese también un sistema de reservas para largo recorrido, en el que incluso una pérdida mínima de la memoria podría tener serios efectos acumulativos; el tener que reiniciar el sistema a causa de una situación de falta de memoria podría resultar en una pérdida de servicio inaceptable.

desechó este diseño, porque demostró ser conceptualmente difícil para los clientes, y tampoco admitía nuestra piedra de toque para la herencia: una Cola-Limitada no es un tipo de Limitada, en el sentido de que se deseé tratarlas como del mismo tipo.

Nótese que estas diversas formas introducen un segundo argumento para el modelo. La forma limitada requiere un entero sin signo Tamanio, que denota el tamaño estático de cada instancia. La forma no limitada requiere una clase GestorAlmacenamiento, cuya responsabilidad es proporcionar una política concreta de gestión del almacenamiento, como se discutirá en la próxima sección.

El protocolo proporcionado por ambas clases de nivel inferior debe ser suficiente para implantar las responsabilidades de las clases cola concretas en particular, y lo bastante completo como para implantar las responsabilidades de todas las demás estructuras de la biblioteca. Por razones de eficiencia, se decide no utilizar funciones virtuales en ninguna de estas clases. Esto tiene dos implicaciones: no se pueden unificar significativamente NoLimitada y Limitada con una superclase común, aunque sí que tienen un protocolo común; y no se puede crear correctamente ninguna subclase. En cierto modo, se ha decidido intercambiar flexibilidad por velocidad de ligadura y uso razonable de memoria. Como parte de este intercambio, se decide también hacer que ciertas funciones sean inline por razones de velocidad; los selectores suelen ser buenos candidatos para ser inline, especialmente cuando sólo implican el retorno de un valor simple.

Por ejemplo, considérese la declaración de la clase Limitada:

```
template<class Elemento, unsigned int Tamanio>
class Limitada {
public:

    Limitada();
    Limitada(const Limitada<Elemento, Tamanio>&);
    ~Limitada();

    Limitada<Elemento, Tamanio>& operator=(const Limitada<Elemento,
                                              Tamanio>&)
    int operator==(const Limitada<Elemento, Tamanio>&) const
    int operator!=(const Limitada<Elemento, Tamanio>&) const;
    const Elemento& operator[](unsigned int indice) const;
    Elemento& operator[](unsigned int indice);

    void borrar();
    void insertar(const Elemento&);
    void insertar(const Elemento&, unsigned int antes);
    void anadir(const Elemento&);
    void anadir(const Elemento&, unsigned int despues);
    void eliminar(unsigned int en);
    void reemplazar(unsigned int en, const Elemento&);
```

```

        unsigned int disponible() const;
        unsigned int longitud() const;
        const Elemento& primero() const;
        const Elemento& ultimo() const;
        const Elemento& elementoEn(unsigned int) const;
        Elemento& elementoEn(unsigned int);
        int posicion(const Elemento&) const;

        static void* operator new(size_t);
        static void operator delete(void*, size_t);

protected:

    Elemento rep[Tamanio];
    unsigned int comienzo;
    unsigned int final;

    unsigned int expandirIzquierda(unsigned int desde);
    unsigned int expandirDerecha(unsigned int desde);
    void encogerIzquierda(unsigned int desde);
    void encogerDerecha(unsigned int desde);

};


```

Esta declaración de clase sigue el estilo descrito anteriormente. ¿Cómo se llegó exactamente a este interfaz particular? La respuesta honesta es que el diseño de clases aisladas, como se explicó en el Capítulo 6, proporcionó el 80 % de la solución, pero después se desarrolló este interfaz hasta la forma final mostrada arriba, después de tener que usar esta clase para implantar tres o cuatro de las familias estructurales de clases. La parte difícil de esta evolución fue la identificación de operaciones primitivas adecuadas que pudieran utilizarse para llevar a cabo todas las políticas diferentes necesarias para todas las estructuras.

Nótese la última representación de esta clase, es decir, el objeto miembro `protected rep`, que se declara como una matriz de elementos con una longitud estática `Tamanio`. Considérese la siguiente declaración:

```
Limitada<char, 100> secuenciaCaracteres;
```

La elaboración de esta declaración da lugar a una matriz de tamaño fijo, de 100 elementos, en la pila. Los objetos miembro `protected comienzo` y `final` se usan como índices en esta matriz, denotando el comienzo y el final de la secuencia. De este modo, se implanta la abstracción usando un buffer circular. El añadir elementos a la cabecera y a la cola de la secuencia no requiere ningún movimiento de elementos que ya estén en la misma; el añadir elementos en un punto intermedio de la secuencia sólo requiere copiar (en promedio) la mitad de los elementos existentes.

El diseño de las clases de soporte limitada y no limitada plantea algunos

problemas sutiles que conciernen al uso de las referencias, a las que se aludió en el Capítulo 3. Aquí se exploran esas cuestiones un poco más en detalle, no sólo porque causan impacto en el interfaz de todas las clases modelo de la biblioteca, sino también porque son problemas fundamentales que debe afrontar el arquitecto de cualquier biblioteca de clases no trivial. En realidad, este es un ejemplo clásico de cómo la semántica particular del lenguaje puede afectar a las decisiones arquitecturales.

En C++, las referencias proporcionan un mecanismo de alias que puede mejorar la eficiencia. Sin embargo, las referencias deben utilizarse con cuidado para evitar la creación de situaciones poco seguras en tiempo de ejecución. En esta biblioteca se usan referencias para mejorar la eficiencia del paso de argumentos a las funciones miembro. Nótese, por ejemplo, la declaración de `Limitada`, en la que se pasan instancias de `Limitada` y `Elemento` por referencia. Como regla general, no se pasan por referencia objetos primitivos (como los enteros de la declaración de la función miembro `elementoEn`), porque es probable que eso hiciera el código más lento, y además la semántica del C++ introduce algunos problemas potencialmente peligrosos cuando se usan objetos temporales.

Sin embargo, se decide que todas las estructuras almacenen *valores*, no referencias, en sus respectivas formas concretas. Este estilo previene la creación de referencias a objetos transitorios en la pila de ejecución. Por la misma razón se rechazó un diseño alternativo que implicaba almacenar punteros a elementos, porque este enfoque exhibe un comportamiento muy poco deseable cuando se instancia un modelo con tipos construidos. Estos problemas son significativos cuando se diseña el interfaz de un marco de referencia que involucre clases modelo, ya que los clientes pueden instanciar los modelos con tipos arbitrarios. Hay tres casos que considerar, y hay que diseñar la biblioteca de forma que consiga un equilibrio entre los tres.

Primero, los tipos definidos pueden pasarse por referencia y copiarse en representaciones concretas sin dificultad. La declaración de los tipos de los argumentos como referencias constantes evita advertencias debidas a elementos temporales involucrados en la conversión de tipos [12].

Segundo, los tipos definidos por el usuario pueden pasarse por referencia y copiarse, pero sólo si proporcionan constructores de copia y el operador de asignación. Aunque las referencias permiten operaciones polimórficas (pasar un objeto de una clase derivada en vez de la declarada en la instancia), la copia no será polimórfica. Asignar el objeto a la representación «recortará» el objeto a una instancia de la clase base [13].

Tercero, los usos polimórficos de la biblioteca tendrán que instanciar los modelos con punteros a las clases base implicadas. Aunque el pasar los punteros por referencia no mejora necesariamente la eficacia, el copiar los punteros en la representación conserva el polimorfismo de los objetos derivados implicados.

Por ejemplo, dada la clase `colaLimitada`, se puede escribir lo siguiente:

```
class Evento ...  
typedef Evento* PtrEvento;
```

```
ColaLimitada<int, 100> colaEnteros;
ColaLimitada<Evento, 50U> colaEventos1;
ColaLimitada<PtrEvento, 100U> colaEventos2;
```

Con el objeto `colaEventos1`, los clientes pueden construir de forma segura colas de eventos, aunque el añadir instancias de cualquier subclase de `Evento` introducirá un recorte, y por tanto el comportamiento polimórfico de tales elementos se perderá. Por otra parte, el objeto `colaEventos2` contiene punteros a objetos de la clase `Evento`, y por tanto se pueden almacenar y liberar objetos de la clase `Evento` o sus subclases sin peligro de recortes.

La decisión de almacenar valores en vez de referencias o punteros a elementos da lugar a ciertas responsabilidades sobre la construcción y destrucción de elementos en una estructura. En particular, las clases de elementos usadas para instanciar una estructura deben proporcionar al menos un constructor por defecto, un constructor de copia y un operador de asignación. Además, no pueden destruirse los elementos inmediatamente después de su eliminación de una estructura. Por ejemplo, en las formas limitadas, los elementos (que al fin y al cabo se almacenan en matrices) no se destruyen hasta que se destruye la propia estructura.

Considérese ahora cómo se usa la clase `Limitada` para formar una clase concreta como `ColaLimitada`. Como se ve en la declaración siguiente, `ColaLimitada` tiene un objeto miembro `protected rep` de la clase `Limitada`:

```
template<class Elemento, unsigned int Tamanio>
class ColaLimitada : public Cola<Elemento> {
public:
    ColaLimitada();
    ColaLimitada(const ColaLimitada<Elemento, Tamanio>&);
    virtual ~ColaLimitada();

    virtual Cola<Elemento>& operator=(const Cola<Elemento>&);
    virtual Cola<Elemento>& operator=(const ColaLimitada<Elemento,
                                         Tamanio>&);
    virtual int operator==(const Cola<Elemento>&) const;
    virtual int operator==(const ColaLimitada<Elemento, Tamanio>&)
                           const;
    int operator!=(const ColaLimitada<Elemento, Tamanio>&) const;

    virtual void borrar();
    virtual void anadir(const Elemento&);
    virtual void extraer();
    virtual void eliminar(unsigned int en);

    virtual unsigned int disponible() const;
    virtual unsigned int longitud() const;
```

```

virtual int estaVacia() const;
virtual const Elemento& cabecera() const;
virtual int posicion(const Elemento&) const;

protected:

    Limitada<Elemento, Tamanio> rep;

    virtual void purgar();
    virtual void anadir(const Elemento&);
    virtual unsigned int cardinalidad() const;
    virtual const Elemento& elementoEn(unsigned int) const;

    static void* operator new(size_t)
    static void operator delete(void *, size_t);

};

```

La responsabilidad principal de esta clase es completar el protocolo definido en la clase base. Frecuentemente, esto implica poco más que la delegación de responsabilidad en la clase de nivel inferior `Limitada`, como se sugiere en la siguiente implantación:

```

template<class Elemento, unsigned int Tamanio>
unsigned int ColaLimitada<Elemento, Tamanio>::longitud() const
{
    return rep.longitud();
}

```

Nótese que la clase `ColaLimitada` introduce algunas operaciones adicionales sobre las definidas en su superclase. En concreto, se añade el selector `disponible`, que devuelve el número de elementos libres en la estructura (calculados como `Tamanio - longitud()`). No se incluyó esta operación en la clase base, más que nada porque el calcular la cantidad de espacio disponible en el heap no es una operación tan clara. También se han sobrecargado los operadores de asignación y prueba de igualdad. Como se mencionó antes, este hábito permite a las subclases proporcionar implantaciones más eficientes de estas operaciones que las que puede proporcionar la clase base, porque las subclases tienen un conocimiento detallado sobre su propia representación. Finalmente, se han añadido los operadores `new` y `delete`, pero se les ha declarado como miembros protegidos, lo que previene de forma efectiva a los clientes de reservar espacio para instancias de `ColaLimitada` (lo que es consistente con la semántica de almacenamiento estático de esta forma concreta).

La clase `NoLimitada` tiene sustancialmente el mismo protocolo que la clase `Limitada`, aunque su implantación es radicalmente diferente.

```
template<class Elemento, class GestorAlmacenamiento>
```

```

class NoLimitada {
public:
    ...
protected:

    Nodo<Elemento, GestorAlmacenamiento>* rep;
    Nodo<Elemento, GestorAlmacenamiento>* ultimo;
    unsigned int tamano;

    Nodo<Elemento, GestorAlmacenamiento>* cache;
    unsigned int indiceCache;

};

```

La forma `NoLimitada` proporciona la implantación de una lista enlazada de nodos, donde `Nodo` se declara como sigue:

```

template<class Elemento, class GestorAlmacenamiento>
class Nodo {
public:

    Nodo(const Elemento& e,
          Nodo<Elemento, GestorAlmacenamiento>* anterior,
          Nodo<Elemento, GestorAlmacenamiento>* siguiente);

    Elemento elemento;
    Nodo<Elemento, GestorAlmacenamiento>* anterior;
    Nodo<Elemento, GestorAlmacenamiento>* siguiente;

    static void* operator new(size_t);
    static void operator delete(void*, size_t);

};

```

La principal responsabilidad de esta clase es manejar un solo elemento junto con punteros a los nodos anterior y siguiente. Ya que esta es una clase de soporte y por tanto no la utiliza ningún cliente fuera de la biblioteca, se ha decidido relajar las reglas estrictas habituales sobre el encapsulamiento y exponer todo lo referente a su estado como miembros public, intercambiando así seguridad por eficiencia.

Puesto que las clases `Limitada` y `NoLimitada` ofrecen prácticamente el mismo protocolo público, se sabe que son funcionalmente la misma, y de aquí que también la implantación de todas las estructuras concretas limitada y no limitada parecerá la misma en gran medida. Sin embargo, la representación de estas dos clases de soporte es lo que lleva a semánticas de tiempo y espacio radicalmente diferentes. En particular, para la representación de la lista enlazada, la manipulación de nodos es muy rápida, pero encontrar elementos particulares

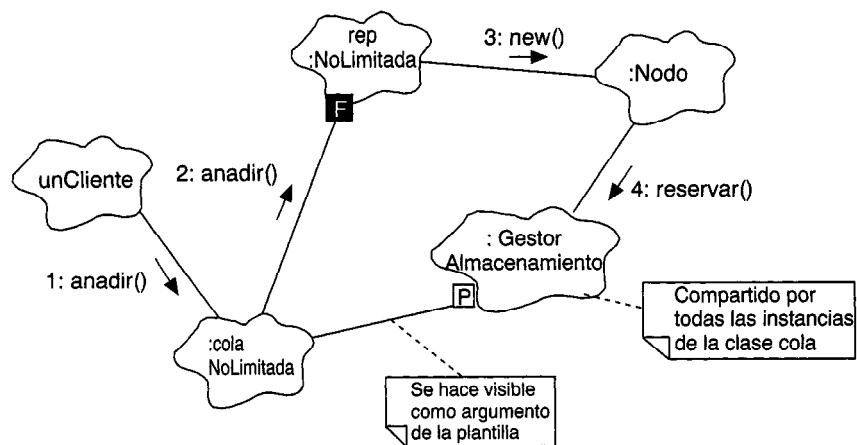


Figura 9.5. Mecanismo de gestión del almacenamiento.

puede ser lento (del orden de $O(n)$). Por esta razón, la implantación anota el último nodo referenciado, en la esperanza de que es probable que él o sus vecinos se toquen a continuación. Para la implantación basada en matriz, la manipulación de elementos puede ser lenta, siendo el caso peor $O(n/2)$, cuando se inserta o borra en el medio de una secuencia, mientras la búsqueda de un elemento particular es muy rápida (del orden de $O(1)$).

Gestión del almacenamiento

La gestión del almacenamiento es un problema para todas las formas no limitadas, porque el diseñador de la biblioteca debe considerar políticas específicas para reservar y liberar nodos del heap. Un enfoque ingenuo usará simplemente las funciones globales `new` y `delete`, pero esta estrategia muchas veces exhibirá una eficacia muy pobre en tiempo de ejecución. Además, la gestión del almacenamiento en ciertas plataformas puede ser bastante compleja (como con espacios de direcciones segmentados bajo ciertos sistemas operativos de computadoras personales), y requerir por tanto que todas las clases utilicen una política adaptada a la plataforma, en vez de usar una general que el diseñador de bibliotecas asuma que funcionará en todas las circunstancias. Aislando claramente estos patrones de gestión del almacenamiento, se puede construir una biblioteca robusta pero adaptable.

La Figura 9.5 ilustra el mecanismo que se ha elegido para proporcionar gestión del almacenamiento a esta biblioteca⁶. Pasamos a efectuar un recorrido del escenario descrito en este diagrama de objetos:

⁶ Una nota histórica: fueron necesarias algo así como cuatro iteraciones de la arquitectura de la biblioteca para llegar a este mecanismo, que —y no es sorprendente— resultó ser el más simple de todos. Alternativas anteriores, que se rechazaron finalmente, demostraron ser menos adaptables, así como difíciles de explicar, porque tendían a exponer las interioridades de la implementación a los clientes poco cuidadosos.

- unCliente invoca la operación anadir sobre una instancia de ColaNoLimitada (más precisamente, sobre una instancia de una instancia de ColaNoLimitada).
- El objeto ColaNoLimitada a su vez delega la responsabilidad de esta operación en su miembro objeto rep, una instancia de la clase NoLimitada.
- NoLimitada reserva espacio para una nueva instancia de Nodo invocando su función miembro static new.
- La instancia Nodo delega a su vez la responsabilidad de la reserva de memoria en su gestor de almacenamiento, que se ha hecho visible a la clase ColaNoLimitada (y a su vez a las clases NoLimitada y Nodo) como argumento de la plantilla. Este gestor de almacenamiento es compartido por todas las instancias de la clase, y sirve así para proporcionar una política de gestión de almacenamiento consistente para todas las clases.

Tratando el gestor de almacenamiento como argumento para todas las estructuras no limitadas concretas, se desconecta efectivamente esa política de gestión de memoria de su implantación, y se hace posible para los usuarios de la biblioteca la inserción de su propia política de gestión del almacenamiento sin cambiar la biblioteca. Éste es un ejemplo clásico de extensibilidad mediante instanciación en vez de herencia.

El último requisito que se plantea para los gestores de almacenamiento es que proporcionen el mismo protocolo bien definido. Específicamente, se requiere que todos los gestores de memoria exporten las funciones miembro reservar y liberar, que suministran y liberan memoria, respectivamente. Por ejemplo, considérese la política de gestión de almacenamiento más simple (y más inocente):

```
class SinGestionar {
public:

    static void* reservar(size_t tam)
    { return ::operator new(tam); }
    static void liberar(void* p, size_t)
    { ::operator delete(p); }

private:
    SinGestionar() {}
    SinGestionar(SinGestionar&) {}
    void operator=(SinGestionar&) {}
    void operator==(SinGestionar&) {}
    void operator!=(SinGestionar&) {}
};
```

Nótese la pauta que se usa para prevenir que los clientes copien, asignen o prueben la igualdad de instancias de esta clase.

Se implanta el protocolo para la clase `SinGestionar` mediante llamadas inline a los operadores globales `new` y `delete`. Se llama a esta política *sin gestionar*, porque efectivamente no hace nada más allá que la política por defecto proporcionada por el lenguaje. Una política mucho mejor se llama *gestionada*. Bajo esta política, los nodos se reservan y liberan desde un embalse común de memoria. Los nodos sin utilizar de cualquier tipo se devuelven a una lista libre, y la asignación toma nodos de esta lista libre a menos que esté vacía, en cuyo caso se toma del heap otro trozo de memoria libre. De este modo, se reduce enormemente el número de veces que hay que recurrir a los servicios de gestión del almacenamiento del sistema operativo: la asignación de memoria implica ahora la manipulación de algunos punteros, lo que es mucho más rápido⁷.

Pensando un poco en su semántica, se puede hacer mucho mejor la abstracción del embalse. Por ejemplo, se podrían proporcionar operaciones que permitiesen a un cliente preasignar un trozo de memoria, antes de su uso. Análogamente, se podría permitir a un cliente desfragmentar sus bloques, y posiblemente devolver los bloques no utilizados al montículo (*heap*). También se podrían suministrar operaciones que permitiesen a un cliente elegir el tamaño del bloque, de forma que se ajustase la asignación de memoria a las necesidades de la implantación (por ejemplo, fijando un tamaño del bloque óptimo para el tamaño de las clases que se están creando, o haciendo que los bloques se alineasen con las fronteras de palabra).

Dadas estas decisiones de diseño, se podría expresar la abstracción del almacén en la siguiente clase (que no es una plantilla) de soporte:

```
class Almacen {
public:

    Almacen(size_t tamTrozo);
    ~Almacen();

    void* reservar(size_t);
    void liberar(void*, size_t);

    void preasignar(unsigned int numeroDeTrozos);
    void recobrarBloquesSinUsar();
    void purgarBloquesSinUsar();

    size_t tamanioBloque() const;
    unsigned int bloquesTotales() const;
    unsigned int numeroDeBloquesSucios() const;
    unsigned int numeroDeBloquesSinUsar() const;

protected:

    struct Elemento ...
```

⁷ En C++, el operador global `new` invoca en última instancia algún tipo de servicio `malloc`, que es una operación relativamente costosa.

```

struct Bloque ...

Bloque* cabeza;
Bloque* bloquesSinUsar;
size_t repTamBloque;
size_t tamBloqueUsable;

Bloque* obtenerBloque(size_t tam);

};

```

Esta clase usa dos clases anidadas `Elemento` y `Bloque`. Cada instancia de la clase `Almacen` gestiona una lista enlazada de objetos `Trozo` que son en realidad meros trozos de memoria, pero se tratan como si ellos mismos fueran listas enlazadas de instancias de `Elemento` (este es uno de los secretos importantes gestionados por la clase `Almacen`). Cada bloque puede manejar elementos de diferente tamaño, y así, por eficiencia, se ordena la lista de bloques de menor a mayor.

La clase de gestión del almacenamiento gestionada puede escribirse ahora como sigue:

```

class Gestionada {
public:

    static Almacen& almacen;

    static void* reservar(size_t tam)
        {return almacen.reservar(tam);}
    static void liberar(void* p, size_t tam)
        {almacen.liberar(p, tam);}

private:

    Gestionada() {}
    Gestionada(Gestionada&) {}
    void operator=(Gestionada&) {}
    void operator==(Gestionada&) {}
    void operator!=(Gestionada&) {}

};

```

Esta clase proporciona el mismo protocolo público que la clase `sinGestionar`. A causa de la subespecificación intencionada del C++ en la semántica de los modelos, la conformidad con el protocolo sólo se comprueba cuando se compila una instancia de una clase como `ColaNoLimitada`, momento en el que se empareja una clase concreta con el argumento formal del modelo, `GestorAlmacenamiento`.

Nótese que la clase `Gestionada` tiene un objeto miembro static de la clase

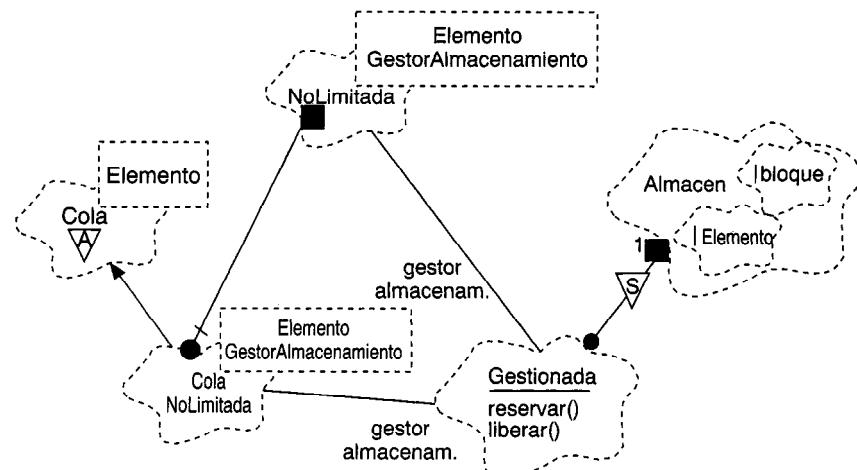


Figura 9.6. Clases de gestión del almacenamiento.

Almacen. De este modo, es posible tener varias estructuras concretas no gestionadas que comparten el mismo almacén. Por supuesto, las diversas estructuras no gestionadas pueden definir su gestor de almacenamiento y por tanto su propio almacén, dando así al desarrollador un control completo sobre la política de gestión del almacenamiento.

La Figura 9.6 muestra un diagrama de clases que ilustra las diversas clases que colaboran para proporcionar una política de almacenamiento gestionada. Sólo se muestra una asociación entre las clases Gestionada y sus clientes ColaNoLimitada y NoLimitada, porque esta asociación sólo se manifestará en una instancia específica de las clases.

Una parte de las decisiones arquitectónicas debe implicar el empaquetamiento físico de estas clases de soporte. En la Figura 9.7 se ilustra la arquitectura de módulos de estas clases. La partición concreta que se elige aísla aquellas clases que es más probable que cambien.

Condiciones excepcionales

Aunque se puede usar el propio lenguaje C++ para reforzar la mayoría de las suposiciones estáticas sobre una abstracción (las violaciones a estas suposiciones pueden detectarse en tiempo de compilación), hay que usar algún otro mecanismo para informar de cualquier violación dinámica, como intentar añadir un elemento a una cola limitada que ya está llena, o eliminar un elemento de una cola vacía. En esta biblioteca se decide aplicar las utilidades de excepción del C++ [14]. La arquitectura usa una jerarquía de clases de excepciones, y las separa de los mecanismos implicados en la información acerca de ellas.

Se comienza con una clase base excepción, cuyo protocolo es evidente:

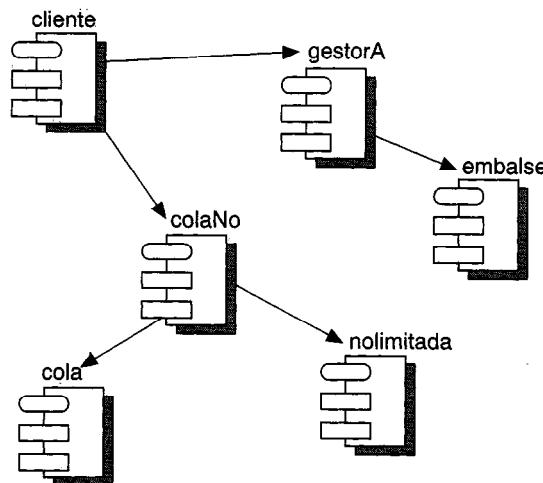


Figura 9.7. Módulos de gestión del almacenamiento.

```

class Excepcion {
public:
    Excepcion(const char* nombre, const char* quien,
              const char* que);
    void mostrar() const;
    const char* nombre() const;
    const char* quien() const;
    const char* que() const;

protected:
    ...
};

  
```

Para toda condición excepcional, se puede asignar su nombre, quién la lanzó y por qué se lanzó. Además, se proporcionan medios para visualizar una excepción en algún flujo de salida oculto para el cliente.

Un análisis del dominio de las diversas clases de la biblioteca revela las siguientes condiciones excepcionales, que se declaran como subclases de la clase base `Excepcion`:

- `ErrorContenedor`
- `Duplicado`
- `PatronIllegal`

- `EsNull`
- `ErrorLexico`
- `ErrorMatematico`
- `NoEncontrado`
- `NoNull`
- `NoRaiz`
- `Desbordamiento`
- `ErrorRango`
- `ErrorAlmacenamiento`
- `DesbordamientoInferior`

Por ejemplo, la declaración de la excepción `Desbordamiento` aparece como sigue:

```
class Desbordamiento : public Expcion {
public:

    Desbordamiento(const char* quien, const char* que)
        : Expcion(<>Desbordamiento>, quien, que) {}

};
```

La responsabilidad de esta clase sólo exige que conozca su nombre, que pasa al constructor de su superclase.

Bajo este mecanismo, las funciones miembro de las clases de la biblioteca sólo *lanzan* excepciones; ninguna de ellas las recoge, principalmente porque no hay nada que ninguna de ellas pueda hacer para responder significativamente a una situación excepcional. Por convención, sólo se lanzan excepciones como parte de una aserción sobre alguna condición. Una *aserción* no es más que una expresión Booleana de alguna condición cuyo valor de verdad debe conservarse. Para simplificar la implantación de la biblioteca, se introduce por tanto la siguiente función no miembro:

```
inline void _asercion(int expresion, const Expcion& excepcion)
{
    if (!expresion)
        throw(excepcion);
}
```

Se declara esta función como inline por razones de eficiencia.

La ventaja de proporcionar esta función es que localiza todos los lanzamientos de excepción (en C++, `throw` tiene la sintaxis de una llamada a función). Así, para compiladores que aún no soportan excepciones, se puede usar una directiva del compilador (`-D` para la mayoría de los compiladores de C++) para redefinir esta mención de `throw` como una llamada a alguna otra función no miembro que muestre la excepción y haga finalizar el programa:

```
void _recoger(const Excepcion& e)
{
    cerr << "EXCEPCION: ";
    e.mostrar();
    exit(1);
}
```

Ahora considérese la implantación de la función miembro `insertar` de `Limitada`:

```
template<class Elemento, unsigned int Tamanio>
void Limitada<Elemento, Tamanio>::insertar(const Elemento&
                                             elemento)
{
    unsigned int cuenta = longitud();
    _asercion((cuenta < Tamanio), Desbordamiento("Limitada::insertar",
                                                   "estructura llena"));
    if (!cuenta)
        comienzo = final = 1;
    else {
        comienzo--;
        if (!comienzo)
            comienzo = Tamanio;
    }
    rep[comienzo - 1] = elemento;
}
```

En esta implantación, se establece (*aserción**) que la longitud actual de la estructura debe ser menor que su tamaño limitado. Si esta condición se evalúa a falso, se lanza la excepción `Desbordamiento`.

Un aspecto muy importante de este uso de las excepciones es que se garantiza que no corrompen el estado de ningún objeto que lance una excepción, excepto en el caso de condiciones de falta de memoria (situación en la cual lo más fácil es que la suerte esté echada de todos modos). En el diseño, las funciones miembro siempre realizan una aserción antes de que se haga ningún cambio al estado del objeto. Por ejemplo, en la implantación de la función miembro `insertar` vista arriba, se llama primero a un selector (el cual, por diseño, se garantiza que conserva el estado del objeto), a continuación se comprueba que se satisfacen todas las precondiciones para la función, y sólo entonces se altera el estado del objeto. Éste es un estilo que se sigue escrupulosamente y consistentemente, y debería conservarse por parte de todas las subclases derivadas de esta biblioteca.

La Figura 9.8 ilustra las clases que colaboran para formar este mecanismo.

* También se podría traducir por asertos. (N. del T.)

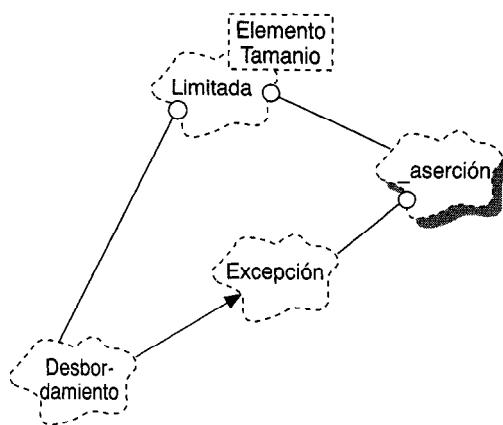


Figura 9.8. Clases de excepción.

Iteración

La iteración es otro patrón arquitectónico que se encuentra en esta biblioteca. Como se definió en el Capítulo 3, un iterador es una operación que permite el acceso a todas las partes de un objeto en algún orden bien definido. No son sólo los clientes de la biblioteca los que necesitan este comportamiento, sino que también se necesitan los iteradores en la implantación de la propia biblioteca, para llevar a cabo ciertas responsabilidades de cada clase base.

Cuando se introducen iteradores, hay dos alternativas de diseño: se puede definir la iteración como parte del protocolo de un objeto, o se pueden inventar objetos separados que actúen como agentes responsables de iterar a lo largo de una estructura. Se elige la segunda alternativa por dos razones poderosas:

- Proporcionando clases iterador separadas, se posibilita tener varios objetos iteradores trabajando sobre el mismo objeto.
- La iteración rompe ligeramente el encapsulamiento del estado de un objeto; separando el comportamiento de la iteración del resto del protocolo de una abstracción, se proporciona una separación de intereses mucho más clara.

Para cada estructura se proporcionan dos formas de iteración. En concreto, un iterador *activo* requiere que los clientes avancen explícitamente el iterador; en una expresión lógica, un iterador *pasivo* aplica una función suministrada por el cliente, y por tanto requiere menos colaboración por parte de éste⁸. Por razones de seguridad de tipos, se definen iteradores diferentes para cada tipo de estructura.

⁸ Los iteradores pasivos implantan una función «aplicar» (*apply*), un modismo utilizado de forma habitual en lenguajes de programación funcional.

Por ejemplo, considérese el iterador activo para la clase Cola:

```
template <class Elemento>
class IteradorActivoCola {
public:

    IteradorActivoCola(const Cola<Elemento>&);
    ~IteradorActivoCola();

    void reiniciar();
    int siguiente();

    int estaHecho() const;
    const Elemento* elementoActual() const;

protected:

    const Cola<Elemento>& cola;
    int indice;

};
```

En el momento de su construcción, todo iterador se liga a un objeto particular. La iteración comienza en el «extremo superior» de una estructura, signifique lo que signifique eso para la abstracción dada.

Un cliente obtiene un puntero al elemento actual mediante la función miembro `elementoActual`; el puntero es null si la iteración se ha completado o si la estructura está vacía. Un cliente avanza el iterador al siguiente elemento sucesivo mediante la función miembro `siguiente` (que devuelve 0 si el iterador no pudo avanzar, quizás porque la iteración ya había sido completada). El selector `estaHecho` permite al cliente informarse sobre el progreso de la iteración, y devuelve 0 si la iteración se ha completado o si la estructura está vacía. La función miembro `reiniciar` permite múltiples recorridos sobre el mismo objeto.

Como ejemplo, dadas las declaraciones siguientes:

```
ColaLimitada<EventoRed> colaEventos;
```

el siguiente fragmento de código usa un iterador activo para visitar todos los elementos de la cola, desde la cabecera hasta el final:

```
IteradorActivoCola<EventoRed> iter(colaEventos);

while (!iter.estaHecho()) {
    iter.elementoActual()->despachar();
    iter.siguiente();
}
```

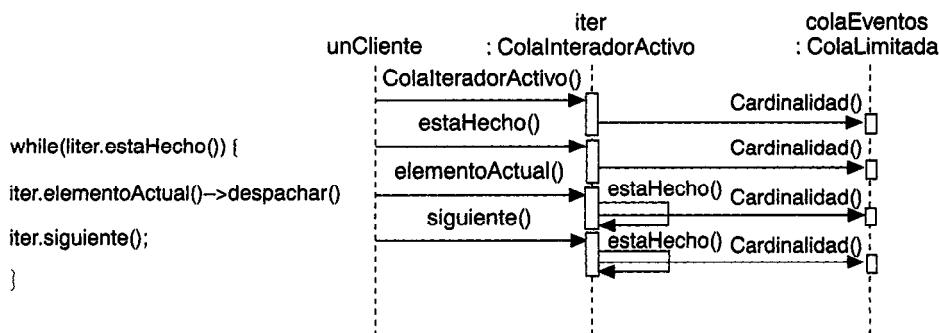


Figura 9.9. Mecanismo de iteración.

El diagrama de interacción de la Figura 9.9 ilustra este escenario, y además revela algunos de los secretos de la implantación del iterador. De hecho, vamos a considerar con más detalle la implantación de este mecanismo.

El constructor de la clase `IteradorActivoCola` se liga a sí mismo a la cola dada y llama a la función miembro `protected cardinalidad` para determinar cuántos elementos hay en la cola. Así, se puede escribir:

```

template<class Elemento>
IteradorActivoCola<Elemento>::IteradorActivoCola(const Cola
<Elemento>& c)
: cola(c),
indice(c.cardinalidad() ? 0 : -1) {}
  
```

La clase `IteradorActivoCola` es una amiga de la clase `Cola`, y esa es la razón por la que el iterador puede invocar la función miembro `protected cardinalidad`.

La operación del iterador `estaHecho` comprueba que su índice está actualmente dentro de la extensión del objeto `cola` al que está ligado:

```

template<class Elemento>
int IteradorActivoCola<Elemento>::estaHecho() const
{
    return ((indice < 0) || (indice >= cola.cardinalidad()));
}
  
```

`elementoActual` devuelve un puntero al elemento sobre el cual se centra el iterador. Implantando la clase iterador como un índice en un objeto `cola`, es posible añadir y eliminar elementos de la cola durante la iteración de forma segura.

```

template<class Elemento>
const Elemento* IteradorActivoCola<Elemento>::elementoActual()
const
  
```

```

{
    return estaHecho() ? 0 : &cola.elementoEn(indice);
}

```

Una vez más, la clase iterador invoca una función miembro protected expuesta de la cola. Nótese que esta operación es eficiente cuando se usa ya sea con una cola limitada o no limitada. Para colas limitadas, `elementoEn` es simplemente una operación de indexación. Para colas no limitadas, `elementoEn` conllevaría en el caso peor el recorrer toda su representación de lista enlazada. Recuérdese, además, que el diseño de la clase `NoLimitada` toma nota del último elemento referenciado, lo que significa que encontrar el siguiente elemento (que es lo que sucede cuando avanza el iterador) es una simple operación con punteros.

La operación del iterador `siguiente`, de hecho, simplemente avanza su índice y comprueba entonces si se ha salido del extremo de la cola:

```

template<class Elemento>
int IteradorActivoCola<Elemento>::siguiente()
{
    indice++;
    return !estaHecho();
}

```

El diseño de la clase iterador motiva así dos de las funciones miembro protected que se proporcionaron en la clase base abstracta `Cola`, a saber, `cardinalidad` y `elementoEn`. Construyendo estos miembros funciones virtuales puras, se hace responsabilidad de cada clase de cola concreta proporcionar una implantación consistente con su representación óptima para ésta.

Anteriormente se indicó que una implicación importante de las decisiones arquitectónicas es que un cliente puede copiar, asignar y probar la igualdad entre instancias de la misma clase base abstracta, incluso aunque cada una tenga una representación diferente. Se consigue esta capacidad mediante un uso elegante de los iteradores y las funciones auxiliares. Este estilo permite, en la clase base abstracta, recorrer cualquier estructura de forma independiente de la implantación. Por ejemplo, en la clase `cola` se encuentra:

```

template<class Elemento>
Cola<Elemento>& Cola<Elemento>::operator=(const Cola<Elemento>& c)
{
    if (this == &c)
        return *this;
    ((Cola<Elemento>&)c).bloquear();
    purgar();
    IteradorActivoCola<Elemento> iter(c);
    while (!iter.estaHecho()) {
        anadir(*iter.elementoActual());

```

```

    iter.siguiente();
{
((Cola<Elemento>&)c).desbloquear();
return *this;
}

```

Este algoritmo usa un modismo para bloquear y desbloquear el objeto cola, que se explicará en la sección siguiente.

La asignación procede recorriendo la estructura del argumento c, utilizando un iterador activo de cola. Se aplica la función auxiliar protected purgar para borrar inicialmente la cola, y se añaden entonces nuevos elementos a la estructura mediante la función auxiliar protected anadir. El hecho de que la iteración dependa del comportamiento polimórfico de las funciones auxiliares de la clase base es lo que hace posible la copia, asignación y prueba de igualdad entre objetos que tienen la misma estructura, pero representaciones diferentes.

Un iterador pasivo es una *aplicación*, lo que significa que aplica alguna función a todos los elementos de la estructura. La siguiente declaración proporciona el iterador para la clase Cola:

```

template <class Elemento>
class IteradorPasivoCola {
public:

    IteradorPasivoCola(const Cola<Elemento>&);

    ~IteradorPasivoCola();

    int aplicar(int (*)(const Elemento&));

protected:

    const Cola<Elemento>& cola;

};

```

Los iteradores pasivos operan sobre todos los elementos de la estructura con una operación (lógicamente) única. La función aplicar visita cada elemento de la estructura e invoca la función suministrada sobre cada uno. Continúa hasta que alcanza el último elemento, o hasta que la función suministrada devuelve un valor de 0 (en tal caso, la propia aplicar devuelve 0, lo que indica que la iteración no fue completa).

Sincronización

Todo marco de referencia general debe considerar los problemas de la concurrencia. Bajo sistemas operativos como UNIX, OS/2 y Windows NT, por ejem-

plo, las aplicaciones pueden estar formadas de múltiples procesos ligeros⁹. A menos que se otorgue una consideración especial, la mayoría de las clases simplemente no funcionarán en tal entorno: cuando dos o más tareas interactúan con el mismo objeto, los objetos activos deben cooperar de alguna forma para evitar corromper el estado del objeto compartido. Como se describió anteriormente, hay básicamente dos enfoques para la gestión de procesos, representados por las formas vigilada y sincronizada de una clase.

El diseño de esta biblioteca realiza la siguiente suposición: los desarrolladores que se preocupen por la concurrencia habrán transportado o implantado al menos una clase *Semaforo* para sincronizar procesos ligeros. Otros clientes no se preocuparán, y no echarán de menos el disponer de las formas vigilada o sincronizada de las estructuras (y apreciarán no tener que pagar la sobrecarga). Las formas vigilada y sincronizada son por tanto una parte independiente, una capa de la biblioteca, y cuentan con que habrá implantaciones locales de este mecanismo de concurrencia. Las únicas dependencias de la biblioteca sobre la implantación local están aisladas intencionadamente en la implantación de la clase *Semaforo*, cuyo interfaz aparece como sigue:

```
class Semaforo {
public:

    Semaforo ();
    Semaforo (const Semaforo &);
    Semaforo (unsigned int cuenta);
    ~Semaforo ();

    void coger();
    void soltar();

    unsigned int ningunoPendiente() const;

protected:
    ...

};
```

Al igual que se hizo para la gestión del almacenamiento, se decide separar las políticas de sincronización de procesos de su implantación. Por esta razón, la presentación del modelo de toda forma vigilada importa un *guardián*, responsable de proporcionar una ligadura con la implantación local de un semáforo o

⁹ Un proceso ligero es aquel que se ejecuta en el mismo espacio de direcciones que sus hermanos. En contraste, la función *fork* de UNIX proporciona procesos pesados que requieren servicios especiales del sistema operativo para la comunicación interproceso. Para C++, la biblioteca de tareas de AT&T proporciona una abstracción semitransportable de los procesos ligeros bajo UNIX. Los procesos ligeros también están disponibles directamente bajo OS/2 y Windows NT. La biblioteca de clases de Smalltalk proporciona la clase *Process* en soporte de su modelo para los procesos ligeros.

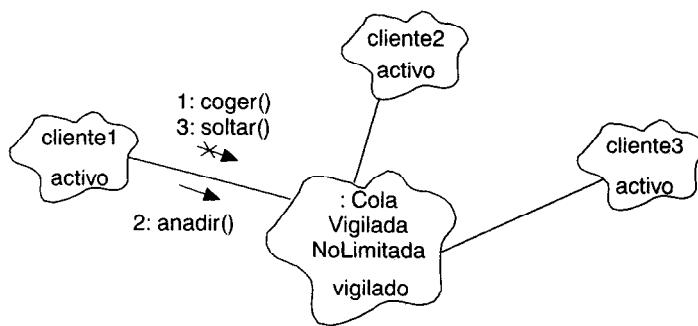


Figura 9.10. Mecanismo de proceso vigilado.

su equivalente. Análogamente, la presentación del modelo para cualquier forma sincronizada importa un *monitor*, que es parecido a un semáforo pero, como se verá, permite un grado mayor de concurrencia.

Como se ilustró en la Figura 9.3, una clase vigilada es una subclase directa de su clase limitada o no limitada concreta; una clase vigilada contiene un guardián como objeto miembro. Todas las clases vigiladas introducen las funciones miembro *coger* y *soltar*, que permiten a un agente activo conseguir acceso exclusivo al objeto. Por ejemplo, considérese la clase *ColaVigiladaNoLimitada*, que es un tipo de *ColaNoLimitada*:

```
template<class Elemento, class GestorAlmacenamiento, class
Guardian>
class ColaVigiladaNoLimitada : public ColaNoLimitada<Elemento,
GestorAlmacenamiento> {
public:

    ColaVigiladaNoLimitada ();
    virtual ~ColaVigiladaNoLimitada ();

    virtual void coger();
    virtual void soltar();

protected:

    Guardian guardian;
};
```

Para esta biblioteca se proporciona el interfaz de un guardián predefinido: la clase *semaforo*. Los usuarios de esta biblioteca deben completar la implantación de esta clase, de acuerdo con las necesidades de la definición local de los procesos ligeros.

Como se ilustra en la Figura 9.10, los clientes que usan objetos vigilados deben seguir el simple protocolo de coger primero el objeto, operar sobre él, y des-

pués soltarlo (especialmente en presencia de cualquier excepción que se haya lanzado). Hacer otra cosa se considera socialmente inapropiado, porque un comportamiento aberrante por parte de un agente deniega el uso honrado por parte de otros agentes. Coger un objeto vigilado y no soltarlo adecuadamente bloquea indefinidamente ese objeto; soltar un objeto que el agente no había cogido es subversivo. Por último, ignorar por completo el protocolo coger/soltar es simplemente irresponsable, porque las tareas intercaladas pueden corromper el estado del objeto compartido.

El principal beneficio que ofrece la forma vigilada es su simplicidad, aunque requiere una acción colectiva honrada de todos los agentes que manipulen el mismo objeto. Otra característica clave de la forma vigilada es que permite a los agentes formar regiones críticas, en las que se garantiza que diversas operaciones realizadas sobre el mismo objeto se tratan como una transacción atómica.

Igual que en el mecanismo de gestión del almacenamiento, la presentación del modelo de las formas vigiladas importa el guardián en vez de hacerlo una característica inmutable. Esto hace posible para los desarrolladores de la biblioteca el introducir nuevas políticas de sincronización. Usando la clase predefinida `Semaforo` como guardián, la política por omisión de la biblioteca es dar a cada objeto de la clase su propio semáforo. Esta política es aceptable sólo hasta el punto en que el número total de procesos alcance algún límite práctico fijado por la implantación local.

Una política alternativa implica que varios objetos vigilados comparten el mismo semáforo. Un desarrollador sólo necesita producir una nueva clase guardián que proporcione el mismo protocolo que `Semaforo` (pero no es necesariamente una subclase de `Semaforo`). Este nuevo guardián podría contener entonces un `semaforo` como objeto miembro estático, lo que quiere decir que el semáforo es compartido por todas sus instancias. Instanciando una forma vigilada con este nuevo guardián, el desarrollador de la biblioteca introduce una política diferente, por la que todos los objetos de esa clase instanciada comparten el mismo guardián, en vez de haber un guardián por objeto. La potencia de esta política surge cuando la nueva clase guardián se usa para instanciar otras estructuras: todos esos objetos comparten en última instancia el mismo guardián. El desplazamiento de la política es sutil, pero muy poderoso: no sólo reduce el número de procesos en la aplicación, sino que también permite a un cliente bloquear globalmente un grupo de objetos que de otro modo no están relacionados. Coger un objeto de este tipo bloquea todos los demás objetos que comparten este nuevo guardián, incluso si esos objetos son de tipos completamente diferentes.

Una clase sincronizada es también una subclase directa de su clase limitada o no limitada concreta. Una clase sincronizada contiene un monitor como objeto miembro, cuyo protocolo está definido por la siguiente clase base abstracta:

```
class Monitor {  
public:  
    Monitor();
```

```

    Monitor(const Monitor&);
    virtual ~Monitor();

    virtual void cogerParaLectura() = 0;
    virtual void cogerParaEscritura() = 0;
    virtual void soltarDeLectura() = 0;
    virtual void soltarDeEscritura() = 0;

protected:
    ...
};


```

Existen dos tipos básicos de sincronización de procesos mediante monitores como éste:

- Simple Garantiza la semántica de una estructura en presencia de múltiples hilos de control, con un solo lector o escritor.
- Múltiple Garantiza la semántica de una estructura en presencia de múltiples hilos de control, con múltiples lectores simultáneos o un solo escritor.

Un *escritor* es un agente que altera el estado de un objeto; los escritores son aquellos agentes que invocan funciones miembro modificadoras. Un *lector* es un agente que opera sobre un objeto, pero conserva su estado; los lectores son aquellos objetos que sólo invocan funciones selectoras. La forma múltiple proporciona por tanto la mayor cantidad de paralelismo real. Se pueden implantar estas dos políticas como subclases de la clase base abstracta `Monitor`. Ambas formas, simple y múltiple, pueden construirse sobre la clase `Semaforo`.

A diferencia de la forma vigilada, las clases sincronizadas no introducen ninguna función miembro nueva; en vez de eso, redefinen todas las funciones miembro virtuales heredadas de su superclase. La semántica añadida por una clase sincronizada hace que cada función miembro se trate como una transacción atómica. Mientras los clientes de formas vigiladas deben coger y soltar explícitamente un objeto para conseguir acceso exclusivo, las formas sincronizadas proporcionan esta exclusividad sin requerir ninguna acción especial por parte de sus clientes.

Se consigue esta exclusividad mediante un mecanismo de bloqueo, como se mostró en la Figura 9.11. Los monitores colaboran con las instancias de las clases predefinidas `BloqueoLectura` y `BloqueoEscritura` para lograr la invocación exclusiva de cada función miembro individual. En este mecanismo, un bloqueo contiene un semáforo o un monitor como agente responsable de la sincronización de procesos, y el bloqueo es responsable de coger este agente cuando se realiza la construcción y soltarlo cuando se realiza la destrucción. Por ejemplo, considérese la declaración de la clase `BloqueoLectura`:

```
class BloqueoLectura {
```

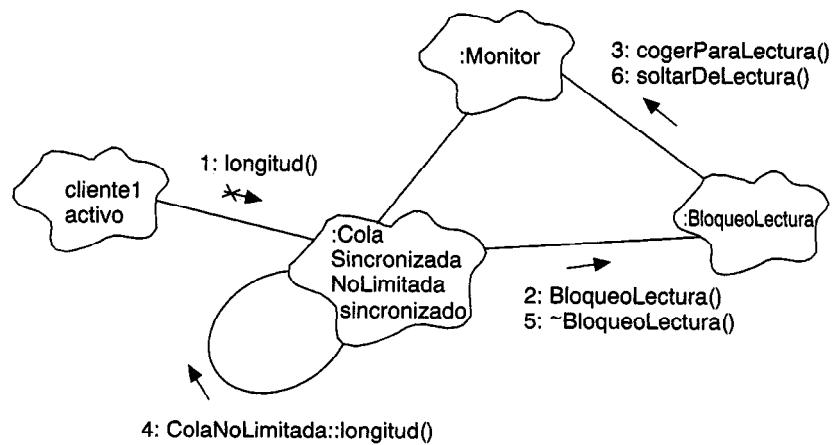


Figura 9.11. Mecanismo de proceso sincronizado.

```

public:

    BloqueoLectura (const Monitor& m)
        : monitor(m) {monitor.cogerParaLectura();}
    ~BloqueoLectura ()
        {monitor.soltarDeLectura();}

private:
    Monitor& monitor;
};
  
```

Separando las abstracciones del bloqueo y su monitor, el diseño permite a un cliente enganchar una política diferente al mecanismo de bloqueo. La declaración de la clase `BloqueoEscritura` es igual de simple, excepto en que usa el protocolo del monitor para escritura.

La definición de cada función miembro en una forma sincronizada usa bloqueos que envuelven la operación correspondiente heredada de su superclase. Por ejemplo, considérese la implantación de la función miembro `longitud` para la cola sincronizada y no limitada:

```

template<class Elemento, class GestorAlmacenamiento, class
Monitor>
unsigned int ColaSincronizadaNoLimitada<Elemento,
GestorAlmacenamiento,
Monitor>::longitud()
const
{
    BloqueoLectura bloquea(monitor);
  
```

```

    return ColaNoLimitada<Elemento,
        GestorAlmacenamiento>::longitud();
}

```

Este código implanta directamente el mecanismo descrito en la Figura 9.11. En general, se usan instancias de la clase `BloqueoLectura` para todos los selectores sincronizados, y se usan instancias de la clase `BloqueoEscritura` para todos los modificadores sincronizados. La elegancia simple de este diseño es lo que garantiza que toda función miembro representa una acción atómica, incluso en presencia de excepciones y sin ninguna actuación explícita por parte del lector o escritor.

En realidad, los clientes que usan objetos sincronizados no necesitan seguir ningún protocolo especial, porque el mecanismo de sincronización de procesos está manejado de forma implícita, y por tanto es menos propenso a los interbloqueos y bloqueos activos que pueden resultar del uso incorrecto de las formas vigiladas. Sin embargo, un desarrollador debería elegir una forma vigilada en vez de una forma sincronizada si es necesario invocar varias funciones miembro juntas como una transacción atómica; la forma sincronizada sólo garantiza que sean atómicas las funciones miembro individuales.

La arquitectura deja las formas sincronizadas relativamente libres de cualquier circunstancia que pueda llevar a un bloqueo entre procesos*. Por ejemplo, asignar un objeto a sí mismo o probar la igualdad de un objeto consigo mismo es potencialmente peligroso porque en concepto requiere bloquear los elementos izquierdo y derecho de tales expresiones, que en estos casos son el mismo objeto. Una vez construido, un objeto no puede cambiar su identidad, por lo que estas pruebas de auto—identidad se realizan primero, antes de que se bloquee uno u otro objeto. Esta es precisamente la razón por la que la implantación anterior de `operator=` incluía una prueba de auto-identidad, como indica la siguiente versión abreviada:

```

template<class Elemento>
Cola<Elemento>& Cola<Elemento>::operator=(const Cola<Elemento>& c)
{
    if (this == &c)
        return *this;
    ...
}

```

Incluso entonces, las funciones miembro que tienen instancias de la propia clase como argumentos deben diseñarse cuidadosamente para asegurar que tales argumentos se bloquean correctamente. La solución se apoya en el comportamiento polimórfico de dos funciones auxiliares, `bloquear` y `desbloquear`, definidas en todas las clases base abstractas. Cada clase base abstracta proporciona una implantación por defecto de estas dos funciones que no hace nada; las formas sincronizadas proporcionan una implantación que coge y suelta el argu-

mento. Esta es precisamente la razón por la que la implantación anterior de `operator=` incluye llamadas a estas dos funciones, como indica la siguiente versión abreviada:

```
template<class Elemento>
Cola<Elemento>& Cola<Elemento>::operator=(const Cola<Elemento>& c)
{
    ...
    ((Cola<Elemento>&)c).bloquear();
    ...
    ((Cola<Elemento>&)c).desbloquear();
    return *this;
}
```

Aquí se usa el modismo de aplicar una conversión obligada y explícita de tipos para eliminar la propiedad `const` del argumento.

9.3. Evolución

Diseño del interface de las clases

En un marco de referencia como éste, una vez que se han seleccionado los patrones que constituyen su arquitectura, el trabajo que queda es relativamente simple, aunque quizás tedioso. El paso siguiente es tomar tres o cuatro familias de clases (como la cola, conjunto y árbol), implantarlas según esta arquitectura y probarlas respecto a aplicaciones de clientes reales¹⁰.

La parte difícil de esta actividad es decidir sobre un interfaz adecuado para cada clase base. Esto conlleva un diseño de clases aisladas, como se describió en el Capítulo 6, pero también requiere que el diseñador mantenga una perspectiva global para asegurar la consistencia. Por ejemplo, se podría seleccionar el siguiente protocolo para Conjunto:

- **fijarFuncionClaves** Fija una función de claves (*hash*) para los elementos.
- **borrar** Vacía el conjunto.
- **anadir** Añade un elemento al conjunto.
- **eliminar** Elimina un elemento del conjunto.
- **unionConjuntos** Realiza una unión de conjuntos.
- **interseccion** Realiza la intersección con el conjunto dado.
- **diferencia** Elimina los elementos que están en el conjunto dado.

¹⁰ Wirs-Brock ha observado que son necesarias al menos tres aplicaciones de un marco de referencia para validar sus decisiones estratégicas y tácticas [15].

• <code>cardinal</code>	Devuelve el número de elementos del conjunto.
• <code>estaVacio</code>	Devuelve 1 si no hay elementos en el conjunto.
• <code>esMiembro</code>	Devuelve 1 si el elemento dado está en el conjunto.
• <code>esSubconjunto</code>	Devuelve 1 si el conjunto es subconjunto del conjunto dado.
• <code>esSubconjuntoPropio</code>	Devuelve 1 si el conjunto es subconjunto propio del conjunto dado.

Análogamente, se podría seleccionar el siguiente protocolo para la clase `ArbolBinario`:

• <code>borrar</code>	Destruye el árbol y sus hijos.
• <code>insertar</code>	Añade un nodo al extremo superior del árbol.
• <code>anadir</code>	Añade un hijo al árbol.
• <code>eliminar</code>	Elimina un hijo del árbol.
• <code>compartir</code>	Comparte estructuralmente el árbol dado.
• <code>intercambiarHijo</code>	Intercambia el hijo con el árbol dado.
• <code>hijo</code>	Devuelve el hijo dado.
• <code>hijoIzquierdo</code>	Devuelve el hijo izquierdo.
• <code>hijoDerecho</code>	Devuelve el hijo derecho.
• <code>padre</code>	Devuelve el padre del árbol.
• <code>fijarElemento</code>	Fija el elemento asociado con el árbol.
• <code>tieneHijos</code>	Devuelve 1 si el árbol tiene hijos.
• <code>esNull</code>	Devuelve 1 si el árbol es null.
• <code>esCompartido</code>	Devuelve 1 si el árbol está compartido estructuralmente.
• <code>esRaiz</code>	Devuelve 1 si el árbol tiene raíz.
• <code>elementoEn</code>	Devuelve el elemento asociado con el árbol.

Nótese que se usan nombres similares para tipos de operación similares. Se usan también las medidas de calidad dadas por la suficiencia, completud (estado completo) y condición de primitiva (como se describió en el Capítulo 3) para guiar el diseño del interfaz de cada familia.

Clases de soporte

La implantación de la clase cadena revela que el rango de semánticas de tiempo y espacio ofrecido por las clases de soporte `Limitada` y `NoLimitada` no es suficiente para nuestros propósitos. En concreto, la forma limitada es ineficiente respecto al espacio para las cadenas de caracteres, porque hay que instanciar esta forma para la longitud máxima esperada, desperdiando así una tremenda cantidad de espacio en todas las cadenas más cortas. Análogamente, la forma no

limitada es ineficiente respecto al tiempo para las cadenas, porque buscar o insertar un elemento en el medio de la cadena puede requerir un recorrido de toda su estructura de lista encadenada subyacente. Por esta razón, se introduce una tercera forma de representación, que se llama **dinámica**, con las siguientes responsabilidades:

- **Dinámica** La estructura se almacena en el heap como una matriz cuya longitud puede encogerse o crecer.

De este modo, la clase de soporte **Dinámica** ofrece una solución intermedia entre la eficiencia temporal de la forma limitada (ya que los elementos pueden indexarse directamente) y la eficiencia espacial de la forma no limitada (ya que sólo se reserva almacenamiento para tantos elementos como sea necesario).

Puesto que el protocolo de esta clase es idéntico al de las clases **LIMITADA** y **NoLIMITADA**, es trivial añadir este nuevo comportamiento a la biblioteca. En concreto, hay que añadir tres nuevas clases a cada familia (por ejemplo, **CadenaDinamica**, **CadenaDinamicaVigilada** y **CadenaDinamicaSincronizada**). Se incluye así la clase de soporte **Dinámica**, cuya declaración abreviada aparece como sigue:

```
template<class Elemento, class GestorAlmacenamiento>
class Dinamica {
public:
    Dinamica(unsigned int tamBloque);

    ...

protected:
    Elemento* rep;
    unsigned int tamanio;
    unsigned int bloquesTotales;
    unsigned int tamBloque;
    unsigned int comienzo;
    unsigned int final;

    void redimensionar(unsigned int longActual, unsigned int
                        longNueva, int preservar = 1);

    unsigned int expandirIzquierda(unsigned int desde);
    unsigned int expandirDerecha(unsigned int desde);
    void encogerIzquierda(unsigned int desde);
    void encogerDerecha(unsigned int desde);

};

}
```

Las secuencias se dimensionan en múltiplos del argumento del constructor,

tamBloque. De este modo, un cliente puede adaptar cada instancia de esta clase a un tamaño óptimo para su uso.

Como sugiere esta declaración, la implantación de la clase `Dinamica` comparte muchas de las características de las clases `Limitada` y `NoLimitada`. De hecho, puesto que las tres clases tienen sustancialmente el mismo protocolo público, la implantación de cada una de las clases concretas de una familia también es la misma en gran medida.

La implantación de la clase mapa también revela que las formas limitada, no limitada y dinámica necesitan un ajuste para satisfacer la semántica del mapa. En concreto, la búsqueda de la asociación de un elemento en un mapa es inaceptablemente costosa si hay que buscar secuencialmente en una larga secuencia. Se puede mejorar muchísimo la eficiencia si se usa en vez de eso una tabla por claves abierta*.

La abstracción de una tabla por claves abierta es muy clara. Básicamente, tal tabla consta de una matriz de secuencias; cada secuencia se llama un *cubo* o *bucket*. Cuando se pone un elemento en la tabla, se genera primero un valor de clave partiendo del propio elemento, que se usa entonces para seleccionar un cubo específico. Se introduce el elemento en ese cubo, al igual que con las formas limitada, dinámica y no limitada. De este modo, una tabla por claves abierta divide una secuencia larga en varias más pequeñas, acelerando así notablemente las búsquedas.

Se puede capturar esta semántica en la siguiente declaración abreviada:

```
template<class Elemento, class Valor, unsigned int Cubos,
          class Contenedor>
class Tabla {
public:
    Tabla(unsigned int (*clave)(const Elemento&))

    ...

    void fijarFuncionClaves(unsigned int (*clave)(const Elemento&));
    void borrar();
    int ligar(const Elemento&, const Valor&);
    int religar(const Elemento&, const Valor&);
    int desligar(const Elemento&);

    Contenedor* cubo(unsigned int cubo);

    unsigned int tamano() const;
    int estaLigado(const Elemento&) const;
    const Valor* valorDe(const Elemento&) const;
    const Contenedor *const cubo(unsigned int cubo) const;

protected:
```

* *Open hash table* en el original en inglés. (N. del T.)

```
Contenedor rep[Cubos];
...
};
```

Nótese el uso de `Contenedor` como un argumento del modelo, que permite definir la abstracción de una tabla por claves abierta independientemente de la secuencia particular que se use. Por ejemplo, considérese la declaración (muy abreviada) del mapa no limitado, que se construye sobre las clases `Tabla` y `NoLimitada`:

```
template<class Elemento, class Valor, unsigned int Cubos,
          class GestorAlmacenamiento>
class MapaNoLimitado : public Mapa<Elemento, Valor> {
public:
    MapaNoLimitado();
    ...
    virtual int ligar(const Elemento&, const Valor&);
    virtual int religar(const Elemento&, const Valor&);
    virtual int desligar(const Elemento&);

    ...
protected:
    Tabla<Elemento, Valor, Cubos, NoLimitada<Par<Elemento, Valor>,
           GestorAlmacenamiento> > rep;
    ...
};
```

Aquí se instancia la clase `Tabla` con un contenedor `NoLimitada`. La Figura 9.12 ilustra la colaboración de estas clases.

Como medida de la aplicabilidad general de esta abstracción, se puede aplicar también la clase `Tabla` a la implantación de las clases `Conjunto` y `Bolsa`.

Herramientas

En esta biblioteca, el uso principal de los modelos es parametrizar cada estructura con el tipo de elemento que contiene; esta es la razón por la que tales estructuras se llaman con frecuencia *clases contenedor*. Como ilustra la declara-

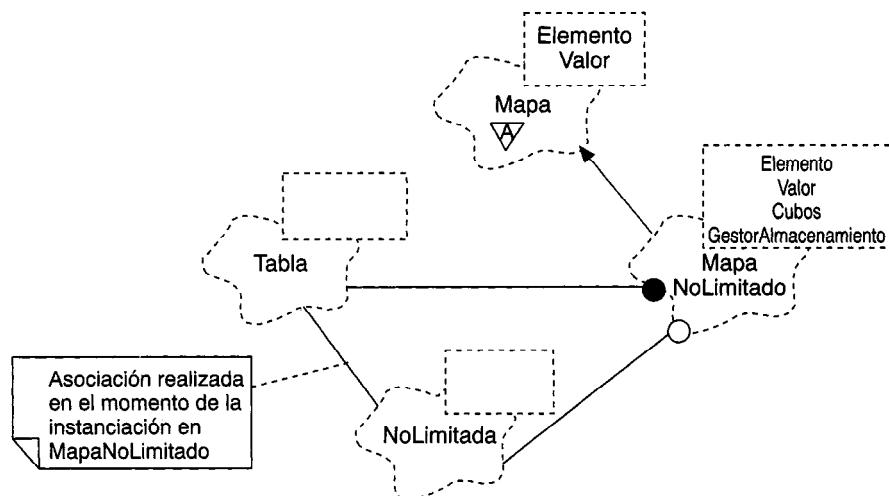


Figura 9.12. Clases de soporte.

ción de la clase **Tabla**, los modelos pueden usarse también para proporcionar cierta información de implantación a una clase.

Una situación aún más sofisticada involucra herramientas que operan sobre otras estructuras. Como se explicó anteriormente, se pueden «objetificar» algoritmos inventando clases cuyas instancias actúan como agentes responsables de llevar a cabo el algoritmo. Este enfoque sigue la idea de Jacobson de un *objeto de control*, cuyo comportamiento proporciona el pegamento por el que otros objetos colaboran en un caso de uso [16]. La ventaja de este enfoque es que permite aprovechar patrones dentro de ciertas familias de algoritmos, formando tramas de herencias. Esto no sólo simplifica su implantación, sino que proporciona una vía para unificar conceptualmente algoritmos similares desde la perspectiva de sus clientes.

Por ejemplo, considérense los algoritmos que buscan patrones dentro de una secuencia. Existen varios de estos algoritmos, con semánticas de tiempo diferentes:

- Simple Se busca la estructura secuencialmente para el patrón dado; en el caso peor, este algoritmo tiene una complejidad temporal del orden de $O(pn)$, donde p es la longitud del patrón, y n es la longitud de la secuencia.
- Knuth-Morris-Pratt Se busca la estructura para el patrón dado, con una complejidad temporal de $O(p + n)$; la búsqueda no requiere mantener copia, lo que hace este algoritmo adecuado para ficheros.

- Boyer-Moore Se busca la estructura para el patrón dado, con una complejidad temporal sublineal de $O(c * (p + n))$, donde $c < 1$ y es inversamente proporcional a p .
- Expresión regular Se busca la estructura para el patrón de expresión regular dado.

Hay al menos tres características comunes de estos algoritmos: todos operan sobre secuencias (y por tanto esperan ciertos protocolos de los objetos que buscan), todos requieren la existencia de una función de igualdad para los elementos que se buscan (porque la operación de igualdad por defecto puede ser insuficiente) y todos tienen sustancialmente la misma firma para su invocación (requieren un objetivo, un patrón y un índice de comienzo).

La necesidad de una operación de igualdad requiere algunas explicaciones. Supóngase, por ejemplo, que se tiene una colección ordenada de registros personales. Se podría desear buscar esta secuencia para un cierto patrón de registros, como grupos de tres registros que son todos del mismo departamento. El uso del operador == de la clase RegistroPersonal no funcionará, porque este operador comprobará seguramente la igualdad basada en algún identificador único. En vez de eso, hay que suministrar una prueba especial de igualdad al algoritmo que interroga sobre el departamento de cada persona (invocando el selector adecuado). Ya que todos los agentes de emparejamiento de patrones requieren una función de igualdad, se puede proporcionar un protocolo común para fijar la función como parte de alguna clase base abstracta. Por ejemplo, se podría usar la declaración siguiente:

```
template<class Elemento, class Secuencia>
class EmparejarPatrones {
public:
    EmparejarPatrones();
    EmparejarPatrones(int (*esIgual)(const Elemento& x, const
                                         Elemento& y));
    virtual ~EmparejarPatrones();

    virtual void
        fijarFuncionEsIgual (int (*)(const Elemento& x, const
                                         Elemento& y));
    virtual int
        emparejar(const Secuencia& objetivo, const Secuencia& patron,
                  unsigned int comienzo = 0) = 0;
    virtual int
        emparejar(const Secuencia& objetivo, unsigned int
                  comienzo = 0) = 0;

protected:
    Secuencia rep;
```

```

int (*esIgual)(const Elemento& x, const Elemento& y);

private:

void operator=(const EmparejarPatrones&) {}
void operator==(const EmparejarPatrones&) {}
void operator!=(const EmparejarPatrones&) {}

};

```

Nótese que se usa otra vez el modismo para la asignación y prueba de igualdad, que evita que los objetos de esta clase o sus subclases se asignen o comparan con otros. Se hace así porque estas operaciones no tienen un significado real cuando se aplican a tales abstracciones agentes.

A continuación se pueden idear subclases concretas, como por ejemplo para el algoritmo Boyer-Moore:

```

template<class Elemento, class Secuencia>
class EmparejarPatronesBM : public EmparejarPatrones<Elemento,
                           Secuencia> {
public:

    EmparejarPatronesBM();
    EmparejarPatronesBM(int (*esIgual)(const Elemento & x,
                                         const Elemento& y));
    virtual ~EmparejarPatronesBM();

    virtual int
    emparejar(const Secuencia& objetivo, const Secuencia&
              patron, unsigned int comienzo = 0);
    virtual int
    emparejar(const Secuencia& objetivo, unsigned int
              comienzo = 0);

protected:

    unsigned int longitud;
    unsigned int* tablaSalto;

    void preprocesar(const Secuencia& patron);
    unsigned int saltoElementos(const Secuencia& patron,
                               const Elemento& elemento);
};


```

El protocolo público para esta clase implanta el de su superclase. Además, se proporcionan dos objetos miembro y dos funciones miembro auxiliares. Uno de los secretos de esta clase es la creación de una tabla temporal que utiliza para

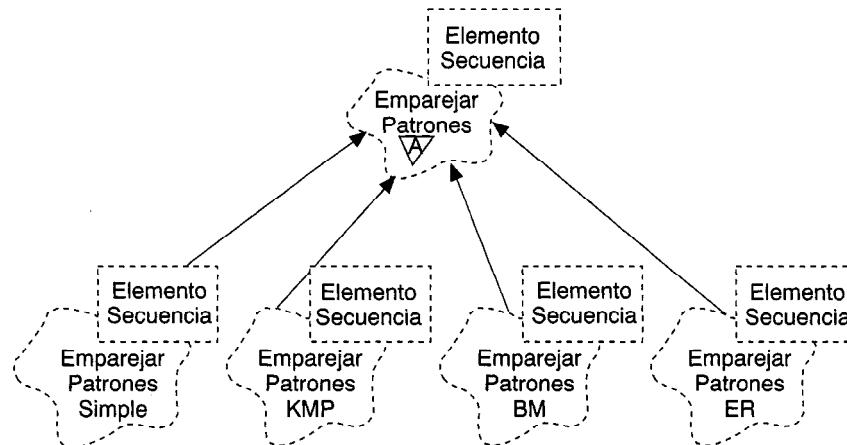


Figura 9.13. Clases de emparejamiento de patrones.

saltar sobre secuencias largas y no emparejadas; estos miembros sirven para implantar este secreto.

Como ilustra la Figura 9.13, se puede construir una jerarquía de clases de emparejamiento de patrones. De hecho, este tipo de jerarquía se aplica a todas las herramientas de la biblioteca, dándole una estructura regular que hace mucho más fácil para los clientes el encontrar las abstracciones que satisfacen mejor su semántica de espacio y tiempo.

9.4. Mantenimiento

Una característica fascinante de los marcos de referencia es que —si están bien hechos— tienden a alcanzar una especie de masa crítica de funcionalidad y adaptabilidad. En otras palabras, si se han seleccionado las abstracciones correctas, y si se ha poblado la biblioteca con un conjunto de mecanismos que funcionan bien juntos, se encontrará que los clientes pronto descubren medios para construir sobre la biblioteca de maneras que sus diseñadores nunca habían imaginado o esperado. A medida que se descubren patrones en el modo en que los clientes utilizan el marco de referencia, va teniendo sentido el codificar estos patrones haciéndolos parte de la biblioteca formalmente. Un signo de que el marco de referencia se ha diseñado bien es que se pueden introducir estos nuevos patrones durante el mantenimiento reusando mecanismos existentes y conservando así la integridad de su diseño.

Un patrón de este tipo que aparece en la biblioteca tiene que ver con el problema de la persistencia. Se podrían encontrar clientes que no quieren o no necesitan toda la potencia de una base de datos orientada a objetos, pero que de vez en cuando necesitan salvar el estado de estructuras como colas y conjuntos, y reconstruir entonces esos objetos en una invocación posterior del programa,

o quizás desde un programa completamente distinto. Puesto que este patrón de uso es tan común, tiene sentido aumentar la biblioteca con un mecanismo simple de persistencia.

Se harán dos suposiciones sobre esta utilidad. Primero, los clientes son responsables de proporcionar un archivo en el cual se introducen y del cual se restauran los elementos. Segundo, los clientes son responsables de asegurar que los elementos tienen el comportamiento necesario para que se los almacene secuencialmente.

Se nos ocurren dos diseños alternativos para esta utilidad. Se podría idear una clase aditiva que ofreciese semántica de persistencia; éste es el enfoque utilizado por muchas bases de datos orientadas a objetos. Otra posibilidad es inventar una clase cuyas instancias actúan como agentes responsables de almacenar varias estructuras. Como parte de la exploración, se podrían intentar ambos enfoques, para ver cuál es más adecuado.

Evidentemente, el estilo aditivo no funciona bien para esta simple forma concreta de persistencia (aunque está indicado para bases de datos orientadas a objetos completos). El usar un estilo aditivo requiere que los clientes que mezclan una abstracción la conecten con sus clases definidas por el usuario, frecuentemente redefiniendo ciertas funciones auxiliares aditivas. Para un agente tan simple, sin embargo, los clientes acabarían escribiendo más código que si hubiesen construido el mecanismo a mano. Esto está claro que no es aceptable, y por eso hay que volverse hacia el segundo enfoque, que requiere poco más que una instancia por parte del cliente.

La Figura 9.14 ilustra el diseño de este mecanismo, en el que se proporciona persistencia mediante el comportamiento de un agente separado. La clase `Persistir` es una amiga de la clase `Cola`, pero se puede aplazar esta asociación introduciendo la siguiente declaración friend en la clase `Cola`:

```
friend class Persistir<Elemento, Cola<Elemento> >;
```

De este modo, la amistad se establece sólo en el momento en que se instancia la clase `Cola`. De hecho, introduciendo una declaración friend similar en todas las clases base abstractas se puede reusar la clase `Persistir` para todas las estructuras de la biblioteca.

La clase parametrizada `Persistir` proporciona las operaciones `colocar` y `obtener`, así como operaciones para fijar sus corrientes de entrada y salida. Se puede capturar esta abstracción en la declaración siguiente:

```
template<class Elemento, class Estructura>
class Persistir {
public:
    Persistir();
    Persistir(iostream& input, iostream& output);
    virtual ~Persistir();
```

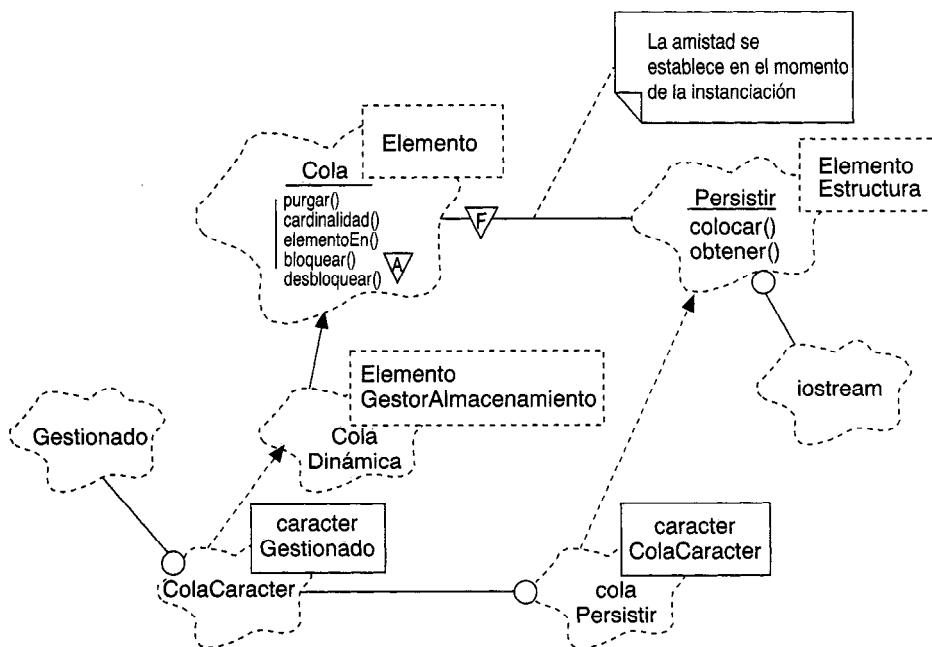


Figura 9.14. Clases de persistencia.

```
    virtual void fijarFlujoEntrada(iostream&);  
    virtual void fijarFlujoSalida(iostream&);  
    virtual void colocar(Estructura&);  
    virtual void obtener(Estructura&);  
  
protected:  
  
    iostream* flujoEnt;  
    iostream* flujoSal;  
  
    ...  
};
```

La implantación de esta clase depende de su amistad con la clase `Estructura`, que se importa como un argumento del modelo. En concreto, `Persistir` depende de la existencia de las funciones auxiliares de la estructura: `purgar`, `cardinalidad`, `elementoEn`, `bloquear` y `desbloquear`. Aquí la regularidad de la biblioteca rinde beneficios: ya que todas las clases base abstractas `Estructura` proporcionan estas funciones auxiliares, se puede usar la clase `Persistir` sin ningún cambio a la arquitectura existente de la biblioteca.

Considérese, por ejemplo, la implantación de `Persistir:colocar:`

```
template<class Elemento, class Estructura>
```

```
void Persistir<Elemento, Estructura>::colocar(Estructura& e)
{
    e.bloquear();
    unsigned int cuenta = e.cardinalidad();
    (*flujoSal) << cuenta << endl;
    for (unsigned int i = 0; i < cuenta; i++)
        (*flujoSal) << e.elementoEn(i);
    e.desbloquear();
}
```

Esta operación usa el mecanismo de bloqueo ya visto, de forma que su semántica funciona para las formas vigilada y sincronizada. El algoritmo procede almacenando el tamaño de la estructura y a continuación sus elementos individuales por orden. Análogamente, la implantación de `Persistir::obtener` es inversa a esta acción:

```
template<class Elemento, class Estructura>
void Persistir<Elemento, Estructura>::obtener(Estructura& e)
{
    e.bloquear();
    unsigned int cuenta;
    Elemento elemento;
    if (!flujoEnt-->eof()) {
        (*flujoEnt) >> cuenta;
        s.purgar();
        for (unsigned int i = 0; (i < cuenta) && (!flujoEnt-->eof()));
            i++) {
                (*flujoEnt) >> elemento;
                e.anadir(elemento);
            }
    }
    e.desbloquear();
}
```

Para usar esta forma simple de consistencia de la persistencia a lo largo de la biblioteca, el cliente no tiene por tanto más que instanciar una clase adicional por estructura.

Construir marcos de referencia es difícil. Al crear bibliotecas generales de clases, hay que equilibrar las necesidades de funcionalidad, flexibilidad y simplicidad. Hay que intentar construir bibliotecas flexibles, porque nunca se puede saber exactamente cómo usarán los programadores las abstracciones. Además, es buena cosa construir bibliotecas que hagan el menor número posible de suposiciones sobre el entorno, para que los programadores puedan combinarlas fácilmente con otras bibliotecas de clases. El arquitecto también debe idear abstracciones simples, para que sean eficientes, y para que los programadores puedan comprenderlas. El marco de referencia más profundamente elegante nunca se reutilizará a menos que el coste de entenderlo y de usar después sus abstrac-

ciones sea menor que el coste que el programador percibe en escribirlas desde cero. El beneficio real llega cuando esas clases y mecanismos se reutilizan una y otra vez, indicando que otros están aprovechando el trabajo duro del desarrollador, permitiéndoles centrarse en las partes únicas de su propio problema particular.

Lecturas recomendadas

Biggerstaff y Perlis [H 1989] proporcionan un amplio tratado sobre la reutilización del software. Wirfs-Brock [C 1988] ofrece una buena introducción a los marcos de referencia orientados a objetos. Johnson [G 1992] examina aproximaciones a la documentación de la arquitectura de marcos de referencia mediante el reconocimiento de sus patrones.

MacApp [G 1989] ofrece un ejemplo de un marco de referencia orientado a objetos para el desarrollo de aplicaciones para el Macintosh, específico y bien construido. Puede encontrarse una introducción a una versión inicial de esta biblioteca de clases en Schmucker [G 1986]. En un trabajo más reciente, Goldstein y Alger [C 1992] discuten las actividades del desarrollo de software orientado a objetos para el Macintosh.

Abundan otros ejemplos de marcos de referencia, cubriendo una variedad de dominios de problema, incluyendo hipermedia (Meyrowitz [C 1986]), reconocimiento de patrones (Yoshida [C 1988]), gráficos interactivos (Young [C 1987]) y autoedición (Ferrrel [K 1989]). Los marcos de referencia generales incluyen ET++ (Weinand, [K 1989]) y arquitecturas MVC dirigidas por eventos (Shan [G 1989]). Coggins [C 1990] estudia los problemas que conciernen al desarrollo de bibliotecas C++ en particular.

Puede encontrarse un estudio empírico de arquitecturas orientadas a objetos y sus efectos en la reutilización en Lewis [C 1992].

Notas bibliográficas

- [1] *C++ Booch Components Class Catalog*. 1992. Santa Clara, CA: Rational.
- [2] Knuth, D. 1973. *The Art of Computer Programming*, vol. 1-3. Reading, MA: Addison-Wesley.
- [3] Aho, A., Hopcroft, J. and Ullman, J. 1974. *The Design and Analysis of Computer Programs*. Reading, MA: Addison-Wesley.
- [4] Kernighan, B. and Plauger, P. 1981. *Software Tools in Pascal*. Reading, MA: Addison-Wesley.
- [5] Sedgewick, R. 1983. *Algorithms*. Reading, MA: Addison-Wesley.
- [6] Stubbs, D. and Webre, N. 1985. *Data Structures with Abstract Data Types and Pascal*. Monterey, CA: Brooks/Cole.
- [7] Tenenbaum, A. and Augenstein, M. 1981. *Data Structures Using Pascal*. Englewood Cliffs, NJ: Prentice-Hall.
- [8] Wirth, N. 1986. *Algorithms and Data Structures*, Second Edition. Englewood Cliffs, NJ: Prentice-Hall.

- [9] Wirfs-Brock, R. October 1991. Object-Oriented Frameworks. *American Programmer* vol. 4(10), p. 27.
- [10] Stroustrup, Bjarne. 1991. *The C++ Programming Language, Second Edition*. Reading, Massachusetts: Addison-Wesley, p. 429.
- [11] Coggins, J. September 1990. *Design and Management of C+ Libraries*. Chapel Hill, North Carolina: University of North Carolina, p. 1.
- [12] Ellis, M. and Stroustrup, B. 1990. *The Annotated C++ Reference Manual*. Reading, Massachusetts: Addison-Wesley, p. 155.
- [13] Ellis and Stroustrup, p. 297.
- [14] Ellis and Stroustrup, p. 90.
- [15] Wirfs-Brock, 1993. Comunicación privada.
- [16] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. 1992. *Object-Oriented Software Engineering*. Workingham, England: Addison-Wesley Publishing Company, p. 184.

Computación cliente/servidor: Mantenimiento de catálogos

Una empresa utilizará para muchas aplicaciones de gestión un moderno sistema de gestión de bases de datos (SGBD), para proporcionar una solución genérica a los problemas de almacenamiento de datos persistentes, acceso concurrente a la base de datos, integridad y seguridad de datos, y copias de seguridad. Por supuesto, cualquier SGBD debe adaptarse a la empresa concreta, y las organizaciones han enfocado este problema tradicionalmente separándolo en otros dos diferentes: el diseño de los datos se encomienda a expertos en bases de datos, y el diseño del software para procesar las transacciones sobre la base de datos se encomienda a desarrolladores de aplicaciones. Esta técnica tiene ciertas ventajas, pero conlleva algunos problemas muy reales. Francamente, hay diferencias culturales entre los diseñadores de bases de datos y los programadores, que reflejan sus diferentes tecnologías y habilidades. Los diseñadores de bases de datos tienden a ver el mundo en forma de tablas de información persistentes y monolíticas, mientras que los desarrolladores de aplicaciones tienden a ver el mundo en términos de su flujo de control.

Es imposible conseguir integridad de diseño en un sistema complejo a menos que los intereses de esos dos grupos se reconcilien. En un sistema en el que dominan los problemas de los datos, hay que ser capaces de hacer equilibrios inteligentes entre una base de datos y sus aplicaciones. Un esquema de base de datos diseñado sin atención a su uso es tan ineficaz como chapucero. Análogamente, las aplicaciones desarrolladas de forma aislada plantean demandas irrationales sobre la base de datos y con frecuencia dan lugar a serios problemas de integridad de datos debido a la redundancia de los mismos.

En el pasado, la computación tradicional en mainframes planteó algunos obstáculos muy claros alrededor de las ventajas de las bases de datos para una

compañía. Sin embargo, con el advenimiento de la computación de bajo coste, que pone herramientas de productividad personal en manos de una multitud de trabajadores, junto con redes que sirven para enlazar los ubicuos computadores personales a través de oficinas y de naciones enteras, el rostro de los sistemas de gestión de información ha cambiado irreversiblemente. Está muy claro que una parte importante de este cambio fundamental es la aplicación de arquitecturas cliente/ servidor. Como apunta Mimno, «El rápido movimiento hacia la disminución de tamaño y la computación cliente—servidor viene dado por imperativos económicos. De cara a incrementar rápidamente la competitividad y acortar los ciclos del producto, los gestores de las empresas están buscando formas para obtener más rápidamente los productos a vender, incrementar los servicios a los clientes, responder más rápido a retos competitivos y reducir costes» [1]. En este capítulo, se aborda una aplicación de sistema de gestión de información (SGI) y se muestra cómo la tecnología orientada a objetos puede atacar los problemas del diseño de bases de datos y aplicaciones de forma unificada, en el contexto de una arquitectura cliente/servidor.

10.1. Análisis

Definición de los límites del problema

El recuadro de texto proporciona los requisitos para un sistema de mantenimiento de catálogos. Esta es una aplicación muy compleja cuyo uso toca virtualmente cualquier aspecto del flujo de trabajo en un almacén. El almacén físico existe para almacenar productos, pero es este software el que sirve como alma de ese almacén, porque sin él el almacén dejaría de funcionar como un centro de distribución eficaz.

Parte del reto que se plantea en el desarrollo de un sistema tan amplio es que requiere que los planificadores reconsideren todo el proceso de la empresa, pero equilibrando esto con la inversión de capital que ya han realizado en código legado, como se discutió en el Capítulo 7. Mientras las ganancias de productividad pueden conseguirse a veces de forma simple automatizando procesos manuales existentes, las ganancias radicales sólo pueden lograrse normalmente cuando se desafía alguna de las suposiciones básicas que se tienen sobre cómo debería funcionar el negocio. Cómo se reconduce este negocio es una actividad de planificación del sistema, y por tanto está fuera del ámbito de este texto. Sin embargo, al igual que la arquitectura del software limita el problema de la implantación, de la misma forma la visión de la empresa limita todo el problema del software. Por tanto se comienza por considerar un plan operacional para el funcionamiento del almacén. El análisis de sistemas sugiere que hay siete actividades funcionales principales en este negocio:

Requisitos del sistema de mantenimiento de catálogos

Como parte de su expansión a mercados nuevos y especializados, una compañía de venta por correo ha decidido establecer una serie de almacenes regionales relativamente autónomos. Cada uno de ellos adquiere responsabilidad local para la gestión de catálogos y el procesamiento de pedidos. Para alcanzar con eficiencia los mercados que se pretende, cada almacén se dedica a realizar un mantenimiento de catálogos que sea lo más adecuado posible al mercado local. La línea específica de productos que gestiona cada almacén puede cambiar de una región a otra; además, la línea de productos gestionada por una región dada tiende a actualizarse casi cada año para adaptarse a gustos cambiantes de los consumidores. Por razones de economías de escala, la compañía central desea tener un sistema de seguimiento de catálogos y pedidos común para todos los almacenes.

Las funciones clave de este sistema incluyen:

- Gestionar las mercancías según entran al almacén, expedidas de una variedad de proveedores.
- Gestionar los pedidos según se reciben de una organización de venta a distancia, central pero remota; los pedidos también pueden recibirse por correo, y se procesan localmente.
- Generar listas de embalaje, utilizadas para dirigir al personal del almacén en la confección y expedición de un pedido.
- Generar facturas y controlar las facturas recibidas.
- Generar peticiones de suministro y controlar las facturas a pagar.

Además de automatizar gran parte del trabajo diario del almacén, el sistema debe proporcionar una utilidad de generación de informes general y abierta, de forma que el equipo gestor pueda seguir las tendencias de las ventas, identificar los clientes y proveedores apreciados y problemáticos, y efectuar programas de promoción especiales.

- | | |
|----------------------|--|
| • Entrada de pedidos | Responsable de recibir pedidos del cliente y responder a preguntas de este sobre el estado de un pedido. |
| • Contabilidad | Responsable de enviar facturas y controlar los pagos de los clientes (facturas recibidas) así como de pagar a los suministradores por pedidos de compras (facturas a pagar). |
| • Expedición | Responsable de ensamblar paquetes para la expedición en apoyo del cumplimiento de pedidos de clientes. |
| • Almacenaje | Responsable de situar nuevas mercancías en almacén así como de recuperar mercancías en apoyo del cumplimiento de pedidos de clientes. |
| • Compras | Responsable de solicitar artículos a los proveedores y llevar cuenta de los envíos de los suministradores. |

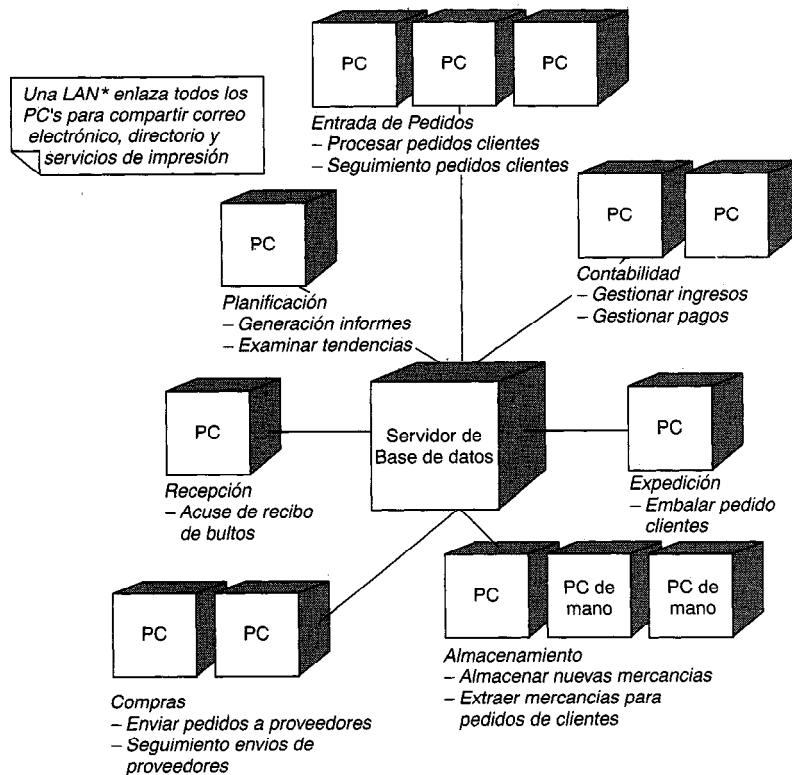


Figura 10.1. Red del sistema de mantenimiento de catálogos.

- **Recepción** Responsable de aceptar artículos de los suministradores.
- **Planificación** Responsable de generar informes para los gestores y estudiar las tendencias en los niveles de inventario y en la actividad de los clientes.

No sorprende el hecho de que hay un isomorfismo entre la arquitectura del sistema y estas unidades funcionales. La Figura 10.1 muestra un diagrama de procesos que ilustra todos los elementos computacionales principales de la red. Esta red es en realidad una estructura SGI bastante habitual: bancos de computadores personales que alimentan un servidor de base de datos central, que a su vez sirve como almacén central para todos los datos de interés de la empresa.

Hay que reseñar algunos detalles acerca de esta red. Primero, aunque se muestra una serie de PCs distintos cada uno de los cuales está ligado a una unidad funcional particular, ésta es una consideración meramente operacional. No debería haber nada en la arquitectura del software que restringiese un PC específico a una sola actividad: el equipo de contabilidad debería ser capaz de rea-

* El término LAN (Local Area Network) o red local se utiliza habitualmente sin traducir. (N. del T.)

lizar consultas generales, y el departamento de compras debería poder consultar sobre asientos contables involucrados en pagos a suministradores. De este modo, según el dictado de las condiciones cambiantes del negocio, la dirección puede añadir o reasignar recursos de computación según sea necesario para equilibrar el trabajo diario. Por supuesto, los requisitos de seguridad dictan la necesidad de alguna disciplina de gestión: a una persona del almacén no debería permitírsela emitir cheques. Se delega la responsabilidad de este tipo de restricciones como una consideración operacional, realizada por mecanismos generales de control de acceso de la red que o bien restringen o bien garantizan el derecho a ciertos datos y aplicaciones.

Como parte de la arquitectura de este sistema, también se asume la existencia de una red de área local (LAN, Local Area Network) que enlaza todos los recursos de computación, y sirve para proporcionar servicios habituales de red como correo electrónico, acceso a directorios compartidos, impresión y comunicaciones. Desde la perspectiva del software del sistema de mantenimiento de catálogos, la elección de una LAN concreta es bastante indiferente, mientras proporcione estos servicios con fiabilidad y eficiencia.

La presencia de los PCs como parte de la función de almacenaje añade un nuevo capricho a esta red. Lo económico de los PCs especializados y portátiles transportados en el cinturón, junto con comunicaciones sin cables, posibilitan el considerar un plan operacional que aproveche estas tecnologías para incrementar la productividad. Básicamente, el plan será dar a cada almacenista un PC de mano. A medida que se ubican nuevas mercancías en el almacén, utilizarán estos dispositivos para informar del hecho de que los artículos están en su lugar, y notificar también al sistema dónde se sitúan; a medida que se asignan pedidos diarios para cumplimentar, se transmiten órdenes de embalaje a estos dispositivos, diciendo a los trabajadores dónde se encuentra cierto artículo, así como cuántos hay que recoger para llevarlos a expedición.

Ahora, ninguna de estas tecnologías es cosa del otro mundo —todo lo que hay en la red es esencialmente hardware comercial. En realidad, se espera utilizar algo más que otro poco de software comercial. Tiene un evidente sentido cara al negocio el comprar —y no construir— hojas de cálculo comerciales, productos de trabajo en grupo (*groupware*) y paquetes de contabilidad. Sin embargo, lo que hace funcionar este sistema es su software de mantenimiento de catálogos, que sirve como el pegamento que lo une todo operacionalmente.

Las aplicaciones de este tipo realizan un trabajo computacional muy pequeño. En vez de eso, hay que almacenar, recuperar y mover grandes volúmenes de datos. La mayor parte del trabajo arquitectónico implicará por tanto decisiones sobre conocimiento declarativo (qué entidades existen, qué significan y dónde están) en vez de conocimiento procedimental (cómo suceden las cosas). El alma del diseño se encontrará en los intereses centrales del desarrollo orientado a objetos: las abstracciones clave que forman el vocabulario del dominio del problema y los mecanismos que lo manipulan.

Las demandas del negocio requieren que el sistema de mantenimiento de catálogos debe ser, por su propia naturaleza, abierto. Durante el análisis, se lle-

gará a la comprensión de las abstracciones clave que son importantes para la empresa en ese momento: se identificarán los tipos de datos que deben almacenarse, los informes a generar, las consultas a procesar, y todas las demás transacciones demandadas por los procedimientos administrativos de la compañía. La frase operativa aquí es *en ese momento*, porque los negocios no son entidades estáticas. Deben actuar y reaccionar a un mercado cambiante, y sus sistemas de gestión de la información deben seguir el ritmo de esos cambios. Un sistema de software obsoleto puede dar lugar a negocios perdidos o a un desperdicio de recursos humanos preciosos. Por tanto, hay que diseñar el sistema de mantenimiento de catálogos contando con que va a cambiar en el tiempo. La observación muestra que son dos los elementos que con mayor probabilidad cambiarán a lo largo del tiempo de vida de este sistema:

- Los tipos de los datos a almacenar.
- El hardware sobre el que se ejecuta esta aplicación.

Con el tiempo, cada almacén gestionará nuevas líneas de productos, se añadirán nuevos clientes y proveedores, y se eliminarán otros viejos. El uso operacional de este sistema puede revelar la necesidad imprevista de capturar información adicional sobre un cliente¹. Además, la tecnología del hardware sigue cambiando a un ritmo más rápido que la del software, y los computadores siguen quedándose obsoletos en cuestión de pocos años. Sin embargo, simplemente no es ni rentable ni inteligente reemplazar con frecuencia un sistema de software grande y complejo. No es rentable porque el tiempo y el coste de desarrollar el software puede muchas veces superar el tiempo y el coste de conseguir el hardware. No es inteligente porque el introducir un nuevo sistema cada vez que uno viejo empieza a parecer gastado añade riesgo al negocio; la estabilidad y la madurez son características valiosas del software que desempeña tan importante papel en las actividades cotidianas de una organización.

Un corolario a este segundo factor es la probabilidad de que el interfaz de usuario de la aplicación necesitará cambiar con el tiempo. En el pasado, para muchas aplicaciones SGI, demostraron ser adecuados simples interfaces orientados a líneas o a pantallas. Sin embargo, los costes decrecientes del hardware y los avances pasmosos en los interfaces gráficos de usuario han hecho práctica y deseable la incorporación de tecnología más sofisticada. Para poner las cosas en perspectiva, el interfaz de usuario del sistema de mantenimiento de catálogos

¹ Considerese, por ejemplo, el impacto de las tecnologías emergentes que ofrecerán servicios interactivos de video a cada hogar. No sería descabellado pensar que, en el futuro, los clientes podrán hacer pedidos electrónicamente a la compañía de venta por correo, y pagar directamente de sus cuentas. Puesto que los estándares para estos dominios cambian casi a diario según se posicionan las compañías de cara a convertirse en los suministradores dominantes de tales servicios, es imposible para el desarrollador de aplicaciones de usuario predecir con exactitud el protocolo para interactuar con tales sistemas. Lo mejor que se puede esperar como arquitectos del sistema es hacer suposiciones inteligentes y encapsular estas decisiones en el software de modo que se pueda adaptar cuando desaparezca la polvareda de la batalla por el dominio de las vías de información —una batalla en la que el desarrollador individual de aplicaciones no deja de ser un peón con influencia mínima—. Esto nos lleva en realidad a una motivación primaria para usar tecnología orientada a objetos: como se ha visto, el desarrollo orientado a objetos ayuda a construir arquitecturas elásticas y adaptables, características que son esenciales para la supervivencia en este mercado.

sólo es una parte pequeña (aunque crítica) de la aplicación. El núcleo de este sistema involucra a su base de datos; su interfaz de usuario es en gran medida una piel alrededor de este núcleo. De hecho, es posible (y muy deseable) permitir una variedad de interfaces de usuario para este sistema. Por ejemplo, un interfaz orientado a menús, simple e interactivo es probablemente el adecuado para clientes que presentan sus propios pedidos. Los interfaces modernos y basados en ventanas son seguramente mejores para las funciones de planificación, compras y contabilidad. Los informes impresos pueden generarse mejor en un entorno por lotes, aunque algunos gestores pueden desear utilizar un interfaz gráfico para ver interactivamente las tendencias. Los almacenistas necesitan un interfaz que sea simple: los sistemas de ventanas dirigidos por ratón no funcionarán bien en el entorno industrial de un almacén, y además los costes de entrenamiento son una cuestión a tener en cuenta. Para los propósitos de esta aplicación, no nos ocuparemos demasiado de la naturaleza del interfaz de usuario; puede emplearse casi cualquier tipo de interfaz sin alterar la arquitectura fundamental del sistema de mantenimiento de catálogos.

En la base de esta discusión, se decide realizar dos decisiones estratégicas del sistema. Primero, se decide usar una base de datos relacional comercial (SGBDR) alrededor del cual se construirá la aplicación. El diseñar una base de datos *ad hoc* no tiene ningún sentido en esta situación; la naturaleza de la aplicación llevaría a implantar la mayor parte de la funcionalidad de un SGBD a un coste muchísimo mayor y con mucha menos flexibilidad en el producto resultante. Un SGBDR también tiene la ventaja de ser razonablemente transportable. Los SGBDR más populares tienen implantaciones que funcionan en un espectro de plataformas hardware, desde los computadores personales hasta las máquinas grandes, transfiriendo así del desarrollador al vendedor la responsabilidad de transportar los SGBDR genéricos. Segundo, como se vio en la Figura 10.1, se decide que el mantenimiento de catálogos se ejecute en una red distribuida. Por simplicidad, se harán planes para una base de datos centralizada que reside en una máquina. Sin embargo, se permitirá que las aplicaciones se dirijan a una variedad de máquinas desde las cuales puedan acceder a esta base de datos. Este diseño representa un modelo cliente/servidor; la máquina dedicada a la base de datos actúa como el servidor, y puede tener muchos clientes. La máquina concreta en la que se ejecuta un cliente (incluso si es la propia máquina de base de datos local) es completamente indiferente para el servidor. Así, la aplicación puede operar sobre una red heterogénea y permite que se incorpore nueva tecnología hardware con un impacto mínimo sobre el funcionamiento del sistema.

Computación cliente/servidor

Aunque no es el propósito de este capítulo proporcionar un examen amplio de la computación cliente/servidor, es conveniente hacer algunas observaciones, porque influyen en las decisiones arquitectónicas.

Qué es y qué no es computación cliente/servidor es un tema debatido con

fervor². Para nuestros propósitos, es suficiente decir que la computación cliente/servidor comprende «una arquitectura descentralizada que permite a los usuarios finales obtener acceso a la información de forma transparente en un entorno multivendedor. Las aplicaciones cliente/servidor acoplan un IGU a un SGBDR basado en el servidor» [2]. La propia naturaleza de las aplicaciones cliente/servidor sugiere una forma de procesamiento cooperativo, en la que la responsabilidad de efectuar las funciones del sistema está distribuida entre varios elementos computacionales casi independientes que existen como parte de un sistema abierto. Berson afirma además que toda aplicación cliente/servidor puede dividirse típicamente en uno de cuatro componentes:

- Lógica de presentación La parte de una aplicación que interactúa con un dispositivo de un usuario final como un terminal, un lector de código de barras o un computador portátil. Sus funciones incluyen «formateo de la pantalla, lectura y escritura de la información de la pantalla, y gestión de ventanas, teclado y ratón».
- Lógica del negocio La parte de una aplicación que utiliza información del usuario y de la base de datos para efectuar transacciones de acuerdo con las restricciones de las reglas del negocio.
- Lógica de bases de datos La parte de una aplicación que «manipula los datos dentro de la aplicación... La manipulación de datos en SGBDs relacionales se realiza mediante algún dialecto del SQL (Structured Query Language, Lenguaje de Consultas Estructurado)».
- Procesamiento de bases de datos El «verdadero procesamiento de los datos de la base de datos realizada por el SGBD... Idealmente, el procesamiento del SGBD es transparente a la lógica de negocios de la aplicación» [3].

El problema fundamental para el arquitecto es cómo y dónde distribuir estos elementos computacionales a lo largo de una red abierta. Para complicar bastante el proceso de decisión, está el hecho de que los estándares y herramientas cliente/servidor evolucionan a un ritmo que causa confusión. El arquitecto debe encontrar su camino a través de una serie de propuestas como POSIX (Portable Operating System Interface), el modelo de referencia Open System Interconnection (OSI), el Object Management Group common object request broker (CORBA), y extensiones orientadas a objetos del SQL (SQL3), así como solu-

² Lo mismo que la cuestión de qué y qué no es orientado a objetos.

ciones específicas de cada fabricante, como el mecanismo Object Linking and Embedding (OLE) de Microsoft³.

No sólo los estándares tienen su impacto en las decisiones de los arquitectos, sino que deben sopesarse también problemas como la seguridad, rendimiento y capacidad. Berson sugiere algunas reglas prácticas para el arquitecto de un sistema cliente/servidor:

- En general, se sitúa en un sistema cliente un componente de la lógica de presentación con sus facilidades de entrada y salida por pantalla.
- Dada la potencia disponible de las estaciones de trabajo clientes, y el hecho de que la lógica de presentación reside en el sistema cliente, tiene sentido situar también alguna parte de la lógica del negocio en un sistema cliente.
- Si la lógica de procesamiento de bases de datos está embebida en la lógica del negocio, y si los clientes mantienen algunos datos de baja interacción y cuasi-estáticos, la lógica de procesamiento de bases de datos puede ubicarse en el sistema cliente.
- Dado el hecho de que una red de área local típica conecta clientes dentro de un grupo de trabajo con propósitos comunes, y suponiendo que el grupo comparta una base de datos, todos los fragmentos comunes y compartidos de la lógica del negocio y de las bases de datos y el propio SGBD deberían emplazarse en el servidor [4].

Si se adoptan las decisiones arquitectónicas correctas y se ha tenido éxito al llevar a su término los detalles tácticos de su implantación, el modelo cliente/servidor ofrece una serie de beneficios, como observa Berson:

- Permite a las corporaciones aprovechar mejor la tecnología emergente de computadores personales.
- Permite al procesamiento residir cerca de la fuente de datos que se está procesando... Por tanto, el tráfico de la red (y el tiempo de respuesta) pueden reducirse en gran medida.
- Facilita el uso de interfaces gráficos de usuario disponibles en estaciones de trabajo potentes.
- Permite y promueve la aceptación de los sistemas abiertos [5].

Por supuesto, hay riesgos:

- Si se traslada al servidor una porción significativa de la lógica de la aplicación, el servidor puede convertirse en un cuello de botella de la misma forma en que lo hace un mainframe en una arquitectura maestro-esclavo.
- Las aplicaciones distribuidas... son más complejas que las no distribuidas [6].

³ Por esta razón los buenos arquitectos de sistemas de información tienden a recibir grandes sumas de dinero por sus habilidades, o si no, al menos se divierten mucho intentando encasar tecnologías tan dispares para formar un todo coherente.

Se mitigan esos riesgos mediante el uso de una arquitectura y un proceso de desarrollo orientados a objetos.

Escenarios

Ahora que se ha establecido el ámbito del sistema, se continúa el análisis estudiando varios escenarios de su uso. Se comienza enumerando una serie de casos de uso básicos, tal como se ven desde los diversos elementos funcionales del sistema:

- Un cliente telefonea a la organización de venta a distancia para efectuar un pedido.
- Un cliente envía un pedido por correo.
- Un cliente llama para conocer el estado de un pedido.
- Un cliente llama para añadir elementos o eliminar elementos de un pedido existente.
- Un almacenista recibe una orden de embalaje para recoger mercancías para un pedido de un cliente.
- Expedición recibe un pedido ensamblado y lo prepara para el envío.
- Contabilidad prepara una factura para un cliente.
- Compras efectúa un pedido de nuevas mercancías.
- Compras añade o elimina un nuevo proveedor.
- Compras consulta el estado de un pedido a proveedores ya existente. Recpción acepta un cargamento de un proveedor, fruto de una orden de compra.
- Un almacenista introduce nuevos artículos en el catálogo.
- Contabilidad emite un cheque frente a un pedido de compras de nuevas mercancías.
- El departamento de planificación genera un informe de tendencias, que muestra las actividades de ventas para varios productos.
- Para propósitos de informes de impuestos, el departamento de planificación genera un resumen que muestra los niveles de mercancías actuales.

Para cada uno de esos escenarios principales, se pueden considerar otros secundarios:

- Un artículo pedido por un cliente está fuera de stock o en una orden atrasada.
- Un pedido de un cliente está incompleto, o menciona números de producto incorrectos u obsoletos.
- Un cliente llama para preguntar sobre o para cambiar una orden, pero no recuerda qué se pidió exactamente, por parte de quién, o cuándo.
- Un almacenista recibe una orden de embalaje para determinadas mercancías, pero no encuentra el artículo.
- Expedición recibe un pedido incompleto.

- Un cliente deja sin pagar una factura.
- Compras emite una orden para nuevas mercancías, pero el proveedor ha dejado el negocio o ya no maneja ese género.
- Recepción acepta un envío incompleto de un proveedor.
- Recepción acepta un envío de un proveedor para el que no encuentra ningún pedido de compras.
- Un almacenista introduce nuevos artículos en el catálogo, para descubrir que no hay espacio para ese elemento.
- Cambia el código de identificación fiscal (en España, CIF) del negocio, lo que requiere que el departamento de planificación genere una serie de nuevos informes de inventario.

Para un sistema de esta complejidad, se esperaría identificar docenas de escenarios principales y muchos más escenarios secundarios. De hecho, esta parte del proceso de análisis llevaría probablemente varias semanas para que se completase a un nivel de detalle razonable⁴. Por esta razón, se sugiere firmemente que se aplique la regla del 80 %: no esperar a generar una lista completa de escenarios (ningún espacio de tiempo será suficiente), sino en vez de eso estudiar algo así como el 80 % de los interesantes, y si se puede, intentar una prueba «rápida y sucia» de concepto para ver si esta parte del análisis va por buen camino. Para los propósitos de este capítulo, se trabajará sobre dos de los escenarios principales del sistema.

La Figura 10.2 proporciona un caso de uso primario para un cliente que plantea un pedido a la organización de venta remota. Aquí se ve que hay una serie de objetos que colaboran para llevar a cabo esta función del sistema. Aunque el control se centra en la interacción cliente/agente, otros tres objetos clave (a saber, `unRegistroCliente`, la `baseDatosInventario` y `unaOrdenEmbalaje`, todos los cuales son artefactos del sistema de seguimiento de catálogos) juegan un papel fundamental. Se añaden esas abstracciones a la «lista de cosas» que se salen del proceso de planificación del escenario.

La Figura 10.3 continúa este escenario con una elaboración sobre la interacción de orden de embalaje/almacenista, otro comportamiento crítico del sistema. Aquí se ve que el almacenista está en el centro de la actividad de este escenario, y colabora con otros objetos, a saber, `expedicion`, que no representaba ningún papel en el escenario anterior. De hecho, la mayor parte de los objetos que colaboran en la Figura 10.3 son los mismos que aparecían en la Figura 10.2, aunque es importante darse cuenta de que estos objetos comunes juegan papeles muy diferentes. Por ejemplo, en el escenario de las órdenes, se usa `unaOrden` para llevar cuenta de las peticiones de un cliente, pero en el escenario del embalaje se usa `unaOrden` como una comprobación y balance de las órdenes de embalaje.

A medida que se pasa por cada uno de estos escenarios, hay que plantearse

⁴ Pero hay que guardarse de la parálisis del análisis: si el ciclo de análisis del software se extiende más que el margen de oportunidad para el negocio, abandonad toda esperanza, quienes sigáis este camino, porque os encontrareis al final fuera del negocio.

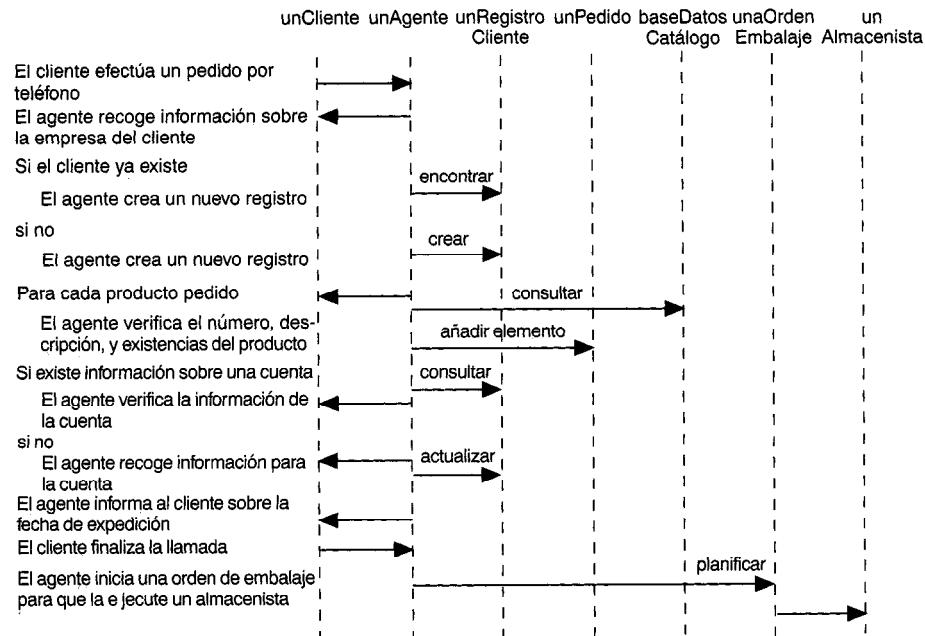


Figura 10.2. Escenario de un pedido.

continuamente una serie de preguntas. ¿Qué objeto debería ser responsable de cierta acción? ¿Tiene un objeto suficiente conocimiento para efectuar una operación dirigida a él, o debe delegar el comportamiento? ¿Está el objeto intentando hacer demasiado? ¿Qué podría ir mal? Es decir, ¿qué pasa si se violan ciertas precondiciones, o si no pueden satisfacerse las postcondiciones?

Dando un carácter antropomórfico a las abstracciones en cada uno de los

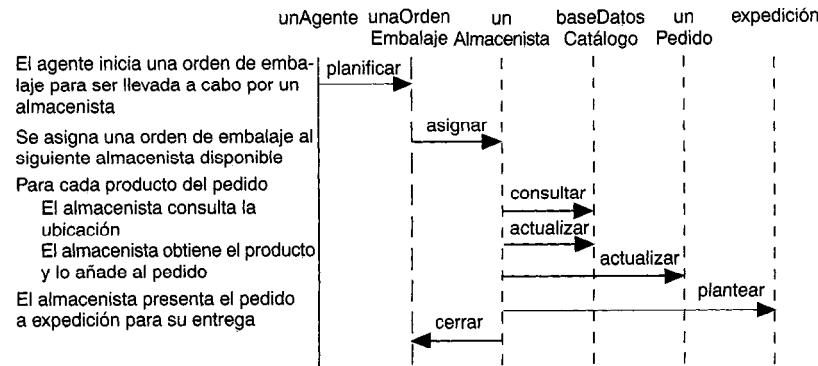


Figura 10.3. Escenario de orden de embalaje.

puntos funcionales del sistema se llegaría finalmente a descubrir gran parte de los objetos de alto nivel interesantes dentro del sistema. En concreto, el análisis nos lleva a descubrir las siguientes abstracciones. Primero, se listan las diversas personas que interactúan con el sistema:

- Cliente
- Proveedor
- AgenteDeOrdenes
- Contable
- AgenteDeEnvios
- Almacenista
- AgenteDeCompras
- AgenteDeRecepción
- Planificador

Es importante identificar estos tipos de personas, porque representan los diferentes papeles que las personas desempeñan cuando interactúan con el sistema. Si se desea llevar cuenta del quién, el cuándo y el por qué de ciertos eventos que tuvieron lugar dentro del sistema, entonces hay que formalizar estos papeles. Por ejemplo, cuando se resuelve una reclamación, se podría desear identificar qué persona de la compañía interactuó recientemente con el cliente insatisfecho, y sólo haciendo esto parte del modelo de la empresa se retendrá suficiente información para realizar un análisis inteligente. Además de ofrecer un papel visible exteriormente, es importante para nosotros distinguir entre estas clases de personas para el propósito de restringir operacionalmente o garantizar el acceso a partes de la funcionalidad del sistema. Con una red abierta, esta forma de control centralizado es un modo razonablemente efectivo de controlar usos erróneos accidentales o maliciosos.

El análisis también revela las siguientes abstracciones clave, cada una de las cuales representa alguna información manipulada por el sistema:

- RegistroCliente
- RegistroProducto
- RegistroProveedor
- Pedido
- OrdenCompra
- Factura
- OrdenEmbalaje
- OrdenAlmacenaje
- EtiquetaEnvio

Las clases RegistroCliente, RegistroProducto y RegistroProveedor paralelizan las abstracciones Cliente, Producto y Proveedor, respectivamente. Se mantienen ambos conjuntos de abstracciones porque, como se verá, cada una juega un papel sutilmente distinto en el sistema.

Nótese que puede haber dos tipos de facturas: las enviadas por la compañía a los clientes buscando el pago de un pedido, y las recibidas por la compañía

por mercancías pedidas a los proveedores. Ambas son materialmente el mismo tipo de cosa, aunque cada una juega un papel muy distinto en el sistema.

Nuestra abstracción de las clases `OrdenEmbalaje` y `OrdenAlmacenaje` requieren una explicación un poco más extensa. Como se describió en la discusión sobre los dos primeros escenarios, la siguiente acción que comprende un `AgenteDeOrdenes` tras aceptar un `Pedido` de un `Cliente` es ordenar a un `Almacenista` que lleve a cabo el `Pedido`. La decisión de sistema es capturar formalmente esta transacción como una instancia de la clase `OrdenEmbalaje`. La responsabilidad de esta clase es recoger toda la información necesaria para dirigir a un almacenista para cumplir un pedido del cliente. Operacionalmente, esto significa que el sistema planifica y transmite esta orden al computador de mano del siguiente almacenista disponible. Tal información incluiría, como mínimo, la identificación de algún número de pedido y los elementos a recuperar del almacén. No es difícil pensar cómo se podría mejorar enormemente este simple escenario: la empresa contiene suficiente información para transmitir la ubicación de cada uno de esos elementos al almacenista, y quizás incluso ofrecer sugerencias como el orden en que el almacenista debería atravesar el almacén para recoger esos elementos con la mayor eficiencia⁵. También hay disponible en el sistema suficiente información incluso para ofrecer ayuda a un almacenista recién contratado, quizás proyectando una imagen del elemento que hay que recoger en la pantalla del computador de mano. Esta utilidad de ayuda general podría ser también útil al almacenista experimentado, de cara a un cambio en la línea de productos.

La Figura 10.4 muestra un diagrama de clases que recoge nuestra comprensión de las asociaciones entre algunas de esas abstracciones, en relación con la función del sistema de tomar y cumplimentar pedidos y órdenes. Se ha adornado este diagrama con algunos de los atributos relevantes de cada clase.

Los intereses particulares de esta estructura de clases están dirigidos en gran parte por el requerimiento de navegar entre instancias de estas clases. Dado un pedido, se desearía generar una etiqueta de expedición para el cliente asociado; para hacerlo, se va marcha atrás de la orden al cliente. Dada una orden de embalaje, sería deseable remontarse al cliente y al agente de pedidos para informar del hecho de que algunos elementos están en órdenes atrasadas; esto requiere un recorrido desde la orden de embalaje hasta quien la emitió y de ahí hasta el cliente y el agente de órdenes (pedidos). Dado un cliente, se desearía determinar qué productos solicita ese cliente con más frecuencia durante ciertas épocas del año. Esta consulta requiere un recorrido desde el cliente hasta todos los pedidos pendientes y previos.

Es importante explicar algunos otros detalles de este diagrama. ¿Por qué se

⁵ Por supuesto, en el caso más general, esto es algo parecido al problema del viajante, que es *np-completo*. Sin embargo, es posible restringir el problema lo suficiente como para que pueda calcularse una solución razonable. Por ejemplo, las reglas de la empresa podrían dictar un orden parcial: se empaquetan primero todos los artículos pesados, y después los más ligeros. Además se podrían recoger juntos los artículos relacionados: los pantalones van con las camisetas, los martillos van con los clavos, las llantas con las cubiertas (¡ya se dijo que éste era un sistema de mantenimiento de catálogos de carácter general!).

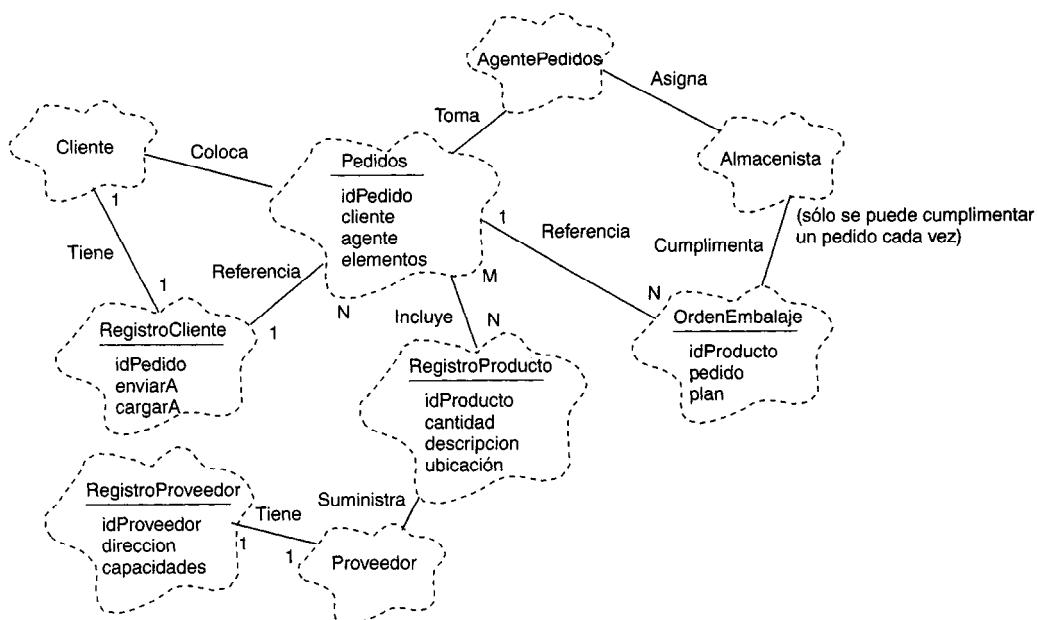


Figura 10.4. Clases clave para tomar y cumplimentar pedidos.

tiene una relación 1:N entre las clases Pedido y OrdenEmbalaje? Las reglas de la empresa establecen que cada orden de embalaje es única para un pedido dado (la parte 1 de la expresión de cardinalidad). Sin embargo, supongamos que el almacén está sin mercancías para ciertos artículos referidos en el pedido original: hay que preparar una segunda orden de embalaje en el momento en que vuelven a estar disponibles esos artículos.

Nótese también la restricción sobre la asociación de un Almacenista y una OrdenEmbalaje: por razones de control de calidad, las reglas de la empresa dicen que un almacenista debe atender sólo un pedido de cada vez.

Para completar esta fase del análisis, se introducen dos clases clave finales:

- Informe
- Transaccion

Se incluye la abstracción **Informe** para denotar la clase base de todos los diversos tipos de copia impresa y consultas en línea que pueden generar los usuarios. El análisis detallado mediante escenario descubrirá probablemente muchos de los tipos concretos de informes que demanda el flujo de trabajo, pero a causa de la naturaleza abierta del sistema, es más aconsejable desarrollar un mecanismo de generación de informes más general, de forma que puedan añadirse informes nuevos de manera consistente. En realidad, identificando los aspectos comunes entre informes, se hace posible para todos ellos compartir un comportamiento y estructura comunes, simplificando así la arquitectura y permitiendo al sistema presentar a sus usuarios un aspecto y sensación homogéneos.

La lista de cosas del sistema no es completa en absoluto, pero se tiene suficiente información en este punto para comenzar a entrar en un diseño arquitectónico. Antes de proceder, sin embargo, hay que considerar algunos principios sobre la estructura de los datos en el sistema que influirán en las decisiones de diseño.

Modelos de bases de datos

Como lo describe Date, una base de datos «es un depósito para datos almacenados. En general, está integrada y compartida. Por “integrada” quiere decirse que la base de datos puede entenderse como una unificación de varios ficheros de datos de otro modo distintos, con cualquier redundancia entre esos ficheros eliminada total o parcialmente... Por “compartida” quiere decirse que los elementos de datos individuales de la base de datos pueden ser compartidos entre varios usuarios diferentes» [7]. Con un control centralizado sobre una base de datos, «puede reducirse la inconsistencia, pueden reforzarse los estándares, pueden aplicarse restricciones de seguridad, y puede mantenerse la integridad de la base de datos» [8].

El diseñar una base de datos efectiva es una tarea difícil a causa de la existencia de tantos requisitos que compiten entre sí. El diseñador de la base de datos no sólo debe satisfacer los requisitos funcionales de la aplicación, sino también afrontar factores de tiempo y espacio. Una base de datos ineficiente respecto al tiempo que recupere los datos mucho después de que sean necesarios es prácticamente inútil. Análogamente, una base de datos que requiera un edificio lleno de computadores y todo un enjambre de personas para soportarla no es muy efectiva respecto al coste.

El diseño de bases de datos tiene muchos paralelismos con el desarrollo orientado a objetos. En la tecnología de bases de datos, el diseño suele verse como un proceso incremental e iterativo que implica decisiones lógicas y físicas [9]. Como apuntan Wiorkowski y Kull, «los objetos que describen una base de datos en la forma en que piensan sobre ella los usuarios y los desarrolladores se llaman objetos lógicos. Aquellos que se refieren al modo en que los datos se almacenan realmente en el sistema se llaman objetos físicos» [10]. En un proceso no muy distinto del diseño orientado a objetos, los diseñadores de bases de datos rebasan entre el diseño lógico y físico a lo largo del desarrollo de la base de datos. Además, las formas en que describimos los elementos de una base de datos son muy parecidas a las formas en las que describimos las abstracciones clave en una aplicación que utilice diseño orientado a objetos. Los diseñadores de bases de datos usan a menudo notaciones como los diagramas entidad-relación para ayudarse a analizar su problema. Como se ha visto, pueden escribirse diagramas de clases que se corresponden directamente con diagramas entidad-relación, pero tienen aún más potencia expresiva.

Como sugiere Date, cada tipo de base de datos generalizada debe afrontar la siguiente cuestión: «¿Qué estructuras de datos y operadores asociados debería

soportar el sistema?» [11]. Las diferentes respuestas a esta cuestión nos aportan tres modelos de bases de datos claramente distintos:

- Jerárquico.
- En red.
- Relacional.

Recientemente, ha surgido un cuarto tipo de base de datos, a saber, las *bases de datos orientadas a objetos (SGBDOO)*. Un SGBDOO representa una mezcla de la tecnología de bases de datos tradicionales y del modelo de objetos. Los SGBDOO han demostrado ser particularmente útiles en dominios como las aplicaciones de ingeniería asistida por computador (computer-aided engineering, CAE) y la ingeniería del software asistida por computador (CASE, computer-aided software engineering), para las cuales hay que manipular cantidades significativas de datos con un rico contenido semántico. Para ciertas aplicaciones, las bases de datos orientadas a objetos pueden ofrecer mejoras de rendimiento significativas sobre las bases de datos relacionales tradicionales. En concreto, en circunstancias en las que haya que realizar uniones múltiples sobre muchas tablas distintas, las bases de datos orientadas a objetos pueden ser mucho más rápidas que las bases de datos relacionales similares. Además, las bases de datos orientadas a objetos proporcionan un modelo coherente y casi sin grietas para integrar los datos con reglas de la empresa. Para conseguir semánticas parecidas, los SGBDR suelen requerir complejas funciones de disparo, generadas mediante una combinación de lenguajes de tercera y cuarta generación —de todas formas, no resulta un modelo muy limpio.

Sin embargo, por una serie de razones, muchas compañías pueden encontrar que el encajar un SGBDR en el contexto de una arquitectura orientada a objetos reduce el riesgo de desarrollo. La tecnología de bases de datos relacionales está mucho más madura y disponible en una gran variedad de plataformas, y con frecuencia es más completa, ofreciendo soluciones a cuestiones de seguridad, versiones e integridad referencial. Además, una compañía puede tener invertido un capital significativo en personas y herramientas para el soporte del modelo relacional, y así para los sistemas de la siguiente generación simplemente no puede permitirse el transformar toda su organización de la noche a la mañana.

El modelo relacional de bases de datos ha tenido realmente mucho éxito popular. Puesto que su uso está tan extendido, puesto que hay una extensa infraestructura de productos y estándares que lo soportan, puesto que satisface los requisitos funcionales del sistema de mantenimiento de catálogos, se decide emplear una base de datos relacional en la arquitectura; esta es una decisión estratégica del sistema. Así, se ha seleccionado una arquitectura híbrida: se construirá una piel orientada a objetos sobre una base de datos relacional tradicional, derivándose así los beneficios de los dos paradigmas. Brevemente, consideremos algunos principios generales para el diseño de bases de datos relacionales, que servirán de guía en la realización de esta piel orientada a objetos.

Los elementos básicos de una base de datos relacional «son tablas en las que

las columnas representan cosas y los atributos que las describen y las filas representan instancias específicas de las cosas descritas... El modelo también ofrece operadores para generar nuevas tablas a partir de otras anteriores, que constituyen el medio por el que los usuarios manipulan la base de datos y recuperan información de ella» [12].

Considérese por un momento una base de datos de productos para una versión del sistema de mantenimiento de catálogos adaptado para gestionar un almacén de elementos electrónicos como resistencias, condensadores y circuitos integrados. De acuerdo con el diagrama de clases previo, se tienen productos identificados de forma única por un identificador de producto, junto con un nombre de elemento descriptivo. Un ejemplo:

Productos

<u>idProducto</u>	<u>descripcion</u>
0081735	Resistencia, 100 a 1/4 vatio
0081736	Resistencia, 140 a 1/4 vatio
3891043	Condensador, 100 pF
9074000	CI 7400 quad NAND
9074001	CI 74LS00 quad NAND

Esta es una tabla con dos columnas, cada una de las cuales representa un atributo diferente. En una relación como ésta, el orden de las filas y columnas es indiferente; puede haber cualquier número de filas, pero no filas duplicadas. La cabecera idProducto representa una clave primaria, lo que significa que se puede usar su valor para identificar de forma inequívoca un elemento concreto.

Los productos provienen de proveedores, así que para cada proveedor hay que mantener un identificador único, un nombre de compañía, una dirección, y posiblemente un número de teléfono. Así, se puede escribir lo siguiente:

Proveedores

<u>idProveedor</u>	<u>compañia</u>	<u>direccion</u>	<u>telefono</u>
00056	Interstate Supply	2222 Fannin, Amarillo, TX	806-555-0036
03107	Interstate Supply	3320 Scott, Santa Clara, CA	408-555-3600
78829	Universal Products	2171 Parfet Ct, Lakewood, CO	303-555-2405

idProveedor es una clave primaria, lo que significa que su valor puede utilizarse para identificar de forma inequívoca a un proveedor. Nótese que cada fila de esta tabla es única, aunque dos filas pueden tener el mismo nombre de proveedor.

Diferentes proveedores suministran diferentes productos a precios distintos, y por tanto también se podría mantener una tabla de precios. Dada una combinación producto/proveedor, esta tabla incluye el precio actual:

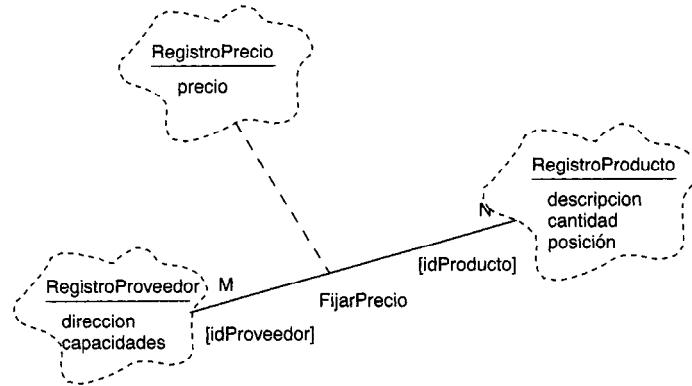


Figura 10.5. Asociación atribuida.

Precios

<u>idProducto</u>	<u>idProveedor</u>	<u>Precio</u>
0081735	03107	15,00 ptas.
0081735	78829	13,50 ptas.
0156999	78829	55.162,00 ptas.
7775098	03107	1.635,00 ptas.
6889655	00056	13,50 ptas.
9074001	03107	262,50 ptas.

Esta tabla no tiene una clave primaria simple. En vez de eso, hay que usar una combinación de las claves idProducto y idProveedor para identificar de forma única una fila de esta tabla. Una clave formada por la combinación de valores de las columnas se llama una *clave compuesta*. Nótese que no se incluyen nombres de elementos y proveedores porque serían redundantes; esta información puede encontrarse siguiendo el rastro desde idProducto o idProveedor hasta el elemento de la tabla de proveedores. idProducto e idProveedor se denominan por tanto *claves ajenas*, porque sus valores representan las claves primarias de otras tablas.

La Figura 10.5 ilustra la estructura de clases que corresponde a esta tabla. Aquí se usa una asociación atribuida para denotar registros que tienen significado sólo en asociación con instancias de otros dos registros. Nótese que también se usa la marca de clave de la notación para indicar la clave primaria de cada clase.

Continuando, se puede mantener un catálogo mediante una tabla que contenga la cantidad de todos los productos de los que se dispone actualmente:

Catalogo

idProducto	cantidad
0081735	1000
0097890	2000
0156999	34
7775098	46
6889655	1
9074001	192

Esta tabla ilustra el hecho de que nuestra visión orientada a objetos de los datos del sistema puede diferir de su visión como base de datos. Mientras que el esquema de la Figura 10.4 consideraba a **cantidad** un atributo de la clase **RegistroProducto**, aquí se ha decidido mantener **cantidad** en una tabla separada, por razones de eficiencia. En concreto, la **descripcion** de un producto tiende a cambiar muy raramente, pero se espera que la **cantidad** de un producto cambie constantemente, a medida que se cumplimentan pedidos y se colocan nuevas mercancías en el almacén. Operacionalmente, son conceptos muy diferentes, y se puede optimizar el acceso y actualización de **cantidad** generando una tabla separada.

En presencia de un esquema orientado a objetos como el que se muestra en la Figura 10.4, este secreto de implantación está oculto para todos los clientes de la aplicación. Pasa a ser por tanto responsabilidad de la clase **RegistroProducto** proporcionar la ilusión de que **cantidad** es una parte integrante de la abstracción.

El objetivo más simple, pero más importante del diseño de bases de datos es el concepto de que cada hecho debería almacenarse en un sitio y sólo uno. Esto elimina la redundancia, simplifica el proceso de actualizar la base de datos, facilita el mantenimiento de la integridad de la base de datos (es decir, la auto-consistencia y la corrección), y reduce las necesidades de espacio de almacenamiento. Conseguir este objetivo no es precisamente fácil (y a su vez, no siempre es importante). Sin embargo, es la característica más deseable que se pretende en nuestro diseño.

La teoría de la normalización ha evolucionado como una técnica para conseguir este objetivo (aunque no es el único principio relevante [13]). La normalización es una propiedad de una tabla; se dice que una tabla concreta está en *forma normal* si satisface ciertas propiedades. Existen varios niveles de formas normales, cada una de las cuales se construye sobre la otra [14]:

- Primera forma normal (1FN)* Cada atributo representa un valor atómico (atributos no descomponibles).

* En inglés, 1NF (First Normal Form). (*N. del T.*)

- Segunda forma normal (2FN) La tabla está en 1FN, y cada atributo depende enteramente de la clave (atributos independientes funcionalmente).
- Tercera forma normal (3FN) La tabla está en 2FN, y ningún atributo representa un hecho respecto de otro atributo (atributos mutuamente independientes).

Las tablas en 3FN «constan de “propiedades de la clave, toda la clave, y nada más que la clave”» [15].

Las tablas que se han mostrado como ejemplo están todas en 3FN. Existen formas superiores de normalización, que se refieren sobre todo a hechos multi-valuados, pero no son de gran interés aquí.

A la hora de llenar el vacío semántico desde el esquema orientado a objetos hasta una vista relacional, a veces se pueden desnormalizar tablas de forma intencionada, lo que significa que se las diseña explícitamente con alguna redundancia. Esto requiere un mayor esfuerzo para mantener los datos redundantes en consonancia, pero es peor el coste computacional si la eficiencia del acceso es una cuestión primordial.

SQL

Especialmente teniendo en cuenta nuestra visión orientada a objetos del mundo, en la que se unen los datos y los aspectos de comportamiento de las abstracciones, un usuario podría desechar realizar una serie de transacciones habituales sobre esas tablas. Por ejemplo, se podría desechar añadir nuevos proveedores, eliminar productos, o actualizar cantidades en el catálogo. También se podría desechar consultar a esas tablas de diversas formas. Por ejemplo, se podría querer un informe que listase todos los productos que pueden pedirse a un suministrador concreto. Se podría querer también un informe que relacionase los productos cuyo nivel de existencias es demasiado bajo o demasiado alto, de acuerdo con algún criterio dado. Por último, se podría desechar un informe extenso que reflejase el coste de recuperar los niveles de existencias hasta determinados niveles, usando las fuentes más baratas de productos. Estos tipos de transacciones son comunes a prácticamente todas las aplicaciones de un SGBDR, y por eso ha surgido un lenguaje estándar llamado SQL (Structured Query Language, lenguaje estructurado de consultas) para interactuar con bases de datos relacionales. El SQL puede usarse bien interactivamente o inmerso en programas.

La construcción más importante en SQL es la cláusula select o de selección, que toma la forma siguiente:

```
SELECT <atributo>
FROM   <relación>
WHERE  <condición>
```

Por ejemplo, para recuperar los números de los productos para los cuales el inventario es menor de 100 unidades, se podría escribir:

```
SELECT IDPRODUCTO, CANTIDAD
FROM   CATALOGO
WHERE  CANTIDAD < 100
```

Se pueden realizar selecciones mucho más complicadas. Por ejemplo, se podría querer el mismo informe para incluir el nombre del elemento en vez del número de elemento:

```
SELECT NOMBRE, CANTIDAD
FROM   CATALOGO, PRODUCTOS
WHERE  CANTIDAD < 100
AND    CATALOGO.IDPRODUCTO = PRODUCTOS.IDPRODUCTO
```

Esta cláusula representa una *unión* o *join*, por la cual se combinan dos o más relaciones en una sola relación. La cláusula select de arriba no genera una nueva tabla, sino que devuelve un conjunto de filas. Puesto que una sola selección podría devolver un número de columnas arbitrariamente largo, hay que disponer de los medios para visitar una columna de cada vez. El mecanismo que usa SQL es el cursor, cuya semántica es similar a la de la operación de iteración de la que se habló en el Capítulo 3. Por ejemplo, se podría declarar un cursor como sigue:

```
DECLARE C CURSOR
FOR SELECT NOMBRE, CANTIDAD
      FROM CATALOGO, PRODUCTOS
     WHERE CANTIDAD < 100
       AND CATALOGO.IDPRODUCTO = PRODUCTOS.IDPRODUCTO
```

Para provocar una evaluación de esta unión, se escribe

```
OPEN C
```

A continuación, para visitar cada fila de la unión, se escribe

```
FETCH C INTO NOM, CANT
```

Por último, cuando hemos terminado, se cierra el cursor ejecutando

```
CLOSE C
```

En vez de usar un cursor, se puede generar una tabla virtual que mantiene el resultado de la selección. Tal tabla virtual se llama una *vista*, y se puede operar sobre ella igual que si fuese una verdadera tabla. Por ejemplo, para crear una

vista que contenga el nombre de elemento, el nombre de proveedor y el coste, se podría escribir:

```
CREATE VIEW V (NOMBRE, COMPAÑIA, COSTE)
AS SELECT PRODUCTOS.NOMBRE, PROVEEDORES.COMPAÑIA,
          PRECIOS.PRECIO
     FROM  PRODUCTOS, PROVEEDORES, PRECIOS
    WHERE PRODUCTOS.IDPRODUCTO = PRECIOS.IDPRODUCTO
      AND PROVEEDORES.IDPROVEEDOR = PRECIOS.IDPROVEEDOR
```

Las vistas son particularmente importantes, porque hacen posible para diferentes usuarios el tener visiones diferentes de la base de datos. Las vistas pueden ser bastante diferentes de las relaciones subyacentes de la base de datos, y permitir así cierto grado de independencia de los datos. También pueden garantizarse vista a vista derechos de acceso para los usuarios, permitiendo así la redacción de transacciones seguras. Las vistas son un poco diferentes de las tablas base, sin embargo, en éstas las vistas que representan uniones no pueden actualizarse de forma directa.

Para nuestros propósitos, el SQL representa un bajo nivel de abstracción. No se espera que los usuarios finales sepan SQL; el SQL no es directamente una parte del vocabulario del dominio del problema. En vez de eso, se usará SQL dentro de la implantación de la aplicación, exponiéndolo sólo a los constructores de herramientas sofisticadas, pero ocultándoselo a los meros mortales que deben interactuar con el sistema a diario.

Considérese el problema siguiente: dado un pedido, se desearía determinar el nombre de compañía del cliente que lo emitió. Desde la perspectiva de su implantación, el llevar a cabo este proceso requiere una modesta cantidad de SQL; desde la perspectiva externa, el cliente de la aplicación preferiría permanecer en el contexto del C++, y escribir expresiones como la siguiente:

```
pedidoActual.cliente().nombre()
```

Desde nuestra perspectiva del mundo orientada a objetos, esta expresión invoca al selector `cliente` para referenciar al cliente que hizo el pedido; se invoca entonces al selector `nombre` para que devuelva el nombre de ese cliente. No hay aquí grandes sorpresas para el cliente externo, excepto que lo que está produciéndose realmente es una consulta a una base de datos, como ésta:

```
SELECT NOMBRE
FROM   PEDIDOS, CLIENTES
WHERE  PEDIDOS.IDCLIENTE = PEDIDOACTUAL.IDCLIENTE
AND    PEDIDOS.IDCLIENTE = CLIENTES.IDCLIENTE
```

El ocultar este secreto al cliente de la aplicación nos permite ocultar todos los detalles sucios del trabajo con SQL.

El hacer corresponder una visión del mundo orientada a objetos con una

relacional es conceptualmente directo, aunque en la práctica conlleva un montón de detalles tediosos⁶. Como observa Rumbaugh, «la correspondencia entre un modelo de objetos y una base de datos relacional es simple excepto para el manejo de la generalización» [16]. Rumbaugh continúa ofreciendo algunas reglas para la correspondencia de clases y asociaciones (incluyendo relaciones de agregación) a tablas:

- Cada clase se corresponde con una o más tablas.
- Cada asociación muchos a muchos se corresponde con una tabla distinta.
- Cada asociación uno a muchos se corresponde con una tabla distinta o puede insertarse como una clave ajena [17].

Además sugiere una de tres alternativas para corresponder las jerarquías clase/subclase a tablas:

- La superclase y cada subclase se corresponden con una tabla.
- Los atributos de la superclase se repiten en cada tabla (y cada subclase se corresponde con una tabla distinta).
- Elevar todos los atributos de las subclases hacia el nivel de la superclase (y tener una tabla para toda la jerarquía superclase/subclases) [18].

Como era de esperar, existen limitaciones al uso de SQL en la implantación subyacente⁷. En particular, SQL sólo define un conjunto muy limitado de tipos de datos, a saber, caracteres, cadenas de longitud fija, enteros, y números en punto fijo y punto flotante. Las implantaciones extienden ocasionalmente este conjunto de tipos; sin embargo, la representación de datos como imágenes o grandes fragmentos de texto no está soportada de forma directa.

Análisis del esquema

Como se pregunta Date, «dada una masa de datos para ser representados en una base de datos, ¿cómo se decide acerca de una estructura lógica adecuada para ellos? En otras palabras, ¿cómo se decide qué relaciones se necesitan y qué atributos deberían tener? Éste es el problema del diseño de bases de datos» [19]. Como puede verse, la identificación de las abstracciones clave de una base de datos es muy parecida al proceso de identificar clases y objetos en el desarrollo orientado a objetos. Por esta razón, en aplicaciones de manejo intensivo de datos como el sistema de mantenimiento de catálogos, se comienza con un análisis

⁶ Gran parte del valor de una base de datos orientada a objetos se deriva del hecho de que oculta estos detalles sucios del SQL al desarrollador. La correspondencia de clases a tablas es lo suficientemente codificable como para que sea posible un enfoque alternativo al uso de un SGBDOO: existen herramientas que toman declaraciones de clases en C++ y generan automáticamente el esquema SGBDR y el código SQL necesario para llenar este hueco semántico. A continuación, por ejemplo, cuando una ampliación intenta acceder a un atributo de un objeto dado, este código generado envía las instrucciones SQL necesarias al SGBDR comercial, extrae los datos importantes y se los devuelve al cliente en una forma consistente con el interfaz C++.

⁷ Recientemente se ha propuesto el estándar SQL3, que ofrece extensiones orientadas a objetos. Estas extensiones reducen en gran medida el hueco semántico entre una visión del mundo orientada a objetos y la visión relacional; estas extensiones también ayudan a mitigar muchas otras limitaciones del SQL.

sis orientado a objetos y se usa su proceso para dirigir el diseño de la base de datos, en vez de la opción inversa de centrarse primero en el esquema de la base de datos y derivar de él una arquitectura orientada a objetos.

La «lista de cosas» que se ha conformado hasta aquí en el análisis es un comienzo. Tomando esta lista y aplicando las reglas de Rumbaugh, se proporcionan las siguientes tablas para la base de datos de la aplicación. Primero se tienen tablas que reproducen los papeles de varios grupos que interactúan con el sistema:

- TablaClientes
- TablaProveedores
- TablaAgentesDeOrdenes
- TablaContable
- TablaAgentesDeEnvios
- TablaAlmacenistas
- TablaAgentesDeRecepcion
- TablaPlanificacion

A continuación, se tienen algunas tablas que se refieren a productos y catálogos:

- TablaProductos
- TablaCatalogos

Por último, se tienen algunas tablas que se refieren a los artefactos de trabajo del almacén:

- TablaPedidos
- TablaOrdenesCompra
- TablaFacturas
- TablaOrdenesEmbalaje
- TablaOrdenesAlmacenaje
- TablaEtiquetasExpedicion

No se incluyen tablas para las clases Informe o Transaccion, porque el análisis revela que sus instancias son transitorias, lo que significa que no hay ningún requisito para hacerlas persistentes.

La siguiente fase en el análisis sería decidir sobre los atributos aplicables a cada una de las tablas de arriba. No se tratarán aquí estas cuestiones, porque ya se han expuesto algunos de los atributos más interesantes de estas abstracciones (como, por ejemplo, en la Figura 10.4), y los atributos restantes no proporcionan ninguna nueva percepción arquitectónica del sistema.

10.2. Diseño

En la formulación de la arquitectura del sistema de mantenimiento de catálogos, hay que abordar tres elementos organizativos: la división en funcionalidad

cliente/servidor, un mecanismo para controlar las transacciones y una estrategia para construir aplicaciones cliente.

Arquitectura cliente/servidor

El problema interesante aquí no es tanto *dónde* se fija exactamente la línea entre responsabilidades del cliente y del servidor, sino más bien *cómo* se toma tal decisión de manera inteligente. Un retorno a los principios iniciales nos da la respuesta al cómo: centrarse primero en el comportamiento de cada abstracción, como se deriva de un análisis de casos de uso de cada entidad, y sólo entonces decidir dónde ubicar el comportamiento de cada abstracción. Una vez que se ha hecho esto para unos pocos objetos interesantes, surgirán algunos patrones de comportamiento, y se puede entonces codificar estos patrones para servir de guía al asignar funcionalidad para todas las abstracciones restantes.

Por ejemplo, considérese el comportamiento de las dos clases Pedido y RegistroProducto. Un análisis más detallado de la primera clase, junto con algo de diseño de clases aisladas, sugiere que son aplicables las siguientes operaciones:

- construir
- fijarCliente
- fijarAgenteDeOrdenes
- anadirElemento
- eliminarElemento
- idPedido
- cliente
- AgenteDeOrdenes
- numeroDeElementos
- elementoIesimo
- cantidadDe
- valorTotal

Estos servicios pueden corresponderse directamente con una declaración de clases C++, como la siguiente. Se comienza con dos definiciones de tipos que se enmarcan en el vocabulario del espacio del problema:

```
// tipos IDentificador
typedef unsigned int IdPedido;

// Tipo que denota dinero en la moneda local
typedef float Dinero;
```

A continuación se ofrece una declaración para la clase Pedido:

```
class Pedido {
```

```
public:

    Pedido();
    Pedido(IdPedido);
    Pedido(const Pedido&);
    ~Pedido();

    Pedido& operator=(const Pedido&);
    int operator==(const Pedido&) const;
    int operator!=(const Pedido&) const;

    void fijarCliente(Cliente&);
    void fijarAgenteDeOrdenes(OrderAgent&);
    void anadirElemento(Producto&, unsigned int cantidad = 1);
    void eliminarElemento(unsigned int indice, unsigned int
        cantidad = 1);

    IdPedido idPedido() const;
    Cliente& cliente() const;
    AgenteDeOrdenes& AgenteDeOrdenes() const;
    unsigned int numeroDeElementos() const;
    Producto& elementoIesimo(unsigned int) const;
    unsigned int cantidadDe(unsigned int) const;
    Dinero valorTotal() const;

protected:
    ...
};
```

Nótese los diversos constructores que se proporcionan, cada uno con una semántica sutilmente diferente. El constructor por defecto (`Pedido()`) crea un nuevo objeto pedido y le asigna un nuevo valor `IdPedido` único. El constructor de copia también crea un nuevo objeto pedido con un nuevo valor `IdPedido`, pero además copia el resto de su estado del argumento suministrado.

El constructor restante toma un argumento `IdPedido` que denota un objeto pedido existente, y a continuación obtiene el objeto dado, numerado de forma única. En otras palabras, este constructor particular hace que se acceda a la base de datos de pedidos y (transparentemente) se rematerialice el objeto correspondiente. Esto por supuesto requiere un poco de trabajo bajo la superficie: si alguna actividad anterior en la misma o en otra aplicación había reconstruido previamente el mismo objeto pedido, el mecanismo SQL subyacente tendría que hacer realidad el que esos dos objetos o bien comparten el mismo estado subyacente, o al menos mantuviesen sus estados sincronizados. Este detalle es, por supuesto, un secreto de la implantación de la clase, y por tanto tiene poco interés para los clientes que en realidad utilizan el objeto desde la perspectiva de su interfaz orientado a objetos.

Afortunadamente, el implantar esta estrategia no es tan malo como parece.

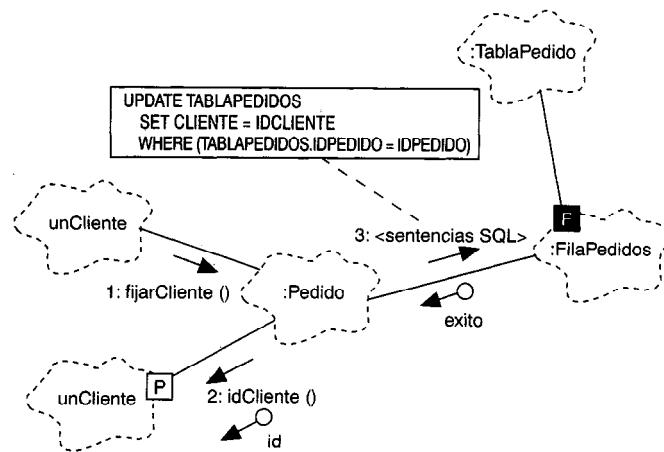


Figura 10.6. Mecanismo SQL.

Si se define la clase `Pedido` de forma que su único estado interesante sea un valor de `idPedido`, entonces todas las operaciones conllevarían más que nada el despacho de sentencias SQL que leyesen o escribiesen en la base de datos. Las copias del mismo objeto se mantienen sincronizadas, porque la tabla correspondiente de la base de datos sirve como el único depósito para el estado de todas las vistas del mismo objeto.

El diagrama de objetos de la Figura 10.6 ilustra este mecanismo SQL, con un escenario que muestra una aplicación cliente⁸ fijando el cliente para un pedido. El escenario sigue este orden de los eventos:

- `unCliente` invoca `fijarCliente` sobre alguna instancia de `Pedido`; se proporciona una instancia de la clase `Cliente` como parámetro para esta función método.
- El objeto `Pedido` invoca al selector `idCliente` sobre el parámetro `cliente`, para obtener su correspondiente clave primaria.
- El objeto `Pedido` despacha una sentencia SQL `UPDATE` para fijar la identificación del cliente en la base de datos de pedidos.

Este mecanismo también significa que se puede confiar en los mecanismos existentes en la base de datos para el bloqueo y la exclusión mutua (por ejemplo, considérense dos aplicaciones intentando actualizar simultáneamente el cliente de un pedido). Si se necesita hacer visibles a los clientes esos mecanismos de bloqueo, se pueden exponer para la sincronización de procesos términos y modismos similares a los utilizados en la biblioteca de clases básicas del Capítulo 9. Como se verá después, la estrategia de transacción ofrece una vía unificada para

⁸ No confundir las dos acepciones del término *cliente* utilizadas aquí; en ocasiones se habla de aplicaciones clientes del sistema cliente/servidor, y en otras ocasiones se habla de clientes como personas que utilizan los servicios de venta por correo. El contexto suele hacer clara la distinción. (N. del T.)

actualizar una serie de objetos de la base de datos de cada vez, protegiendo así la autoconsistencia de la base de datos.

Con este mecanismo a punto, en gran parte se convierte en una cuestión táctica la decisión de dónde situar la lógica que refuerza las reglas de la empresa para la aplicación. Esta situación no es distinta de la que se daría sin una arquitectura orientada a objetos, pero envolviendo las abstracciones en clases se hace posible aplazar estas decisiones y, cuando se adopten, ocultar sus detalles a los clientes. De este modo, los clientes son insensibles a cualquier cambio que se pueda hacer a medida que se ajusta el sistema.

Considérense dos casos distintos. Primero, añadir o eliminar un producto de la base de datos de productos es claramente una acción que requiere una comprobación de reglas significativa, y así se habrá asignado probablemente la mayor parte del refuerzo de las reglas de la empresa al servidor. Añadir un nuevo producto requiere comprobar que se ha definido y descrito de forma única este elemento; también se podría tener que difundir la existencia del nuevo producto a varios clientes, de forma que pudiesen actualizar cualquier tabla temporal que mantuviesen por razones de eficiencia. Análogamente, el eliminar un producto existente requiere comprobar que no se invalida ningún pedido pendiente, y, si se hace, difundir el cambio a los clientes apropiados que podrían quedarse obsoletos por esta acción⁹.

En contraste, las reglas que conciernen al cálculo del valor total de un nuevo pedido se aplican a una actividad mucho más local, y por tanto estarían quizás mejor asignadas al cuidado del cliente. Con este proceder, hay que consultar el precio actual de cada elemento del pedido, realizar conversiones a la moneda local si es necesario, y comprobar reglas locales para descuentos, límites de crédito y cosas así.

Para resumir, se aplican dos reglas para elegir la asignación de responsabilidades cliente/servidor: primero, asignar el cumplimiento de las reglas de la empresa según qué parte del sistema tiene el mayor conocimiento por lo que se refiere al impacto de esas reglas, y, segundo, ocultar estas decisiones bajo una capa orientada a objetos, de forma que si se cambia de idea no importa.

Continuando con el ejemplo, volvamos a la abstracción de una clase diferente, pongamos *Producto*. Un análisis más detallado, junto con un diseño de la clase aislada, sugiere que son aplicables las siguientes operaciones:

- construir
- fijarDescripción
- fijarCantidad
- fijarUbicación
- fijarProveedor
- idProducto
- descripción

⁹ Estos tipos de semántica son exactamente los mismos que la de los disparadores: representa la ligazón de una acción con algún evento significativo de la base de datos. Adoptando una visión del mundo orientada a objetos, se formaliza esta convención de utilizar disparadores encapsulándolos como parte de la semántica de operar sobre un objeto de la base de datos.

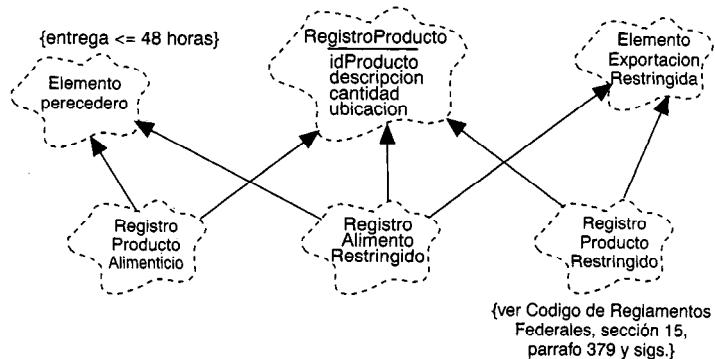


Figura 10.7. Clases de productos.

- cantidad
- ubicacion
- proveedor

Estas operaciones son comunes a todos los tipos de producto. Sin embargo, el análisis de casos de uso revela que esta semántica no es suficiente para ciertos tipos de productos. Por ejemplo, dada la naturaleza abierta del sistema de mantenimiento de catálogos junto con el hecho de que las líneas de productos pueden cambiar, la aplicación puede tener que tratar con los siguientes tipos de productos, que tienen su propio comportamiento único:

- Plantas y productos alimenticios que son perecederos y que por tanto requieren una manipulación y almacenamiento especiales.
- Productos químicos que también requieren un manejo especial porque son cáusticos o tóxicos.
- Distintos productos, como transmisores y receptores de radio, que deberían embalarse emparejadamente y que por tanto dependen uno de otro.
- Componentes de alta tecnología, cuyo embalaje está restringido por las leyes locales de exportación del país.

Estas variaciones sugieren la necesidad de una jerarquía de clases de productos. Sin embargo, como se puede ver, las propiedades representadas por los ejemplos de arriba son ortogonales, y por tanto no se prestan a una jerarquía estricta. En vez de eso, como se muestra en la Figura 10.7, es una estrategia mucho más flexible el usar un estilo aditivo de diseño. Nótese que en este diagrama se ha utilizado la marca de restricción de la notación para plasmar la comprensión que se tiene de la semántica particular de cada abstracción.

¿Qué valor ofrece una trama de herencias para clases que en la práctica son abstracciones de nivel superior sobre entidades de una base de datos relacional? Existe una cantidad tremenda de valor añadido: mediante la formulación de tales jerarquías, se extrae comportamiento común y se eleva a superclases comunes, que son responsables entonces de proporcionar este comportamiento de

manera consistente para todas las instancias, a menos que esas instancias especialicen ese comportamiento (mediante una superclase intermedia) o lo aumenten (mediante una superclase aditiva). Esta estrategia simplifica la arquitectura y la hace más elástica frente a los cambios, porque reduce la redundancia y localiza la estructura y el comportamiento comunes.

Mecanismo de transacción

La computación cliente/servidor implica una colaboración entre un cliente y un servidor, y por eso hay que tener algún mecanismo común mediante el cual partes dispares del sistema se comuniquen entre sí. Como hace notar Berson, «existen tres tipos básicos de técnicas de comunicación en proceso cooperativo que puede utilizar una arquitectura cliente/servidor» [20]:

- Tuberías (*pipes*).
- Llamadas remotas a procedimientos.
- Interacciones SQL cliente/servidor.

Hasta aquí, se ha utilizado sólo el tercer modelo, es decir, todas las interacciones SQL. Sin embargo, para un sistema tan diverso como este, es probable que sea necesario utilizar estos tres patrones en uno u otro momento por razones de eficiencia o, más pragmáticamente, porque sea ese el convenio que nos obligue a utilizar el software adquirido a terceros. Con miras a mantener una visión arquitectónica clara y consistente, sería mejor idear una abstracción de nivel superior que ocultase la elección de los patrones de comunicación.

Anteriormente se introdujo el concepto de una clase transacción, pero no se trabajó sobre su semántica. Como la define Berson, una transacción es «una unidad de procesamiento y de intercambio de información entre un programa local y otro remoto que efectúa una acción o resultado particular» [21]. Ésta es precisamente la abstracción que se necesita: un objeto transacción sirve como el agente responsable de llevar a cabo alguna acción remota, y al hacerlo proporciona una clara separación de intereses entre la propia acción y los mecanismos para realizarla.

En realidad, las transacciones son *el* modelo central de nivel superior del modelo de comunicaciones entre el cliente y el servidor y entre clientes. Las transacciones tienden a caer fuera de un análisis de casos de uso. Todas las funciones principales de la empresa en el sistema de mantenimiento de catálogos pueden, por lo general, abstraerse como una transacción sobre el sistema. Por ejemplo, plantear un pedido, hacer acuse de recibo de nuevas mercancías, o actualizar información de proveedores son todas ellas transacciones aplicables al sistema.

Desde fuera, se observa que las siguientes operaciones capturan nuestra percepción básica del comportamiento de una transacción:

- asignarOperacion
- despachar

- realizar (*commit*)
- deshacer (*rollback*)
- estado

Para cada transacción, se identifica un conjunto completo de operaciones que debe realizar. En programación, esto significa que hay que proporcionar funciones miembro como `asignarOperacion` para la clase `Transaccion` que permitan a otros objetos empaquetar una colección de sentencias SQL para su ejecución como una unidad simple.

Es agradable subrayar que esta visión orientada a objetos del mundo está en armonía conceptualmente con la forma en que el mundo de las bases de datos ve las transacciones. Como dice Date, «una transacción es una secuencia de operaciones SQL (y posiblemente otras) que se garantiza que será atómica para los propósitos de recuperación y control de la concurrencia» [22].

El concepto de atomicidad es una parte importante de la semántica de las transacciones. Si la acción de una transacción particular requiere que se manipulen varias filas de una tabla, hay que agrupar esas operaciones; de otro modo se podría dejar a la base de datos en un estado inconsistente. Por tanto, cuando se despacha una transacción, quiere decirse que se ejecutan sus operaciones asociadas como un todo mutuamente exclusivo.

Si una transacción se completa con normalidad, hay que realizarla, haciendo que tengan efecto todas sus actualizaciones. Sin embargo, el despacho de una transacción puede fallar por una serie de razones, quizás a causa de un fallo de la red que haga imposible abrir una base de datos concreta, o quizás porque otro cliente haya bloqueado ciertos registros críticos necesarios para la actualización. Bajo estas condiciones, hay que deshacer la transacción, lo que hace que se abandonen todas las actualizaciones que la transacción había comenzado. El selector `estado` devuelve un valor que informa si una operación se completó o no con normalidad.

El despacho de una transacción es mucho más complicado en presencia de bases de datos distribuidas. El simple protocolo realizar/deshacer funciona bien si hay que actualizar sólo una base de datos local, pero ¿qué pasa si hay que actualizar varias bases de datos en distintos servidores? La solución general es utilizar lo que se llama un *protocolo de realización en dos fases* [23]. Bajo este protocolo, un agente (en el diseño, una instancia de la clase `Transaccion`) asigna primero las diversas partes de las operaciones de la transacción a sus servidores distribuidos correspondientes; esta es la *fase de preparación*. Si todos esos participantes informan de que están listos para realizar, entonces la transacción central que comenzó esa acción difunde a todos ellos una acción `realizar`; ésta recibe el nombre de *fase de realización*. Si en vez de eso cualquier servidor informa, tras la fase de preparación, de que no estaba listo para la realización, se difunde un `deshacer` de forma que se vuelve atrás al completo la transacción distribuida. Esto es posible en gran medida gracias a que cada instancia de `Transaccion` encapsula suficiente conocimiento sobre su comportamiento como para ser capaz de invertir su acción original.

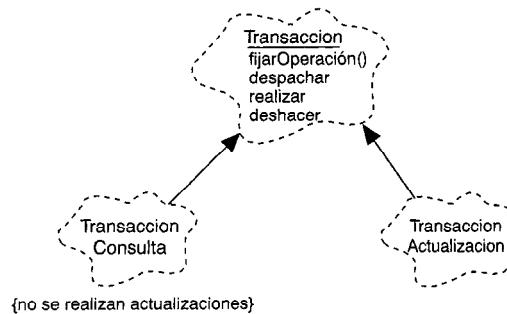


Figura 10.8. Clases de transacción.

La Figura 10.8 muestra un diagrama de clases que ilustra nuestra abstracción de las transacciones. Aquí se ve una jerarquía de transacciones. La clase base `Transaccion` de arriba captura la estructura y comportamiento comunes a todas las transacciones, mientras que las subclases incorporan la semántica de ciertas transacciones especializadas. Se distingue entre `TransaccionActualizacion` y `TransaccionConsulta`, por ejemplo, porque cada una de ellas proporciona una semántica muy distinta: la primera clase modifica el estado del servidor, mientras que la segunda no lo hace. Mediante la distinción entre estos y otros tipos de transacciones, se eleva el comportamiento común a clases comunes, y además se incorpora más del vocabulario del problema.

A medida que se continúa con el desarrollo del sistema, se encontrarán probablemente otros modelos de transacción que constituyen subclases adecuadas. Por ejemplo, si se encuentra que el añadido y eliminación de elementos de ciertas bases de datos tiene sustancialmente la misma semántica, se podría desechar inventar las operaciones `TransaccionAniadir` y `TransaccionBorrar`, para capturar este comportamiento común.

En última instancia, la existencia de la clase base `Transaccion` proporciona una vía abierta para la construcción y despacho de cualquier tipo de acción atómica. Por ejemplo, en C++ se podría escribir:

```

class Transaccion {
public:
    Transaccion();
    virtual ~Transaccion();

    virtual void fijarOperacion(const ColeccionNoLimitada<<SentenciaSQL>> &);
    virtual int despachar();
    virtual void realizar();
    virtual void deshacer();
    virtual int estado() const;
}
  
```

```
protected:  
...  
};
```

Nótese cómo esta clase se apoya en una de las clases básicas que se trataron en el Capítulo 9. Aquí, nuestra abstracción ve a una transacción como la responsable de gestionar una colección *indexable* (indizable) de sentencias. Las operaciones para manipular esta colección se delegan a la clase parametrizada `colecciónNoLimitada`.

En última instancia, este patrón arquitectónico permite a sofisticadas aplicaciones cliente despachar sentencias SQL puras. Las subclases de transacción ocultan esta potencia (y su complejidad asociada) a los clientes más simples que sólo necesitan realizar ciertas transacciones habituales.

Construcción de aplicaciones cliente

En gran medida, el construir una aplicación cliente es un problema de construir un programa de IGU intensivo. Construir un interfaz de usuario intuitivo y amigable tiene, sin embargo, tanto de arte como de ciencia. En aplicaciones cliente/servidor como esta, es frecuentemente el aspecto y sensación del interfaz de usuario lo que marca la diferencia entre un sistema ampliamente difundido y uno que se descarta rápidamente. Factores humanos, restricciones técnicas, razones históricas y las preferencias personales del equipo de desarrollo conspiran para que la realización de un interfaz hombre/máquina útil, expresivo y autoconsistente sea algo realmente difícil.

Como arquitectos del sistema de mantenimiento de inventarios, tenemos por tanto dos riesgos clave de desarrollo a los que enfrentarnos cuando se trata de crear aplicaciones cliente. Primero, ¿cómo se despliega el interfaz de usuario «correcto»? Segundo, ¿qué arquitectura común se puede usar para construir una familia de aplicaciones cliente?

La primera cuestión tiene una respuesta simple, aunque su ejecución es tediosa y requiere una dirección activa de los gestores: prototipos, prototipos y prototipos. Esto significa que hay que usar cualquier facilidad disponible para conseguir poner versiones tempranas de prototipos ejecutables del interfaz de usuario en manos de usuarios reales, de forma que el equipo de desarrollo pueda conseguir una realimentación de calidad por lo que respecta al aspecto y sensación del sistema. El proceso orientado a objetos ayuda enormemente en esta cuestión, porque promueve este tipo de desarrollo incremental e iterativo. El uso de una arquitectura orientada a objetos también tiene mucho que ver al facilitar este uso de prototipos, porque permite ajustar el interfaz de usuario sin hacer pedazos el entramado del sistema.

La segunda cuestión, la de diseñar una arquitectura común para los clientes, es en última instancia una decisión estratégica de diseño, pero, afortunadamente, es una de las que pueden aprovecharse de marcos de referencia de IGUs. Comercialmente, existen productos como el sistema X Window de MIT, Open

Look, Windows de Microsoft, MacApp de Apple, NextStep de Next, y el Presentation Manager de IBM. Cada uno de estos sistemas de ventanas es diferente: algunos están basados en redes, y otros en núcleos (*kernel*); algunos tratan los pixels individuales como el elemento gráfico más primitivo, y otros manipulan abstracciones de nivel superior, como rectángulos, óvalos y arcos. En cualquier caso, todos estos productos tienen un objetivo común: existen para simplificar la tarea de implantar esa parte de una aplicación que forma el interfaz hombre/máquina. Habría que apuntar que ninguno de esos productos surgió de un día para otro. Antes bien, los sistemas de ventanas más útiles evolucionaron con el tiempo, partiendo de sistemas más pequeños y ya probados. Ha llevado años de éxitos y fracasos el que surgiese suficiente consenso en la industria respecto al conjunto útil de abstracciones para el problema de construir interfaces de usuario. Si se ven muchos sistemas de ventanas es porque no hay una sola respuesta correcta al problema del diseño de interfaces de usuario.

Como se dice en el Capítulo 9, la parte verdaderamente difícil de convivir con cualquier biblioteca grande e integrada, especialmente las de interfaces de usuario, es aprender qué mecanismos incorpora. Quizás los mecanismos más importantes que hay que comprender, al menos en el contexto de computación cooperativa cliente/servidor, es cómo responde una aplicación IGU a los eventos. Como hace notar Berson, los clientes IGU tienen que enfrentarse a los siguientes tipos de eventos [24]:

- Eventos de ratón.
- Eventos de teclado.
- Eventos de menú.
- Eventos de actualización de ventana.
- Eventos de cambio de tamaño.
- Eventos de activación/desactivación.
- Eventos de inicialización/terminación.

Se añaden a esta lista los eventos de red¹⁰. Esta última categoría es central en nuestro enfoque del procesamiento cooperativo: las aplicaciones cliente pueden recibir eventos de red de otras aplicaciones, ya sea consultándolas para emprender alguna acción o proporcionándoles algún dato. Esta semántica se integra bien con la invención anterior de la clase Transaccion, porque ahora se pueden ver las transacciones como agentes que envían eventos de una aplicación a otra. En el contexto de cada aplicación cliente, lo bueno de tratar a un evento de red como cualquier otro tipo de evento es que permite usar un mecanismo consistente para reaccionar a cualquier tipo de evento.

Berson observa que existen varios modelos arquitectónicos para tratar con eventos [25]:

¹⁰ Por ejemplo, DDE (Dynamic Data Exchange) y OLE (Object Linking and Embedding) de Microsoft son protocolos uno a uno, basados en mensajes, que proveen a las aplicaciones Windows de un medio de intercambiar información.

- Modelo de bucle de eventos El bucle de eventos comprueba los eventos pendientes y despacha una rutina de manejo de eventos apropiada.
- Modelo de callback de eventos La aplicación registra una función callback o de retorno para cada elemento del IGU que sabe cómo responder a cierto evento; la función se llama cuando el elemento detecta un evento.
- Modelo híbrido Una combinación de los modelos de bucle de eventos y de callback de eventos.

Aunque el decir esto supone una cierta simplificación, se observa que en general MacApp utiliza el modelo de bucle de eventos, Motif sigue el modelo de callback de eventos y Microsoft Windows aplica el modelo híbrido.

Además de este mecanismo primario, hay que seguir otros varios mecanismos comunes del IGU, incluyendo los de dibujo, desplazamiento de pantalla, arrastre y respuesta a acciones del ratón, respuesta a comandos de menú, salvar y recuperar el estado de una aplicación, imprimir, editar (incluyendo el corte, copia, borrado y pegado), gestión de diálogos, recuperación de errores y gestión de memoria. Obviamente, una discusión completa de cada uno de estos mecanismos está mucho más allá del ámbito de este texto, porque cada marco de referencia de IGU tiene sus propios convenios al respecto.

Una regla que se propone para el desarrollador de aplicaciones clientes es seleccionar un marco de referencia IGU apropiado, aprender sus patrones, y aplicarlos consistentemente.

10.3. Evolución

Gestión de versiones

Ahora que se ha establecido un marco de referencia arquitectónico para el sistema de mantenimiento de inventarios, se puede proceder con el desarrollo incremental del sistema. Se comienza este proceso seleccionando primero un pequeño número de transacciones interesantes, tomando un corte vertical a través de la arquitectura, e implantando entonces lo suficiente del sistema como para producir un producto ejecutable que al menos simule la ejecución de estas transacciones.

Por ejemplo, se podría seleccionar solamente tres transacciones simples: añadir un cliente, añadir un producto, y tomar un pedido. Juntándolas, la implantación de estas tres transacciones requiere tocar casi todos los interfaces arquitectónicos críticos, forzándonos así a validar las suposiciones estratégicas. Una vez que se supera este hito con éxito, se podría generar una serie de nuevas versiones, de acuerdo con la siguiente secuencia:

- Modificar o borrar un cliente; modificar o borrar un producto; modificar un pedido; consultar un cliente, pedido y producto.
- Integrar todas las transacciones similares de proveedores, crear una orden de almacenamiento, crear una factura.
- Integrar las transacciones de almacenamiento restantes, crear un informe, crear un embalaje.
- Integrar las transacciones contables restantes, crear una orden de recepción.
- Integrar las transacciones de embalaje restantes.
- Completar las transacciones previstas restantes.

Para un ciclo de desarrollo de 12 a 18 meses, esto significa probablemente generar una versión razonablemente estable cada 3 meses, construyéndose cada una sobre la funcionalidad de la otra. Cuando se haya finalizado, se habrán cubierto todas las transacciones del sistema.

Como se discutió en el Capítulo 6, la clave para tener éxito en esta estrategia es la gestión del riesgo, por la cual para cada versión se identifica el mayor riesgo de desarrollo y se le ataca directamente. Para aplicaciones cliente/servidor como esta, esto significa introducir pruebas de capacidad en momentos tempranos del ciclo evolutivo (de forma que se identifiquen todos los cuellos de botella del sistema lo bastante pronto como para que se pueda hacer algo al respecto de ellos). Como sugiere la secuencia de versiones descrita, esto también significa seleccionar de forma amplia transacciones para cada versión que comprendan los elementos funcionales del sistema, de forma que no se queden en el tintero grietas sin descubrir en el análisis.

Generadores de aplicaciones

Los dominios como el sistema de mantenimiento de catálogos incluyen a menudo muchos tipos diferentes de modelos de pantalla y de informes impresos que hay que generar. Para sistemas grandes, estas partes de la aplicación no son técnicamente difíciles de escribir, sólo horriblemente aburridas. Esta es precisamente la razón por la que los generadores de aplicaciones (o *4GLs = fourth-generation languages o lenguajes de cuarta generación*) están tan extendidos en las empresas de gestión. El uso de 4GLs no es inconsistente con una arquitectura orientada a objetos. En realidad, el uso controlado de 4GLs puede eliminar la escritura de una cantidad de código considerable.

Generalmente, se usan 4GLs para crear automáticamente pantallas e informes. Dada la especificación de la distribución de una pantalla o informe, un 4GL puede generar el código necesario para producir la pantalla o informe verdaderos. Se integra este código con el resto del sistema envolviéndolo manualmente en una capa muy fina orientada a objetos. Los productos del 4GL se convierten así en utilidades de clase que el resto de la aplicación puede utilizar sin saber cómo se crearon.

De este modo, se aprovechan los beneficios del 4GL, pero manteniendo la

ilusión de una arquitectura completamente orientada a objetos. La otra ventaja que ofrece esta estrategia es que, a medida que los 4GLs van siendo progresivamente orientados a objetos y ofreciendo interfaces para la programación de aplicaciones (APIs) para lenguajes orientados a objetos como C++, va menguando el vacío semántico de esta grieta particular del sistema.

Se puede aplicar esta misma estrategia para tratar con todos los diversos diálogos que se podrían encontrar en las aplicaciones clientes. El escribir código para cajas de diálogo modales y no modales es aburrido, porque hay que tratar con todo tipo de características de distribución detalladas. En vez de escribir código para los diálogos¹¹, es mucho mejor solución usar un constructor de IGUs que permita «pintar» los diálogos. Igual que con un generador de aplicaciones de informes, se coloca una delgada piel orientada a objetos sobre los productos del constructor de IGUs, proporcionando una clara separación de intereses.

10.4. Mantenimiento

Un sistema cliente/servidor útil rara vez se da por terminado. Esto no quiere decir que nunca se llegue a un sistema estable. Antes bien, la realidad es que para aplicaciones que son centrales en una empresa, el software debe adaptarse a medida que cambian las reglas del negocio, de otro modo el software se convierte en una carga en vez de en un bien competitivo.

Para el sistema de mantenimiento de catálogos, se pueden intuir varias mejoras que las condiciones cambiantes del negocio pueden plantear:

- Permitir a los clientes enviar electrónicamente sus propios pedidos y consultar el estado de pedidos pendientes.
- Generar automáticamente catálogos personalizados de la base de datos del inventario, adaptada al objetivo de grupos específicos de clientes, o incluso clientes individuales.
- Automatizar completamente todas las funciones del almacén, eliminando por tanto al almacenista humano, así como a la mayoría del personal de recepción y embalaje.

En realidad, el riesgo dominante en cada uno de estos cambios particulares no es técnico, sino social y político. Teniendo una arquitectura orientada a objetos flexible, la organización de desarrollo ofrece al menos a la compañía muchos grados de libertad a la hora de ser capaz de adaptarse con agilidad al mercado cambiante.

¹¹ El escribir software orientado a objetos puede ser divertido, pero es mucho más importante centrarse en satisfacer los requerimientos del problema que se trata. Esto significa evitar tener que escribir nuevo código siempre que sea posible. Los generadores de aplicaciones y los constructores de IGUs no son más que dos formas de hacer esto. Los marcos de referencia, como se describen en el Capítulo 9, son otro elemento esencial en este mismo camino.

Lecturas recomendadas

Se ha escrito más sobre computación cliente/servidor de lo que la mayoría de los mortales desearía leer en toda su vida. Dos referencias particularmente útiles son Dewire [H 1992] y Berson [H 1992], ofreciendo ambos un extenso y legible examen del espectro de tecnologías cliente/servidor. Bloom [H 1993] proporciona un recuento corto pero lúcido sobre los conceptos y problemas básicos de las arquitecturas cliente/servidor.

*Downsizing** no es lo mismo que la computación cliente/servidor, aunque el downsizing de una sistema de gestión de la información corporativo conlleva a menudo el uso de una arquitectura cliente/servidor. Puede encontrarse un estudio de las motivaciones y los riesgos que afectan al downsizing en Guengerich [H 1992].

Puede encontrarse un extenso tratamiento de la tecnología de bases de datos relacionales en Date [E 1981, 1983; 1986]. Además, Date [E 1987] ofrece una descripción del estándar SQL. Pueden encontrarse varios enfoques al análisis de datos en Veryard [B 1984], Hawryszkiewycz [E 1984] y Ross [F 1987].

Las bases de datos orientadas a objetos representan la mezcla de la tecnología de bases de datos convencionales y del modelo de objetos. Pueden encontrarse informes sobre trabajos en este campo en Cattell [E 1991], Atwood [E 1991], Davis et al. [H 1983], Kim y Lochovsky [E 1989], y Zdonik y Maier [E 1990].

La bibliografía ofrece varias referencias de varios sistemas de ventanas e interfaces de usuario orientados a objetos (véase sección K, «Herramientas y Entornos»). Pueden encontrarse detalles sobre el API de Microsoft Windows en Windows [G 1992]. Puede encontrarse una referencia similar para MacApp de Apple en Macapp [G 1992].

Notas bibliográficas

- [1] Mimno, P., April 1993. Client-Server Computing. *American Programmer*, Arlington MA: Cutter Information Corporation, p. 19.
- [2] Mimno, p. 21.
- [3] Berson, A. 1992. *Client/Server Architecture*. New York, NY: McGraw-Hill, p. 34.
- [4] Berson, p. 37.
- [5] Berson, p. 12.
- [6] Berson, p. 13.
- [7] Date, C. 1981. *An Introduction to Database Systems*. Vol. 1. Reading, MA: Addison-Wesley, p. 4.
- [8] Date. *An Introduction*, p. 10.
- [9] Hawryszkiewycz, I. 1984. *Database Analysis and Design*. Chicago, IL: Science Research Associates, p. 425.
- [10] Wiorkowski, G. and Kull, D. 1988. *DB2 Design and Development Guide*. Reading, MA: Addison-Wesley, p. 29.
- [11] Date. *An Introduction*, p. 63.

* Downsizing es un término que significa literalmente «reducción de tamaño» y que se aplica al hardware. Habitualmente no se traduce. (*N. del T.*)

- [12] Wiorkowski and Kull. *DB2 Design*, p. 2.
- [13] Date. *An Introduction*, p. 238.
- [14] Wiorkowski and Kull. *DB2 Design*, p. 15.
- [15] Date, C. 1986. *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley, p. 461.
- [16] Rumbaugh, J. July/August 1992. Onward to OOPSLA. *Journal of Object-Oriented Programming*, vol. 5(4).
- [17] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenzen, W. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall, p. 386.
- [18] Ibid.
- [19] Date. *An Introduction*. p. 237.
- [20] Berson, p. 39.
- [21] Berson, p. 441.
- [22] Date, C. 1987. *The Guide to the SQL Standard*. Reading, MA: Addison-Wesley, p. 32.
- [23] Berson, p. 244.
- [24] Berson, p. 61.
- [25] Ibid.

Inteligencia artificial: Criptoanálisis

Las criaturas sensibles exhiben un conjunto de comportamientos enormemente complejo que emana de la mente a través de mecanismos que comprendemos sólo muy pobemente. Por ejemplo, piénsese en cómo se resuelve el problema de planificar una ruta a través de una ciudad para efectuar una serie de recados. Considérese también cómo, cuando se camina a través de una habitación oscura, uno es capaz de reconocer los límites de los objetos y evitar tropezones. Más aún, piénsese cómo una persona se puede concentrar en una conversación en una fiesta mientras hay docenas de personas hablando a la vez. Ninguno de estos tipos de problemas se prestan a una solución algorítmica obvia. La planificación óptima de rutas se conoce como un problema *np-completo*. La navegación a través de un terreno oscuro implica deducir determinada percepción partiendo de una entrada visual que es (muy literalmente) difusa e incompleta. Identificar a un solo hablante de entre docenas de fuentes requiere que quien escucha distinga los datos significativos del ruido y que filtre entonces todas las conversaciones indeseadas de la cacofonía de fondo.

Los investigadores en el campo de la inteligencia artificial han atacado estos y otros problemas similares para mejorar nuestra comprensión sobre el proceso cognitivo humano. La actividad en este campo conlleva a menudo la construcción de sistemas inteligentes que copian ciertos aspectos del comportamiento humano. Erman, Lark y Hayes-Roth apuntan que «los sistemas inteligentes difieren de los sistemas convencionales en una serie de atributos, los cuales no están siempre presentes:

- Persiguen metas que varían con el tiempo.
- Incorporan, usan y mantienen conocimiento.
- Explotan diversos subsistemas ad hoc que incorporan diversos métodos seleccionados.

- Interaccionan inteligentemente con los usuarios y otros sistemas.
- Asignan sus propios recursos y atención» [1].

Cualquiera de estas propiedades es causa suficiente para que la construcción de sistemas inteligentes sea una tarea muy difícil. Cuando se considera que los sistemas inteligentes se están desarrollando para un espectro de dominios que afectan tanto a la vida como a la propiedad, como para diagnosis médica o dirección de aviones, la tarea se vuelve aún más exigente porque hay que diseñar estos sistemas de forma que nunca sean activamente peligrosos: la inteligencia artificial rara vez incorpora cualquier tipo de conocimiento de sentido común.

Aunque este campo se ha vendido a veces con optimismo exagerado por parte de una prensa demasiado entusiasta, el estudio de la inteligencia artificial ha proporcionado algunas ideas muy buenas y prácticas, entre las que se cuentan aproximaciones a la representación del conocimiento y la evolución de arquitecturas comunes de resolución de problemas para sistemas inteligentes, incluyendo sistemas expertos basados en reglas o el modelo de la pizarra [2]. En este capítulo, nos volvemos hacia el diseño de un sistema inteligente que resuelve criptogramas usando un marco de referencia de pizarra en una forma que paraleliza el modo en que un humano resolvería el mismo problema. Como se verá, el uso del desarrollo orientado a objetos es muy adecuado para este dominio.

11.1. Análisis

Definición de los límites del problema

Como se subraya en el recuadro, el problema es de criptoanálisis, el proceso de reconvertir texto cifrado en texto normal. En su forma más general, el descifrar criptogramas es un problema intratable que desafía incluso la más sofisticada de las técnicas. Por ejemplo, el DES (el estándar de cifrado de datos o *Data Encryption Standard*, un algoritmo de cifrado —o codificado— de clave privada que usa múltiples aplicaciones de sustitución y transposición) parece estar libre de cualquier debilidad matemática y, por tanto, es seguro frente a todos los flancos de ataque conocidos. Felizmente, nuestro problema es mucho más simple, porque nos limitamos a simples sustituciones.

Como parte del análisis, va a recorrerse un escenario de la resolución de un criptograma simple. Pierda el lector los próximos minutos en resolver el siguiente problema¹ y, a medida que avance, tome nota de cómo lo hace (sin trampas y sin leer más adelante):

Q AZWS DSSC KAS DXZNN DASNN

Como ayuda, se advierte que la letra w representa lo que en realidad es la v.

¹ El ejemplo es el del original. (*N. del T.*)

Requisitos del criptoanálisis

La criptografía «abraza métodos para que los datos aparezcan como inteligibles para entidades no autorizadas» [3]. Usando algoritmos criptográficos, los mensajes (texto normal) pueden transformarse en criptogramas (texto cifrado) y a la inversa.

Uno de los tipos de algoritmos de criptografía más básicos, empleados desde tiempos de los romanos, se llama cifrado por sustitución. Con este cifrado, cada letra del alfabeto del texto normal se corresponde con una letra diferente. Por ejemplo, se podría desplazar cada letra a su sucesor: A se transforma en B, B se transforma en C, Z vuelve a ser A y así sucesivamente. Así, el texto

CLOS es un lenguaje de programación orientado a objetos

puede cifrarse como el criptograma

DMPT ft vñ mfñhvbkf ef qspbsbnbdjpñ psjfñubep b pckfupt

Con mayor frecuencia, la sustitución de letras se desordena. Por ejemplo, A pasa a ser G, B pasa a ser J, etc. Como ejemplo, considérese el criptograma siguiente:

BS TBCEAS KG KACL GL ZHS NGELCHS NBGHSJGHPG IAIS²

Ayuda: la letra s representa a la letra normal A.

Es una suposición enormemente simplificadora el saber que sólo se utilizó un cifrado por sustitución al codificar un mensaje de texto; sin embargo, descifrar el criptograma resultante no es una tarea algorítmicamente trivial. El descifrado a veces exige realizar intento y error, realizándose suposiciones sobre una sustitución particular y se evalúan entonces sus implicaciones. Por ejemplo, se puede comenzar con las palabras de una y dos letras del criptograma y hacer la hipótesis de que representan palabras comunes como a, o, y³, o también es, se, un, el o la. Sustituyendo las otras ocurrencias de estas letras cifradas, pueden encontrarse ayudas para descifrar otras palabras. Por ejemplo, una palabra de tres letras que empieza con u podría ser razonablemente una, uno o uso.

También puede usarse el conocimiento que se tiene sobre ortografía y gramática para atacar un cifrado por sustitución. Por ejemplo, una ocurrencia de dobles letras no es probable que represente la secuencia qq. Análogamente, se podría intentar expandir un final de palabra con la letra e al sufijo -mente. A un nivel más alto de abstracción, se podría asumir que la secuencia de palabras en que es más probable que aparezca en cierto contexto que la secuencia el que. Además, se podría suponer que la estructura de una sentencia incluye normalmente un nombre y un verbo. Así, si el análisis ha identificado un verbo pero no un agente, se buscarían adjetivos y nombres.

² Este ejemplo está traducido al español, puesto que el traductor ha encontrado la clave. La frase inglesa era PDG TBCER CQ TCK ALS NGELCH QBBR SBAJG, y la pista, que c=o. (N. del T.)

³ Téngase en cuenta a lo largo de todo este capítulo que los ejemplos originales están en inglés, y algunos de ellos se han traducido cuando pueden representar el mismo concepto. (N. del T.)

A veces hay que retroceder. Por ejemplo, se puede haber supuesto que una palabra era el, pero si la sustitución causa contradicciones en otras palabras, habría que probar con en, y deshacer así otras suposiciones basadas en esta sustitución anterior.

Esto lleva al requisito del problema: diseñar un sistema que, dado un criptograma, lo transforme de nuevo a su texto normal original, suponiendo que sólo se empleó un cifrado por sustitución.

Intentar una búsqueda exhaustiva es algo que no tiene sentido. Suponiendo que el alfabeto normal abarcase sólo los 26 caracteres ingleses en mayúsculas, ¡hay aproximadamente 26 (del orden de $4,03 \times 10^{26}$) combinaciones posibles! Así, hay que probar algo que no sea un ataque mediante la fuerza bruta. Una técnica alternativa es realizar una suposición basada en el conocimiento que se tiene sobre la estructura de frase, palabra y letra, y seguirla hasta sus conclusiones naturales. Una vez que no se puede ir más allá, se elige la siguiente suposición más prometedora que se construye sobre la primera, y así sucesivamente, siempre y cuando cada suposición nos acerque a una solución. Si se llega a un atasco, o se alcanza una conclusión que contradice a otra anterior, hay que volver atrás y modificar alguna suposición anterior.

Esta es nuestra solución, mostrando los resultados de cada paso:

1. De acuerdo con la ayuda, se puede sustituir directamente v por w:

Q AZVS DSSC KAS DXZNN DASNN

2. La primera palabra es pequeña, así que será probablemente una A o una I; supóngase que es A.

A AZVS DSSC KAS DXZNN DASNN

3. La tercera palabra necesita una vocal, y probablemente serán las letras dobles. Sería raro que fuese II o UU, y no puede ser AA porque ya se ha usado una A. Así, se podría intentar EE.

A AZVE DEEC KAE DXZNN DAENN

4. La cuarta palabra tiene tres letras, y acaba por E; es probable que sea la palabra THE.

A HZVE DEEC THE DXZNN DHENN

5. La segunda palabra necesita una vocal, pero sólo puede ser I, O o U (ya se ha usado la A). Sólo la I da una palabra con significado.

A HIVE DEEC THE DXZNN DHENN

6. Hay pocas palabras de cuatro letras que tengan una doble E, incluyendo DEER, BEER y SEEN. La gramática sugiere que la tercera palabra debería ser un verbo, así que se elige SEEN.

A HIVE SEEN THE SXINN SHENN

7. Esta sentencia no está adquiriendo ningún sentido (los enjambres⁴ no pueden ver), así que probablemente se ha hecho alguna suposición errónea en algún punto del trayecto. El problema parece estribar en la vocal de la segunda palabra, y así se podría reconsiderar la suposición inicial.

I HAVE SEEN THE SXANN SHENN

8. Ataquemos la última palabra. Las dobles letras no pueden ser ss (ya se ha usado una s, y además SHESS no significa nada), pero LL forma una palabra correcta.

I HAVE SEEN THE SXALL SHELL

9. La última palabra es parte de una expresión nominal, y por eso es probablemente un adjetivo (STALL, por ejemplo, se rechaza por esta razón). Buscando palabras que encajen con el modelo S?ALL aparece SMALL.

I HAVE SEEN THE SMALL SHELL⁵

Así, se ha llegado a una solución.

Se pueden hacer las tres observaciones siguientes sobre este proceso de resolución de problemas:

- Se han aplicado muchas fuentes de conocimiento diferentes, como conocimiento sobre gramática, ortografía y vocales.
- Se han registrado las suposiciones en un lugar central y se han aplicado las fuentes de conocimiento a esas suposiciones para razonar sobre sus consecuencias.
- Se ha razonado de forma oportunista. A veces, se ha razonado de reglas generales a reglas específicas (si la palabra tiene tres letras y acaba en E, es probablemente THE), y otras veces se ha razonado de lo específico a lo general (?EE? podría ser DEER, BEER o SEEN, pero ya que la palabra debe ser un verbo y no un nombre, sólo SEEN satisface la hipótesis).

Lo que se ha descrito es un enfoque de resolución de problemas conocido como el *modelo de pizarra*⁶. El modelo de pizarra fue propuesto en primer lugar por Newell en 1962, e incorporado después por Reddy y Erman en los proyectos

⁴ La palabra «hive» significa «enjambre» y «seen» es el participio de «ver». (N. del T.)

⁵ Se podría traducir como «he visto la concha pequeña». (N. del T.)

⁶ «Blackboard model» en el original. (N. del T.)

Hearsay y Hearsay II, que se enfrentaban a problemas de reconocimiento del habla [4]. El modelo de pizarra demostró ser útil en este dominio, y el marco de referencia se aplicó pronto con éxito a otros dominios, incluyendo interpretación de señales, modelado de estructuras moleculares tridimensionales, interpretación de imágenes y planificación [5]. Los marcos de referencia de pizarra han demostrado ser especialmente aplicables a la representación del conocimiento declarativo, y son eficientes respecto a espacio y tiempo cuando se los compara con enfoques alternativos [6].

Un marco de referencia de pizarra satisface la definición de marco de referencia, como se describió en el Capítulo 9. Se puede codificar por tanto su arquitectura en términos de un conjunto de clases y mecanismos que describen cómo colaboran las instancias de esas clases.

Arquitectura del marco de referencia de pizarra

Englemore y Morgan explican el modelo de pizarra por analogía con el problema de un grupo de personas que resuelven un puzzle:

Imagínese una habitación con una gran pizarra y a su alrededor un grupo de personas, cada una de las cuales tiene grandes piezas de un puzzle. Se comienza con voluntarios que ponen en la pizarra (suponiendo que sea adherente) sus piezas más ‘prometedoras’. Cada miembro del grupo mira sus piezas y ve si cualquiera de ellas encaja con las piezas que ya están en la pizarra. Los que tienen piezas apropiadas van al encerado y actualizan la solución que se va desplegando. Las nuevas actualizaciones hacen que otras piezas encuentren su sitio, y otras personas van al encerado a añadir sus piezas. No importa si una persona tiene más piezas que otra. Todo el puzzle puede resolverse en completo silencio; es decir, no es necesario que haya una comunicación directa entre los miembros del grupo. Cada persona se activa por sí misma, sabiendo cuándo sus piezas contribuirán a la solución. No existe un orden establecido *a priori* para que las personas se acerquen a la pizarra. El comportamiento aparentemente cooperativo está mediado por el estado de la solución que hay en la pizarra. Si uno mira la tarea que se está haciendo, la solución se construye incrementalmente (una pieza de cada vez) y oportunísticamente (cuando surge una oportunidad para añadir una pieza), en oposición a comenzar, digamos, sistemáticamente desde la esquina superior izquierda e ir intentando todas las piezas [7].

Como indica la Figura 11.1, el marco de referencia de pizarra consta de tres elementos: una pizarra, múltiples fuentes de conocimiento, y un controlador que media entre estas fuentes de conocimiento [8]. Nótese cómo la siguiente descripción paraleliza los principios del modelo de objetos. De acuerdo con Nii, «el propósito de la pizarra es mantener los datos computacionales y de estado de la solución necesitados y producidos por las fuentes de conocimiento. La pizarra

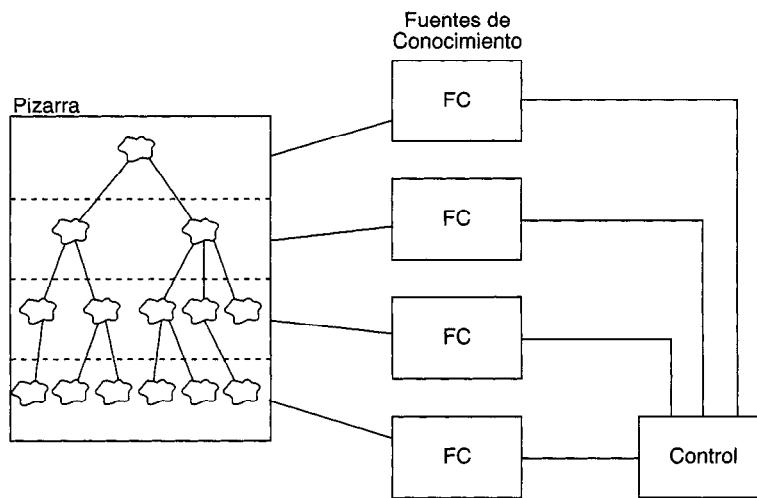


Figura 11.1. Un marco de referencia de pizarra.

consta de objetos del espacio de la solución. Los objetos de la pizarra están organizados jerárquicamente en niveles de análisis. Los objetos y sus propiedades definen el vocabulario del espacio de la solución» [9].

Como explican Englemore y Morgan, «el conocimiento del dominio que se necesita para resolver un problema se divide en fuentes de conocimiento que se mantienen separadas e independientes. El objetivo de cada fuente de conocimiento es contribuir con información que llevará a una solución del problema. Una fuente de conocimiento toma un conjunto de información actual de la pizarra y la actualiza tal como esté codificado en su conocimiento especializado. Las fuentes de conocimiento se representan como procedimientos, conjuntos de reglas o aserciones lógicas» [10].

Las fuentes de conocimiento, o FCs para abreviar, son específicas del dominio. En sistemas de reconocimiento del habla, las fuentes de conocimiento podrían incluir agentes que puedan razonar sobre fonemas, palabras y frases. En sistemas de reconocimiento de imágenes, las fuentes de conocimiento incluirían agentes que supiesen cosas sobre elementos simples de esas imágenes, como bordes y regiones de textura similar, así como abstracciones de nivel superior que representasen los objetos de interés en cada escena, como casas, caminos, campos, coches y personas.

Hablando en términos generales, las fuentes de conocimiento paralelizan la estructura jerárquica de los objetos de la pizarra. Además, cada fuente de conocimiento usa objetos de un nivel en su entrada y genera y/o modifica objetos de otro nivel en su salida. Por ejemplo, en un sistema de reconocimiento del habla, una fuente de conocimiento que incorpore conocimiento sobre las palabras podría observar una corriente de fonemas (a un nivel bajo de abstracción)

para formar una nueva palabra (a un nivel más alto de abstracción). Alternativamente, una fuente de conocimiento que incorporase conocimiento sobre la estructura de las frases podría hacer hipótesis sobre la necesidad de un verbo (a un nivel alto de abstracción); filtrando una lista de posibles palabras (a un nivel más bajo de abstracción), esta fuente de conocimiento puede verificar la hipótesis.

Estos dos enfoques hacia el razonamiento representan, respectivamente, encadenamiento hacia delante y encadenamiento hacia atrás. El *encadenamiento hacia delante* representa un razonamiento desde afirmaciones específicas hasta una afirmación general, y el *encadenamiento hacia atrás* comienza con una hipótesis, e intenta entonces verificarla partiendo de afirmaciones existentes. Esta es la razón por la que se dice que el control en el modelo de pizarra es oportunista: dependiendo de las circunstancias, podría seleccionarse para activación una fuente de conocimiento que usase o bien encadenamiento hacia adelante o bien encadenamiento hacia atrás.

Las fuentes de conocimiento suelen incorporar dos elementos, a saber, precondiciones y acciones. Las precondiciones de una fuente de conocimiento representan el estado de la pizarra en la que la fuente de conocimiento muestra un interés. Por ejemplo, una precondición para una fuente de conocimiento en un sistema de reconocimiento de imágenes podría ser el descubrimiento de una región relativamente lineal de elementos de dibujo (quizás representando una carretera). El disparo de una precondición hace que la fuente de conocimiento centre su atención en esta parte de la pizarra y emprenda entonces una acción procesando sus reglas o su conocimiento procedimental.

Bajo estas circunstancias, no es necesaria una consulta: cuando una fuente de conocimiento piensa que tiene algo interesante con lo que contribuir, se lo notifica al controlador de la pizarra. Hablando en sentido figurado, es como si cada fuente de conocimiento levantase la mano para indicar que tiene algo útil que hacer; entonces, de entre las fuentes de conocimiento impacientes, el controlador llama a la que parece más prometedora.

Análisis de fuentes de conocimiento

Volvamos al problema específico, y considérense las fuentes de conocimiento que pueden contribuir a una solución. Como ocurre típicamente con la mayoría de las aplicaciones de ingeniería del conocimiento, la mejor estrategia es sentarse con un experto del dominio y registrar los heurísticos que esta persona aplica para resolver los problemas del dominio. Para el problema presente, esto podría implicar intentar resolver una serie de criptogramas y registrar el proceso de pensamiento sobre la marcha.

El análisis sugiere que hay trece fuentes de conocimiento relevantes; aparecen con el conocimiento que incorporan en la lista siguiente⁷:

⁷ A partir de este momento, ya no se trata con ejemplos de fondo, sino que la idiosincrasia del idioma inglés puede influir poderosamente en la arquitectura del sistema. Por tanto, ya no se traducirán los ejemplos o las cuestiones idiomáticas, que en este capítulo son muy relevantes. (*N. del T.*)

• Prefijos comunes	Comienzos de palabra habituales como <i>re</i> , <i>anti</i> y <i>un</i> .
• Sufijos comunes	Finales de palabra habituales como <i>ly</i> , <i>ing</i> , <i>es</i> y <i>ed</i> .
• Consonantes	Letras no vocales.
• Sustitución directa	Ayudas que se dan como parte del enunciado del problema.
• Letras dobles	Letras dobles habituales, como <i>tt</i> , <i>ll</i> y <i>ss</i> .
• Frecuencia de las letras	Probabilidad de aparición de cada letra.
• Cadenas permitidas	Combinaciones legales e ilegales de letras, como <i>qu</i> y <i>zg</i> , respectivamente.
• Emparejamiento de patrones	Palabras que encajan con un patrón concreto de letras.
• Estructura de frase	Gramática, incluyendo los significados de expresiones nominales y verbales.
• Palabras pequeñas	Emparejamientos posibles para palabras de una, dos, tres y cuatro letras.
• Resuelto	Cuándo el problema se ha resuelto y cuándo no, o si no puede hacerse ningún progreso más.
• Vocales	Letras no consonantes.
• Estructura de palabra	La ubicación de las vocales y la estructura común de nombres, verbos, adjetivos, adverbios, artículos, conjunciones, etc.

Desde una perspectiva orientada a objetos, cada una de estas trece fuentes de conocimiento representa una clase candidata de la arquitectura: cada instancia incorpora algún estado (su conocimiento), cada una exhibe cierto comportamiento específico de la clase (una fuente de conocimiento sobre sufijos puede reaccionar ante palabras que se sospecha que tienen un final habitual), y cada una se identifica de forma única (una fuente de conocimiento sobre palabras pequeñas existe con independencia de la fuente de conocimiento sobre emparejamiento de patrones).

También se pueden disponer estas fuentes de conocimiento en una jerarquía. En concreto, algunas fuentes de conocimiento operan sobre frases, otras sobre palabras, otras sobre grupos contiguos de letras, y las de nivel inferior sobre las letras individuales. En realidad, esta jerarquía refleja los objetos que pueden aparecer en la pizarra: frases, palabras, cadenas de letras y letras.

11.2. Diseño

Arquitectura de la pizarra

Ahora se está en disposición de diseñar una solución al problema de criptoanálisis usando el marco de referencia de pizarra que se ha descrito. Este es un ejemplo clásico de reutilización a gran escala, en la que se puede reutilizar como fundamento del diseño un patrón arquitectónico probado. La estructura del marco de referencia de pizarra sugiere que entre los objetos del nivel más alto del sistema están una pizarra, varias fuentes de conocimiento y un controlador. La siguiente tarea es identificar las clases y objetos específicos del dominio que especializan estas abstracciones generales.

Objetos de la Pizarra. Los objetos que aparecen en una pizarra existen en una jerarquía estructural que va en paralelo con los diferentes niveles de abstracción de las fuentes de conocimiento. Así, se tienen las tres clases siguientes:

- Frase Un criptograma completo.
- Palabra Una sola palabra del criptograma.
- LetraCifrada Una sola letra de una palabra.

Las fuentes de conocimiento también deben compartir conocimiento sobre las suposiciones que hace cada una, de modo que se incluye la siguiente clase de objetos de pizarra:

- Suposicion Una suposición hecha por una fuente de conocimiento.

Finalmente, es importante saber qué letras de texto normal y de texto cifrado del alfabeto se han usado en suposiciones hechas por las fuentes de conocimiento, así que se incluye la clase siguiente:

- Alfabeto El alfabeto del texto normal, el del texto cifrado, y la correspondencia entre ambos.

¿Hay algo en común entre estas cinco clases? La respuesta es un resonante sí: cada una de estas clases representa objetos que pueden ponerse en una pizarra, y esa importantísima propiedad los distingue de, por ejemplo, las fuentes de conocimiento y los controladores. Así, se inventa la siguiente clase como la superclase de todos los objetos que pueden aparecer en una pizarra:

```
class ObjetoDePizarra ...
```

Observando esta clase desde una vista externa, se pueden definir dos operaciones aplicables:

- registrar Añade el objeto a la pizarra.

- renunciar Elimina el objeto de la pizarra.

¿Por qué se definen `registrar` y `renunciar` como operaciones sobre instancias de `ObjetoDePizarra`, en vez de sobre la propia pizarra? Esta situación no es diferente de la de decir a un objeto que se dibuje a sí mismo en una ventana. La piedra de toque para decidir dónde situar estos tipos de operaciones es si la propia clase tiene o no suficiente conocimiento o responsabilidad para efectuar la operación. En el caso de `registrar` y `renunciar`, ésta es realmente la situación: el objeto de pizarra es la única abstracción con conocimiento detallado sobre cómo añadirse o eliminarse a sí mismo de la pizarra (aunque esto ciertamente requiere colaboración con el objeto de tipo pizarra). De hecho, es una responsabilidad importante de esta abstracción que cada objeto de pizarra esté auto-enterado de que está unido a la pizarra, porque sólo entonces puede comenzar a participar en la resolución oportunística del problema de la pizarra.

Dependencias y afirmaciones. Las frases individuales, palabras individuales y letras cifradas tienen otra cosa en común: cada una tiene ciertas fuentes de conocimiento que dependen de ella. Una fuente de conocimiento dada puede expresar un interés sobre uno o más de esos objetos, y, por tanto, una frase, palabra o letra cifrada debe mantener una referencia a cada una de tales fuentes de conocimiento, de forma que cuando cambia una suposición sobre el objeto pueda notificarse a las fuentes de conocimiento apropiadas que ha ocurrido algo interesante. Este mecanismo es similar al mecanismo de dependencia de Smalltalk que se mencionó en el Capítulo 4. Para proporcionar este mecanismo, se introduce una simple clase aditiva:

```
class Dependiente {
public:

    Dependiente();
    Dependiente(const Dependiente&);
    virtual ~ Dependiente();

    ...

protected:

    ColeccionNoLimitada<FuenteConocimiento*> referencias;

};
```

Se ha saltado a la implantación de esta clase para mostrar que se construye sobre la biblioteca de clases básicas que se describió en el Capítulo 9. Aquí se ve

que la clase `Dependiente` tiene un solo objeto miembro, que representa una colección de punteros a fuentes de conocimiento⁸.

Se definen las siguientes operaciones para esta clase:

- `anadir` Añadir una referencia a la fuente de conocimiento.
- `eliminar` Eliminar una referencia a la fuente de conocimiento.
- `numeroDeDependientes` Devuelve el número de dependientes.
- `notificar` Difunde una operación de cada dependiente.

La operación `notificar` tiene la semántica de un iterador pasivo, lo que significa que cuando se lo invoca, se puede suministrar una operación que se desearía realizar sobre todos los dependientes de la colección.

La dependencia es una propiedad independiente que puede «mezclarse» con otras clases. Por ejemplo, una letra cifrada es un objeto de pizarra, así como un dependiente, así que se pueden combinar estas dos abstracciones para conseguir el comportamiento deseado. El uso de mezclas o adiciones de esta forma incrementa la reutilizabilidad y separación de intereses en la arquitectura.

Las letras y alfabetos cifrados tienen otra propiedad en común: las instancias de ambas clases pueden tener suposiciones que se han hecho sobre ellas (y recuérdese que una aserción también es un tipo de `ObjetoDePizarra`). Por ejemplo, una fuente de conocimiento determinada podría suponer que la letra de texto cifrado `K` representa a la letra de texto normal `P`. A medida que nos acercamos a la resolución del problema, se podría realizar la suposición inmodificable de que `G` representa a `J`. Así, se incluye la clase siguiente:

```
class Afirmacion...
```

Las responsabilidades de esta clase son mantener las suposiciones o aserciones sobre el objeto asociado. No se usa `Afirmacion` como una clase aditiva, sino que en vez de eso se usa para agregación. Las letras *tienen* afirmaciones que se han hecho sobre ellas, no son *tipos de* afirmaciones.

En la arquitectura, sólo se harán afirmaciones sobre letras individuales como en letras cifradas y alfabetos. Como implicaba el escenario anterior, las letras cifradas representan letras simples sobre las que podrían hacerse afirmaciones, y los alfabetos comprenden muchas letras, cada una de las cuales podría tener hechas diferentes afirmaciones sobre ella. La definición de `Afirmacion` como una clase independiente sirve así para capturar el comportamiento común a esas dos clases separadas.

Se definen las siguientes operaciones para instancias de esta clase:

⁸ En la arquitectura de las clases básicas del Capítulo 9 se comprobó que las estructuras no limitadas requieren un gestor de almacenamiento. Por simplicidad, se omite este argumento de la plantilla en esta y otras declaraciones similares de este capítulo. Por supuesto, una implantación completa tendría que respetar los mecanismos del marco de referencia básico.

● <code>hacer</code>	Hacer una afirmación.
● <code>retractar</code>	Retractarse de una afirmación.
● <code>textoCifrado</code>	Dada una letra de texto normal, devuelve su equivalente en texto cifrado.
● <code>textoNormal</code>	Dada una letra de texto cifrado, devuelve su equivalente en texto normal.

Un análisis posterior sugiere que se debería distinguir claramente entre los dos papeles desempeñados por una afirmación: una suposición, que representa una correspondencia temporal entre una letra de texto cifrado y su equivalente de texto normal, y una aserción, que es una correspondencia permanente, lo que significa que la correspondencia está definida y por tanto no se puede cambiar. Durante la solución de un criptograma, las fuentes de conocimiento harán muchas suposiciones y, a medida que se llega más cerca de una solución final, estas correspondencias llegarán eventualmente a ser aserciones. Para modelizar estos papeles cambiantes, se va a refinar la clase previamente identificada `Suposicion`, e introducir una nueva subclase llamada `Asencion`, cuyas instancias (de ambas) son gestionadas por instancias de la clase `Afirmacion`, así como situadas en la pizarra. Se comienza por completar la firma de las operaciones `hacer` y `retractar` para incluir un argumento `Suposicion` o `Asencion`, y se añaden entonces los selectores siguientes:

- `seHaDefinidoLetraNormal` Un selector: ¿está definida la letra de texto normal?
- `seHaDefinidoLetraCifrada` Un selector: ¿está definida la letra de texto cifrado?
- `letraNormalTieneSuposicion` Un selector: ¿existe una suposición sobre la letra de texto normal?
- `letraCifradaTieneSuposicion` Un selector: ¿existe una suposición sobre la letra de texto cifrado?

A continuación, se define la clase `Suposicion`. Puesto que esta abstracción es en gran medida una abstracción estructural, se hace parte de su estado no encapsulado:

```
class Suposicion : public ObjetoDePizarra {
public:
    ...
    ObjetoDePizarra* destino;
    FuenteConocimiento* creador;
    Cadena<char> razon;
    char letraNormal;
    char letraCifrada;
};
```

Nótese que se reusa otra clase del marco de referencia descrito en el Capítulo 9, la clase plantilla Cadena.

Las suposiciones son tipos de objetos de pizarra porque representan estado que es de interés general para todas las fuentes de conocimiento. Los diversos objetos miembros representan las siguientes propiedades:

- **destino** El objeto de pizarra sobre el cual se hizo la suposición.
- **creador** La fuente de conocimiento que creó la suposición.
- **razon** La razón por la que la fuente de conocimiento hizo la suposición.
- **letraNormal** La letra de texto normal sobre la cual se está haciendo la suposición.
- **letraCifrada** El valor supuesto de la letra de texto normal.

La necesidad de cada una de estas propiedades se deriva en gran medida de la propia naturaleza de una suposición: una fuente de conocimiento particular realiza una suposición sobre una correspondencia texto normal/texto cifrado, y lo hace por una razón determinada (normalmente porque se disparó alguna regla). La necesidad del primer miembro, **destino**, es menos obvia. Se incluye por el problema de la vuelta atrás (*backtracking*). Si hay que revocar una suposición, hay que notificárselo a todos los objetos de pizarra para los cuales se hizo originalmente esa suposición, de forma que puedan a su vez alertar a las fuentes de conocimiento de las que dependen (por medio del mecanismo de dependencia) sobre el cambio de su significado.

A continuación, se tiene la subclase llamada **Asercion**:

```
class Asercion : public Suposicion ...
```

Las clases **Suposicion** y **Asercion** comparten la siguiente operación, entre otras:

- **esRetractable** Un selector: ¿es temporal la correspondencia?

Todos los objetos suposición responden cierto al predicado **esRetractable**, mientras que todos los objetos aserción responden falso. Además, una vez que se ha hecho, una aserción no puede ni volver a establecerse ni retractarse.

La Figura 11.2 proporciona un diagrama de clases que ilustra las colaboraciones de las clases de dependencia y de afirmación. Póngase especial atención en los papeles que desempeña cada abstracción en las diversas asociaciones. Por ejemplo, una **FuenteConocimiento** es el creador de una **Suposicion**, y también es quien hace referencia a una **LetraCifrada**. Puesto que un papel representa una vista diferente que una abstracción presenta al mundo, se esperaría ver un protocolo diferente entre las fuentes de conocimiento y las suposiciones que entre las fuentes de conocimiento y las letras.

Diseño de los objetos de pizarra. Va a completarse el diseño de las clases **Frase**, **Palabra** y **LetraCifrada**, seguidas por la clase **Alfabeto**, haciendo un

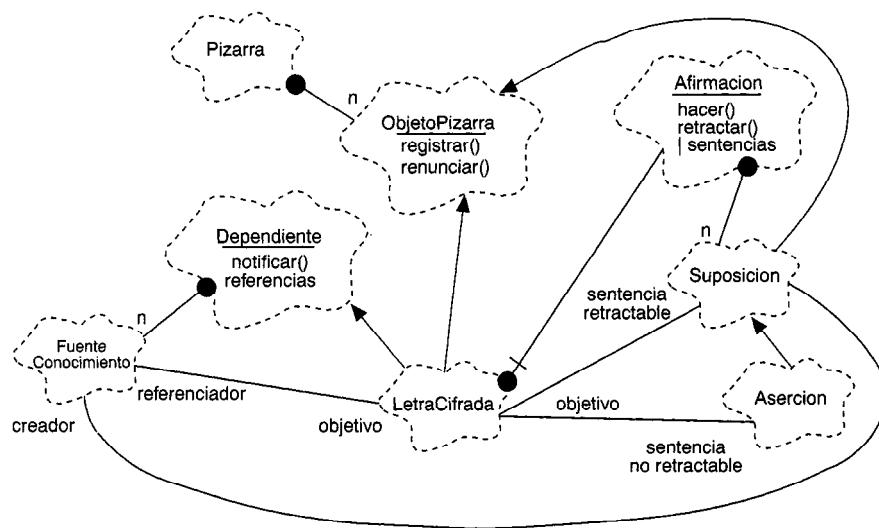


Figura 11.2. Clases de dependencia y de afirmación.

pequeño diseño de clases aisladas. Una frase es bastante simple: es un objeto de pizarra así como un dependiente, y denota una lista de palabras que componen la frase. Así, se puede escribir:

```
class Frase : public ObjetoDePizarra,
    virtual public Dependiente {
public:
    ...
protected:
    Lista<Palabra*> palabras;
};
```

Se hace `virtual` la superclase `Dependiente`, porque se espera que haya otras subclases `Frase` que intenten heredar también de `Dependiente`. Marcando esta relación de herencia como `virtual`, se hace que tales subclases comparten una sola superclase `Dependiente`.

Además de las operaciones `registrar` y `renunciar` definidas por su superclase `ObjetoDePizarra`, más las cuatro operaciones definidas en `Dependiente`, se añaden las siguientes dos operaciones específicas de las frases:

- `valor` Devuelve el valor actual de la frase.
- `estaResuelto` Devuelve `cierto` si existe una aserción para todas las palabras de la frase.

Al comienzo del problema, `valor` devuelve una cadena que representa el cri-

tograma original. Una vez que `estaResuelto` se evalúa como cierto, la operación `valor` puede usarse para recuperar la solución en texto normal. Si el acceso a `valor` antes de que `estaResuelto` es cierto proporcionará soluciones parciales.

Al igual que la clase frase, una palabra es un tipo de objeto de pizarra así como un tipo de dependiente. Además, una palabra denota una lista de letras. Para ayudar a las fuentes de conocimiento que manipulan palabras, se incluye una referencia de una palabra a su frase, así como desde una palabra a la palabra anterior y siguiente en esa frase. Así, se puede escribir lo siguiente:

```
class Palabra : public ObjetoDePizarra,
    virtual public Dependiente {
public:
    ...
    Frase& frase() const;
    Palabra* anterior() const;
    Palabra* siguiente() const;

protected:
    Lista<LetraCifrada*> letras;
};
```

Como se hizo con las operaciones de frase, se definen las siguientes dos operaciones para la clase `Palabra`:

- `valor` Devuelve el valor actual de la palabra.
- `estaResuelto` Devuelve cierto si existe una aserción para todas las letras de la palabra.

Se puede definir a continuación la clase `LetraCifrada`. Una instancia de esta clase es un tipo de objeto de pizarra y un tipo de dependiente. Además de sus comportamientos heredados, cada objeto de letra cifrada tiene un valor (tal como la letra de texto cifrado `H`) junto con una colección de suposiciones y aserciones respecto a la letra de texto normal que le corresponde. Se puede usar la clase `Afirmacion` para recoger estas declaraciones. Así, se puede escribir lo siguiente:

```
class LetraCifrada : public ObjetoPizarra,
    virtual public Dependiente {
public:
    ...
    char valor() const;
    int estaResuelto() const;...
```

```
protected:
```

```
    char letra;
    Afirmacion afirmaciones;

};
```

Nótese que se incluyen los selectores `valor` y `estaResuelto` análogamente al diseño de `Frase` y `Palabra`. También hay que proporcionar eventualmente operaciones para que los clientes de `LetraCifrada` accedan a sus suposiciones y aserciones de forma segura.

Un comentario sobre el objeto miembro `afirmaciones`: se espera que sea una colección de suposiciones y aserciones ordenadas de acuerdo con su momento de creación, de forma que la afirmación más reciente de esta colección representa la suposición o aserción actual. La razón por la que se elige mantener una historia de todas las suposiciones es permitir a las fuentes de conocimiento examinar suposiciones anteriores que se rechazaron, de forma que puedan aprender de errores pasados. Esta decisión influye en las decisiones de diseño sobre la clase `Afirmacion`, a la que se añaden las siguientes operaciones:

- `masReciente` Un selector: devuelve la suposición o aserción más reciente.
- `sentenciaEn` Un selector: devuelve la afirmación *enésima*.

Ahora que se ha refinado su comportamiento, se puede tomar a continuación una decisión de implantación razonable sobre la clase `Afirmacion`. En concreto, se puede incluir el siguiente miembro objeto `protected`:

```
ColeccionOrdenadaNoLimitada<Suposicion*> sentencias;
```

`ColeccionOrdenadaNoLimitada` es otra clase reutilizable del marco de referencia de clases básicas del Capítulo 9.

Considérese a continuación la clase llamada `Alfabeto`. Esta clase representa el alfabeto completo del texto normal y del cifrado, más las correspondencias entre ambos. Esta información es importante porque todas las fuentes de conocimiento pueden usarla para determinar qué correspondencias se han hecho y cuáles quedan por hacer. Por ejemplo, si ya se tiene una aserción de que la letra de texto cifrado `c` es en realidad la letra `M`, entonces un objeto alfabeto registra esta correspondencia de forma que ninguna otra fuente de conocimiento pueda aplicar la letra de texto normal `M`. Por eficiencia, se necesita interrogar sobre la correspondencia de ambas maneras: dada una letra de texto cifrado, devolver su correspondencia en texto normal, y dada una letra de texto normal, devolver su correspondencia en texto cifrado. Se puede definir la clase `Alfabeto` como sigue:

```
class Alfabeto : public ObjetoDePizarra {
```

```

public:

...
char textonormal(char) const;
char textocifrado(char) const;
int estaLigado(char) const;

};

```

Como en la clase `LetraCifrada`, se incluye también un objeto miembro `protected afirmaciones`, y se proporcionan operaciones adecuadas para acceder a su estado.

Ahora se está en disposición de definir la clase `Pizarra`. Esta clase tiene la única responsabilidad de recoger instancias de la clase `ObjetoDePizarra` y sus subclases. Así se puede escribir:

```
class Pizarra : public ColeccionDinamica<ObjetoDePizarra*> ...
```

Se ha decidido heredar en lugar de contener una instancia de la clase `ColeccionDinamica`, porque `Pizarra` pasa la prueba de la herencia: una pizarra es realmente un tipo de colección.

La clase `Pizarra` proporciona operaciones como `anadir` y `eliminar`, que hereda de la clase `Coleccion`. El diseño incluye cinco operaciones específicas de la pizarra.

- `reinicializar` Borrar la pizarra.
- `asercionProblema` Colocar un problema inicial en la pizarra.
- `conectar` Unir la fuente de conocimiento con la pizarra.
- `estaResuelto` Devuelve cierto si la frase está resuelta.
- `recuperarSolucion` Devuelve la frase de texto normal resuelta.

La segunda operación se necesita para crear una dependencia entre una pizarra y sus fuentes de conocimiento.

En la Figura 11.3 se resume el diseño de las clases que colaboran con `Pizarra`. Este diagrama muestra principalmente relaciones de herencia; por simplicidad, omite las relaciones «de uso», como entre una suposición y un objeto de pizarra.

En este diagrama, nótese que se muestra la clase `Pizarra` como instanciando y heredando simultáneamente de la clase plantilla `ColeccionDinamica`. Este diagrama también muestra con claridad por qué fue una buena decisión el introducir la clase `Dependiente` como aditiva. En concreto, `Dependiente` representa un comportamiento que abarca sólo un conjunto parcial de subclases `ObjetoDePizarra`. Se podría haber introducido `Dependiente` como una superclase intermedia, pero haciéndola aditiva en vez de enlazarla en la jerarquía de `ObjetoDePizarra` se incrementan sus oportunidades de reutilización.

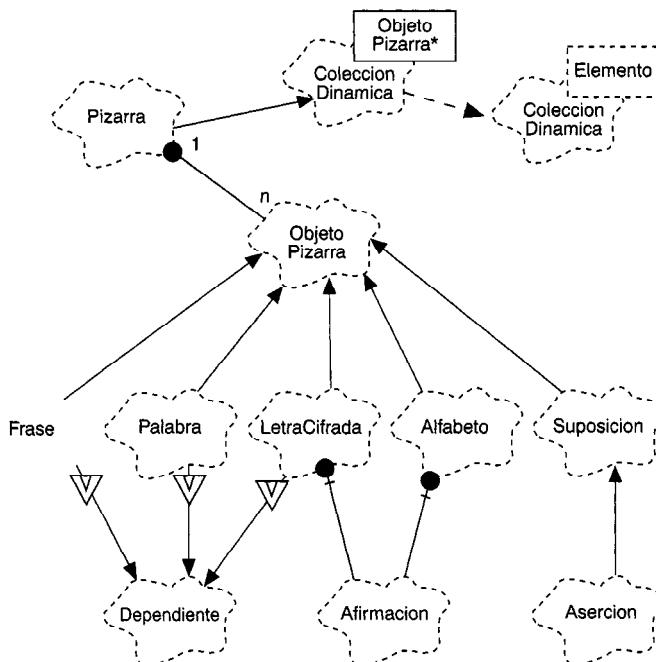


Figura 11.3. Diagrama de las clases pizarra.

Diseño de las fuentes de conocimiento

En una sección anterior, se identificaron trece fuentes de conocimiento relevantes para este problema. Al igual que se hizo para los objetos de pizarra, se puede diseñar una estructura de clases que abarque estas fuentes de conocimiento y eleve por tanto todas las características comunes hacia clases más abstractas.

Diseño de fuentes de conocimiento especializadas. Supóngase por ahora la existencia de una clase abstracta llamada `FuenteConocimiento`, cuyo propósito es muy parecido al de la clase `ObjetoDePizarra`. En vez de tratar cada una de las trece fuentes de conocimiento como subclase directa de esta clase más general, es útil realizar primero un análisis de dominios y ver si existe cualquier agrupamiento de fuentes de conocimiento. En realidad, existen tales grupos: algunas fuentes de conocimiento operan sobre frases enteras, otras sobre palabras enteras, otras sobre cadenas contiguas de letras, y otras sobre letras individuales. Se pueden capturar estas decisiones de diseño escribiendo lo siguiente:

```

class FuenteConocimientoFrase : public FuenteConocimiento ...
class FuenteConocimientoPalabra : public FuenteConocimiento ...
class FuenteConocimientoLetra : public FuenteConocimiento ...
  
```

Para cada una de esas clases abstractas se pueden proporcionar subclases específicas. Por ejemplo, las subclases de la clase abstracta `FuenteConocimientoFrase` incluyen

```
class FuenteConocimientoEstructuraFrage : public
    FuenteConocimientoFrage...
class FuenteConocimientoResuelta : public
    FuenteConocimientoFrage...
```

Análogamente, las subclases de la clase intermedia `FuenteConocimientoPalabra` incluyen

```
class FuenteConocimientoEstructuraPalabra : public
    FuenteConocimientoPalabra...
class FuenteConocimientoPalabraPequena : public
    FuenteConocimientoPalabra...
class FuenteConocimientoEmparejamientoPatrones : public
    FuenteConocimientoPalabra...
```

La última clase requiere algunas explicaciones. Anteriormente, se dijo que el propósito de esta clase era proponer palabras que encajasen en cierto patrón. Se pueden usar símbolos de emparejamiento de patrones por expresiones regulares como los usados por la herramienta *grep* del UNIX:

- Cualquier elemento ?
- Excepto elemento ~
- Elemento de cierre *
- Comienzo de grupo {
- Fin de grupo }

Con estos símbolos se podría dar a una instancia de esta clase el patrón `?E~{ A E I O U }`, pidiendo así que devolviera de su diccionario todas las palabras de tres letras que comiencen con cualquier letra, seguida de una E, y que acabe con cualquier letra que no sea una vocal.

El emparejamiento de patrones es una facilidad generalmente útil, así que no es extraño que la recogida de clases similares nos lleve a las clases de emparejamiento de patrones que se describen como parte de la biblioteca básica del Capítulo 9. Así, se puede delinear la fuente de conocimiento sobre el emparejamiento de patrones como sigue, copiándola de algunas clases existentes:

```
class FuenteConocimientoEmparejamientoPatrones : public
    FuenteConocimientoPalabra {
public:
    ...
protected:
```

```

    static ColeccionLimitada<Palabra*> palabras;
    EmparejamientoPatronesER emparejadorPatrones;
}

```

Todas las instancias de esta clase comparten un diccionario de palabras, y cada instancia tiene su propio agente de emparejamiento de patrones por expresiones regulares.

El comportamiento detallado de esta clase no es importante en este momento del diseño, así que se aplaza la invención del resto de su interfaz e implantación.

Continuando, se pueden declarar las subclases de la clase `Fuente-ConocimientoCadena` como sigue:

```

class FuenteConocimientoPrefijosComunes : public
    FuenteConocimientoCadena ...
class FuenteConocimientoSufijosComunes : public
    FuenteConocimientoCadena ...
class FuenteConocimientoLetraDoble : public
    FuenteConocimientoCadena ...
class FuenteConocimientoCadenaLegal : public
    FuenteConocimientoCadena ...

```

Por último, se introducen las subclases de la clase abstracta `Fuente-ConocimientoLetra`:

```

class FuenteConocimientoSustitucionDirecta : public
    FuenteConocimientoLetra ...
class FuenteConocimientoVocales : public
    FuenteConocimientoLetra ...
class FuenteConocimientoConsonantes : public
    FuenteConocimientoLetra ...
class FuenteConocimientoFrecuenciaLetras : public
    FuenteConocimientoLetra ...

```

Generalización de las fuentes de conocimiento. El análisis sugiere que hay sólo dos operaciones principales que se aplican a todas estas clases especializadas:

- **reiniciar** Reiniciar la fuente de conocimiento.
- **evaluar** Evaluar el estado de la pizarra.

La razón de este interfaz tan simple es que las fuentes de conocimiento son entidades relativamente autónomas: se apunta una hacia un objeto interesante de la pizarra, y se le dice que evalúe sus reglas de acuerdo con el estado global actual de la pizarra. Como parte de la evaluación de sus reglas, una fuente de conocimiento dada podría hacer cualquiera de entre varias cosas:

- Proponer una suposición sobre el cifrado por sustitución.
- Descubrir una contradicción entre suposiciones previas, y hacer que se retracte la suposición errónea.
- Proponer una aserción sobre el cifrado por sustitución.
- Decir al controlador que tiene algún conocimiento interesante que apor tar.

Todas estas son acciones generales que son independientes del tipo específico de fuente de conocimiento. Para generalizar más aún, estas acciones representan el comportamiento de un *motor de inferencias*. Dicho de forma sencilla, un motor de inferencias es un objeto que, dado un conjunto de reglas, evalúa estas reglas ya sea para generar nuevas reglas (encadenamiento hacia delante) o para demostrar alguna hipótesis (encadenamiento hacia atrás). Así, se propone la clase siguiente:

```
class MotorInferencias {
public:
    MotorInferencias(ConjuntoDinamico<Reglas*>);

    ...
};
```

La responsabilidad básica del constructor es crear una instancia de esta clase y poblarla con un conjunto de reglas, que usa entonces para la evaluación.

De hecho esta clase sólo tiene una operación crítica que resulta visible para las fuentes de conocimiento:

- **evaluar** Evaluar las reglas de un motor de inferencias.

Este es entonces el modo en que colaboran las fuentes de conocimiento: cada fuente de conocimiento especializada define sus propias reglas específicas de ese conocimiento, y delega la responsabilidad de evaluar esas reglas en la clase *MotorInferencias*. De forma más precisa, se puede decir que la operación *FuenteConocimiento::evaluar* invoca en última instancia la operación *MotorInferencias::evaluar*, cuyos resultados se usan para llevar a cabo cualquiera de las cuatro acciones que se discutieron anteriormente. En la Figura 11.4 se ilustra un escenario habitual de esta colaboración.

¿Qué es exactamente una regla? Usando un formato al estilo Lisp, se podría componer la regla siguiente para la fuente de conocimiento de los sufijos comunes:

```
((* I ? ?)
 (* I N G)
 (* I E S)
 (* I E D))
```

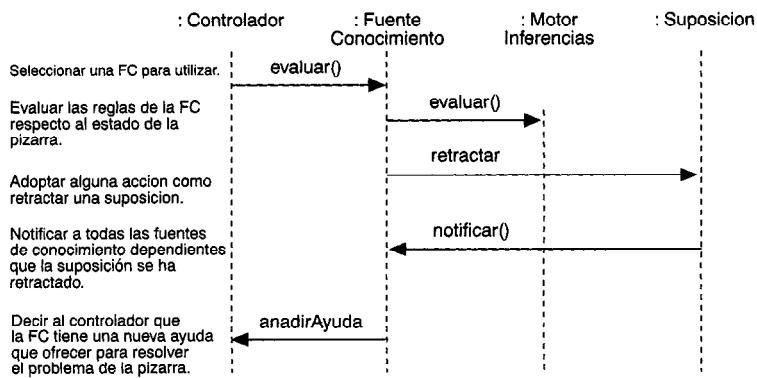


Figura 11.4. Escenario para evaluar reglas de la fuente de conocimiento.

Esta regla significa que, dada una cadena de letras que se empareje con el patrón de expresión regular *I?? (el antecedente), entre los sufijos candidatos se incluyen ING, IES y IED (los consecuentes). En C++, se podría definir una clase que representase una regla como sigue:

```

class Regla {
public:
    ...
    int ligar(Cadena<char>& antecedente, Cadena<char>&
              consecuente);
    int eliminar(Cadena<Char>& antecedente);
    int eliminar(Cadena<Char>& antecedente, Cadena<char>&
                  consecuente);

    int tieneConflicto(const Cadena<char>& antecedente) const;

protected:
    Cadena<char> antecedente;
    Lista<Cadena<char> > consecuentes;
};


```

La semántica que se pretende para estas operaciones sigue a la de sus nombres. Una vez más, se reutilizan algunas de las clases descritas en el Capítulo 9.

En términos de su estructura de clases, se puede decir por tanto que una fuente de conocimiento es un tipo de motor de inferencias. Además, cada fuente de conocimiento debe tener alguna asociación con un objeto de tipo pizarra, porque ahí es donde encuentra los objetos sobre los que opera. Por último, cada

fuente de conocimiento debe tener una asociación con un controlador, con el cual colabora enviando propuestas de soluciones; a su vez, el controlador podría disparar la fuente de conocimiento de vez en cuando.

Se pueden expresar estas decisiones de diseño como sigue:

```
class FuenteConocimiento : public MotorInferencias,
                           public Dependiente {
public:

    FuenteConocimiento(Pizarra*, Controlador*);

    ...

    void reinicializar();
    void evaluar();

protected:

    Pizarra* pizarra;
    Controlador* controlador;
    ColeccionOrdenadaNoLimitada<Suposicion*> suposicionesPasadas;

};
```

Se introduce también el objeto protected suposicionesPasadas, de forma que la fuente de conocimiento pueda llevar cuenta de todas las suposiciones y aser- ciones que ha hecho hasta ahora, con el fin de aprender de sus errores.

Las instancias de la clase Pizarra sirven como un depósito de objetos de pizarra. Por una razón similar, se necesita una clase FuentesConocimiento, que denote la colección completa de fuentes de conocimiento para un problema particular. Así, se puede escribir:

```
class FuentesConocimiento : public
    ColeccionDinamica<FuenteConocimiento*> ...
```

Una de las responsabilidades de esta clase es que cuando se crea una instancia de FuentesConocimiento, se crean también los trece objetos fuente de conocimiento individuales. Se pueden hacer tres operaciones sobre instancias de esta clase:

- | | |
|--|--|
| <ul style="list-style-type: none"> ● reiniciar ● comenzarFuenteConocimiento ● conectar | <p>Reiniciar las fuentes de conoci-
miento.</p> <p>Dar a una fuente de conocimiento
concreta sus condiciones iniciales.</p> <p>Unir la fuente de conocimiento a la
pizarra o al controlador.</p> |
|--|--|

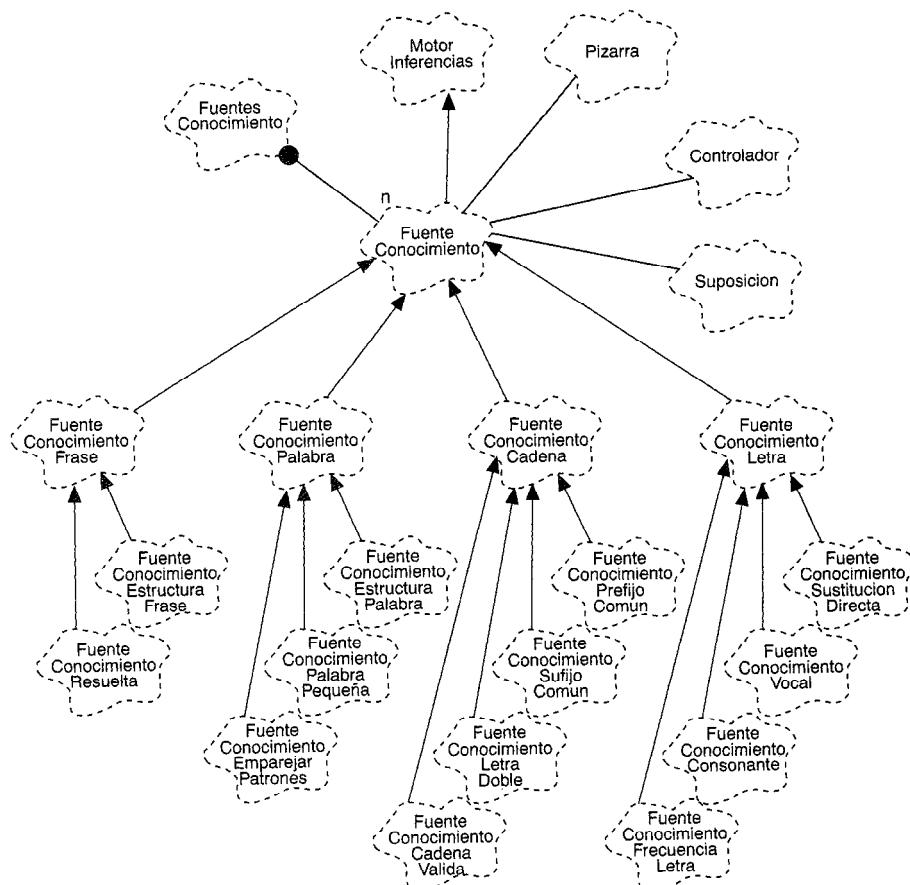


Figura 11.5. Diagrama de clases de las fuentes de conocimiento.

La Figura 11.5 ofrece la estructura de clases de las clases `FuenteConocimiento`, de acuerdo con estas decisiones de diseño.

Diseño del controlador

Considérese por un momento cómo interactúan el controlador y las fuentes de conocimiento individuales. En cada etapa de la solución de un criptograma, una fuente de conocimiento particular podría descubrir que tiene una contribución útil que hacer y, por tanto, hace una sugerencia al controlador. Por el contrario, la fuente de conocimiento podría decidir que su última sugerencia ya no se aplica, y puede eliminarla. Una vez que se ha dado una oportunidad a todas las fuentes de conocimiento, el controlador selecciona la sugerencia más promete-

dora y activa la fuente de conocimiento apropiada invocando su operación `evaluar`.

¿Cómo decide el controlador qué fuente de conocimiento activar? Se pueden idear unas pocas reglas convenientes:

- Una asercción tiene más prioridad que una suposición.
- La fuente de conocimiento resolvedora proporciona las sugerencias más útiles.
- La fuente de conocimiento de emparejamiento de patrones proporciona sugerencias de mayor prioridad que la fuente de conocimiento de estructura de frase.

Un controlador actúa así como un agente responsable de mediar entre las diversas fuentes de conocimiento que operan sobre una pizarra.

El controlador debe tener una asociación con sus fuentes de conocimiento, a las que puede acceder a través de la clase (denominada de forma apropiada) `FuentesConocimiento`. Además, el controlador debe tener como una de sus propiedades una colección de sugerencias, ordenadas de acuerdo con su prioridad. De este modo, el controlador puede seleccionar para activación con facilidad a la fuente de conocimiento que ofrece la pista más interesante.

Ocupándose de un diseño de clases un poco más aislado, se ofrecen las siguientes operaciones para la clase `Controlador`:

• <code>reiniciar</code>	Reiniciar el controlador.
• <code>anadirPista</code>	Añadir una sugerencia de una fuente de conocimiento.
• <code>eliminarPista</code>	Eliminar una sugerencia de una fuente de conocimiento.
• <code>procesarPistaSig</code>	Evaluuar la sugerencia siguiente de prioridad mayor.
• <code>estaResuelto</code>	Un selector: devuelve <code>cierto</code> si el problema está resuelto.
• <code>incapazDeProceder</code>	Un selector: devuelve <code>cierto</code> si las fuentes de conocimiento están atascadas.
• <code>conectar</code>	Unir el controlador con la fuente de conocimiento.

Se pueden capturar estas decisiones de diseño en la declaración siguiente:

```
class Controlador {
public:

    ...
    void reiniciar();
    void conectar(FuenteConocimiento&);
    void anadirPista(FuenteConocimiento&);
    void eliminarPista(FuenteConocimiento&);
```

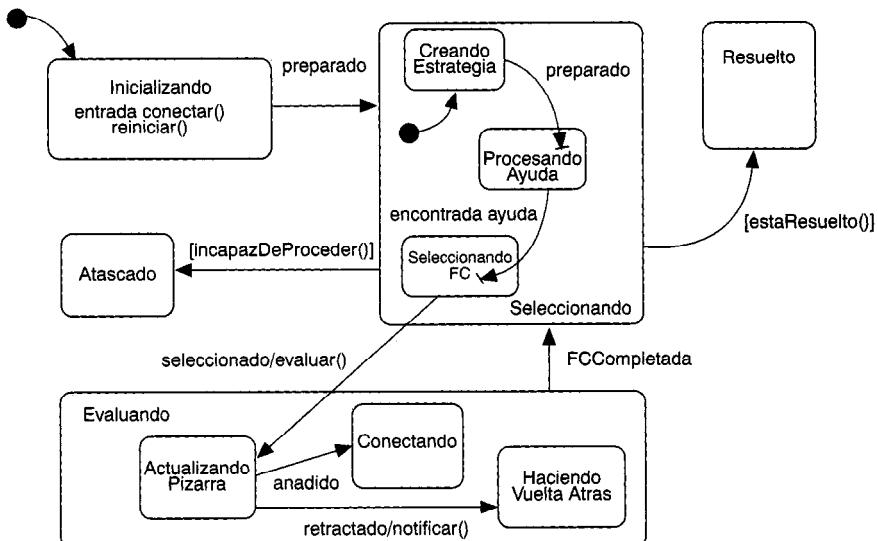


Figura 11.6. Máquina de estados finitos del controlador.

```

    void procesarPistaSig();

    int estaResuelto() const;
    int incapazDeProceder() const;

};

```

El controlador es, en cierto sentido, dirigido por las pistas o sugerencias que recibe de diversas fuentes de conocimiento. Siendo así, las máquinas de estados finitos están indicadas para capturar el comportamiento dinámico de esta clase.

Por ejemplo, considérese el diagrama de transición de estados que aparece en la Figura 11.6. Aquí se ve que un controlador puede estar en uno de cinco estados principales: Inicializando, Seleccionando, Evaluando, Atascado y Resuelto. La actividad más interesante del controlador se da entre los estados Seleccionando y Evaluando. Mientras está seleccionando, el controlador evoluciona de forma natural desde el estado CreandoEstrategia a ProcesandoPista y eventualmente a SelecciónFC. Si se ha seleccionado de hecho una fuente de conocimiento, el controlador pasa al estado Evaluando, dentro del cual está primeramente en ActualizandoPizarra. Pasa a Conectando si se añaden objetos, y a VueltaAtras si se retractan suposiciones, momento en el cual también avisa a todos los dependientes.

El controlador pasa incondicionalmente a Atascado si no puede seguir, y a Resuelto si encuentra un problema de pizarra resuelto.

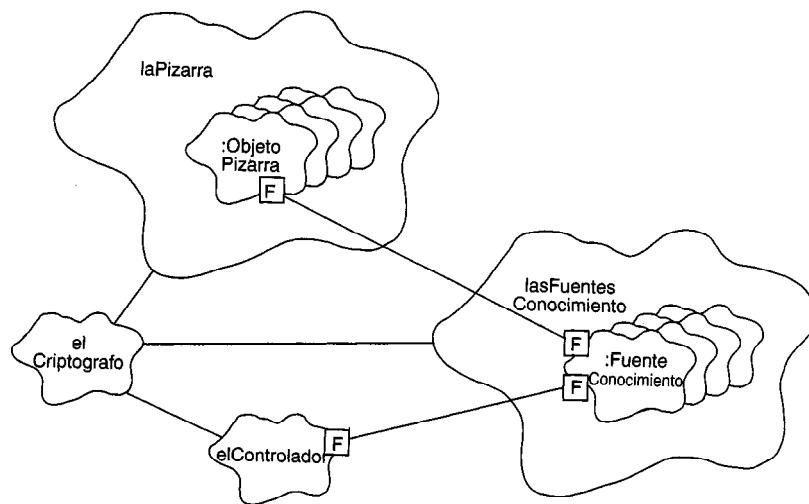


Figura 11.7. Diagrama de objetos del criptoanálisis.

11.3. Evolución

Integración del marco de referencia de pizarra

Ahora que se han definido las abstracciones clave del dominio, se puede continuar poniéndolas juntas para formar una aplicación completa. Se procederá implantando y probando un corte vertical a través de la arquitectura, y completando después el sistema mecanismo a mecanismo.

Integración de los objetos de nivel superior. La Figura 11.7 es un diagrama de objetos que captura el diseño del objeto de nivel superior del sistema, reproduciendo la estructura del marco de referencia de pizarra genérico de la Figura 11.1. En la Figura 11.7, se muestra la contención física de objetos de pizarra por parte de la colección `laPizarra` y de las fuentes de conocimiento por parte de la colección `lasFuentesConocimiento`, usando un estilo taquigráfico idéntico al que se usa para mostrar las clases anidadas.

En este diagrama, se introduce una instancia de una nueva clase que se llama `Criptografo`. El propósito de esta clase es servir como un agregado que abarque la pizarra, las fuentes de conocimiento y el controlador. De este modo, la aplicación podría proporcionar varias instancias de esta clase, y tener por tanto varias pizarras funcionando a la vez.

Se definen dos operaciones principales para esta clase:

- reiniciar Reiniciar la pizarra.

• descifrar

Resolver el criptograma dado.

El comportamiento que se requiere por parte del constructor de esta clase es crear las dependencias entre la pizarra y sus fuentes de conocimiento, así como entre las fuentes de conocimiento y el controlador. El método de reinicialización es similar pues simplemente reinicia estas conexiones y devuelve a la pizarra, las fuentes de conocimiento y el controlador a su estado inicial estable.

Aunque no se mostrarán sus detalles aquí, la seña de identidad de la operación `descifrar` incluye una cadena, mediante la cual se proporciona el texto cifrado que hay que resolver. De este modo, la raíz del programa principal pasa a ser embarazosamente simple, como suele ocurrir en sistemas orientados a objetos bien diseñados:

```
char* resolverProblema(char* textoCifrado)
{
    Criptografo elCriptografo;
    return elCriptografo.descifrar(textoCifrado);
}
```

La implantación de la operación `descifrar` es, como era de esperar, ligeramente más complicada. Básicamente, hay que invocar primero la operación `asercionProblema` para fijar el problema en la pizarra. A continuación, hay que iniciar las fuentes de conocimiento reclamando su atención sobre este nuevo problema. Finalmente, hay que iterar, diciendo al controlador que procese la siguiente pista en cada nueva pasada, ya sea hasta que se resuelva el problema o hasta que todas las fuentes de conocimiento sean incapaces de seguir. Se podría usar un diagrama de interacción o un diagrama de objetos para mostrar este flujo de control, aunque los fragmentos de código C++ funcionan igualmente bien para un algoritmo tan simple:

```
laPizarra.asercionProblema();
lasFuentesConocimiento.reiniciar();
while (!elControlador.estaResuelto() ||
       elControlador.incapazDeProceder())
    elControlador.procesarPistaSig();
if (laPizarra.estaResuelto())
    return laPizarra.recuperarSolucion();
```

Como parte de la evolución, sería más aconsejable completar lo suficiente de los interfaces arquitectónicos relevantes como para poder completar este algoritmo y ejecutarlo. Aunque en este momento tendría una funcionalidad mínima, su implantación como un corte vertical a través de la arquitectura obligaría a validar ciertas decisiones arquitectónicas clave.

Continuando, considérense dos de las operaciones clave usadas en `descifrar`, a saber, `asercionProblema` y `recuperarSolucion`. La operación `asercionProblema` es especialmente interesante, porque debe generar todo un

conjunto de objetos pizarra. En forma de un simple guión, el algoritmo es como sigue:

```

recortar todos los blancos que preceden y siguen a la cadena
volver si la cadena resultante es vacía
crear un objeto frase
añadir la frase a la pizarra
crear un objeto palabra (será la palabra más a la izquierda de la frase)
añadir la palabra a la pizarra
añadir la palabra a la frase
para cada carácter de la cadena, de izquierda a derecha
    si el carácter es un espacio
        hacer la palabra actual palabra previa
        crear un objeto palabra
        añadir la palabra a la pizarra
        añadir la palabra a la frase
    si no
        crear un objeto letra cifrada
        añadir la letra a la pizarra
        añadir la letra a la palabra

```

Como se describió en el Capítulo 6, el propósito del diseño es simplemente proporcionar un plano para la implantación. Este guión ofrece un algoritmo suficientemente detallado, así que no es necesario mostrar su implantación completa en C++.

La operación `recuperarSolucion` es mucho más simple; no hay más que devolver el valor de la frase de la pizarra. La llamada a `recuperarSolucion` antes de que `estaResuelto` se evalúe como cierto producirá soluciones parciales.

Implantación del mecanismo de suposición. En este punto se han implantado los mecanismos que permiten fijar y recuperar valores para objetos de pizarra. El siguiente punto funcional principal lleva el mecanismo de realizar suposiciones sobre objetos de pizarra. Esta es una cuestión especialmente significativa, porque las suposiciones son dinámicas (lo que significa que se crean y destruyen rutinariamente durante el proceso de formar una solución) y su creación o retractación dispara eventos del controlador.

La Figura 11.8 ilustra el escenario principal de cuando una fuente de conocimiento establece una suposición. Como muestra este diagrama, una vez que la fuente de conocimiento crea una suposición, se lo notifica a la pizarra, que a su vez realiza la suposición para su alfabeto y, a continuación, para todos los objetos de pizarra a los que se aplica la suposición. Usando el mecanismo de dependencia, el objeto de pizarra afectado debería notificarlo a su vez a sus fuentes de conocimiento dependientes.

En su implantación más ingenua, el retractarse de una suposición simplemente deshace el trabajo de este mecanismo. Por ejemplo, para retractarse de una suposición sobre una letra cifrada, simplemente se elimina su colección de

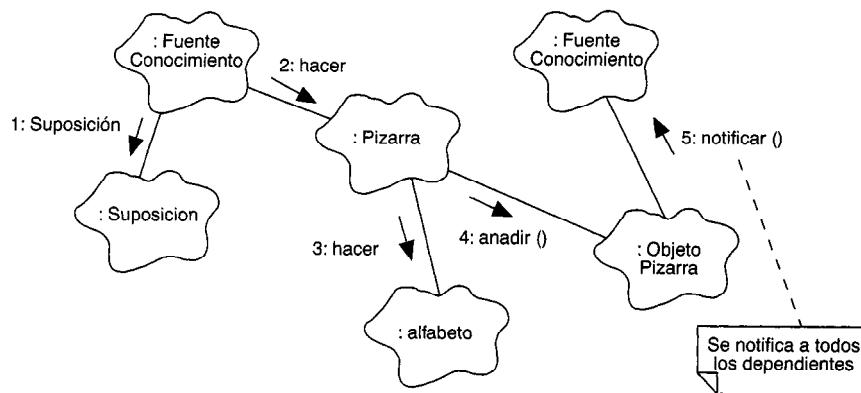


Figura 11.8. Mecanismo de suposición.

suposiciones, llegando hasta la suposición que se está retractando (inclusive). De este modo, se deshacen la suposición dada y todas las que se hicieron basándose en ella.

Es posible un mecanismo más sofisticado. Por ejemplo, supóngase que se hizo una suposición de que cierta letra de una palabra es realmente la letra *i* (suponiendo que se necesita una vocal). Se podría hacer una suposición posterior de que cierta letra doble de una palabra es *NN* (suponiendo que se necesitan consonantes). Si se encuentra entonces que hay que retractarse de la primera suposición, es probable que no haya que retractarse de la segunda. Este enfoque exige que se añada un nuevo comportamiento a la clase *Suposicion*, de forma que pueda llevar cuenta de qué suposiciones son dependientes de otras. Se puede aplazar razonablemente esta mejora hasta mucho más tarde en la evolución de este sistema, porque el añadido de este comportamiento no tiene impacto arquitectónico.

Añadido de nuevas fuentes de conocimiento

Ahora que se tienen colocadas las abstracciones clave para el marco de referencia de pizarra, y una vez que funcionan los mecanismos para establecer suposiciones y retractarse, el paso siguiente es implantar la clase *MotorInferencias*, ya que todas las fuentes de conocimiento dependen de ella. Como se mencionó anteriormente, esta clase sólo tiene una operación verdaderamente interesante, a saber, *evaluarReglas*. No se mostrarán sus detalles aquí, porque este método particular no revela nuevas cuestiones de diseño importantes.

Una vez que se tiene confianza en el correcto funcionamiento del motor de inferencias, se puede añadir incrementalmente cada fuente de conocimiento. Se pone énfasis en el uso de un proceso incremental por dos razones:

- Para una fuente de conocimiento determinada, no está claro qué reglas

son verdaderamente importantes hasta que se las aplica a problemas reales.

- La depuración de la base de conocimiento es mucho más fácil si se implantan y prueban conjuntos relacionados de reglas más pequeños, en vez de intentar probarlos todos a la vez.

Fundamentalmente, la implantación de cada fuente de conocimiento es en gran medida un problema de ingeniería del conocimiento. Para una fuente de conocimiento determinada, hay que dialogar con un experto (quizás un criptólogo) para decidir qué reglas son significativas. A medida que se prueba cada fuente de conocimiento, el análisis puede revelar que ciertas reglas son inútiles, otras son demasiado específicas o demasiado generales, y puede que falten algunas otras. Se puede elegir entonces alterar las reglas de una fuente de conocimiento determinada o incluso añadir nuevas fuentes de conocimiento.

A medida que se implanta cada fuente de conocimiento, se puede descubrir la existencia de reglas comunes así como de un comportamiento común. Por ejemplo, se podría descubrir que *FuenteConocimientoEstructuraPalabra* y *Fuente Conocimiento Estructura Frase* comparten un comportamiento común, en el sentido de que ambos deben saber cómo evaluar reglas respecto al orden admisible de ciertas construcciones. La primera fuente de conocimiento se interesa por la disposición de las letras; la segunda se interesa por la disposición de las palabras. En ambos casos, el procesamiento es el mismo; así, es razonable alterar la estructura de clases de las fuentes de conocimiento desarrollando una nueva clase aditiva, llamada *FuenteConocimientoEstructura*, en la que se ubica este comportamiento común.

Esta nueva jerarquía de clases de las fuentes de conocimiento subraya el hecho de que la evaluación de un conjunto de reglas es dependiente tanto del tipo de fuente de conocimiento como del tipo de objeto de pizarra. Por ejemplo, dada una fuente de conocimiento específica, podría usar encadenamiento hacia adelante para cierto tipo de objeto de pizarra, y encadenamiento hacia atrás para otro. Además, dado un objeto de pizarra específico, la forma en que se evalúa dependerá de sobre qué fuente de conocimiento se aplica.

11.4. Mantenimiento

Añadido de nueva funcionalidad

En esta sección se considera una mejora a la funcionalidad del sistema de criptoanálisis y se observa cómo el diseño resiste el cambio.

En cualquier sistema inteligente es importante saber cuál es la respuesta final a un problema, pero con frecuencia es igual de importante saber cómo llegó el sistema a esa solución. Así, se desea que la aplicación haga examen de conciencia: debería llevar cuenta de cuándo se activaron las fuentes de conoci-

miento, qué suposiciones se hicieron y por qué, y así sucesivamente, de forma que después se le pueda interrogar, por ejemplo, sobre el porqué de una suposición, cómo llegó a otra suposición, y cuándo se activó determinada fuente de conocimiento.

Para añadir esta nueva funcionalidad, hay que hacer dos cosas. Primero, hay que idear un mecanismo para seguir la pista al trabajo que realizan el controlador y cada una de las fuentes de conocimiento, y segundo, hay que modificar las operaciones apropiadas para que registren esta información. Básicamente, el diseño requiere que las fuentes de conocimiento y el controlador registren lo que han hecho en algún depósito central.

Comencemos por inventar las clases que se necesitan para soportar este mecanismo. Primero, se podría definir la clase `Accion`, que sirve para registrar lo que ha hecho una fuente de conocimiento o controlador concretos:

```
class Accion {
public:

    Accion(FuenteConocimiento* quien, ObjetoDePizarra* que, char*
           porque);
    Accion(Controlador* quien, FuenteConocimiento* que, char*
           porque);

    ...
};
```

Por ejemplo, si el controlador seleccionase una fuente de conocimiento particular para la activación, crearía una instancia de esta clase, fijaría el argumento `quien` como él mismo, fijaría el argumento `que` como la fuente de conocimiento, y daría al argumento `porque` el valor de alguna explicación (quizás incluyendo la prioridad actual de la sugerencia).

Ya está la primera parte de la tarea, y la segunda parte es casi igual de fácil. Considérese por un momento dónde tienen lugar eventos importantes en la aplicación. Puede verse que hay cinco tipos principales de operaciones afectadas:

- Métodos que establecen una suposición.
- Métodos que retractan una suposición.
- Métodos que activan una fuente de conocimiento.
- Métodos que hacen que se evalúen reglas.
- Métodos que registran sugerencias de una fuente de conocimiento.

En realidad, estos eventos están restringidos, en gran medida, a dos lugares de la arquitectura: como parte de la máquina de estados finitos del controlador, y como parte del mecanismo de suposición. La tarea de mantenimiento, por tanto, implica tocar todos los métodos que representan un papel en estos dos lugares, una tarea tediosa pero que tampoco tiene mayor misterio. En realidad, el des-

cubrimiento más importante es que el añadir este nuevo comportamiento no requiere cambios arquitectónicos significativos.

Para completar el trabajo aquí, también hay que implantar una clase que pueda responder a preguntas del usuario sobre quién, qué, cuándo y por qué. El diseño de tal objeto no es terriblemente difícil, porque toda la información que necesita puede encontrarse como parte del estado de la clase `acciones`.

Cambios en los requisitos

Una vez que se tiene en situación una implantación estable, pueden incorporarse muchos requisitos nuevos con cambios mínimos en el diseño. Considerense tres tipos de nuevos requisitos:

- La capacidad de descifrar lenguajes distintos del inglés.
- La capacidad de descifrar usando cifrados por transposición así como cifrados por sustitución simple.
- La capacidad de aprender de la experiencia.

El primer cambio es evidentemente sencillo, porque el hecho de que la aplicación use el inglés es bastante irrelevante de cara al diseño. Suponiendo que se use el mismo conjunto de caracteres, es más que nada cuestión de cambiar las reglas asociadas con cada fuente de conocimiento. En realidad, el cambio del conjunto de caracteres no es tampoco difícil, porque incluso la clase `alfabeto` no depende de qué caracteres manipula.

El segundo cambio es mucho más difícil, pero sigue siendo posible en el contexto del marco de referencia de pizarra. Básicamente, el enfoque es añadir nuevas fuentes de conocimiento que incorporen información sobre cifrados por transposición. Una vez más, este cambio no altera ninguna abstracción o mecanismo clave existente en el diseño; en vez de eso, conlleva la adición de nuevas clases que usan facilidades existentes, como la clase `MotorInferencias` y el mecanismo de suposición.

El tercer cambio es el más difícil de todos, principalmente porque el aprendizaje automático está al borde del conocimiento que se tiene sobre inteligencia artificial. Como una aproximación, cuando el controlador descubre que ya no puede seguir, podría pedir al usuario una sugerencia o pista. Registrando esta pista, junto con las acciones que llevaron al atasco del sistema, la aplicación de pizarra puede evitar un problema similar en el futuro. Se puede incorporar este mecanismo simplista de aprendizaje sin alterar demasiado ninguna de las clases existentes; al igual que los otros cambios, éste puede construirse sobre facilidades existentes.

Lecturas recomendadas

En el contexto de los patrones arquitectónicos, Shaw [A 1991] trata los marcos de referencia de pizarra así como otros tipos de marcos de referencia de aplicaciones.

Englemore y Morgan [C 1988] ofrecen un amplio tratamiento de los sistemas de pizarra, incluyendo su evolución, teoría, diseño y aplicación. Entre otros temas, hay descripciones de dos sistemas de pizarra orientados a objetos, BB1 de Stanford y BLOB, desarrollado por el Ministerio de Defensa Británico. Otras fuentes útiles de información respecto a los sistemas de pizarra pueden encontrarse en Hayes-Roth [J 1985] y Nii [J 1986].

Pueden encontrarse discusiones detalladas que conciernen al encadenamiento hacia adelante y hacia atrás en sistemas basados en reglas en Barr y Feigenbaum [J 1981]; Brachman y Levesque [J 1985]; Hayes-Roth, Waterman y Lenat [J 1983]; y Winston y Horn [G 1989].

Meyer y Matyas [I 1982] cubren las fortalezas y debilidades de varios tipos de cifrados, junto con aproximaciones algorítmicas para romperlos.

Notas bibliográficas

- [1] Erman, L., Lark, J., and Hayes-Roth, F. December 1988. ABE: An Environment for Engineering Intelligent Systems. *IEEE Transactions on Software Engineering* vol. 14(2), p. 1758.
- [2] Shaw, M. 1991. *Heterogeneous Design Idioms for Software Architecture*. Pittsburgh, Pennsylvania: Carnegie Mellon University.
- [3] Meyer, C. and Matyas. 1982. *Cryptography*. New York, NY: John Wiley and Sons, p. 1.
- [4] Nii, P. Summer 1986. Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Magazine* vol. 7(2), p. 46.
- [5] Englemore, R. and Morgan, T. 1988. *Blackboard System*. Wokingham, England: Addison-Wesley, p. 16.
- [6] Ibid., p. 19.
- [7] Ibid., p. 6.
- [8] Ibid., p. 12.
- [9] Nii. *Blackboard Systems*, p. 43.
- [10] Englemore and Morgan. *Blackboard Systems*, p. 11.



Dirección y control: Gestión de tráfico

La economía del desarrollo del software ha progresado de tal forma que ahora es factible la automatización de muchos más tipos de aplicaciones que nunca, desde microcomputadores empotrados que controlan motores de automóviles hasta herramientas que eliminan gran parte del trabajo monótono asociado con la producción de una película animada, pasando por sistemas que gestionan la distribución de servicios interactivos de vídeo a millones de consumidores. La característica distintiva de todos estos sistemas grandes es que son extremadamente complejos. El construir sistemas de forma que su implantación sea pequeña es ciertamente una tarea honorable, pero la realidad nos dice que ciertos problemas grandes demandan implantaciones grandes. Para algunas aplicaciones masivas no es extraño encontrar organizaciones de desarrollo de software que emplean varios cientos de programadores, que deben colaborar para producir un millón o más de líneas de código frente a un conjunto de requisitos que, de seguro, no serán estables durante el desarrollo. Tales proyectos rara vez conllevan el desarrollo de programas únicos; con más frecuencia abarcan programas múltiples y cooperativos que deben ejecutarse a través de un sistema final distribuido que consta de muchos computadores conectados entre sí de diversas formas. Para reducir el riesgo de desarrollo, tales proyectos suelen implicar una organización central que es responsable de la arquitectura y la integración de los sistemas; el trabajo que queda se subcontrata a otras compañías. Así, el equipo de desarrollo en su conjunto nunca se reúne en uno sólo; suele estar distribuido en el espacio y, puesto que el intercambio de personal es habitual en grandes proyectos, en el tiempo.

Los desarrolladores que están contentos escribiendo herramientas pequeñas, autosuficientes, monousuario y basadas en ventanas pueden sentir vértigo ante los problemas de construir aplicaciones masivas —tanto que consideran una locura incluso el intentarlo. Sin embargo, la realidad del mundo científico y de

los negocios es tal que hay que construir sistemas de software complejos. En realidad, en ciertos casos, es una locura no intentarlo. Imagínese el uso de un sistema manual para controlar el tráfico aéreo alrededor de un centro metropolitano importante o para gestionar el sistema de apoyo a la vida en una nave espacial tripulada o las actividades de contabilidad en un banco multinacional. La automatización con éxito de tales sistemas no sólo soluciona los propios problemas reales que se manejan, sino que también conduce a una serie de beneficios tangibles e intangibles, como costes operacionales más bajos, mayor seguridad e incremento de la funcionalidad. Por supuesto, la palabra operativa aquí es *éxito*. Construir sistemas complejos es un trabajo evidentemente duro, y requiere la aplicación de las mejores prácticas de ingeniería que se conocen, junto con la perspicacia creativa de algunos pocos grandes diseñadores.

Este capítulo afronta tal problema, para demostrar que la notación y el proceso del desarrollo orientado a objetos soportan el cambio de escala para la programación a escalas muy grandes.

12.1. Análisis

Definición de los límites del problema

Para la mayoría de la gente que vive en los Estados Unidos, los trenes son un artefacto del pasado remoto; en Europa y muchas partes de Oriente, la situación es completamente opuesta. Al contrario que en los Estados Unidos, por ejemplo, Europa tiene pocas autopistas nacionales e internacionales, y los precios de la gasolina y los automóviles son comparativamente muy altos. Así, los trenes son una parte esencial de la red de transportes del continente; decenas de miles de kilómetros de vías transportan personas y mercancías a diario, tanto dentro de las ciudades como de lado a lado de las naciones. En justicia, los trenes aún proporcionan un medio importante y económico de transporte de mercancías en los Estados Unidos. Además, a medida que los centros metropolitanos principales crecen y se atestan, el transporte ferroviario ligero se ve cada vez más como una opción atractiva para reducir la congestión y enfrentarse a los problemas de polución por motores de combustión interna.

A pesar de todo, los trenes son un negocio y por tanto deben proporcionar beneficios. Las compañías ferroviarias deben equilibrar delicadamente las demandas de economía y seguridad y las presiones para incrementar el tráfico respecto a una planificación de horarios eficiente y fiable. Estas necesidades contradictorias sugieren una solución automatizada para la gestión del tráfico ferroviario, que incluya la informatización de itinerarios y la monitorización de todos los elementos del sistema ferroviario.

Tales sistemas automatizados y semiautomatizados existen en la actualidad en Suecia, Gran Bretaña, Alemania Occidental, Francia y Japón [1]. Un sis-

Requisitos del sistema de gestión de tráfico

El sistema de gestión de tráfico tiene dos funciones principales: dirección de trenes y monitorización de los sistemas ferroviarios. Las funciones relacionadas incluyen planificar el tráfico, seguir la posición de los trenes, monitorizar el tráfico, evitar colisiones, predicción de fallos y registro del mantenimiento. La Figura 12.1 proporciona un diagrama de bloques de los elementos principales del sistema de gestión de tráfico[2].

El sistema de análisis e información de locomotoras incluye varios sensores discretos y analógicos para monitorizar elementos como temperatura del aceite, presión del aceite, cantidad de combustible, voltios y amperios del alternador, posición del acelerador, rpm del motor, temperatura del agua y fuerza de tracción. Estos valores se presentan al maquinista mediante el sistema de visualización de a bordo, y también a los controladores y personal de mantenimiento que están en otros lugares de la red. Se registran condiciones de alerta o alarma siempre que ciertos valores de sensor se salgan del rango normal. Se mantiene un registro de los valores de los sensores para apoyar el mantenimiento y la gestión del combustible.

El sistema de gestión de energía aconseja en tiempo real al maquinista sobre las posiciones más eficientes de acelerador y frenos. Las entradas a este sistema incluyen el perfil y pendiente del trazado, límites de velocidad, horarios, carga del tren y potencia disponible, a partir de las cuales el sistema puede determinar las posiciones de acelerador y frenos eficientes de cara al combustible que son consistentes con los intereses de horarios y seguridad. Las posiciones de acelerador y frenos sugeridas, la pendiente y perfil del trazado y la posición y velocidad del tren pueden verse en el sistema de visualización de a bordo.

El sistema de visualización de a bordo proporciona el interfaz hombre/máquina para el maquinista. Está disponible para visualización de información del sistema de análisis e informes de locomotoras, del sistema de gestión de energía y de la unidad de gestión de datos. El maquinista puede seleccionar distintas pantallas mediante software.

La unidad de gestión de datos sirve como nodo de comunicaciones entre todos los sistemas de a bordo y el resto de la red, a la que están conectados todos los trenes, controladores y otros usuarios.

El seguimiento de la posición del tren se consigue mediante dos dispositivos de la red: *transponders** de posición y el sistema Navstar de posicionamiento global (GPS). El sistema de análisis e información de locomotoras puede determinar la posición aproximada del tren por estimación directa, contando las vueltas de las ruedas. Esta información se completa con información de los transponders de posición, situados a cada kilómetro de la vía y en empalmes críticos de las mismas. Estos *transponders* transmiten su identidad a los trenes que pasan mediante sus unidades de gestión de datos, a partir de las cuales se puede determinar una localización más exacta de los trenes. Estos también pueden estar equipados con re-

* Se ha dejado el término original *transponder*, que se puede definir como: un emisor-receptor de radio o radar que se activa para transmitir cuando recibe una señal predeterminada. En inglés es abreviatura de *tran/smitter* + *(re)sponder*. (N. del T.)

ceptores GPS, que permiten determinar la posición de los trenes con un metro de margen de error.

Se coloca una unidad de interfaz a pie de vía dondequiera que haya algún dispositivo controlable (como una aguja) o un sensor (como un sensor de infrarrojos para detectar excesos de temperatura en los cojinetes de las ruedas). Cada unidad

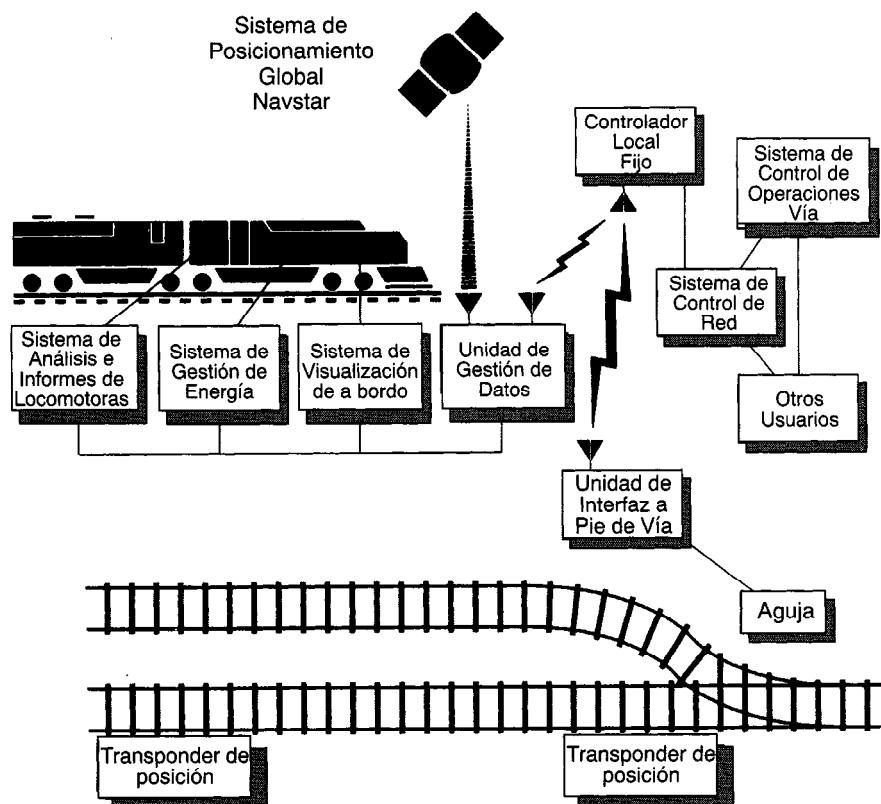


Figura 12.1. Sistema de gestión de tráfico.

de interfaz a pie de vía puede recibir comandos de un controlador local fijo (por ejemplo, para activar o desactivar una señal). Los dispositivos se pueden ignorar mediante controles manuales locales. Cada unidad también puede informar de su estado actual. Hay controladores de terminal fijos a lo largo de la vía, a intervalos lo bastante cortos como para que todos los trenes estén siempre dentro del rango de, al menos, un terminal.

Cada controlador de terminal fijo envía su información a un sistema de control de la red común. Las conexiones entre el sistema de control de la red y cada controlador de terminal fijo puede hacerse mediante enlaces de microondas, cables subterráneos o cables de fibra óptica, según la lejanía de cada controlador de terminal fijo. El sistema de control de la red monitoriza el estado de salud de toda la red y puede dirigir automáticamente información de diversas maneras si el equipo fallase.

El sistema de control de la red se conecta en última instancia a uno o más centros de control, entre los que se cuentan el sistema de control de operaciones ferroviarias y otros usuarios. En el sistema de control de operaciones ferroviarias, los controladores pueden establecer itinerarios de los trenes y seguir el progreso de los trenes individuales. Los controladores individuales supervisan territorios distintos; la consola de cada controlador puede configurarse para controlar uno o más territorios. Los itinerarios de los trenes incluyen instrucciones para cambiarlos automáticamente de una pista a otra, fijar restricciones de velocidad, enviar o recibir vagones y permitir o denegar el acceso del tren a un tramo de vía concreto. Los controladores pueden anotar la posición de las obras a lo largo de la ruta de los trenes para que el maquinista lo vea. Los trenes pueden detenerse desde el sistema de control de operaciones ferroviarias (manualmente por parte de los controladores o automáticamente) cuando se detectan condiciones peligrosas (como un tren sin control, fallos de las vías o una situación de colisión potencial). Los controladores también pueden reclamar cualquier información disponible para los maquinistas individuales, así como enviar autorizaciones de salida, configuraciones de los dispositivos a pie de vía o revisiones de planes.

El trazado de las vías y el equipamiento a pie de vía pueden cambiar con el tiempo. El número de trenes y sus itinerarios pueden cambiar a diario. El sistema debe estar diseñado para permitir la incorporación de nueva tecnología de sensores, redes y procesadores.

tema parecido, llamado el Advanced Train Control System, se ha estado desarrollando en Canadá y los Estados Unidos, con la participación de Amtrak, Burlington, la Canadian National Railway Company, CP Rail, CSX Transportation, la Norfolk and Western Railway Company, la Southern Railway Company y la Union Pacific. La motivación para cada uno de estos sistemas es en gran parte económica y social: los objetivos son: menores costes operativos y utilización más eficiente de los recursos, con un incremento de la seguridad como efecto secundario.

El recuadro proporciona los requisitos básicos para un sistema de gestión de trenes. Obviamente, es una declaración de requisitos muy simplificada. En la práctica, los requisitos detallados para una aplicación tan grande como ésta sólo aparecen después de que se ha demostrado la viabilidad de una solución automatizada, y eso después de muchos cientos de personas/mes de análisis con la participación de numerosos expertos del dominio y los clientes y usuarios eventuales del sistema. En última instancia, los requisitos para un sistema grande pueden abarcar miles de páginas de documentación, especificando no sólo el comportamiento general del sistema, sino también detalles intrincados como las distribuciones de pantalla que se usarán para la interacción hombre/máquina.

Incluso partiendo de estos requisitos del sistema tan resumidos, se pueden hacer dos observaciones sobre el proceso de desarrollo de un sistema de gestión de tráfico:

- La arquitectura debe poder evolucionar en el tiempo.

- La implantación debe basarse cuanto sea posible en estándares ya existentes.

Nuestra experiencia en el desarrollo de sistemas grandes ha sido que una declaración inicial de requisitos nunca está completa, frecuentemente es vaga, y siempre se contradice a sí misma. Por estas razones, hay que interesarse uno mismo deliberadamente por la gestión de la incertidumbre durante el desarrollo, y por tanto se sugiere firmemente que se permita al desarrollo de un sistema tal que pueda evolucionar en el tiempo de forma incremental e iterativa. Como se apuntó en el Capítulo 7, el propio proceso de desarrollo da a los usuarios y a los desarrolladores un conocimiento más depurado sobre qué requisitos son verdaderamente importantes, mucho mejor que cualquier ejercicio sobre el papel escribiendo documentos de requisitos en ausencia de una implantación o prototipo existente. Además, puesto que el desarrollo de software para un sistema grande puede llevar varios años, los requisitos del software deben poder cambiar para aprovechar la tecnología del hardware, rápidamente cambiante¹. Es innegablemente futil idear una elegante arquitectura software dirigida a tecnología hardware con garantías de que será obsoleta para cuando se instale el sistema. Ésta es la razón por la que se aconseja que, sean los que sean los mecanismos que se inventan como parte de la arquitectura software, habría que basarse en estándares existentes para comunicaciones, gráficos, redes y sensores. Para sistemas verdaderamente nuevos, a veces es necesario inaugurar una nueva tecnología hardware o software. Esto añade riesgo a un proyecto grande, sin embargo, el cual ya suele conllevar de por sí un alto riesgo. El desarrollo de software queda claramente como la tecnología de mayor riesgo en el despliegue de cualquier aplicación automatizada de gran tamaño, y el objetivo es limitar este riesgo a un nivel manejable, no incrementarlo.

Obviamente, no se puede llevar a cabo un análisis o diseño completos de este problema en un solo capítulo, ni siquiera en un solo libro. Ya que lo que se intenta aquí es explorar cómo aumenta de escala nuestra notación y proceso, nos centraremos en el problema de construir una arquitectura flexible para este dominio.

Requisitos del sistema versus requisitos software

Los proyectos grandes como éste suelen estar organizados en torno a algún equipo pequeño y localizado, responsable de establecer la arquitectura general del sistema, con el verdadero equipo de desarrollo subcontratado a otras com-

¹ De hecho, para muchos de tales sistemas de esta complejidad es habitual tener que tratar con muchos tipos de computadores diferentes. Tener una arquitectura estable y bien pensada mitiga gran parte del riesgo que supone cambiar hardware a mitad del desarrollo, algo que sucede demasiado a menudo como reflejo de la rápida evolución cambiante del hardware. Los productos hardware vienen y van y, por eso, es importante administrar la frontera hardware/software de un sistema de manera que puedan introducirse nuevos productos que reduzcan el coste del sistema o mejoren su eficacia mientras, al mismo tiempo, se protege la integridad de la arquitectura del sistema.

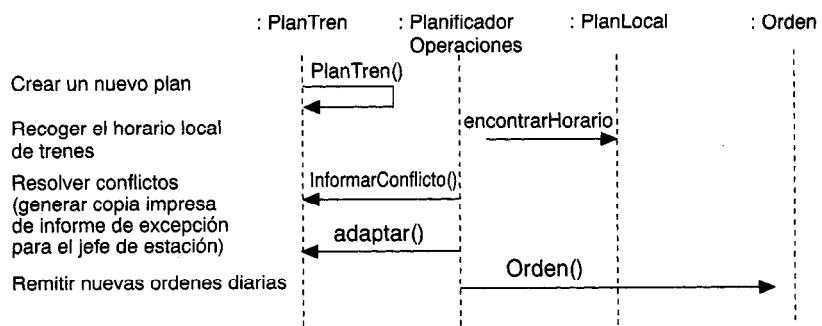


Figura 12.2. Escenario para proceso de órdenes diarias de trenes.

pañías o a diferentes equipos de la misma compañía. Incluso durante el análisis, los arquitectos del sistema suelen tener en mente algún modelo conceptual que divide los elementos hardware y software de la implantación. Se puede argumentar que esto es diseño, no análisis, pero a esto se contestaría que hay que empezar restringiendo el espacio del diseño en algún punto. En realidad, es difícil averiguar si el diagrama de bloques de la Figura 12.1 representa requisitos del sistema o un diseño del mismo. Dejando de lado esta cuestión, el diagrama de bloques sugiere claramente que la arquitectura del sistema en esta etapa del desarrollo es principalmente orientada a objetos. Por ejemplo, muestra objetos complejos como el sistema de gestión de energía y el sistema de control de operaciones ferroviarias, cada uno de los cuales realiza una función principal del sistema. Esto es lo que se dijo en el Capítulo 4: en grandes sistemas, los objetos a niveles más altos de abstracción tienden a agruparse en las líneas de funciones principales del sistema. La forma en que se identifican y refinan estos objetos durante el análisis es ligeramente diferente de la forma en que se hace durante el diseño.

Una vez que se tiene una arquitectura de juguete a nivel de diagrama de bloques como el de la Figura 12.1, se puede comenzar el análisis trabajando con expertos del dominio para articular los escenarios principales que cubren el comportamiento que se desea del sistema, como se describió en el Capítulo 6. Para mayor detalle, se podrían usar diagramas de interacción, diagramas de objetos, guiones simples o prototipos para ilustrar el comportamiento que se espera del sistema. Por ejemplo, en la Figura 12.2, se ofrece un diagrama de interacción que captura un escenario simple para procesar las órdenes diarias de un tren. En este nivel del análisis, es suficiente capturar sólo los eventos e interacciones principales que pueden tener lugar para llevar a cabo cada comportamiento. Los detalles característicos de las operaciones y la representación de las asociaciones son cuestiones tácticas que deberían aplazarse hasta fases posteriores del diseño.

Para un sistema de esta magnitud, no sería extraño identificar unos pocos

de cientos de escenarios principales². Como se sugirió también en el Capítulo 6, aquí se aplica la regla del 80 %: es suficiente capturar el 80 % de los escenarios de un sistema antes de pasar al diseño arquitectónico. Intentar «finalizar» el análisis antes de seguir es futil y engañoso.

Eventualmente, hay que traducir estos requisitos del sistema a requisitos para los segmentos hardware y software del mismo, de modo que diferentes organizaciones, cada una con habilidades diferentes, puedan proceder en paralelo para atacar su parte concreta del problema (pero siempre con algún grupo central promoviendo y protegiendo la visión arquitectónica del sistema). El hacer estos equilibrios de hardware y software es una tarea difícil, particularmente si las organizaciones de hardware y software están débilmente acopladas y, especialmente, si son partes de compañías completamente diferentes. A veces es intuitivamente obvio que debería emplearse cierto hardware. Por ejemplo, se podrían usar modernos terminales o estaciones de trabajo para el sistema de visualización de a bordo y para las visualizaciones en los centros de control de operaciones ferroviarias. Análogamente, por ejemplo, puede ser obvio que el software es el vehículo de implantación correcto para describir los itinerarios de los trenes. Las decisiones sobre qué plataforma utilizar para otras cosas, ya sean implantaciones hardware o software, dependen tanto de las preferencias personales de los arquitectos del sistema como de cualquier otro factor. Se podría lanzar hardware especial contra los problemas en los que sean críticas las necesidades de eficiencia, o usar software si es más importante la flexibilidad.

Para los propósitos de este problema, se supone que se ha elegido una arquitectura hardware inicial por parte de los arquitectos del sistema. Esta decisión no tiene por qué considerarse irreversible, pero al menos da un punto de partida en términos de dónde ubicar los requisitos software. A medida que se avanza en el análisis y luego en el diseño, se necesita la libertad de hacer concesiones entre hardware y software: se podría decidir después que se necesita hardware adicional para satisfacer algún requisito, o que pueden realizarse mejor ciertas funciones mediante software que mediante hardware.

La Figura 12.3 ilustra el hardware de destino para el sistema de gestión de tráfico, usando la notación para los diagramas de procesos. Esta arquitectura de procesos reproduce el diagrama de bloques del sistema en la Figura 12.1. En concreto, hay un computador a bordo de cada tren, que abarca el sistema de análisis e información de locomotoras, el sistema de gestión de energía, el sistema de visualización de abordo y la unidad de gestión de datos. Se espera que algunos de los dispositivos de a bordo, como la pantalla, sean inteligentes, pero se supone que esos dispositivos no son necesariamente programables. Continuando, cada *transponder* de posición está conectado a un transmisor, a través del cual pueden enviarse mensajes a los trenes que pasan; no hay asociado ningún computador con un *transponder* de posición. Por otra parte, cada colección de dispositivos a pie de vía (cada uno de los cuales abarca una unidad de inter-

²Se han encontrado proyectos software cuyos productos del análisis consumieron por sí solos más de 8.000 páginas de documentación, un signo de analistas con exceso de celo. Los proyectos que empiezan de esta manera no suelen tener éxito.

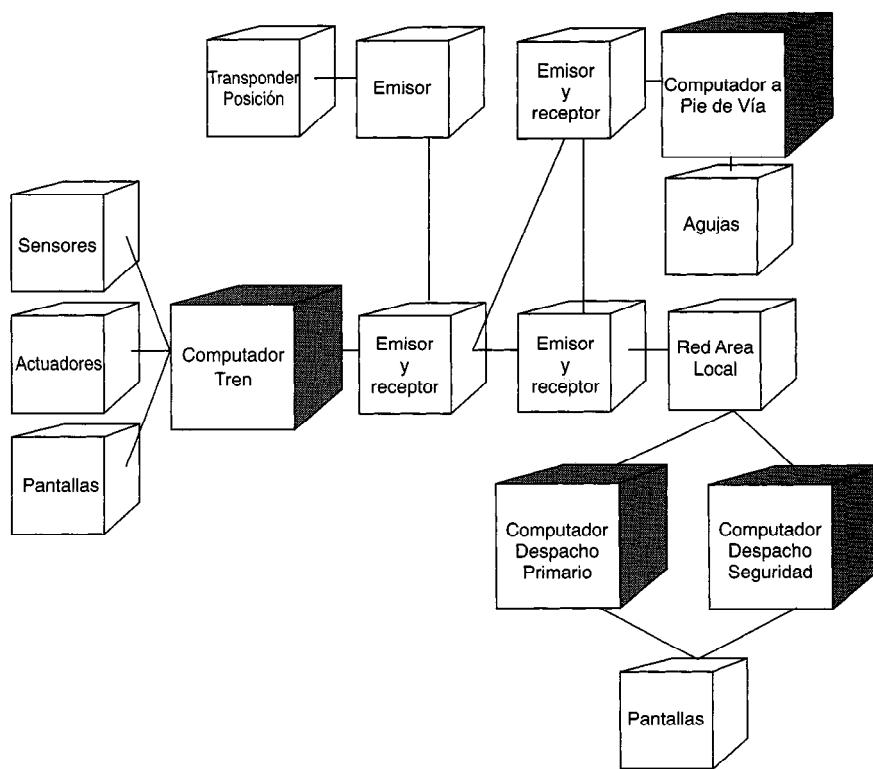


Figura 12.3. Diagrama de procesos del sistema de gestión de tráfico.

faz a pie de vía y sus agujas) es controlada por un computador que puede comunicarse mediante su transmisor y receptor con un tren que pase o con un controlador de terminal fijo. Cada controlador de terminal fijo se conecta en última instancia con una red de área local, una para cada centro de control (incluyendo al sistema de control de operaciones ferroviarias). A causa de la necesidad de un funcionamiento ininterrumpido, se ha decidido poner dos computadores en cada centro de control: un computador principal y uno de seguridad, que se espera que entre en funcionamiento si falla el principal. Durante períodos ociosos, el computador de seguridad puede usarse para atender necesidades computacionales de otros usuarios de menor prioridad.

Cuando está en funcionamiento, el sistema de gestión de tráfico puede involucrar cientos de computadores, incluyendo uno para cada tren, uno para cada unidad de interfaz a pie de vía, y dos en cada centro de control. El diagrama de procesos sólo muestra la presencia de unos pocos de ellos, ya que las configuraciones de computadores similares son completamente redundantes.

Como se discutió en los Capítulos 6 y 7, la clave para mantener la cordura durante el desarrollo de cualquier proyecto complejo es idear interfaces claros y

explicitos entre las partes fundamentales del sistema. Esto es particularmente importante cuando se definen interfaces entre hardware y software. Al principio, los interfaces pueden estar poco definidos, pero deben formalizarse rápidamente de forma que puedan desarrollarse, probarse y entregarse en paralelo las diferentes partes del sistema. Los interfaces bien definidos también hacen mucho más fácil el realizar balances hardware/software cuando surgen oportunidades, sin deshacer partes del sistema que ya se han completado. Además, no se puede esperar que todos los desarrolladores de una organización de desarrollo grande sean atletas de la programación. Hay que dejar por tanto la especificación de esas abstracciones y mecanismos clave a los mejores arquitectos.

Abstracciones y mecanismos clave

Un estudio de los requisitos para el sistema de gestión de tráfico sugiere que en realidad hay que resolver cuatro subproblemas diferentes:

- Redes.
- Bases de datos.
- Interfaz hombre/máquina.
- Control analógico de dispositivos en tiempo real.

¿Cómo se ha llegado a identificar estos problemas como los que acarrean el mayor riesgo de desarrollo?

El hilo que ata los elementos de este sistema es una red de comunicaciones distribuida. Los mensajes pasan por radio de los *transponders* a los trenes, entre trenes y controladores de terminales fijos, entre trenes y unidades de interfaz a pie de vía, y entre controladores de terminales fijos y unidades de interfaz a pie de vía. También deben pasar mensajes entre los centros de control y los controladores de terminales fijos individuales. El funcionamiento seguro de todo este sistema depende de que la transmisión y recepción de mensajes sean fiables y rápidas.

Además, este sistema debe llevar cuenta de las posiciones actuales y de las rutas planeadas para muchos trenes diferentes simultáneamente. Hay que mantener esta información actualizada y autoconsistente, incluso en presencia de actualizaciones concurrentes y de consultas concurrentes de alrededor de la red. Éste es básicamente un problema de base de datos distribuida.

La construcción de los interfaces hombre/máquina plantea un conjunto diferente de problemas. En concreto, los usuarios de este sistema son principalmente maquinistas y controladores, ninguno de los cuales tiene por qué ser un usuario experto de computadores. El interfaz de usuario de un sistema operativo como UNIX o Windows podría ser marginalmente aceptable para un ingeniero del software profesional, pero se suele considerar hostil al usuario por parte de los usuarios finales de aplicaciones como la del sistema de gestión del tráfico. Todas las formas de interacción con el usuario deben ser por tanto cuidadosamente diseñadas para adaptarse a este grupo de usuarios específico del dominio.

Por último, el sistema de gestión de tráfico debe interactuar con gran diversidad de sensores y actuadores. Da igual qué dispositivo sea: los problemas de medir y controlar el entorno son parecidos, y por tanto deberían ser tratados por el sistema de forma consistente.

Cada uno de estos cuatro subproblemas conlleva problemas bastante independientes. Los arquitectos del sistema necesitan identificar los mecanismos y abstracciones clave implicados en cada uno, así que se pueden asignar expertos en cada dominio para abordar su subproblema particular en paralelo con los demás. Nótese que éste no es un problema de análisis o diseño: el análisis de cada problema va a impactar en la arquitectura, y los diseños van a desvelar nuevos aspectos del problema que requieren un análisis posterior. El desarrollo es, por tanto, inevitablemente, incremental e iterativo.

Si se realiza un breve análisis del dominio a lo largo de estas cuatro áreas de problema, se encuentra que hay tres abstracciones comunes de alto nivel:

- Trenes Incluyendo locomotoras y vagones.
- Vías Abarcando perfiles, inclinaciones y dispositivos a pie de vía.
- Planes Incluyendo horarios, órdenes, permisos, autorizaciones y asignación de tripulaciones.

Cada tren tiene una posición actual en las vías, y cada tren tiene exactamente un plan activo. Análogamente, el número de trenes en cada punto de las vías puede ser cero o uno; para cada plan, hay exactamente un tren, lo que involucra muchos puntos de las vías.

Continuando, se puede idear un mecanismo clave para cada uno de esos cuatro subproblemas casi independientes:

- Paso de mensajes.
- Planificación de horarios de trenes.
- Visualización.
- Adquisición de datos de sensores.

Estos cuatro mecanismos forman el alma del sistema. Representan aproximaciones a lo que se ha identificado como las áreas de mayor riesgo de desarrollo. Es, por tanto, esencial que se desplieguen aquí los mejores arquitectos de sistemas para experimentar con enfoques alternativos y eventualmente fijen un marco de referencia a partir del cual otros desarrolladores menos experimentados puedan componer el resto del sistema.

12.2. Diseño

Como se discutió en el Capítulo 6, el diseño arquitectónico implica el establecimiento de la estructura de clases central del sistema, más una especificación de las colaboraciones comunes que animan esas clases. El centrarse pronto en

estos mecanismos ataca directamente los elementos de mayor riesgo del sistema, y sirve para capturar de forma concreta la visión de los arquitectos del mismo. En última instancia, los productos de esta fase sirven como el marco de referencia de clases y colaboraciones sobre el cual se construyen los demás elementos funcionales del sistema.

En esta sección, se examinará la semántica de cada uno de estos cuatro mecanismos clave del sistema.

Paso de mensajes

Con *mensaje*, no se hace referencia a una invocación de métodos, como en un lenguaje de programación orientado a objetos; en vez de eso, nos estamos refiriendo a un concepto en el vocabulario del dominio del problema, a un nivel de abstracción mucho más alto. Por ejemplo, entre los mensajes típicos en el sistema de gestión de tráfico se incluyen señales para activar dispositivos a pie de vía, indicaciones de que los trenes pasan por lugares concretos, y órdenes de los controladores a los maquinistas. En general, estos tipos de mensajes se pasan a dos niveles distintos dentro del sistema de gestión de tráfico:

- Entre computadores y dispositivos.
- Entre computadores.

El interés está en el segundo nivel del paso de mensajes. Puesto que el problema implica una red de comunicaciones distribuida geográficamente, hay que considerar cuestiones como el ruido, fallos de equipos y seguridad.

Se puede hacer un primer corte en la identificación de estos mensajes examinando cada par de computadores que se comunican, como se muestra en el diagrama de procesos de la Figura 12.3. Para cada par, hay que hacer tres preguntas: (1) ¿Qué información gestiona cada computador? (2) ¿Qué información debe pasar de un computador a otro? (3) ¿A qué nivel de abstracción debería estar esta información? No hay solución empírica para estas preguntas. En vez de eso, hay que adoptar un enfoque iterativo hasta que se llegue a la conclusión de que se han definido los mensajes correctos y que no hay cuellos de botella en las comunicaciones del sistema (quizás a causa de demasiados mensajes sobre una sola vía de comunicación, o mensajes demasiado grandes o demasiado pequeños).

Es absolutamente crítico a este nivel del diseño el centrarse en la sustancia, no en la forma, de esos mensajes. Con demasiada frecuencia, se ha visto a los arquitectos de sistemas comenzar por seleccionar una representación de los mensajes a nivel de bits. El verdadero problema de elegir prematuramente una representación de tan bajo nivel es que es seguro que cambiará y que causará, por tanto, rupturas con todos los clientes que dependan de una representación particular. Es más, en este punto del proceso de diseño, no se puede saber suficientemente acerca del modo en que se usarán estos mensajes como para tomar

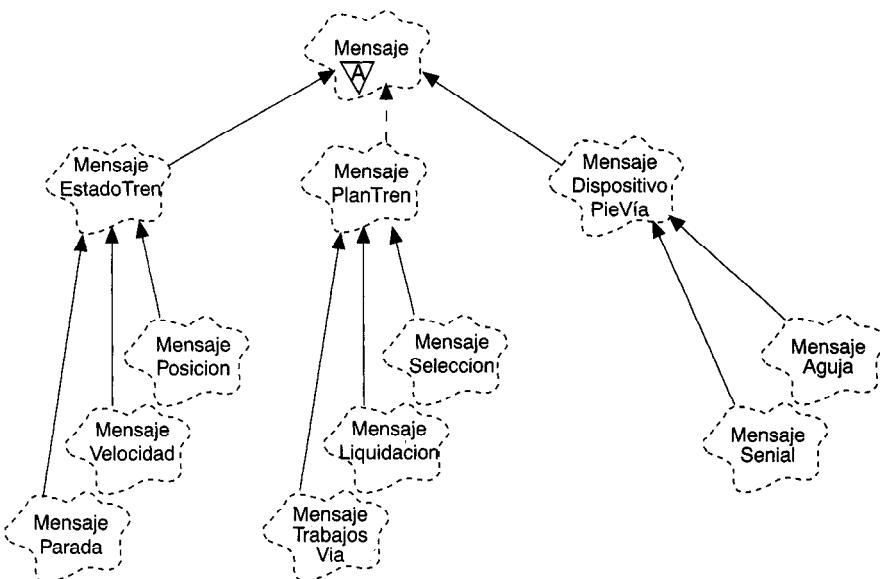


Figura 12.4. Diagrama de clases de mensajes.

decisiones inteligentes sobre representaciones eficientes respecto al tiempo o al espacio.

Al centrarse en la sustancia de esos mensajes, se tiene la intención de concentrar la atención en la vista externa de cada clase de mensajes. En otras palabras, hay que decidir sobre los papeles y responsabilidades de cada mensaje, y qué operaciones pueden realizarse de forma significativa sobre cada mensaje.

El diagrama de clases de la Figura 12.4 captura las decisiones de diseño con respecto a algunos de los mensajes más importantes en el sistema de gestión de tráfico. Nótese que todos los mensajes son, al fin y al cabo, instancias de una clase abstracta generalizada llamada `Mensaje`, que abarca el comportamiento común a todos los mensajes. Tres clases de nivel inferior representan las categorías principales de mensajes, a saber, mensajes sobre el estado del tren, mensajes sobre planes del tren y mensajes de dispositivos a pie de vía. Cada una de estas clases está más especializada. En realidad, el diseño final podría incluir docenas de tales clases especializadas, momento en el cual la existencia de estas clases intermedias pasa a ser aún más importante; sin ellas, se acabaría con muchos módulos sin relación entre sí —y por tanto difíciles de mantener— que representarían cada clase especializada diferente. A medida que se despliega el diseño, probablemente se descubrirán otros agrupamientos importantes de mensajes y, por tanto, se inventarán otras clases intermedias. Afortunadamente, la reorganización de la trama de clases de esta forma tiende a tener un impacto semántico mínimo sobre los clientes que, en última instancia, usan las clases hoja.

Como parte del diseño arquitectónico, sería una medida inteligente establecer pronto el interfaz de las clases de mensajes clave. Se podría comenzar con un análisis del dominio de las clases hoja más interesantes de la jerarquía, con el fin de formular los papeles y responsabilidades de todas esas clases, que se podrían capturar entonces concretamente en declaraciones de clase en C++. Se comienza con la invención de dos definiciones de tipos:

```
// Número que denota un número único de paquete
typedef unsigned int IdPaquete;

// Número que denota un identificador único de red
typedef unsigned int IdNodo;
```

Se continúa con la declaración de la clase base abstracta Mensaje:

```
class Mensaje {
public:

    Mensaje();
    Mensaje(IdNodo emisor);
    Mensaje(const Mensaje&);
    virtual Mensaje();

    virtual Mensaje& operator=(const Mensaje&);
    virtual Booleano operator==(const Mensaje&);
    Booleano operator!=(const Mensaje&);

    IdPaquete id() const;
    Hora marcaHora() const;
    IdNodo emisor() const;
    virtual Boolean estaIntacto() const = 0;

};
```

Las responsabilidades de esta clase incluyen la gestión de un único identificador de mensaje, marca de la hora e identificador del emisor, así como asegurar la integridad de los mensajes (es decir, saber si un mensaje del sistema es o no sintácticamente o semánticamente correcto). Este último comportamiento es lo que hace que los mensajes sean algo más que simples registros de datos. Como de costumbre, los mensajes también son responsables de saber cómo copiarse, asignarse y probarse (para igualdad) a sí mismos.

Una vez que se ha diseñado el interfaz de los mensajes más importantes, se pueden escribir programas que se construyan sobre estas clases para simular la creación y recepción de flujos de mensajes. Se pueden usar estos programas como un andamiaje para probar diferentes partes del sistema durante el desarrollo y antes de que las piezas con las que se conectan se hayan completado.

El diagrama de clases de la Figura 12.4 es incuestionablemente incompleto.

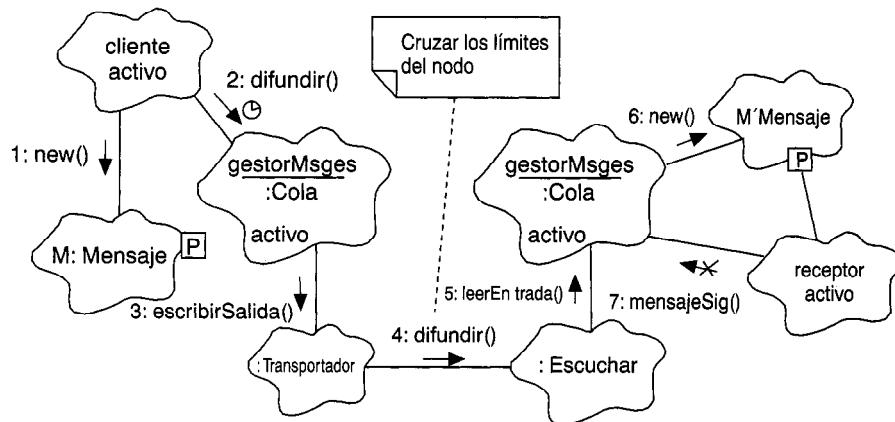


Figura 12.5. Paso de mensajes.

En la práctica, se comprueba que se pueden identificar primero los mensajes más importantes y dejar que los otros evolucionen a medida que se desvelan las formas menos importantes de comunicación. El uso de una arquitectura orientada a objetos permite añadir estos mensajes sin romper el diseño del sistema, porque tales cambios son generalmente compatibles en sentido creciente.

Una vez que se está satisfecho con esta estructura de clases, se puede comenzar a diseñar el propio mecanismo de paso de mensajes. Aquí se tienen dos objetivos que compiten entre sí: idear un mecanismo que proporcione una entrega fiable de los mensajes, y que lo haga a un nivel de abstracción lo bastante alto como para que los clientes no tengan que preocuparse sobre cómo tiene lugar esa entrega de los mensajes. Tal mecanismo permite a sus clientes realizar suposiciones elementales sobre cómo se envían y reciben los mensajes.

La Figura 12.5 ofrece un escenario que captura el diseño del mecanismo de paso de mensajes. Como indica este diagrama, para enviar un mensaje, un cliente crea antes un nuevo mensaje M, y entonces lo difunde al gestor de mensajes de su nodo, cuya responsabilidad es poner el mensaje en una cola para una eventual transmisión. Nótese que el diseño permite al cliente abandonar si el gestor de mensajes no puede efectuar la difusión en un tiempo razonable. Nótese también que el gestor de mensajes recibe el mensaje que va a difundir como un parámetro y usa entonces los servicios de un objeto Transportador para reducir el mensaje a su forma canónica y difundirlo por la red.

Como sugiere este diagrama, se decide que ésta sea una operación asíncrona, porque no se desea hacer al cliente esperar a que el mensaje se envíe a través de un enlace de radio, lo cual requiere un tiempo para la codificación, descodificación y quizás retransmisión a causa del ruido. Eventualmente, algún objeto Escuchador del otro lado de la red recibe este mensaje, y se lo presenta en su forma canónica al gestor de mensajes de su nodo, que a su vez crea un mensaje paralelo y lo pone en una cola. Un receptor puede bloquearse en la cabecera de la cola del gestor de mensajes, esperando a que llegue el siguiente

mensaje, que se entrega como parámetro a la operación `mensajeSig`, una operación síncrona.

El diseño del gestor de mensajes lo sitúa en la capa de aplicación en el modelo ISO OSI para redes [4]. Esto permite que todos los clientes que envían mensajes y los que los reciben operen al nivel más alto de abstracción, es decir, en términos de mensajes específicos de la aplicación.

Se espera que la implantación final de este mecanismo sea un poquito más compleja. Por ejemplo, se podría querer añadir comportamientos de cifrado y descifrado e introducir códigos para detectar y corregir errores, con el fin de asegurar una comunicación fiable en presencia de ruidos o fallos de los equipos.

Planificación de horarios de trenes

Como se hizo notar anteriormente, el concepto de plan de un tren es el centro del funcionamiento del sistema de gestión de tráfico. Cada tren tiene exactamente un plan activo, y cada plan se asigna a exactamente un tren y puede involucrar muchas órdenes y posiciones en la vía diferentes.

El primer paso es decidir exactamente qué partes constituyen el plan de un tren. Para hacerlo, hay que considerar todos los clientes potenciales de un plan y cómo se espera que cada uno de ellos use ese plan. Por ejemplo, a algunos clientes se les podría permitir crear planes, otros podrían modificarlos, y otros podrían tener sólo la capacidad de leer los planes. En este sentido, un plan de un tren actúa como un depósito para toda la información pertinente asociada con el itinerario de un tren concreto y las acciones que tienen lugar en el camino, como enganchar o desenganchar vagones.

La Figura 12.6 captura las decisiones estratégicas sobre la estructura de la clase `PlanTren`. Como en el Capítulo 10, se usa un diagrama de clases para mostrar las partes que componen un plan de un tren (muy parecido a la forma en que lo haría un diagrama entidad-relación tradicional). Así, se ve que cada plan de un tren tiene exactamente una tripulación y puede tener muchas órdenes generales y muchas acciones. Se espera que esas acciones estén ordenadas en el tiempo, con cada acción compuesta de información como una hora, una posición, velocidad, autorización y órdenes. Por ejemplo, un plan de tren específico podría constar de las siguientes acciones:

Hora	Posición	Velocidad	Autorización	Órdenes
0800	Pueblo*	Según indicación	Ver jefe de estación	Salida de terminal
1100	Colorado Springs	60 km/h		Dejar 30 vagones
1300	Denver	70 km/h		Dejar 20 vagones
1600	Pueblo*	Según indicación		Volver a terminal

* En el original viene la ciudad estadounidense Pueblo que puede inducir a error al lector con el nombre común «pueblo» de nuestro idioma. (N. del T.)

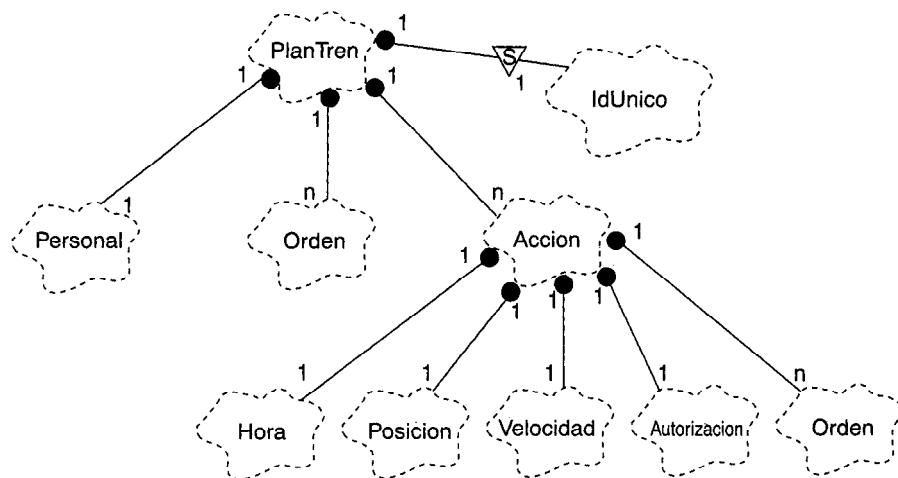


Figura 12.6. Diagrama de clases del PlanTren.

Como indica este diagrama, la clase `PlanTren` tiene un objeto miembro static, del tipo `IdUnico`, cuyo propósito es proporcionar el llamado *número mágico* para identificar de forma única cada instancia `PlanTren`.

Como se hizo para la clase `Mensaje` y sus subclases, se pueden diseñar los elementos más importantes de un plan de tren en fases tempranas del proceso de desarrollo; sus detalles evolucionarán con el tiempo, a medida que se apliquen realmente planes a diversos tipos de clientes.

El hecho de que se pueda tener una gran cantidad de planes de trenes activos e inactivos en cualquier momento dado nos enfrenta con el problema de bases de datos del que se habló anteriormente. El diagrama de clases de la Figura 12.6 puede servir como un bosquejo del esquema lógico de esta base de datos. La siguiente cuestión que se podría plantear por tanto es simplemente: ¿dónde se mantienen los planes de trenes?

En un mundo más perfecto, sin ruidos o retrasos de comunicación y con infinitos recursos de computación, la solución sería colocar todos los planes de trenes en una sola base de datos centralizada. Este enfoque produciría exactamente una instancia de cada plan de trenes. Sin embargo, el mundo real es mucho más perverso, y por eso esta solución no es práctica. Hay que contar con retrasos en la comunicación, y no se dispone de ciclos de procesador ilimitados. Así, la necesidad de acceder a un plan ubicado en el centro de control desde un tren no satisfacería los requisitos de tiempo real y cuasirreal. Sin embargo, se puede crear la ilusión de una base de datos centralizada y única en el software. Básicamente, la solución es tener una base de datos de planes de trenes situados en los computadores del centro de control, con copias de planes individuales distribuidas según sea necesario y situadas alrededor de la red. Por eficiencia,

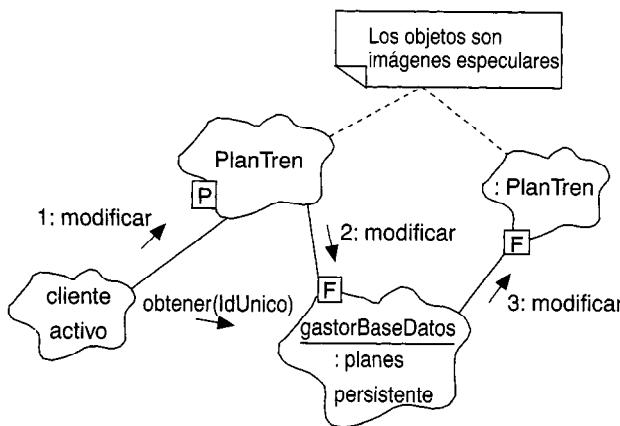


Figura 12.7. Planificación de horarios de trenes.

entonces, cada computador de un tren podría retener una copia de su plan actual. Así, el software de a bordo podría consultar este plan con un retraso despreciable. Si el plan cambiase, ya fuese como resultado de una acción de un controlador o (menos probablemente) de la decisión del maquinista, el software tendría que asegurar que todas las copias de ese plan se actualizan en un tiempo aceptable.

La forma en que funciona este escenario es función del mecanismo de planificación de horarios, mostrado en la Figura 12.7. La versión principal de cada plan de tren reside en una base de datos centralizada en un centro de control, con cero o más copias diseminadas alrededor de la red. Siempre que un cliente requiera una copia de un plan de tren particular (mediante la operación `obtener`, invocada con un valor de `IdUnico` como argumento), el estado de esta versión primaria se clona y se entrega al cliente como parámetro. La ubicación de la copia en la red se registra en la base de datos, y la copia del plan mantiene un enlace con la base de datos. Ahora, supóngase que un cliente de un tren necesita hacer un cambio en un plan particular, quizás como resultado de alguna acción del maquinista. En última instancia, este cliente invocaría operaciones sobre su copia del plan de tren y modificaría por tanto su estado. Estas operaciones también enviarían mensajes a la base de datos centralizada, para modificar de la misma forma el estado de la versión principal del plan. Ya que se registra la posición en la red de cada copia de un plan de tren, también se pueden enviar mensajes al depósito central que fuercen la actualización correspondiente en el estado de todas las copias restantes. Para asegurar que los cambios se realizan consistentemente a lo largo de la red, se podría emplear un mecanismo de bloqueo de registros, de forma que los cambios en los planes de los trenes no se lleven a cabo hasta que se hayan actualizado todas las copias de la versión primaria.

Este mecanismo se aplica igualmente bien si algún cliente en el centro de

control inicia el cambio, quizás como resultado de la acción de algún controlador. Primero, se actualizaría la versión primaria del plan y, a continuación, se difundirían los cambios a todas las copias a lo largo de la red, usando el mismo mecanismo. En cualquier caso, ¿cómo se comunican exactamente esos cambios? La respuesta es que se usa el mecanismo de paso de mensajes inventado anteriormente. En concreto, sería necesario añadir al diseño algunos nuevos mensajes sobre planes de trenes y a continuación construir el mecanismo de planes de trenes sobre este mecanismo de paso de mensajes de nivel inferior.

El uso de sistemas de gestión de base de datos modernos y comerciales en los computadores de control permite afrontar cualquier requisito sobre copias de seguridad de la base de datos, recuperación, seguimiento y seguridad.

Visualización

El uso de tecnologías punteras para las necesidades en cuanto a bases de datos ayuda a centrarse en las partes específicas del dominio del problema. Se puede conseguir un apoyo parecido para las necesidades de visualización usando facilidades gráficas estándar, como Microsoft Windows o X Windows. El uso de software de gráficos comercial eleva efectivamente el nivel de abstracción del sistema, de forma que los desarrolladores nunca necesitan preocuparse de manipular la representación visual de objetos visualizables a nivel de pixeles. Además, es importante encapsular las decisiones de diseño respecto al modo en que se representan visualmente diversos objetos.

Por ejemplo, considérese la visualización de perfil e inclinación de una sección específica de la vía. Los requisitos dictan que tal visualización puede aparecer en dos lugares diferentes: en un centro de control y a bordo de un tren (con la visualización centrada sólo en la vía que está por delante del tren). Suponiendo que se tenga alguna clase cuyas instancias representen secciones de una vía, se podrían adoptar dos enfoques para representar visualmente el estado de tales objetos. Primero, se podría tener algún objeto gestor de pantalla que construyese una representación visual preguntando el estado del objeto que va a visualizarse. Alternativamente, se podría eliminar este objeto externo y tener encapsulado en cada objeto visualizable el conocimiento sobre cómo mostrarse a sí mismo. Se prefiere este segundo enfoque, porque es más simple y está más en el espíritu del modelo de objetos.

Hay una desventaja potencial en este enfoque, sin embargo. Al fin y al cabo, se podrían tener muchos tipos diferentes de objetos visualizables, cada uno implantado por grupos diferentes de desarrolladores. Si se deja proceder independientemente a la implementación de cada objeto visualizable, probablemente se acabará teniendo código redundante, diferentes estilos de implementación, y un revoltijo generalmente inmantenible. Es mucha mejor solución el realizar un análisis del dominio de todos los tipos de objetos visualizables, determinar qué elementos visuales tienen en común e idear un conjunto intermedio de utilidades de clase que proporcionen rutinas de visualización para estos elementos

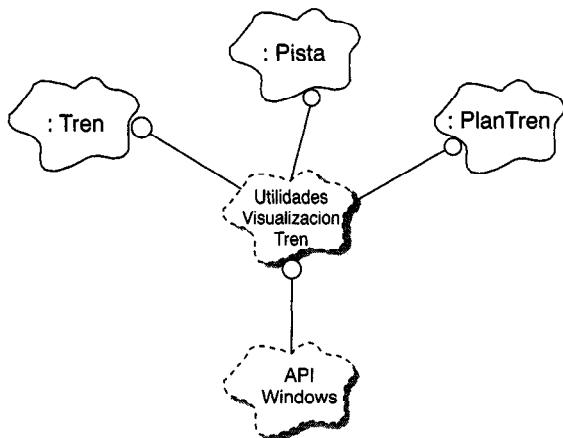


Figura 12.8. Visualización.

de dibujo comunes. Estas utilidades de clase pueden construirse a su vez sobre paquetes gráficos comerciales de nivel inferior.

La Figura 12.8 ilustra este diseño, mostrando que la implementación de todos los objetos visualizables comparte utilidades de clase comunes. Estas utilidades a su vez se construyen sobre interfaces Windows de nivel inferior, que están ocultos para todas las clases de nivel superior. En la práctica, los interfaces como el API de Windows no pueden expresarse fácilmente en una sola clase o utilidad de clase. Por tanto, el diagrama está un poco simplificado: es más probable que la implantación requiera un conjunto de utilidades de clase hermanas para el API de Windows, así como para las utilidades de visualización del tren.

La principal ventaja de este enfoque es que limita el impacto de cualquier cambio de nivel inferior resultante de intercambios hardware/software. Por ejemplo, si se encuentra que hay que sustituir el hardware de visualización por dispositivos más o menos potentes, sólo hay que reimplantar las rutinas de la clase **UtilidadesVisualizacionTren**. Sin esta colección de rutinas, los cambios de bajo nivel obligarían a modificar la implantación de todos los objetos visualizables.

Adquisición de datos de sensores

Como sugieren los requisitos, el sistema de gestión de tráfico incluye muchos tipos diferentes de sensores. Por ejemplo, hay sensores en cada tren que monitorizan la temperatura del aceite, cantidad de combustible, posición del acelerador, temperatura del agua, carga de tracción y demás. Análogamente, hay sensores activos en algunos de los dispositivos a pie de vía que informan, entre otras cosas, de las posiciones actuales de agujas y señales. Los tipos de valores que retornan los diversos sensores son diferentes entre sí, pero el procesamiento

de los datos de diferentes sensores es el mismo en una gran parte. Por ejemplo, suponiendo que los computadores usen E/S por correspondencia de memoria, cada valor de sensor se lee al fin y al cabo como un conjunto de bits de un lugar específico de la memoria, y se convierte después a algún valor específico del sensor. Además, la mayoría de los sensores deben muestrearse periódicamente. Si un valor está dentro de cierto rango, no pasa nada especial aparte de notificar a algún cliente el nuevo valor. Si el valor excede ciertos límites prefijados, habría que avisar a un cliente diferente. Por último, si este valor va mucho más allá de sus límites, podría ser necesario hacer sonar alguna clase de alarma, y notificárselo a otro cliente distinto más para que emprenda una acción drástica (por ejemplo, cuando la presión del aceite de la locomotora baja hasta niveles peligrosos).

Repetir este comportamiento para todos los tipos de sensor no sólo es tedioso y propenso a producir errores, también suele dar lugar a una redundancia de código. A menos que se exploten estos aspectos comunes, diferentes desarrolladores acabarán inventando múltiples soluciones para el mismo problema, lo que conduce a una proliferación de mecanismos de sensor ligeramente distintos y, a su vez, a un sistema que resulta más difícil de mantener. Es muy deseable, por tanto, realizar un análisis de dominios de todos los sensores periódicos y no discretos, de forma que se pueda inventar un mecanismo de sensor común para todos los tipos de sensores.

Se ha encontrado antes este problema, introducido en el Capítulo 8 como parte de la arquitectura del sistema de monitorización del clima. Se encontró una arquitectura que abarcaba una jerarquía de clases de sensor, y un mecanismo basado en marcos que adquiría periódicamente datos de esos sensores. En vez de reinventar esta arquitectura, tiene sentido plagiarla de este capítulo anterior, y aplicarla al sistema de gestión de tráfico.

Este es un ejemplo de reutilización de patrones en dominios cruzados.

12.3. Evolución

Arquitectura de módulos

Como se ha discutido, el módulo es un medio de descomposición necesario pero no suficiente; y así, para un problema del tamaño del sistema de monitorización del tráfico, hay que centrarse en una descomposición a nivel de subsistemas. Hay dos importantes factores que sugieren que en las actividades iniciales de la evolución habría que incluir la invención de la arquitectura de módulos del sistema de gestión de tráfico, que representa su arquitectura física software.

El diseño de software para sistemas muy grandes debe comenzar con frecuencia antes de que se complete el hardware de destino. El diseño de software requiere a menudo mucho más tiempo que el de hardware, y en cualquier caso,

hay que hacer compensaciones entre ambos a lo largo del camino. Esto implica que las dependencias que el software tiene respecto del hardware deben aislarse lo más posible, de forma que el diseño de software pueda proceder en ausencia de un entorno de destino estable. También implica que el software debe diseñarse pensando en sistemas reemplazables. En un sistema de dirección y control como el sistema de gestión de tráfico, se podría desear aprovechar nuevas tecnologías hardware que hubiesen madurado durante el desarrollo del software del sistema.

También hay que tener una descomposición física pronta e inteligente del software del sistema, de forma que los subcontratados que trabajan en diferentes partes del mismo (que puede incluso que usen lenguajes de programación distintos) puedan hacerlo en paralelo. Como se explicó en el Capítulo 7, muchas veces hay razones no técnicas que dirigen la descomposición física de un sistema grande. Quizás el más importante de estos intereses sea la asignación de trabajo a equipos de desarrolladores independientes. Las relaciones de subcontratación suelen establecerse en fases tempranas de la vida de un sistema complejo, a menudo antes de que haya suficiente información para adoptar buenas decisiones técnicas respecto a la descomposición correcta en subsistemas.

Se recomienda dar a los arquitectos del sistema la oportunidad de experimentar con descomposiciones alternativas en subsistemas, de forma que se pueda tener un nivel claramente alto de confianza en el que las decisiones globales de diseño físico son buenas. Esto puede conllevar la construcción de prototipos a gran escala (pero con todas las implantaciones de subsistemas reducidas a sistemas vacíos) y la realización de simulaciones de carga de procesadores, tráfico de mensajes y eventos externos. Estos prototipos y simulaciones pueden llevarse a cabo entonces a lo largo del proceso de maduración, como vehículos para pruebas de regresión.

¿Cómo se selecciona una descomposición en subsistemas correcta? Como se sugirió en el Capítulo 4, los objetos de más alto nivel están muchas veces agrupados en torno a líneas funcionales. Una vez más, esto no es ortogonal al modelo de objetos, porque con el término *funcionales* no se pretende hacer referencia a abstracciones algorítmicas que incorporen simples correspondencias entrada/salida. Se está hablando de puntos funcionales del sistema que representan comportamientos claramente visibles y comprobables, que resultan de la acción cooperativa de colecciones lógicas de objetos. Así, las abstracciones y mecanismos de más alto nivel que se identifican primero son buenos candidatos para organizar a su alrededor los subsistemas. Se puede afirmar primero la existencia de tales subsistemas, y después desplegar sus interfaces con el tiempo.

El diagrama de módulos de la Figura 12.9 representa las decisiones de diseño respecto a la arquitectura de módulos de más alto nivel en el sistema de gestión de tráfico. Aquí se ve una arquitectura dividida fuertemente en capas, en la que cada nivel abarca las funciones de los cuatro subproblemas que se identificaron anteriormente, a saber: redes, bases de datos, control de dispositivos analógicos en tiempo real e interfaz hombre/máquina.

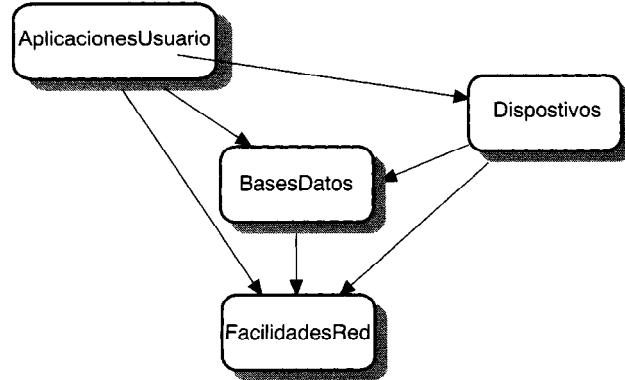


Figura 12.9. Diagrama de módulos de nivel superior para el sistema de gestión de tráfico.

Especificación de subsistemas

Si nos centramos en la vista externa de cualquiera de estos subsistemas, se encontrará que tiene todas las características de un objeto. Tiene una identidad única, aunque sea estática; incorpora una cantidad significativa de estado y exhibe un comportamiento muy complejo. Los subsistemas sirven como los depósitos para otras clases, utilidades de clases, y objetos; así, se caracterizan mejor por los recursos que exportan. En la práctica, con el uso de C++, estos subsistemas se capturan como directorios, que denotan colecciones lógicas de módulos y subsistemas anidados.

El diagrama de módulos de la Figura 12.9 es útil pero incompleto, porque cada subsistema de este diagrama es exageradamente grande como para ser desarrollado por un pequeño equipo de desarrolladores. Hay que examinar con más detalle cada uno de los subsistemas de nivel más alto, y descomponerlos más en sus módulos y subsistemas anidados.

Por ejemplo, considérese el subsistema `FacilidadesRed`. Se decide descomponerlo en otros dos subsistemas, uno privado (que se denomina `ComunicacionRadio`) y uno público (que se denomina `Mensajes`). El subsistema privado oculta los detalles del control software de los dispositivos físicos de radio, mientras que el subsistema exportado proporciona la funcionalidad del mecanismo de paso de mensajes que se diseñó anteriormente.

El subsistema llamado `BasesDatos` se construye sobre los recursos del subsistema `FacilidadesRed` y sirve para implantar el mecanismo de planes de trenes que se creó antes. Se decide descomponer más tarde este subsistema en dos subsistemas exportados, que representan los principales elementos de bases de datos del sistema. Se llama a esos subsistemas anidados `BaseDatosPlanTren` y `BaseDatosVia`, respectivamente. También se espera tener un subsistema privado, `GestorBaseDatos`, cuyo propósito es proporcionar todos los servicios comunes a las dos bases de datos específicas del dominio.

El subsistema **Dispositivos** también se descompone de forma natural en varios subsistemas más pequeños. Se decide agrupar el software relacionado con todos los dispositivos a pie de vía en un subsistema y el software asociado con todos los sensores y actuadores de a bordo de las locomotoras en otro. Estos dos subsistemas están a disposición de los clientes del subsistema **Dispositivos**, y ambos se construyen sobre los recursos de **BaseDatosPlanTren** y **Mensajes**. Así, se ha diseñado el subsistema **Dispositivos** para implantar el mecanismo de sensores que se describió anteriormente.

Por último, se decide descomponer el subsistema de nivel superior **AplicacionesUsuario** en varios más pequeños, incluyendo los subsistemas **AplicacionesMaquinista** y **AplicacionesControlador**, para reflejar los diferentes papeles de los dos usuarios principales del sistema de gestión de tráfico. El subsistema **AplicacionesMaquinista** incluye recursos que proporcionan toda la interacción maquinista/máquina especificada en los requisitos, como la funcionalidad del sistema de análisis e información de locomotoras y el sistema de gestión de energía. Se incluye el subsistema **AplicacionesControlador** para abarcar el software que proporciona la funcionalidad de todas las interacciones controlador/ máquina. Tanto **AplicacionesMaquinista** como **AplicacionesControlador** comparten recursos private comunes, como los exportados por el subsistema **Visualizaciones**, que incorpora el mecanismo de visualización descrito anteriormente.

Este diseño nos deja cuatro subsistemas de nivel superior, que abarcan varios más pequeños, a los cuales se han asignado todas las abstracciones y mecanismos clave que se inventaron previamente. Igualmente importante, como se discutió en el Capítulo 7, estos subsistemas de nivel inferior forman las unidades para asignar el trabajo así como las unidades para la gestión de configuraciones y el control de versiones. Como se sugirió también, cada subsistema debería ser poseído por una persona, aunque pueda ser implantado por muchas más. El propietario del sistema dirige el diseño detallado y la implantación del subsistema y gestiona su interfaz en relación con otros subsistemas al mismo nivel de abstracción. Así, se hace posible la gestión de un proyecto de desarrollo muy grande tomando un problema muy complejo y descomponiéndolo en varios más pequeños.

Como se trató en el Capítulo 7, esta estrategia también hace posible tener varias vistas diferentes simultáneas del sistema que se desarrolla. Un conjunto de generaciones compatibles de cada subsistema forma una versión y se pueden tener muchas de estas versiones: una para cada desarrollador, una para el equipo de garantía de calidad, y quizás una para el uso temprano del cliente. Los desarrolladores individuales pueden crear su propia versión estable en la que integren nuevas generaciones del software del cual son responsables, antes de entregarlas al resto del equipo. De este modo, se tiene una plataforma para la integración continua de nuevo código.

La clave para realizar este trabajo es un desarrollo cuidadoso de los interfaces entre subsistemas. Una vez construidos, estos interfaces deberían protegerse rigurosamente. ¿Cómo se determina la vista externa de cada subsistema? Exa-

minando cada subsistema como un objeto. Así, se plantean las mismas preguntas que se plantearon en el Capítulo 4 para objetos mucho más primitivos: ¿qué estado incorpora este objeto, qué operaciones significativas pueden realizar los clientes sobre él, y qué operaciones requiere de otros objetos?

Por ejemplo, considérese el subsistema `BaseDatosPlanTren`. Se construye sobre otros tres subsistemas (`Mensajes`, `BaseDatosTren` y `BaseDatosVia`) y tiene varios clientes importantes, a saber: los cuatro subsistemas `DispositivosPieVia`, `DispositivosLocomotora`, `AplicacionesMaquinista` y `AplicacionesControlador`. La `BaseDatosPlanTren` incorpora un estado relativamente evidente, concretamente el estado de todos los planes de trenes. Por supuesto, la peculiaridad es que este subsistema debe soportar el comportamiento de los mecanismos distribuidos de planes de trenes. Así, desde fuera, los clientes ven una base de datos monolítica pero, desde dentro, se sabe que esta base de datos está en realidad distribuida y, por tanto, debe construirse sobre la cima del mecanismo de paso de mensajes que se encuentra en el subsistema `Mensajes`.

¿Qué servicios ofrece la `BaseDatosPlanTren`? Parecen aplicarse todas las operaciones habituales de bases de datos: añadir registros, borrar registros, modificar registros e interrogar sobre registros. Como se hizo para el problema de bases de datos del Capítulo 10, se capturarían eventualmente todas estas decisiones de diseño que constituyen este subsistema en forma de clases C++ que proporcionan las declaraciones de todas estas operaciones.

En esta etapa del diseño, se continuaría el proceso de diseño para cada subsistema. Una vez más, se espera que estos interfaces no sean exactamente correctos a la primera; hay que permitir que evolucionen con el tiempo. Felizmente, al igual que para objetos más pequeños, la experiencia sugiere que la mayoría de los cambios que habrá que hacer a esos interfaces serán compatibles en sentido creciente, suponiendo que se haya hecho un buen trabajo previo en la caracterización del comportamiento de cada subsistema de forma orientada a objetos.

12.4. Mantenimiento

Añadido de nueva funcionalidad

El viejo software nunca muere, simplemente se mantiene o se conserva, especialmente si se trata de sistemas tan grandes como éste. Ésta es la razón por la que se sigue hallando software en producción que se desarrolló a lo largo de los últimos veinte años (lo que es una edad absolutamente anciana cuando se habla de software). A medida que más usuarios aplican el sistema de gestión de tráfico y a medida que se adapta el diseño a nuevas implantaciones, los clientes descubrirán nuevos e imprevistos usos para mecanismos existentes, creando presión para ampliar la funcionalidad del sistema.

Considérese una adición significativa a los requisitos, a saber, el procesamiento de nóminas. En concreto, supóngase que el análisis muestra que las nóminas de la compañía ferroviaria las soporta actualmente un elemento hardware que ya no se fabrica y que se está corriendo un gran riesgo de perder la capacidad de gestión de nóminas porque un solo fallo serio del hardware pondrá el sistema de contabilidad fuera de combate para siempre. Por esta razón, se podría decidir integrar el proceso de nóminas con el sistema de gestión de tráfico. En principio, no es difícil concebir cómo podrían coexistir estos dos sistemas aparentemente sin relación; se los podría ver simplemente como aplicaciones separadas, con el proceso de nóminas corriendo como una actividad de segundo plano.

Un examen más detallado muestra que se ganaría realmente un tremendo valor con la integración del procesamiento de nóminas. Se puede recordar la discusión anterior de que, entre otras cosas, los planes de trenes contienen información sobre asignaciones de tripulación. Así, es posible hacer un seguimiento de las asignaciones reales y las planeadas, y partiendo de aquí se pueden calcular las horas trabajadas, horas extra, etc. Obteniendo directamente esta información, los cálculos de nóminas serán más precisos y ciertamente más rápidos.

¿Qué impacto tiene el añadir esta funcionalidad en el diseño existente? Muy poco. El enfoque sería añadir otro subsistema dentro del subsistema **AplicacionesUsuario**, que representase la funcionalidad del procesamiento de nóminas. En este punto de la arquitectura, tal subsistema tendría acceso a todos los mecanismos importantes sobre los que podría construirse. Esto es en realidad bastante habitual en sistemas orientados a objetos bien estructurados: una adición significativa de requisitos para el sistema puede tratarse con evidente facilidad construyendo nuevas aplicaciones sobre mecanismos existentes.

Considérese un cambio aún más radical. Supóngase que se desea introducir tecnología de sistemas expertos en el sistema, construyendo un ayudante del controlador que podría aconsejar sobre la forma de dirigir el tráfico y las respuestas ante emergencias. ¿Cómo afectaría este nuevo requisito a la arquitectura? Una vez más, la respuesta es que muy poco. La solución sería añadir un nuevo subsistema entre los subsistemas **BaseDatosPlanTren** y **AplicacionesControlador**, porque la base de conocimiento incorporada por este sistema experto reproduce los contenidos de la **BaseDatosPlanTren**; además, el subsistema **AplicacionesControlador** es el único cliente de este sistema experto. Sería necesario inventar algunos mecanismos nuevos para establecer la forma en que se presentan las sugerencias al usuario final. Por ejemplo, se podría usar una arquitectura de pizarra, como se hizo en el Capítulo 11.

Cambios en el hardware de destino

Como ya se mencionó, la tecnología del hardware sigue avanzando a un ritmo más rápido que nuestra capacidad para generar software. Además, es probable

que diversas razones políticas e históricas nos hayan hecho adoptar ciertas decisiones sobre el hardware y el software en momentos tempranos del proceso de desarrollo de las que puede que nos arrepintamos después³. Por esta razón, el hardware de destino para sistemas grandes se queda obsoleto mucho antes que su software. Por ejemplo, tras varios años de uso operacional, se podría decidir que era necesario reemplazar las pantallas de cada tren y de cada centro de control. ¿Cómo podría afectar esto a la arquitectura existente? Si se hubiesen mantenido los interfaces de los subsistemas a un nivel de abstracción alto durante la evolución del sistema, este cambio de hardware afectaría al software sólo en aspectos mínimos. Ya que se decidió encapsular todas las decisiones de diseño respecto a pantallas específicas, ningún otro subsistema se escribió de forma que dependiese de las características específicas de una estación de trabajo determinada; el sistema encapsula todos esos secretos del hardware. Esto significa que el comportamiento de las estaciones de trabajo estaba oculto en el subsistema llamado visualizaciones. Así, este subsistema actúa como un cortafuegos de abstracciones, que protege a todos los demás clientes de las complicaciones de la tecnología particular usada para las visualizaciones.

De manera similar, un cambio radical en los estándares de telecomunicaciones afectaría a la implantación, pero sólo de forma limitada. En concreto, el diseño asegura que sólo el subsistema Mensajes sabe algo sobre las comunicaciones en red. Así, incluso un cambio fundamental en las redes nunca afectaría a ningún cliente de alto nivel; el subsistema Mensajes los protege de la perversidad del mundo real.

Ninguno de los cambios que se han introducido rasga el tejido de la arquitectura ya existente. Esta es en realidad la característica definitiva de un sistema orientado a objetos bien diseñado.

Lecturas recomendadas

Los requerimientos para el sistema de gestión de tráfico están basados en los del Advanced Train Control System, descritos por Murphy [C 1988].

La traducción y verificación de mensajes se da en prácticamente todos los sistemas de dirección y control. Plinta, Lee y Rissman [C 1989] proporcionan un tratamiento excelente de los problemas, y ofrecen el diseño de un mecanismo de paso de mensajes de forma segura a través de varios procesadores en un sistema distribuido.

³Por ejemplo, el proyecto podría haber elegido un producto hardware o software concreto de una tercera parte vendedora, para encontrar después que el producto no proporciona lo que prometía. Peor aún, se podría encontrar que el único suministrador de un producto crítico ha cerrado la empresa. En esos casos, el director del proyecto tiene normalmente una de estas dos opciones: (1) correr gritando en la noche, (2) elegir otro producto, y esperar que la arquitectura del sistema sea lo bastante flexible para acomodarse al cambio. El uso de análisis y diseño orientados a objetos lo ayuda a conseguir (2), aunque a veces uno se sigue quedando muy satisfecho realizándolo (1).

Notas bibliográficas

- [1] Murphy, E. December 1988. All Aboard for Solid State. *IEEE Spectrum* vol. 25(13), p. 42.
- [2] *Rockwell Advanced Railroad Electronic Systems*. 1989. Cedar Rapids, IA: Rockwell International.
- [3] Tanenbaum, A. 1981. *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall.

Epílogo

Porque los libros son tan sólo una parte de las mentes y los cuerpos de sus autores. Gran parte de ellos proviene de otro sitio, y nosotros los autores nos sentamos a la máquina de escribir esperando a que los libros ocurran.

GUY LEFRANCOIS*
Of Children

El desarrollo orientado a objetos es una tecnología ya probada. Nuestro método se ha utilizado para construir y entregar multitud de sistemas complejos en una amplia gama de dominios de problemas.

La demanda de software complejo continúa creciendo a un ritmo vertiginoso. Las crecientes capacidades del hardware y una creciente conciencia social sobre la utilidad de los computadores crea una tremenda presión para automatizar más y más aplicaciones de complejidad todavía mayor. El valor fundamental del desarrollo orientado a objetos, con su notación y proceso bien definidos es que libera al espíritu humano de forma que pueda centrar sus energías creativas sobre las partes realmente difíciles en la construcción de un sistema complejo.

* Lefrançois, G. 1977. *Of Children: An Introduction to Child Development, Second Edition*. Belmont, CA: Wadsworth, p. 371.



Lenguajes de programación orientados a objetos

El uso de tecnología orientada a objetos no se restringe a ningún lenguaje particular; al contrario, es aplicable a un amplio espectro de lenguajes de programación basados en objetos y orientados a objetos. Por importantes que sean el análisis y el diseño, sin embargo, no se pueden ignorar los detalles de la codificación porque, al fin y al cabo, las arquitecturas software deben expresarse en algún lenguaje de programación. En realidad, tal como sugirió Wulf, un lenguaje de programación sirve a tres propósitos:

- Es una herramienta de diseño.
- Es un vehículo para el trabajo humano.
- Es un vehículo para instruir a un computador [1].

Este apéndice es para el lector que pueda no estar familiarizado con algunos de los lenguajes de programación orientados a objetos que se mencionan en este libro. En él se ofrece una descripción resumida de algunos de los lenguajes más importantes, junto con un ejemplo común que proporciona una base para comparar la sintaxis, semántica y modismos de dos de los lenguajes de programación más interesantes, que son C++ y Smalltalk.

A.1. Conceptos

En la actualidad, hay más de 2.000 lenguajes distintos de programación de alto nivel. Aparecen tantos lenguajes diferentes porque cada uno se adaptó a los requisitos particulares del dominio de problema al que se destinaban. Además, la existencia de cada nuevo lenguaje permitió a los desarrolladores pasar a problemas más y más complejos. Con cada aplicación que no se había explorado an-

tes, los diseñadores de lenguajes aprendían nuevas lecciones que cambiaban sus suposiciones básicas sobre lo que era importante en un lenguaje y lo que no lo era. Esta evolución de los lenguajes también se vio fuertemente influenciada por el progreso de la teoría de la computación, que ha llevado a una comprensión formal de la semántica de las declaraciones, módulos, tipos abstractos de datos y procesos.

Como se discutió en el Capítulo 2, los lenguajes de programación pueden agruparse en cuatro generaciones, según soporten abstracciones matemáticas, algorítmicas, de datos u orientadas a objetos. Los avances más recientes en lenguajes de programación se han debido a la influencia del modelo de objetos. Según nuestros datos, hay actualmente más de 100 lenguajes diferentes basados en objetos y orientados a objetos. Como se dijo también en el Capítulo 2, un lenguaje se considera basado en objetos si soporta directamente abstracción de datos y clases. Un lenguaje orientado a objetos es aquél que está basado en objetos, pero también proporciona soporte para la herencia y el polimorfismo.

El antepasado común de casi todos los lenguajes de programación contemporáneos basados en objetos y orientados a objetos es Simula, desarrollado en los años sesenta por Dahl, Myhrhaug y Nygard [2]. Simula se construyó sobre las ideas de ALGOL, pero añadía los conceptos de encapsulamiento y herencia. Quizás más importante aún, Simula —como lenguaje para describir sistemas y desarrollar simulaciones— introducía la disciplina de escribir programas que reflejaban el vocabulario del dominio del problema.

La Figura A.1 se deriva de Schmucker [3] y muestra la genealogía de los lenguajes de programación orientados a objetos y basados en objetos más influyentes y ampliamente utilizados. En las próximas secciones, se examinan algunos de estos lenguajes en relación al soporte que ofrecen para los elementos del modelo de objetos.

A.2. Smalltalk

Origen

Smalltalk fue creado por los miembros del Xerox Palo Alto Research Center Learning Research Group como el elemento software de Dynabook, un visionario proyecto de Alan Kay. Simula fue su principal influencia, aunque Smalltalk también tomó algunas ideas del lenguaje FLEX y del trabajo de Seymour Papert y Wallace Feurzeig. Smalltalk representa tanto un lenguaje como un entorno de desarrollo de software. Es un lenguaje de programación orientado a objetos puros, en el sentido de que todo se ve como un objeto —incluso los enteros son clases—. Tras Simula, Smalltalk es quizás el lenguaje de programación orientado a objetos más importante, porque sus conceptos han influido no sólo el diseño de casi todos los lenguajes de programación orientados a objetos

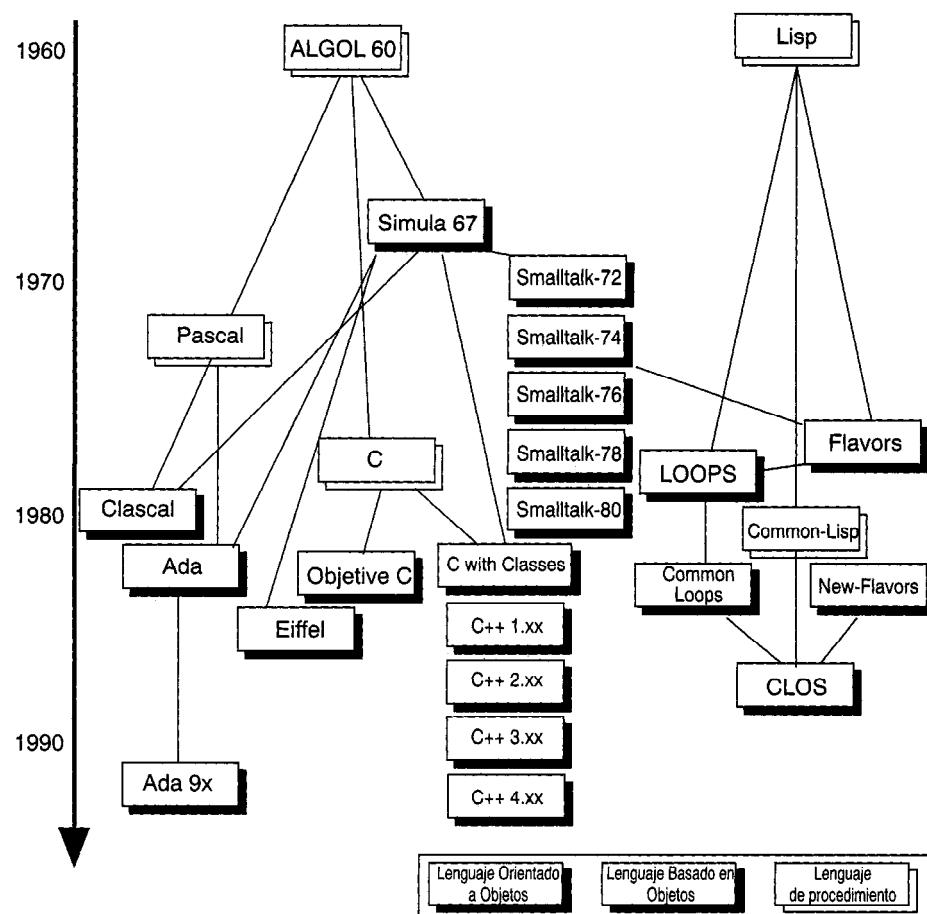


Figura A.1. Una genealogía de los lenguajes de programación orientados a objetos y basados en objetos.

subsiguientes, sino también el aspecto y sensación de interfaces gráficos de usuario como los de Macintosh, Windows y Motif, con los cuales se cuenta de forma natural en la actualidad.

Smalltalk evolucionó a lo largo de casi una década de trabajo y fue el producto de la actividad sinérgica de un grupo. Dan Ingalls fue el arquitecto jefe durante la mayor parte del desarrollo de Smalltalk, pero hubo también contribuciones fundamentales de Peter Deutscher, Glenn Krasner y Kim McCall. En paralelo, los elementos del entorno Smalltalk fueron desarrollados por James Althoff, Robert Flegal, Ted Kaehler, Diana Merry y Steve Putz. Entre otros papeles importantes que desempeñaron, Adele Goldberg y David Robson sirvieron como cronistas del proyecto Smalltalk.

Hay cinco versiones identificables de Smalltalk, indicadas por su año de aparición: Smalltalk-72, -74, -76, -78 y la más reciente, Smalltalk-80. Smalltalk-72 y -74 no proporcionaban soporte para la herencia, pero sentaron gran parte de los fundamentos conceptuales del lenguaje, incluyendo las ideas de paso de mensajes y polimorfismo. Las versiones posteriores del lenguaje convirtieron a las clases en ciudadanos de primera clase, completando así la visión de que todo lo que hay en el entorno puede tratarse como un objeto. Smalltalk-80 se ha transportado a una gran variedad de arquitecturas de máquinas.

También hay un importante dialecto de Smalltalk proporcionado por Digital, Smalltalk/V, muy parecido a Smalltalk-80 y disponible para el IBM PC (Windows y OS/2) y para Macintosh. Excepto para las clases de interfaz de usuario, las bibliotecas de clases son bastante parecidas entre sí. También al igual que Smalltalk-80, existe un entorno de desarrollo y diversas herramientas de desarrollo que son parecidas en capacidad, pero diferentes en estructura y función [4].

Visión general

Ingalls establece que «el propósito del proyecto Smalltalk es apoyar a niños de todas las edades en el mundo de la información. El desafío es identificar y dominar metáforas de potencia y simplicidad suficientes para permitir a una sola persona tener acceso a, y control creativo sobre, información que varía desde números y texto hasta sonidos e imágenes» [5]. Con este objetivo, Smalltalk se construye alrededor de dos conceptos simples: todo se trata como un objeto, y los objetos se comunican mediante el paso de mensajes.

La Tabla A.1 resume las características de Smalltalk, en relación a los siete elementos del modelo de objetos. Aunque la tabla no lo indica, la herencia múltiple es posible por redefinición de ciertos métodos primitivos [6].

Ejemplo

Considérese el problema en el que se tiene una lista heterogénea de formas, en la que cada objeto forma particular podría ser un círculo, un rectángulo o un rectángulo sólido (esto es parecido al problema que se introdujo en el Capítulo 3). Smalltalk tiene una extensa biblioteca de clases que ya incluye clases para círculos y rectángulos, y por eso la solución en este lenguaje sería casi trivial; esto demuestra la importancia de la reutilización. Sin embargo, a efectos de comparación, supóngase que sólo se tienen clases primitivas para dibujar líneas y arcos. Por tanto, se podría definir la clase `Formas` como sigue:

```
Object subclass: #Formas
  instanceVariableNames: 'elCentro'
  classVariableNames: ''
  poolDictionaries: ''
```

<i>Abstracción</i>	Variables de instancia Métodos de instancia Variables de clase Métodos de clase	Sí Sí Sí Sí
<i>Encapsulamiento</i>	De variables De métodos	Privado Público
<i>Modularidad</i>	Tipos de módulos	Ninguno
<i>Jerarquía</i>	Herencia Unidades genéricas Metaclases	Simple No Sí
<i>Tipos</i>	Comprobación estricta Polimorfismo	No Sí (simple)
<i>Concurrencia</i>	Multitarea	Indirectamente (mediante clases)
<i>Persistencia</i>	Objetos persistentes	No

Tabla A.1. Smalltalk.

```

category: 'Apendice'

initialize
    "Inicializar la forma"

    elCentro := Point new

fijarCentro: unPunto
    "Fijar el centro de la forma"

    elCentro := unPunto

centro
    "Devolver el centro de la forma"

    ^elCentro

draw
    "Dibujar la forma"

    self subclassResponsibility

```

Se puede definir a continuación la subclase `unCirculo` como sigue:

```
Formas subclass: #UnCirculo
```

```

instanceVariableNames: 'elRadio'
classVariableNames: ''
poolDictionaries: ''
category: 'Apéndice'

fijarRadio: unEntero
    "Fijar el radio del círculo"

    elRadio := unEntero

radio
    "Devolver el radio del círculo"

    ^elRadio

draw
    "Dibujar el círculo"

    | unArco indice |
    unArco := Arc new.
    indice := 1.
    [indice <= 4]
        whileTrue:
            [unArco
                center: elCentro
                radius: elRadio
                quadrant: indice.
                unArco display.
                indice := indice + 1]

```

Continuando, la subclase Rectangulos puede definirse como sigue:

```

Formas subclass: #Rectangulos
    instanceVariableNames: 'elAlto elAncho'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Apéndice'

draw
    "Dibujar el rectángulo"

    | unaLinea esquinaSupIz |
    unaLinea := Line new.
    esquinaSupIz := elCentro x - (elAncho / 2) @ (elCentro
                                                y - (elAlto / 2)).
    unaLinea beginPoint: esquinaSupIz.
    unaLinea endPoint: esquinaSupIz x + elAncho @ esquinaSupIz y.
    unaLinea display.
    unaLinea beginPoint: unaLinea endPoint.

```

```

unaLinea endPoint: esquinaSupIz x + elAncho @ (esquinaSupIz
y + elAlto).
unaLinea display.
unaLinea beginPoint: unaLinea endPoint.
unaLinea endPoint: esquinaSupIz x @ (esquinaSupIz
y + elAlto).
unaLinea display.
unaLinea beginPoint: unaLinea endPoint.
unaLinea endPoint: esquinaSupIz.
unaLinea display.

fijarAltura: unEntero
    "Fijar la altura del rectangulo"

    elAlto := unEntero

fijarAnchura: unEntero
    "Fijar la anchura del rectangulo"

    elAncho := unEntero

altura
    "Devolver la altura del rectangulo"

    ^elAlto

anchura
    "Devolver la anchura del rectangulo"

    ^elAncho

```

Por último, la subclase RectangulosSolidos puede definirse como:

```

Rectangulos subclass: #RectangulosSolidos
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Apendice'

draw
    "Dibujar el rectangulo sólido"

    | esquinaSupIz esquinaInfDe |
    super draw.
    esquinaSupIz := elCentro x - (elAncho quo: 2) + 1 @
        (elCentro y - (elAlto quo: 2) + 1).
    esquinaInfDe := esquinaSupIz x + elAncho - 1 @
        (esquinaSupIz y + elAlto - 1).

```

```
Display
  fill: (esquinaSupIz corner: esquinaInfDe)
  mask: Form gray
```

Referencias

Las referencias principales para Smalltalk son *Smalltalk-80: The Language*, de Goldberg y Robson [7]; *Smalltalk-80: The Interactive Programming Environment*, de Goldberg [8], y *Smalltalk-80: Bits of History, Words od Advice*, de Krasner [9]. LaLonde y Pugh [10] exploran Smalltalk-80 con gran profundidad, incluyendo tanto las bibliotecas de clases como el desarrollo de aplicaciones.

A.3. Object Pascal

Origen

Object Pascal fue creado por desarrolladores de Apple Computer (algunos de los cuales estuvieron implicados en el desarrollo de Smalltalk), en conjunción con Niklaus Wirth, el diseñador de Pascal. El antecesor inmediato de Object Pascal es Clascal, una versión orientada a objetos del Pascal para el Lisa. Object Pascal se puso a disposición del público en 1986 y es el primer lenguaje de programación orientado a objetos soportado por el Macintosh Programmer's Workshop (MPW), el entorno de desarrollo para la familia Apple de computadores Macintosh. La biblioteca de clases de MPW, llamada MacApp, proporciona el marco de referencia para construir aplicaciones que respeten las líneas maestras del interfaz de usuario Macintosh.

Visión general

Como dice Schmucker, «Object Pascal es el esqueleto de un lenguaje orientado a objetos. No proporciona métodos de clase, variables de clase, herencia múltiple o metaclasses. Estos conceptos se excluyeron deliberadamente en un intento de suavizar la curva de aprendizaje con que se encontraba la mayoría de los novatos en la programación orientada a objetos» [11].

Se resumen las características de Object Pascal en la Tabla A.2, en relación a los siete elementos del modelo de objetos.

Referencias

La referencia principal para Object Pascal es el *MPW Object Pascal Reference* de Apple [12].

<i>Abstracción</i>	Variables de instancia Métodos de instancia Variables de clase Métodos de clase	Sí Sí No No
<i>Encapsulamiento</i>	De variables De métodos	Público Público
<i>Modularidad</i>	Tipos de módulos	Unit
<i>Jerarquía</i>	Herencia Unidades genéricas Metaclases	Simple No No
<i>Tipos</i>	Comprobación estricta Polimorfismo	No Sí (simple)
<i>Concurrencia</i>	Multitarea	No
<i>Persistencia</i>	Objetos persistentes	No

Tabla A.2. Object Pascal.

A.4. C++

Origen

C++ fue diseñado por Bjarne Stroustrup de los AT&T Bell Laboratories. El antecesor inmediato de C++ es un lenguaje llamado C with Classes, desarrollado también por Stroustrup en 1980. A su vez, C with Classes recibió una poderosa influencia de los lenguajes C y Simula. C++ es en gran medida un superconjunto del C. Sin embargo, en cierto sentido, C++ es simplemente un mejor C, en el sentido de que proporciona comprobación de tipos, funciones sobrecargadas y muchas otras mejoras. Más importante, sin embargo, es el hecho de que C++ añade al C características de la programación orientada a objetos.

Han existido varias versiones principales del lenguaje C++. La versión 1.0 y sus versiones menores añadían características básicas de orientación a objetos al C, como la herencia simple y el polimorfismo, además de la comprobación de tipos y la sobrecarga. La versión 2.0, aparecida en 1989, mejoró las versiones anteriores de diversas formas (como la introducción de herencia múltiple), en base a una amplia experiencia con el lenguaje por parte de una comunidad de usuarios relativamente grande. La versión 3.0, aparecida en 1990, introdujo las plantillas (*templates*, clases parametrizadas) y el manejo de excepciones. El comité ANSI X3J16 C++ ha adoptado recientemente propuestas para control del

espacio de nombres (consistente con nuestra noción de categorías de clases) e identificación de tipos en tiempo de ejecución.

La tecnología inicial de traductores para C++ implicaba el uso de un preprocesador para C, llamado *cfront*. Puesto que este traductor generaba código C como representación intermedia, era posible transportar C++ a prácticamente todas las arquitecturas UNIX con bastante rapidez. Ahora, están disponibles comercialmente los traductores de C++ y los compiladores nativos para los conjuntos de instrucciones de casi todas las arquitecturas.

Visión general

Stroustrup afirma que «C++ se diseñó principalmente de forma que el autor y sus amigos no tuviesen que programar en ensamblador C o varios lenguajes de alto nivel modernos. Su propósito principal es hacer más fácil y agradable para el programador individual la escritura de buenos programas. Nunca hubo un diseño de C++ sobre el papel; el diseño, la documentación y la implantación sucedieron simultáneamente» [13]. C++ corrige muchas de las deficiencias del C, y añade al lenguaje soporte para clases, comprobación de tipos, sobrecarga, gestión del espacio libre, tipos constantes, referencias, funciones inline, clases derivadas y funciones virtuales [14].

Se resumen las características de C++ en la Tabla A.3, en relación a los siete elementos del modelo de objetos.

Ejemplo

De nuevo se reimplanta el problema de las formas. El estilo habitual en C++ es colocar la vista externa de cada clase en ficheros cabecera. Así, se puede escribir:

```
struct Punto {
    int x;
    int y;
};

class Forma {
public:
    Forma();
    void fijarCentro(Punto p);
    virtual void draw() = 0;
    Punto centro() const;
private:
    Punto elCentro;
};

class Circulo : public Forma {
public:
```

<i>Abstracción</i>	Variables de instancia Métodos de instancia Variables de clase Métodos de clase	Sí Sí Sí Sí
<i>Encapsulamiento</i>	De variables De métodos	Público, privado, protegido Público, privado, protegido
<i>Modularidad</i>	Tipos de módulos	Fichero
<i>Jerarquía</i>	Herencia Unidades genéricas Metaclases	Múltiple Sí No
<i>Tipos</i>	Comprobación estricta Polimorfismo	Sí Sí (simple)
<i>Concurrencia</i>	Multitarea	Indirectamente (mediante clases)
<i>Persistencia</i>	Objetos persistentes	No

Tabla A.3. C++.

```

Circulo();
void fijarRadio(int r);
virtual void draw();
int radio() const;
private:
    int elRadio;
};

class Rectangulo : public Forma {
public:
    Rectangulo();
    void fijarAltura(int h);
    void fijarAnchura(int a);
    virtual void Draw();
    int altura() const;
    int anchura() const;
private:
    int laAltura;
    int laAnchura;
};

class RectanguloSolido : public Rectangulo {
public:
    virtual void draw();
};

```

La definición de C++ no incluye una biblioteca de clases. Para nuestros propósitos, se supone la existencia de un interfaz de programación con X Windows y de los objetos globales Pantalla, Ventana y ContextoGraficos (que necesita Xlib). Así, se pueden completar los métodos de arriba en un fichero separado, como sigue:

```
Forma::Forma()
{
    elCentro.x = 0;
    elCentro.y = 0;
};

void Forma::fijarCentro(Punto p)
{
    elCentro = p;
};

Punto Forma::centro() const
{
    return elCentro;
};

Circulo::Circulo() : elRadio(0) {}

void Circulo::fijarRadio(int r)
{
    elRadio = r;
};

void Circulo::draw()
{
    int X = (centro().x - elRadio);
    int Y = (centro().y - elRadio);
    xDrawArc(Pantalla, Ventana, ContextoGraficos, X, Y,
              (elRadio * 2), (elRadio * 2), 0, (360 * 64));
};

int Circulo::radio() const
{
    return elRadio;
};

Rectangulo::Rectangulo() : laAltura(0), laAnchura(0) {}

void Rectangulo::fijarAltura (int h)
{
    laAltura = h;
};
```

```
void Rectangulo::fijarAnchura (int a)
{
    laAnchura = a;
};

void Rectangulo::draw()
{
    int X = (centro().x - (laAnchura / 2));
    int Y = (centro().y - (laAltura / 2));
    XDrawRectangle(Pantalla, Ventana, ContextoGraficos, X, Y,
                    laAnchura, laAltura);
};

int Rectangulo::altura() const
{
    return laAltura;
};

int Rectangulo::anchura() const
{
    return laAnchura;
};

void RectanguloSolido::draw()
{
    Rectangulo::draw();
    int X = (centro().x - (anchura() / 2));
    int Y = (centro().y - (altura() / 2));
    gc contextoGraficosViejo = ContextoGraficos;
    XSetForeground(Pantalla, ContextoGraficos, Gray);
    XDrawFilled(Pantalla, Ventana, ContextoGraficos, X, Y,
                anchura(), altura());
    ContextoGraficos = contextoGraficosViejo;
};
```

Referencias

La referencia principal para C++ es el *Annotated C++ Reference Manual** de Ellis y Stroustrup [15]. Stroustrup [16] cubre en profundidad el lenguaje y su uso en el contexto del diseño orientado a objetos.

* Existe versión en español en Addison-Wesley/Díaz de Santos, 1994; «*C++. Manual de referencia con anotaciones*».

A.5. Common Lisp Object System

Origen

Existen literalmente docenas de dialectos de Lisp, incluyendo MacLisp, Standard Lisp, SpiceLisp, S-1 Lisp, Nil, ZetaLisp, InterLisp y Scheme. A comienzos de los años ochenta apareció una multitud de nuevos dialectos de Lisp que soportaban programación orientada a objetos, muchos de los cuales se inventaron para soportar la investigación avanzada en representación del conocimiento. Bajo el estímulo del éxito de la estandarización del Common Lisp, se abordó un esfuerzo similar en 1986 para estandarizar estos dialectos orientados a objetos.

La idea de estandarización se llevó adelante en la ACM Lisp and Functional Programming Conference del verano de 1986, que tuvo como resultado la formación de un subcomité especial como parte del comité ANSI X3J13 (para la estandarización del Common Lisp). Puesto que este nuevo dialecto se concibió como un superconjunto propio del Common Lisp, se llamó Common Lisp Object System o CLOS, en abreviatura. Daniel Bobrow presidió el comité, entre cuyos miembros estaban Sonya Keene, Linda DeMichiel, Patrick Dussud, Richard Gabriel, James Kempf, Gregor Kiczales y David Moon.

El diseño del CLOS recibió una fuerte influencia de los lenguajes New Flavors y CommonLoops. Despues de aproximadamente dos años de trabajo, se publicó la especificación completa de CLOS a finales de 1988.

Visión general

Keene indica que se plantearon tres objetivos de diseño para CLOS:

- CLOS debía ser una extensión estándar del lenguaje que incluyese la mayor parte de los aspectos útiles de los paradigmas existentes orientados a objetos.
- El interfaz del programador de CLOS debía ser lo bastante potente y flexible como para desarrollar la mayoría de los programas de aplicación.
- El propio CLOS debía diseñarse como un protocolo extensible, para permitir la personalización de su comportamiento y para promover investigaciones posteriores sobre programación orientada a objetos [17].

Se resumen las características de CLOS en la Tabla A.4, en relación a los siete elementos del modelo de objetos. Aunque CLOS no soporta directamente objetos persistentes, hay extensiones sencillas que usan el protocolo de metaobjetos para añadir persistencia [18].

<i>Abstracción</i>	Variables de instancia Métodos de instancia Variables de clase Métodos de clase	Sí Sí Sí Sí
<i>Encapsulamiento</i>	De variables De métodos	De lectura, escritura y acceso Público
<i>Modularidad</i>	Tipos de módulos	Paquete
<i>Jerarquía</i>	Herencia Unidades genéricas Metaclases	Múltiple No Sí
<i>Tipos</i>	Comprobación estricta Polimorfismo	Opcional Sí (múltiple)
<i>Concurrencia</i>	Multitarea	Sí
<i>Persistencia</i>	Objetos persistentes	No

Tabla A.4. CLOS.

Referencias

La referencia principal para CLOS es la *Common Lisp Object System Specification* [19].

A.6. Ada

Origen

El Departamento de Defensa de los Estados Unidos (DoD)¹ es quizás el mayor usuario de computadoras del mundo. A mediados de los 70, el desarrollo de software para sus sistemas había alcanzado proporciones críticas: los proyectos con frecuencia se retrasaban, se salían del presupuesto, y fallaban a la hora de satisfacer los requisitos establecidos. Era evidente que los problemas seguirían empeorando a medida que los costes de desarrollo de software continuasen creciendo y la demanda de software continuase incrementándose a un ritmo exponencial. Para ayudar a resolver estos problemas, que se veían potenciados por la proliferación de cientos de lenguajes diferentes, el DoD patrocinó el desarro-

¹ DoD: Department of Defense. (*N. del T.*)

llo de un lenguaje de alto nivel único y común. En cierto sentido, Ada representa uno de los primeros lenguajes de calidad industrial que se han creado. Se desarrolló un conjunto de requisitos a comienzos de 1975 y culminó en el documento Steelman, que se publicó en 1978. Se planteó entonces una solicitud internacional de propuestas, que invitaba a las compañías a diseñar un lenguaje basado en estos requisitos. La solicitud recibió diecisiete respuestas. Este número se redujo a cuatro, después a dos y luego a una, mediante un extenso período de diseño y evaluación que implicó a cientos de informáticos de todo el mundo.

El diseño ganador se llamó originalmente el Lenguaje Verde (llamado así por su código de color durante la competición), y luego se renombró como Ada, en honor de Ada Augusta, Condesa de Lovelace, que destacó por sus pioneras observaciones sobre el potencial de las computadoras. El autor principal de este lenguaje fue el francés Jean Ichbiah. Otros miembros del equipo de diseño fueron Bernd Krieg-Brueckner, Brian Wichmann, Henry Ledgard, Jean-Claude Heliard, Jean-Loup Gailly, Jean-Raymond Abrial, John Barnes, Mike Woodger, Olivier Roubine, S. A. Schuman y S. C. Vestal.

Los antecesores inmediatos de Ada son Pascal y sus derivados, incluyendo Euclid, Lis, Mesa, Modula y Sue. También se incorporaron varios conceptos de ALGOL 68, Simula, CLU y Alphard. El estándar ANSI de Ada se publicó finalmente en 1983. Los traductores para Ada tardaron en llegar, pero hoy en día hay traductores para casi todas las familias importantes de arquitecturas de conjuntos de instrucciones. Aunque Ada fue patrocinado originalmente por el DoD, ha encontrado un importante papel a nivel mundial en proyectos gubernamentales y comerciales, y suele ser el lenguaje elegido para proyectos software a gran escala, como los sistemas de control de tráfico aéreo de Estados Unidos y Canadá. Puesto que los estándares de ANSI deben revisarse cada cinco años, se ha puesto en marcha un proyecto llamado Ada9x para actualizar este estándar. Con Ada9x, la definición original del lenguaje ha cambiado en varios aspectos de pequeña importancia, clarificando cosas, llenando lagunas y corrigiendo errores. En su definición actual, Ada está basado en objetos, no orientado a objetos. Sin embargo, Ada9x añade extensiones de programación orientada a objetos a la definición original del lenguaje.

Visión general

De acuerdo con sus diseñadores, Ada se diseñó con tres objetivos:

- Fiabilidad y mantenimiento de los programas.
- Programación como actividad humana.
- Eficiencia [20].

Se resumen las características de Ada en la Tabla A.5, en relación a los siete elementos del modelo de objetos.

<i>Abstracción</i>	Variables de instancia Métodos de instancia Variables de clase Métodos de clase	Sí Sí No No
<i>Encapsulamiento</i>	De variables De métodos	Público, privado Público, privado
<i>Modularidad</i>	Tipos de módulos	Paquete
<i>Jerarquía</i>	Herencia Unidades genéricas Metaclases	No (parte de Ada9x) Sí No
<i>Tipos</i>	Comprobación estricta Polimorfismo	Sí No (parte de Ada9x)
<i>Concurrencia</i>	Multitarea	Sí
<i>Persistencia</i>	Objetos persistentes	No

Tabla A.5. Ada.

Referencias

La referencia principal para la sintaxis y semántica de Ada es el *Reference Manual for the Ada Programming Language* [21].

A.7. Eiffel

Origen

Eiffel fue creado por Bertrand Meyer no sólo como un lenguaje de programación orientado a objetos, sino también como una herramienta de ingeniería del software. Aunque Eiffel recibe influencias de Simula, se diseñó desde el principio como un lenguaje independiente orientado a objetos y un entorno de desarrollo.

El lenguaje soporta ligadura dinámica y comprobación estática de tipos, ofreciendo flexibilidad en el diseño del interfaz de una clase, pero aprovechando la seguridad respecto al tipo que proporciona la comprobación estática de tipos. Hay varias características significativas que dan apoyo a una ingeniería del software más rigurosa, incluyendo clases parametrizadas, aserciones y excepciones. Meyer sostiene que las clases genéricas complementan la relación de herencia

permitiendo una genericidad horizontal: pueden crearse nuevas clases al mismo nivel de abstracción en una jerarquía de herencia basándose en parámetros de tipo, en vez de duplicar comportamientos en subclases hermanas.

Las precondiciones y postcondiciones, ambas partes integrales del lenguaje, implantan las aserciones sobre la entrada y la salida de un método, respectivamente. Si una precondición falla al entrar en un método, o si una postcondición falla al salir, se eleva una excepción. Existe un mecanismo en el lenguaje para gestionar la excepción mediante el uso de la cláusula rescue y la instrucción retry.

Visión general

Eiffel refuerza los conceptos de la buena ingeniería del software: buena especificación de clases, comprobación estricta de tipos, y facilidades para aprovechar la reutilización mediante la herencia y las clases genéricas. El tratamiento formal de las excepciones permite una especificación rigurosa de los interfaces de las clases en la implantación.

Eiffel también proporciona un entorno de desarrollo completo que incluye un editor dirigido por sintaxis, generación de documentación, bibliotecas de clases y un hojeador (browser). Además, soporta facilidades de gestión del código y de gestión de la construcción.

Las características de Eiffel en relación al modelo de objetos se resumen en la Tabla A.6.

Referencias

El mejor tratamiento del lenguaje Eiffel se encuentra en el libro de Meyer, *Object-Oriented Software Construction* [22].

A.8. Otros lenguajes de programación orientados a objetos

La Figura A.2 proporciona los nombres de muchos otros lenguajes de programación basados en objetos y orientados a objetos que son importantes o influyentes; la Bibliografía Clasificada ofrece referencias para fuentes de información sobre la mayoría de ellos.

Saunders [23] proporciona un examen de unos 80 lenguajes distintos de programación orientados a objetos y basados en objetos. Sugiere que los lenguajes de programación orientados a objetos pueden agruparse en siete categorías [24]:

- De actor
- Concurrentes

Lenguajes que admiten la delegación.
Lenguajes orientados a objetos que enfatizan la concurrencia.

<i>Abstracción</i>	Variables de instancia Métodos de instancia Variables de clase Métodos de clase	Sí Sí No No
<i>Encapsulamiento</i>	De variables De métodos	Privado Público, privado
<i>Modularidad</i>	Tipos de módulos	Unit
<i>Jerarquía</i>	Herencia Unidades genéricas Metaclases	Múltiple Sí No
<i>Tipos</i>	Comprobación estricta Polimorfismo	Sí Sí
<i>Concurrencia</i>	Multitarea	No
<i>Persistencia</i>	Objetos persistentes	No

Tabla A.6. Eiffel.

- Distribuidos Lenguajes orientados a objetos que enfatizan los objetos distribuidos.
- Basados en marcos Lenguajes que soportan la teoría de marcos.
- Híbridos Extensiones orientadas a objetos de lenguajes tradicionales.
- Basados en Smalltalk Smalltalk y sus dialectos.
- Ideológicos Aplicación de características orientadas a objetos a otros dominios.
- Varios Lenguajes orientados a objetos que no entran en ninguna otra categoría.

Notas bibliográficas

- [1] Wulf, W. January 1980. Trends in the Design and Implementation of Programming Languages. *IEEE Computer* vol. 13(1), p. 15.
- [2] Birtwistle, G., Dahl, O-J., Myhrhaug, B., and Nygard, K. 1979. *Stimula begin*. Lund, Sweden: Studentlitteratur.
- [3] Schmucker, K. 1986. *Object-Oriented Programming for the Macintosh*. Hasbrouk Heights, NJ: Hayden, p. 346.
- [4] LaLonde, W. and Pugh, J. 1990. *Inside Smalltalk, Volumes 1 and 2*. Englewood Cliffs, New Jersey: Prentice Hall.

ABCL/1	Concurrent Smalltalk	Lore	Plasma II
ABE	CSSA	Mace	POOL-T
Acore	CST	MELD	PROCOL
Act/1	Director	Mjolner	Quick Pascal
Act/2	Distributed Smalltalk	ModPascal	Quicktalk
Act/3	Eiffel	Neon	ROSS
Actor	Emerald	New Flavors	SAST
Actors	ExperCommonLisp	NIL	SCOOP
Actra	Extended Smalltalk	O-CPU	SCOOPS
Ada	Felix Pascal	OakLisp	Self
Argus	Flavors	Oberon	Simula
ART	FOOPlog	Object Assembler	SINA
Berkeley Smalltalk	FOOPS	Object Cobol	Smalltalk
Beta	FRL	Object Lisp	Smalltalk AT
Blaze	Galileo	Object Logo	Smalltalk V
Brouhaha	Garp	Object Oberon	Smallworld
C with Classes	GLISP	Object Pascal	SPOOL
C++	Gypsy	Objective-C	SR
C_talk	Hybrid	ObjVLisp	SRL
Cantor	Inheritance	OOPC	STROBE
Clascal	InnovAda	OOPS+	T
Classic Ada	Intermission	OPAL	Trellis/Owl
CLOS	Jasmine	Orbit	Turbo Pascal 5.x
Cluster 86	KL-One	Orient84/K	Uniform
Common Loops	KRL	OTM	UNITS
Common Objects	KRS	PCOL	Vulcan
Common ORBIT	Little Smalltalk	PIE	XLISP
Concurrent Prolog	LOOPS	PL/LL	Zoom/VM

Figura A.2. Lenguajes de programación basados en objetos y orientados a objetos.

- [5] Ingalls, D. The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, ACM, p. 9.
- [6] Borning, A. and Ingalls, D. 1982. Multiple Inheritance in Smalltalk-80. *Proceedings of the National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI.
- [7] Goldberg, A. and Robson, D. 1989. *Smalltalk-80: The Language*. Reading, MA: Addison-Wesley.
- [8] Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley.
- [9] Krasner, G. 1983. *Smalltalk-80: Bits of History, Words of Advice*. Reading, MA: Addison-Wesley.
- [10] LaLonde, 1990.
- [11] Schmucker, K. August 1986. Object-Oriented Language for the Macintosh. *Byte* vol. 11(8), p. 179.
- [12] *Macintosh Programmer's Workshop Pascal 3.0 Reference*. 1989. Cupertino, CA: Apple Computer.
- [13] Stroustrup, B. 1986. *The C++ Programming Language, Second Edition*. Reading, MA: Addison-Wesley, p. 4.*
- [14] Gorlen, K. 1989. An Introduction to C++, in *UNIX System V AT&T C++ Language System, Release 2.0 Selected Readings*. Murray Hill, NJ: AT&T Bell Laboratories, p. 2-1.

* Existe versión en español por Addison-Wesley Iberoamericana.

- [15] Ellis, M. and Stroustrup, B. 1990. *The Annotated C++ Reference Manual*. Reading, Massachusetts: Addison-Wesley Publishing Company.
- [16] Stroustrup, B. 1991. *The C++ Programming Language*, Second Edition. Reading, MA: Addison-Wesley.
- [17] Keene, S. 1989. *Object-Oriented Programming in Common Lisp*. Reading, MA: Addison-Wesley, p. 215.
- [18] Bobrow, D. 1990. Comunicación privada.
- [19] Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., and Moon, D. September 1988. Common Lisp Object System Specification X·J13 Document 88-002R. *SIGPLAN Notices* vol. 23.
- [20] *Reference Manual for the Ada Programming Language*. February 1983. Washington, D.C.: Department of Defense, Ada Joint Program Office, p. 1-3.
- [21] Ibid.
- [22] Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall.
- [23] Saunders, J. March/April 1989. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming* vol. 1(6).
- [24] Ibid., p. 6.

- abstracción** Las características esenciales de un objeto que lo distinguen de todos los demás tipos de objetos y proporcionan así fronteras conceptuales definidas con nitidez en relación con la perspectiva del observador; el proceso de centrarse en las características esenciales de un objeto. La abstracción es uno de los elementos fundamentales del modelo de objetos.
- abstracción clave** Clase u objeto que forma parte del vocabulario del dominio del problema.
- acción** Una operación que, a efectos prácticos, precisa de un tiempo cero para realizarse. Una acción puede denotar la invocación de un método, el disparo de otro evento, o el lanzamiento o parada de una actividad.
- actividad** Una operación que precisa algún tiempo para completarse.
- actor** Objeto que puede operar sobre otros objetos pero sobre el que nunca operan otros objetos. En algunos contextos, los términos *objeto activo* y *actor* son equivalentes.
- aditiva (mixin)** Clase que incorpora un comportamiento único y específico, utilizada para aumentar el comportamiento de alguna otra clase mediante herencia; el comportamiento de una clase aditiva suele ser ortogonal al de las clases con que se combina.
- agente** Objeto que puede operar sobre otros objetos y sobre el que pueden operar otros objetos. Un agente suele crearse para realizar algún trabajo en nombre de un actor u otro agente.
- amigo** Clase u operación cuya implantación puede hacer referencia a las partes privadas de otra clase, que es la única que puede extender la oferta de amistad.
- análisis orientado a objetos** Método de análisis en el que los requisitos se examinan desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.
- arquitectura** La estructura lógica y física de un sistema, forjada por todas las decisiones de diseño estratégicas y tácticas aplicadas durante el desarrollo.
- arquitectura de módulos** Grafo cuyos vértices representan módulos y cuyos arcos representan relaciones entre esos módulos. La arquitectura de módulos de un sistema se representa por un conjunto de diagramas de módulos.
- arquitectura de procesos** Grafo cuyos vértices representan procesadores y dispositivos y cuyos arcos representan conexiones entre estos procesadores y dispositivos. La arquitectura de procesos de un sistema se representa mediante un conjunto de diagramas de procesos.
- aserción** Expresión booleana de alguna condición cuyo valor de verdad debe conservarse.
- asociación** Relación que denota una conexión semántica entre dos clases.

- atributo** Una parte de un objeto agregado.
- campo** Depósito para parte del estado de un objeto; colectivamente, los campos de un objeto constituyen su estructura. Los términos *campo*, *variable de instancia*, *objeto miembro* y *ranura o slot* son intercambiables.
- capa** Colección de categorías de clases o subsistemas al mismo nivel de abstracción.
- cardinalidad** El número de instancias que puede tener una clase; el número de instancias que participan en una relación entre clases.
- categoría de clases** Colección lógica de clases, algunas de las cuales son visibles para otras categorías de clases, y otras de las cuales están ocultas. Las clases de una categoría de clases colaboran para proporcionar un conjunto de servicios.
- centinela** Expresión booleana aplicada a un evento; si es cierta, la expresión permite al evento causar un cambio en el estado del sistema.
- clase** Conjunto de objetos que comparten una estructura común y un comportamiento común. Los términos *clase* y *tipo* suelen ser (no siempre) equivalentes; una clase es un concepto ligeramente diferente del de un tipo, en el sentido de que enfatiza la clasificación de estructura y comportamiento.
- clase abstracta** Una clase que no tiene instancias. Una clase abstracta se escribe con la intención de que sus subclases concretas añadan elementos nuevos a su estructura y comportamiento, normalmente implantando sus operaciones abstractas.
- clase base** La clase más generalizada en una estructura de clases. La mayoría de las aplicaciones tiene muchas de tales clases raíz. Algunos lenguajes definen una clase base primitiva, que sirve como la superclase última de todas las clases.
- clase concreta** Clase cuya implantación está completa y por tanto puede tener instancias.
- clase contenedor (*container*)** Clase cuyas instancias son colecciones de otros objetos. Las clases contenedor pueden denotar colecciones homogéneas (todos los objetos de la colección son de la misma clase) o heterogéneas (cada uno de los objetos de la colección puede ser de una clase diferente, aunque generalmente todos deben compartir una superclase común). Las clases contenedor se definen la mayoría de las veces como clases parametrizadas, en las que algún parámetro designa la clase de los objetos contenidos.
- clase genérica** Clase que sirve como plantilla para otras clases, en las que la plantilla puede parametrizarse con otras clases, objetos y/u operaciones. Una clase genérica debe ser instanciada (rellenados sus parámetros) antes de que puedan crearse objetos. Las clases genéricas se usan típicamente como clases contenedor. Los términos *clase genérica* y *clase parametrizada* son intercambiables.
- clase parametrizada** Ver *clase genérica*.
- clave** Atributo cuyo valor identifica de forma única un solo objeto—blanco.
- cliente** Objeto que usa los servicios de otro objeto, ya sea operando sobre él o haciendo referencia a su estado.
- colaboración** El proceso por el cual varios objetos cooperan para proporcionar algún comportamiento de nivel superior.
- complejidad espacial** Tiempo absoluto o relativo en el que se completa alguna operación.
- complejidad temporal** El espacio relativo o absoluto consumido por un objeto.
- comportamiento** Cómo actúa y reacciona un objeto, en términos de sus cambios de estado y su paso de mensajes; la actividad exteriormente visible y comprobable de un objeto.
- comprobación estricta de tipos** Una característica de un lenguaje de programación, de

acuerdo con la cual se garantiza que todas las expresiones son consistentes respecto al tipo.

comprobación de tipos (typing) El hacer cumplir la clase de un objeto, lo que previene el intercambio de objetos de diferentes tipos o, como mucho, permite ese intercambio sólo de maneras muy restringidas.

conurrencia Propiedad que distingue un objeto activo de uno que no es activo.

constructor Operación que crea un objeto y/o inicializa su estado.

control de acceso El mecanismo de control de acceso a la estructura o comportamiento de una clase. Los elementos públicos son accesibles por todo el mundo; los protegidos (protected) son accesibles sólo por las subclases, implantación y amigos de la clase en cuestión; los elementos privados son accesibles sólo por la implantación y amigos de la clase que contiene el elemento; los elementos de implantación son accesibles sólo por la implantación de la clase que contiene el elemento.

decisión de diseño estratégica Una decisión de diseño que tiene vastas implicaciones arquitectónicas.

decisión de diseño táctica Decisión de diseño que tiene implicaciones arquitectónicas locales.

delegación El acto por el cual un objeto transmite una operación a otro objeto, para que este la realice en nombre del primero.

descomposición algorítmica El proceso de dividir un sistema en partes, cada una de las cuales representa algún paso pequeño de un proceso más grande. La aplicación de métodos de diseño estructurado conduce a una descomposición algorítmica, cuyo centro de interés está en el flujo de control dentro de un sistema.

descomposición orientada a objetos El proceso de dividir un sistema en partes, cada una de las cuales representa alguna clase u objeto del dominio del problema. La aplicación de métodos de diseño orientado a objetos lleva a una descomposición orientada a objetos, en la que se ve el mundo como una colección de objetos que cooperan con otros para conseguir alguna funcionalidad que se desea.

destructor Operación que libera el estado de un objeto y/o destruye el propio objeto.

diagrama de clases Parte de la notación del diseño orientado a objetos, utilizada para mostrar la existencia de las clases y sus relaciones en el diseño lógico de un sistema. Un diagrama de clases puede representar todo o parte de la estructura de clases de un sistema.

diagrama de interacción Parte de la notación del diseño orientado a objetos, utilizada para mostrar la ejecución de un escenario en el contexto de un diagrama de objetos.

diagrama de módulos Parte de la notación del diseño orientado a objetos, utilizada para mostrar la asignación de clases y objetos a módulos en el diseño físico de un sistema. Un diagrama de módulos puede representar toda la arquitectura de módulos de un sistema o parte de ella.

diagrama de objetos Parte de la notación del diseño orientado a objetos, utilizada para mostrar la existencia de objetos y sus relaciones en el diseño lógico de un sistema. Un diagrama de objetos puede representar toda la estructura de objetos de un sistema o parte de ella, e ilustra principalmente la semántica de los mecanismos del diseño lógico. Un solo diagrama de objetos representa una instantánea de lo que de otro modo es un evento o configuración transitoria de los objetos.

diagrama de procesos Parte de la notación del diseño orientado a objetos, utilizada para mostrar la asignación de procesos a procesadores en el diseño físico de un sistema. Un diagrama de procesos puede representar todo o parte de la arquitectura de procesos de un sistema.

diagrama de transición de estados Parte de la notación del diseño orientado a objetos, utilizada para mostrar el espacio de estados de una clase dada, los eventos que causan una transición de un estado a otro y las acciones que resultan de un cambio de estado.

diccionario de datos Depósito amplio que enumera todas las clases de un sistema.

diseño estructurado Método de diseño que abarca el proceso de descomposición algorítmica.

diseño global circular (round-trip gestalt design) Un estilo de diseño que enfatiza el desarrollo incremental e iterativo de un sistema, mediante el refinamiento de vistas lógicas (diferentes pero consistentes) del sistema como un todo; el proceso de diseño orientado a objetos está guiado por los conceptos del diseño global circular; el diseño global circular es un reconocimiento del hecho de que la visión global de un diseño influye en sus detalles, y los detalles afectan con frecuencia a la visión global.

diseño orientado a objetos Método de diseño que abarca el proceso de descomposición orientada a objetos y una notación para describir modelos lógicos y físicos, así como estáticos y dinámicos, del sistema que se diseña; en concreto, esta notación incluye diagramas de clases, diagramas de objetos, diagramas de módulos y diagramas de procesos.

dispositivo Elemento de hardware que no tiene recursos computacionales.

encapsulamiento El proceso de introducir en el mismo compartimento los elementos de una abstracción que constituyen su estructura y comportamiento; el encapsulamiento sirve para separar el interfaz contractual de una abstracción y su implantación.

enlace Entre dos objetos, una instancia de una asociación.

escenario Esquema de los eventos que provocan algún comportamiento en el sistema.

espacio de estados Enumeración de todos los estados posibles de un objeto. El espacio de estados de un objeto abarca un número indefinido pero finito de estados posibles (aunque no siempre deseables o esperados).

estado Resultados acumulados del comportamiento de un objeto; una de las condiciones posibles en que puede existir un objeto, caracterizada por cantidades definidas que son distintas de otras; en cualquier momento dado, el estado de un objeto abarca todas las propiedades (normalmente estáticas) del objeto más los valores actuales (normalmente dinámicos) de cada una de esas propiedades.

estructura Representación concreta del estado de un objeto. Un objeto no comparte su estado con ningún otro objeto, aunque todos los objetos de la misma clase comparten la misma representación de su estado.

estructura de clases Grafo cuyos vértices representan clases y cuyos arcos representan relaciones entre esas clases. La estructura de clases de un sistema se representa mediante un conjunto de diagramas de clases.

estructura de objetos Grafo cuyos vértices representan objetos y cuyos arcos representan relaciones entre esos objetos. La estructura de objetos de un sistema se representa mediante un conjunto de diagramas de objetos.

evento Algun suceso que puede hacer que cambie el estado de un sistema.

excepción Indicación de que algún invariante no se ha satisfecho o no puede satisfacerse. En C++, se lanza una excepción para abandonar el procesamiento y alertar a algún otro objeto sobre el problema, que a su vez puede capturar la excepción y gestionar el problema.

fichas CRC Clase/Responsabilidades/Colaboradores; una herramienta simple para efectuar tormentas de ideas sobre las abstracciones y mecanismos clave de un sistema.

función Correspondencia entrada/salida que resulta del comportamiento de algún objeto.

función genérica Una operación sobre un objeto. Una función genérica de una clase puede redefinirse en las subclases; así, para un objeto dado, se implementa mediante un conjunto de métodos declarados en varias clases relacionadas mediante su jerarquía de herencias. Los términos *función genérica* y *función virtual* suelen ser equivalentes.

función miembro Operación sobre un objeto, definida como parte de la declaración de una clase; todas las funciones miembro son operaciones, pero no todas las operaciones son funciones miembro. Los términos *función miembro* y *método* suelen ser equivalentes. En algunos lenguajes, las funciones miembro son independientes y pueden redefinirse en una subclase; en otros lenguajes, las funciones miembro no pueden redefinirse, pero sirven como parte de la implementación de una función genérica o virtual, las cuales pueden redefinirse ambas en una subclase.

función virtual Operación sobre un objeto. Una función virtual puede ser redefinida por las subclases; así, para un objeto dado, se implementa mediante un conjunto de métodos declarados en diversas clases que se relacionan mediante su jerarquía de herencias. Los términos *función genérica* y *función virtual* suelen ser intercambiables.

herencia Relación entre clases, en la que una clase comparte la estructura o comportamiento definido en otra (herencia simple) u otras (herencia múltiple) clases. La herencia define una relación «de tipo» entre clases en la que una subclase hereda de una o más superclases generalizadas; una subclase suele especializar a sus superclases aumentando o redefiniendo la estructura y comportamiento existentes.

hilo de control Un solo proceso. El comienzo de un hilo de control es la raíz a partir de la cual ocurren las acciones dinámicas independientes dentro de un sistema; un sistema dado puede tener muchos hilos simultáneos de control, algunos de los cuales pueden aparecer dinámicamente y dejar después de existir. Los sistemas que se ejecutan en múltiples CPUs permiten hilos de control verdaderamente concurrentes, mientras que los sistemas que corren en una sola CPU sólo pueden conseguir la ilusión de hilos concurrentes de control.

identidad La naturaleza de un objeto que lo distingue de todos los demás.

implementación Vista interna de una clase, objeto o módulo, que incluye los secretos de su comportamiento.

instancia Algo a lo cual se le pueden hacer cosas. Una instancia tiene estado, comportamiento e identidad. La estructura y comportamiento de instancias similares se definen en su clase común. Los términos *instancia* y *objeto* son intercambiables.

interfaz Vista externa de una clase, objeto o módulo, que enfatiza su abstracción mientras que oculta su estructura y los secretos de su comportamiento.

invariante Expresión booleana de alguna condición cuyo valor de verdad debe conservarse.

ingeniería directa Producción de código ejecutable a partir de un modelo físico o lógico.

ingeniería inversa Producción de un modelo lógico o físico a partir de código ejecutable.

instanciación Proceso de llenar la plantilla de una clase genérica o parametrizada para producir una clase a partir de la cual pueden crearse instancias.

iterador Operación que permite visitar las partes de un objeto.

jerarquía Clasificación u ordenación de abstracciones. Las dos jerarquías más habituales en un sistema complejo son su estructura de clases (que incluye jerarquías «de

tipo») y su estructura de objetos (que incluye jerarquías «de partes» y de colaboración); pueden encontrarse también jerarquías en las arquitecturas de módulos y procesos de un sistema complejo.

ligadura dinámica Ligadura denota la asociación de un nombre (como una declaración de variable) con una clase; ligadura dinámica es una ligadura en la que la asociación nombre/ clase no se realiza hasta que el objeto designado por el nombre se crea en tiempo de ejecución.

ligadura estática Ligadura denota la asociación de un nombre (como una declaración de variable) con una clase; ligadura estática es una ligadura en la que la asociación nombre/ clase se realiza cuando se declara el nombre (en tiempo de compilación) pero antes de la creación del objeto que designa el nombre.

marco de referencia Colección de clases que proporcionan un conjunto de servicios para un dominio particular; un marco de referencia exporta por tanto una serie de clases y mecanismos individuales que los clientes pueden usar o adaptar.

mecanismo Estructura por la que los objetos colaboran para proporcionar algún comportamiento que satisface un requisito del problema.

mensaje Operación que un objeto realiza sobre otro. Los términos *mensaje*, *método* y *operación* suelen ser equivalentes.

metaclase La clase de una clase; una clase cuyas instancias son a su vez clases.

método Operación sobre un objeto, definida como parte de la declaración de una clase; todos los métodos son operaciones, pero no todas las operaciones son métodos. Los términos *mensaje*, *método* y *operación* suelen ser equivalentes. En algunos lenguajes, los métodos son independientes y pueden redefinirse en una subclase; en otros, los métodos no pueden redefinirse, pero sirven como parte de la implementación de una función genérica o virtual, que pueden redefinirse ambas en una subclase.

modelo de objetos La colección de principios que forman las bases del diseño orientado a objetos; un paradigma de ingeniería del software que enfatiza los principios de abstracción, encapsulamiento, modularidad, jerarquía, tipos, concurrencia y persistencia.

modificador Operación que altera el estado de un objeto.

modismo Expresión peculiar de cierto lenguaje de programación o cultura de aplicaciones, que representa una convención generalmente aceptada para el uso del lenguaje.

modularidad Propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.

módulo Unidad de código que sirve como bloque de construcción para la estructura física de un sistema; una unidad de programa que contiene declaraciones, expresadas en el vocabulario de un lenguaje de programación concreto, que forma la realización física de algunas o de todas las clases y objetos del diseño lógico del sistema. Un módulo suele tener dos partes: su interfaz y su implementación.

monomorfismo Concepto de teoría de tipos, de acuerdo con el cual un nombre (como una declaración de variable) sólo puede denotar objetos de la misma clase.

nivel de abstracción Clasificación relativa de las abstracciones en una estructura de clases, estructura de objetos, arquitectura de módulos o arquitectura de procesos. En términos de su jerarquía «de partes», una abstracción dada está a un nivel de abstracción más alto que otras si se construye sobre las otras; en términos de su jerarquía «de tipos», las abstracciones de alto nivel están generalizadas, y las de bajo nivel están especializadas.

objeto Algo a lo cual se le pueden hacer cosas. Un objeto tiene estado, comportamiento e identidad; la estructura y comportamiento de objetos similares se definen en su clase común. Los términos *instancia* y *objeto* son intercambiables.

- objeto activo** Objeto que contiene su propio hilo de control.
- objeto agregado** Objeto compuesto de otro u otros objetos, cada uno de los cuales se considera parte del objeto agregado.
- objeto con bloqueo** Objeto pasivo cuya semántica se garantiza en presencia de múltiples hilos de control. La invocación de una operación de un objeto con bloqueo bloquea al cliente mientras dure la operación.
- objeto concurrente** Objeto activo cuya semántica se garantiza en presencia de múltiples hilos de control.
- objeto miembro** Repositorio para parte del estado de un objeto; colectivamente, los objetos miembro de un objeto constituyen su estructura. Los términos , , y (slot) son intercambiables.
- objeto pasivo** Un objeto que no contiene su propio hilo de control.
- objeto secuencial** Objeto pasivo cuya semántica se garantiza sólo en presencia de un único hilo de control.
- ocultación de información** Proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales; típicamente, la estructura de un objeto está oculta, así como la implantación de sus métodos.
- operación** Algun trabajo que un objeto realiza sobre otro con el fin de provocar una reacción. Todas las operaciones sobre un objeto concreto pueden encontrarse en forma de subprogramas libres y funciones miembro o métodos. Los términos , *y suelen ser equivalentes.*
- operación abstracta** Operación que es declarada pero no implantada por una clase abstracta. En C++, una operación abstracta se declara como una función miembro virtual pura.
- operación de clases** Operación, como por ejemplo un constructor o destructor, dirigida a una clase en vez de a un objeto.
- papel (rol)** El propósito o capacidad por el que una clase u objeto participa en una relación con otro; el papel de un objeto denota la selección de un conjunto de comportamientos bien definidos en un solo punto del tiempo; un papel es la cara que un objeto presenta al mundo en un momento dado.
- partición** Las categorías de clases o subsistemas que forman parte de un nivel dado de abstracción.
- persistencia** La propiedad de un objeto por la cual su existencia trasciende en el tiempo (es decir, el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (la ubicación del objeto se mueve del espacio de direcciones en el que se creó).
- polimorfismo** Un concepto de teoría de tipos, de acuerdo con el cual un nombre (como una declaración de variable) puede denotar objetos de muchas clases diferentes que se relacionan mediante alguna superclase común; así, todo objeto denotado por este nombre es capaz de responder a algún conjunto común de operaciones de diferentes modos.
- postcondición** Un invariante satisfecho por una operación.
- precondición** Un invariante supuesto por una operación.
- privada (private)** Declaración que forma parte del interfaz de una clase, objeto o módulo; lo que se declara como privado () no es visible para ninguna otra clase, objeto o módulo.
- proceso** Activación de un solo hilo de control.
- procesador** Elemento de hardware que tiene recursos computacionales.
- programación basada en objetos** Método de programación, en el que los programas se

organizan como colecciones cooperativas de objetos cada uno de los cuales representa una instancia de algún tipo, y cuyos tipos son miembros de una jerarquía de tipos unidos mediante relaciones que no son de herencia. En tales programas, los tipos suelen verse como estáticos, mientras que los objetos suelen tener una naturaleza mucho más dinámica, un tanto restringida por la existencia de la ligadura estática y el monomorfismo.

programación orientada a objetos Método de implantación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son miembros de una jerarquía de clases unidas mediante relaciones de herencia. En tales programas, las clases suelen verse como estáticas, mientras que los objetos suelen tener una naturaleza mucho más dinámica, promovida por la existencia de la ligadura dinámica y el polimorfismo.

protegida (protected) Declaración que forma parte del interfaz de una clase, objeto o módulo, pero que no es visible para ninguna otra clase, objeto o módulo excepto los que representan subclases.

protocolo Las formas de las que un objeto puede actuar y reaccionar, constituyendo la vista estática y dinámica externas completas del objeto; el protocolo de un objeto define la parte externa del comportamiento que se permite a un objeto.

pública (public) Declaración que forma parte del interfaz de una clase, objeto o módulo, que es visible para todas las demás clases, objetos y módulos que tienen visibilidad para él.

punto funcional En el contexto de un análisis de requisitos, una actividad simple, visible y comprobable exteriormente.

ranura (slot) Depósito para parte del estado de un objeto; colectivamente, las ranuras de un objeto constituyen su estructura. Los términos *campo*, *variable de instancia*, *objeto miembro* y *ranura* son intercambiables.

responsabilidad Algun comportamiento para el cual se cuenta con un objeto; una responsabilidad denota la obligación de un objeto para proporcionar cierto comportamiento.

restricción Expresión de alguna condición semántica que debe protegerse.

selector Operación que accede al estado de un objeto pero no lo altera.

servicio Comportamiento proporcionado por una parte dada de un sistema.

servidor Objeto que nunca opera sobre otros, sino que sólo se opera sobre él por parte de otros objetos; un objeto que proporciona ciertos servicios.

signatura (signature) Perfil completo de los argumentos formales y tipo de retorno de una operación.

sincronización La semántica de concurrencia de una operación. Una operación puede ser simple (sólo conlleva un hilo de control), síncrona (cita entre dos procesos), de abandono inmediato (un proceso puede citarse con otro sólo si el segundo proceso ya está esperando), de tiempo limitado (un proceso puede citarse con otro, pero esperará por el segundo sólo durante un tiempo determinado), o asíncrono (los dos procesos operan independientemente).

sistema reactivo Sistema dirigido por los eventos; el comportamiento de un sistema reactivo no es una mera correspondencia entrada/salida.

sistema en tiempo real Sistema cuyos procesos esenciales deben satisfacer ciertas restricciones críticas de tiempo. Un sistema en tiempo real riguroso debe ser determinista; el perder una de esas restricciones puede traer consecuencias catastróficas.

sistema transformacional Sistema cuyo comportamiento es una correspondencia entrada/salida.

- subclase** Clase que hereda de una o más clases (llamadas sus *superclases* inmediatas).
- subprograma libre** Procedimiento o función que sirve como operación no primitiva sobre un objeto u objetos de la misma o de distintas clases. Un subprograma libre es cualquier subprograma que no sea método de un objeto.
- subsistema** Colección de módulos, algunos de los cuales son visibles a otros subsistemas y otros de los cuales están ocultos.
- superclase** La clase de la cual hereda otra clase (llamada su *subclase* inmediata).
- tipo** Definición del dominio de valores admisibles que un objeto puede tener y del conjunto de operaciones que pueden realizarse sobre ese objeto. Los términos *clase* y *tipo* suelen ser (no siempre) equivalentes; un tipo es un concepto ligeramente diferente de una clase, en el sentido de que enfatiza la importancia de la conformidad con un protocolo común.
- transición** El paso de un estado a otro estado.
- usar** Referenciar la vista externa de una abstracción.
- utilidad de clase** Colección de subprogramas libres o, en C++, una clase que sólo proporciona miembros static y/o funciones miembro static.
- variable de clase** Parte del estado de una clase. Colectivamente, las variables de clase de una clase constituyen su estructura. Una variable de clase es compartida por todas las instancias de la misma clase. En C++, una variable de clase se declara como un miembro static.
- variable de instancia** Depósito para parte del estado de un objeto. Colectivamente, las variables de instancia de un objeto constituyen su estructura. Los términos , *variable de instancia*, *objeto miembro* y *ranura* o *slot* son intercambiables.
- visibilidad** La capacidad de una abstracción para ver a otra y referenciar por tanto recursos de su vista externa. Una abstracción es visible a otra sólo donde sus ámbitos se solapan. El control de exportación puede restringir además el acceso a las abstracciones visibles.



Bibliografía clasificada

Esta bibliografía clasificada está dividida en once secciones, etiquetadas de la A a la K. Las referencias a ella en los finales de capítulo son de la forma [<etiqueta> <año>]. Por ejemplo, Brooks [H 1975] se refiere al libro de 1975, *The Mythical Man-Month*, de la sección H (Ingeniería del software) de esta bibliografía.

A. Clasificación

- Allen, T., and Starr, T. 1982. *Hierarchy: Perspectives for Ecological Complexity*. Chicago, Illinois: The University of Chicago Press.
- Aquinas, T. *Summa Theologica*. Vol. 19 of Great Books of the Western World. Chicago, IL: Encyclopedia Britannica.
- Aristotle. *Categories*. Vol. 8 of Great Books of the Western World. Chicago, IL: Encyclopedia Britannica.
- Bateson, G. 1979. *Mind and Nature: A Necessary Unity*. New York, New York Bantam Books.
- Brachman, R., McGuinness, D., Patel-Schneider, P., and Resnick, L. *Living with Classic: Principles of Semantic Networks*. San Mateo, California: Morgan Kaufman Publishers.
- Bulman, D. Enero 1991. Refining Candidate Objects. *Computer Language* vol. 8(1).
- Cant, S., Jeffery, D. and Henderson-Sellers, B. Octubre 1991. *A Conceptual Model of Cognitive Complexity of Elements of the Programming Process*. New South Wales, Australia University of New South Wales.
- Classification Society of North America. *Journal of Classification*. New York, NY: Springer-Verlag.
- Coad, P. Septiembre 1992. Object-Oriented Patterns. *Communications of the ACM* vol. 35(9).
- Coad, P. 1993. *The Object Game*. Austin, TX: Object International.
- Coombs, C., Raiffa, H., and Thrall, R. 1954. Some Views on Mathematical Models and Measurement Theory. *Psychological Review* vol. 61(2).
- Courtois, P. Junio 1985. On Time and Space Decomposition of Complex Structures. *Communications of the ACM* vol. 28(6).
- Cunningham, W., and Beck, K. Julio/Agosto 1989. Constructing Abstractions for Object-Oriented Abstractions. *Journal of Object-Oriented Programming* vol. 2(2).

- Darwin, C. *The Origin of Species*. Vol. 49 of Great Books of the Western World. Chicago, IL: Encyclopedia Britannica.
- Descartes, R. *Rules for the Direction of the Mind*. Vol. 71 of Great Books of the Western Word. Chicago IL: Encyclopedia Britannica.
- Flood, R. and Carson, E. 1988. *Dealing with Complexity*. New York, NY: Plenum Press.
- Gould, S. Junio 1992. We Are All Monkey's Uncles. *Natural History*.
- Johnson, R. *Documenting Frameworks using Patterns*. Vancouver, Canada: OOPSLA'92.
- Lakoff, G. 1987. Women, Fire, and Dangerous Things: What Categories Reveal About the Mind. Chicago, IL: The University of Chicago Press.
- Lefrancois, G. 1977. *Of Children: An Introduction to Child Development*. Second edition. Belmont, CA: Wadsworth.
- Lewin, R. 4 Noviembre 1988. Family Relationships Are a Biological Conundrum. *Science* vol. 242.
- Maccoby, M. Diciembre 1991. The Innovative Mind at Work. *IEEE Spectrum* vol. 28(12).
- Maier, H. 1969. *Three Theories of Child Development: The Contributions of Erik H. Erickson, Jean Piaget, and Robert R. Sears, and Their Applications*. New York, NY: Harper and Row.
- Mayo, R. 16 Septiembre 1988. How Many Species Are There on Earth? *Science* vol. 241.
- Michalski, R. and Stepp, R. 1983. *Learning from Observation: Conceptual Clustering, in Machine Learning: An Artificial Intelligence Approach*. ed. R. Michalski, J. Carbonell, and T. Mitchell. Palo Alto, CA: Tioga.
- Miller, G. Marzo 1956. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review* vol. 63(2).
- Minsky, M. 1986. *The Society of Mind*. New York, NY: Simon and Schuster.
- Abril 1970. Form and Content in Computer Science. *Journal of the ACM* vol. 17(2).
- Moldovan, D., and Wu, C. Diciembre 1988. A Hierarchical Knowledge-Bases System for Airplane Classification. *IEEE Transactions on Software Engineering* vol. 14(12).
- Newell, A. 1990. *Unified Theories of Cognition*. Cambridge, Massachusetts: Harvard University Press.
- Newell, A. and Simon, H. 1972. *Human Problem Solving*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Papert, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY: Basic Books.
- Plato. *Statesman*. Vol. 7 of Great Books of the Western World. Chicago, IL: Encyclopedia Britannica.
- Prieto-Díaz, R. and Arango, G. 1991. *Domain Analysis and Software Systems Modeling*. Las Alamitos, California: Computer Society Press of the IEEE.
- Shaw, M. 1989. Larger Scale Systems Require Higher-Level Abstractions. *Proceedings of the Fifth International Workshop on Software Specification and Desing*. IEEE Computer Society.
- 1990. Elements of a Design Language for Software Architecture. Pittsburgh, PA: Carnegie Mellon University.
- 1991. *Heterogeneous Design Idioms for Software Architecture*. Pittsburgh, Pennsylvania: Carnegie Mellon University.
- Siegler, R., and Richards, D. 1982. The Developmen of Intelligence, in *Handbook of Human Intelligence*. ed. R. Sternberg. Cambridge, London: Cambridge University Press.
- Simon, H. 1962. The Architecture of Complexity. *Proceedings of the American Philosophical Society* vol. 106.

- 1982. *The Sciences of the Artificial*. Cambridge, MA: The MIT Press.
- Sowa, J. 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley.
- 1991. *Principles of Semantic Networks*. San Mateo, California: Morgan Kaufman Publishers.
- Stepp, R., and Michalski, R. 1986. Conceptual Clustering of Structured Objects: A Goal-Oriented Approach. *Artificial Intelligence* vol. 28(1).
- Stevens, S. Junio 1946. On the Theory of Scales of Measurement, *Science* vol. 103(2684).
- Stillings, N., Feinstein, M., Garfield, J., Rissland, E., Rosenbaum, D., Weisler, S., and Baker-Ward, L. 1987. *Cognitive Science: An Introduction*. Cambridge, MA: The MIT Press.
- Waldrop, M. 1992. *Complexity: The Emerging Science at the Edge of Order and Chaos*. New York, New York: Simon and Schuster.

B. Análisis orientado a objetos

- Arango, G. Mayo 1989. Domain Analysis: From Art Form to Engineering Discipline. *SIGSOFT Engineering Notes* vol. 14(3).
- Bailin, S. 1988. *Remarks on Object-Oriented Requirements Specification*. Laurel, MD: Computer Technology Associates.
- Bailin, S., and Moore, J. 1987. *An Object-Oriented Specification Method for Ada*. Laurel, MD: Computer Technology Associates.
- Barbier, F. Mayo 1992. Object-Oriented Analysis of Systems through their Dynamical Aspects. *Journal of Object-Oriented Programming* vol. 5(2).
- Borgida, A., Mylogoulos, J., and Wong, H. 1984. Generalization/Specialization as a Basis for Software Specification, in *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. ed. M. Brodie, J. Mylopoulos and J. Schmidt. New York, NY: Springer-Verlag.
- Cernosek, G., Monterio, E., and Pribyl, W. 1987. *An Entity-Relationship Approach to Software Requirements Analysis for Object-Based Development*. Houston, TX: McDonnell Douglas Astronautics.
- Coad, P. Verano 1989. OOA: Object-Oriented Analysis. *American Programmer*, vol. 2(7-8).
- Abril 1990. *New Advances in Object-Oriented Analysis*. Austin, Texas: Object International.
- Coad, P. and Yourdon, E. 1991. *Object-Oriented Analysis*, Second edition. Englewood Cliffs, New Jersey: Yourdon Press.
- Dahl, O-J. 1987. Object-Oriented Specifications, in *Research Directions in Object-Oriented Programming*. ed. B. Schriever and P. Wegner. Cambridge, MA: The MIT Press.
- deChampeaux, D. Abril 1991. *A Comparative Study of Object-Oriented Analysis Methods*. Palo Alto, California: Hewlett-Packard Laboratories.
- Abril 1991. *Object-Oriented Analysis and Top-Down Software Development*. Palo Alto, California: Hewlett-Packard Laboratories.
- DeMarco, T. 1979. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice-Hall.
- Embley, D., Kurtz, B. and Woodfield, S. 1992. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Englewood Cliffs, New Jersey: Yourdon Press.

- EVB Software Engineering, 1989. *Object-Oriented Requirements Analysis*. Frederick, MD.
- Gane, C., and Sarson, T. 1979. *Structured Systems Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- Hatley, D., and Pirbhai, I. 1988. *Strategies for Real-Time System Specification*. New York, NY: Dorset House.
- Ho, D., and Parry, T. Julio 1991. *The Hewlett-Packard Method of Object-Oriented Analysis*. Palo Alto, California: Hewlett-Packard Laboratories.
- Iscoe, N. 1988. *Domain Models for Program Specification and Generation*. Austin, TX: University of Texas.
- Iscoe, N., Browne, J., and Werth, J. 1989. *Modeling Domain Knowledge: An Object-Oriented Approach to Program Specification and Generation*. Austin, TX: The University of Texas.
- Lang, N. Enero 1993. *Shlaer-Mellor Object-Oriented Analysis Rules*. Software Engineering Notes vol. 18(1).
- Marca, D., and McGowan, C. 1988. *SADTTM: Structured Analysis and Design Technique*. New York, NY: McGraw-Hill.
- Martin, J., and Odell, J. 1992. *Object-Oriented Analysis and Design*. Englewood Cliffs, New Jersey; Prentice Hall.
- McMenamin, S. and Palmer, J. 1984. *Essential Systems Analysis*. New York, NY: Yourdon Press.
- Mellor, S., Hecht, A., Tryon, D., and Hywari, W. Septiembre 1988. *Object-Oriented Analysis: Theory and Practice, Course Notes*. San Diego, CA: OOPSLA'88.
- Moore, J., and Bailin, S. 1988. *Position Paper on Domain Analysis*, Laurel, MD: Computer Technology Associates.
- Page-Jones, M., and Weiss, S. Verano 1989. Synthesis: An Object-Oriented Analysis and Design Method. *American Programmer* vol. 2(7-8).
- Rubin, K., and Goldberg, A. Septiembre 1992. *Object Behavior Analysis*. Communications of the ACM vol. 35(9).
- Saeki, M., Horai, H., and Enomoto, H. Mayo 1989. Software Development Process from Natural Language Specification. *Proceedings of the 11th International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.
- Shemer, I. Junio 1987. Systems Analysis: A Systemic Analysis of a Conceptual Model. *Communications of the ACM* vol. 30(6).
- Shlaer, S. and Mellor, S. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, New Jersey: Yourdon Press.
- Julio 1989. An Object-Oriented Approach to Domain Analysis. *Software Engineering Notes* vol. 14(5).
- Verano 1989. Understanding Object-Oriented Analysis. *American Programmer* vol. 2(7-8).
- 1992. *Object Lifecycles: Modeling the World in States*. Englewood Cliffs, New Jersey: Yourdon Press.
- Stoecklin, S., Adams, E., and Smith, S. 1987. *Object-Oriented Analysis*. Tallahassee, FL: East Tennessee State University.
- Sully, P. Verano 1989. Structured Analysis: Scaffolding for Object-Oriented Development. *American Programmer* vol. 2(7-8).
- Tsai, J., and Ridge, J. Noviembre 1988. Intelligent Support for Specifications Transformation. *IEEE Software* vol. 5(6).
- Varyard, R. 1984. *Pragmatic Data Analysis*. Oxford, England: Blackwell Scientific Publications.

- Ward, P. Marzo 1989. How to Integrate Object Orientation with Structured Analysis and Design. *IEEE Software* vol. 6(2).
- Weinberg, G. 1988. *Rethinking Systems Analysis and Design*. New York, NY: Dorset House.

C. Aplicaciones orientadas a objetos

- Abdali, K., Cherry, G., and Soiffer, N. Noviembre 1986. A Smalltalk System for Algebraic Manipulation. *SIGPLAN Notices* vol. 21(11).
- Abdel-Hamid, T., and Madnick, S. Diciembre 1989. Lessons Learned from Modeling the Dynamics of Software Development. *Communications of the ACM* vol. 32(12).
- Almes, G., and Holman, C. Septiembre 1987. Edmas: An Object-Oriented, Locally Distributed Mail System. *IEEE Transactions on Software Engineering* vol. SE-13(9).
- Anderson, D. Noviembre 1986. Experience with Flamingo: A Distributed, Object-Oriented User Interface System. *SIGPLAN Notices* vol. 21(11).
- Archer, J., and Devlin, M. 1987. *Rational's Experience Using Ada for Very Large Systems*. Mountain View, CA: Rational.
- Bagrodia, R., Chandy, M., and Misra, J. Junio 1987. A Message-Based Approach to Discrete-Event Simulation. *IEEE Transactions on Software Engineering* vol. SE-13(6).
- Barry, B. Octubre 1989. Prototyping a Real-Time Embedded System in Smalltalk. *SIGPLAN Notices* vol. 24(10).
- Barry, B., Altoft, J., Thomas, D., and Wilson, M. Octubre 1987. Using Objects to Design and Build Radar ESM Systems. *SIGPLAN Notices* vol. 22(12).
- Basili, V., Caldiera, G., and Cantone, G. Enero 1992. A Reference Architecture for the Component Factory. *ACM Transactions on Software Engineering and Methodology* vol. 1(1).
- Batory, D., and O'Malley, S. Octubre 1992. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology* vol. 1(4).
- Bezivin, J. Octubre 1987. Some Experiments in Object-Oriented Simulation. *SIGPLAN Notices* vol. 22(12).
- Bhaskar, K., and Peckol, J. Noviembre 1986. Virtual Instruments: Object-Oriented Program Synthesis. *SIGPLAN Notices* vol. 21(11).
- Bihair, T., and Gopinath, P. Diciembre 1992. Object-Oriented Real-Time Systems: Concepts and Examples. *IEEE Computer* vol. 25(12).
- Bjornerstedt, A., and Britts, S. Septiembre 1988. AVANCE: An Object Management System. *SIGPLAN Notices* vol. 23(11).
- Bobrow, D., and Stefik, M. Febrero 1986. Perspectives on Artificial Intelligence Programming. *Science* vol. 231.
- Boltuck-Pasquier, J., Grossman, E., and Collaus, G. Agosto 1988. Phototyping an Interactive Electronic Book System Using an Object-Oriented Approach. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Bonar, J., Cunningham, R. and Schultz, J. Noviembre 1986. An Object-Oriented Architecture of Intelligent Tutoring Systems. *SIGPLAN Notices* vol. 21(11).
- Booch, G. 1987. *Software Components with Ada: Structures, Tools, and Subsystems*. Menlo Park, CA: Benjamin/Cummings.

- Borning, A. Octubre 1981. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems* vol. 3(4).
- Bowman, W. and Flegal, B. Agosto 1981. ToolBox: Smalltalk Illustration System. *Byte* vol. 6(8).
- Britcher, R. and Craig, J. Mayo 1986. Using Modern Design Practices to Upgrade Aging Software Systems. *IEEE Software* vol. 3(3).
- Britton, K. and Parnas, D. Diciembre 8, 1981. *A-7E Software Module Guide*, Report 4702. Washington, D.C.: Naval Research Laboratory.
- Brooks, R. 1987. *A Hardware Retargetable Distributed Layered Architecture for Mobile Robot Control*. Cambridge, Massachusetts MIT Artificial Intelligence Laboratory.
- Brooks, R. and Flynn, A. Junio 1989. *Fast, Cheap, and Out of Control A Robot Invasion of the Solar System*. Cambridge, Massachusetts: MIT Artificial Intelligence Laboratory.
- Bruck, D. 1988. Modeling of Control Systems with C++ and PHIGS. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Budd, T. Enero 1989. The Design of an Object-Oriented Command Interpreter. *Software—Practice and Experience* vol. 19(1).
- C++ Booch Components Class Catalog. 1992. Santa Clara, CA: Rational.
- Call, L., Cohrs, D., and Miller, B. Octubre 1987. CLAM - an Open System for Graphical User Interfaces. *SIGPLAN Notices* vol. 22(12).
- Campbell, R., Islam, N., and Madany, P. 1992. The Design of an *Object-Oriented Operating System: A Case Study of Choices*. Vancouver, Canada: OOPSLA'92.
- Caplinger, M. Octubre 1987. An Information System Based on Distributed Objects. *SIGPLAN Notices* vol. 22(12).
- Cargill, T. Noviembre 1986. Pi: A Case Study in Object-Oriented Programming. *SIGPLAN Notices* vol. 21(11).
- Carroll, M. Septiembre 1990. *Building Reusable C++ Components*. Murray Hills, New Jersey: AT&T Bell Laboratories.
- Cmelik, R., and Genani, N. Mayo 1988. Dimensional Analysis with C++. *IEEE Software* vol. 5(3).
- Coggins, J. Septiembre 1990. *Design and Management of C++ Libraries*. Chapel Hill, North Carolina: University of North Carolina.
- Cointe, P., Briot, J., and Serpette, B. 1987. The Formes System: A Musical Application of Object-Oriented Concurrent Programming, in *Object-Oriented Concurrent Programming*. ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Collins, D. 1990. *What is and Object-Oriented User Interface?* Thornwood, New York: IBM Systems Research Education Center.
- Comeau, G. Marzo 1991. C++ In the Real World Interviews with C++ Application Developers. *The C++ Repor* vol. 3(3).
- Coplien, J. Septiembre 1991. Experience with CRC Cards in AT&T. *The C++ Report* vol. 3(8).
- Coutaz, J. Septiembre 1985. Abstractions for User Interface Design. *IEEE Computer* vol. 18(9).
- Custer, H. 1993. *Inside Windows NT*. Redmond, Washington: Microsoft Press.
- Dasgupta, P. Noviembre 1986. A Probe-Based Monitoring Scheme for an Object-Oriented Operating System. *SIGPLAN Notices* vol. 21(11).
- Davidson, C., and Moseley, R. 1987. *An Object-Oriented Real-Time Knowledge-Based System*. Albuquerque, NM: Applied Methods.

- Davis, J., and Morgan, T. Enero 1993. Object-Oriented Development at Brooklyn Union Gas. *IEEE Software* vol. 10(1).
- deChampeaux, D., Anderson, A., Lerman, D., Gasperina, M., Feldhausen E., Glei, M., Fulton, F., Groh, C., Houston, D., Monroe, C., Raj, Rommel and Shultheis, D. Octubre 1991. *Case Study of Object-Oriented Software Development*. Palo Alto, California Hewlett-Packard Laboratories.
- Dietrich, W., Nackman, L., and Gracer, F. Octubre 1989. Saving a Legacy with Objects. *SIGPLAN Notices* vol. 24(10).
- Dijkstra, E. Mayo 1968. The Structure of the «THE» Multiprogramming System. *Communications of the ACM* vol. 11(5).
- Durand, G., Benkiran, A., Durel, C., Nga, H., and Tag, M. 9 Marzo 1988. *Distributed Mail Service in CSE System*. Paris, France: Synergie Informatique et Development.
- Englemore, R., and Morgan, T. 1988. *Blackboard Systems* Wokingham, England: Addison-Wesley.
- Epstein, D., and LaLonde, W. Septiembre 1988. A Smalltalk Window System Based on Constraints. *SIGPLAN Notices* vol. 23(11).
- Ewing, J. Noviembre 1986. An Object-Oriented Operating System Interface. *SIGPLAN Notices* vol. 21(11).
- Fenton, J., and Beck, K. Octubre 1989. Playground: An Object-Oriented Simulation System with Agent Rules for Children of All Ages. *SIGPLAN Notices* vol. 24(10).
- Fischer, G. 1987. *An Object-Oriented Construction and Tool Kit for Human-Computer Communication*. Boulder, CO: University of Colorado Department of Computer Science and Institute of Cognitive Science.
- Foley, J., and Van Dam, A. 1982. *Fundamentals of Interactive Computer Graphics*. Reading MA: Addison-Wesley.
- Frankowski, E. 20 Marzo 1986. *Advantages of the Object Paradigm for Prototyping*. Golden Valley, MN: Honeywell.
- Freburger, K. Octubre 1987. RAPID: Prototyping Control Panel Interfaces. *SIGPLAN Notices* vol. 22(12).
- Freitas, M., Moreira, A., and Guerreiro, P. Julio/Agosto 1990. Object-Oriented Requirements Analysis in an Ada Project. *Ada Letters* vol. X(6).
- Funk, D. 1986. Applying Ada to Beech Starship Avionics. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.
- Garrett, N., and Smith, K. Noviembre 1986. Building a Timeline Editor from Prefab Parts: The Architecture of an Object-Oriented Application. *SIGPLAN Notices* vol. 21(11).
- Goldberg, A. 1978. *Smalltalk in the Classroom*. Palo Alto, CA: Xerox Palo Alto Research Center.
- Goldberg, A., and Rubin, K. Octubre 1990. Taming Object-Oriented Technology. *Computer Language* vol. 7(10).
- Goldstein, N., and Alger, J. 1992. *Developing Object-Oriented Software for the Macintosh*. Reading Massachusetts: Addison-Wesley Publishing Company.
- Gorlen, K. Diciembre 1987. An Object-Oriented Class Library for C++ Programs. *Software-Practice and Experience* vol. 17(12).
- Gray, L. 1987. *Transferring Object-Oriented Design Techniques into Use: AWIS Experience*. Fairfax, VA: TRW Federal Systems Group.
- Grimshaw, A., and Liu, J. Octubre 1987. Mentat: An Object-Oriented Macro Data Flow System. *SIGPLAN Notices* vol. 22(12).

- Grossman, M., and Ege, R. Octubre 1987. Logical Composition of Object-Oriented Interfaces. *SIGPLAN Notices* vol. 22(12).
- Gutfreund, S. Octubre 1987. ManipIcons in ThinkerToy. *SIGPLAN Notices* vol. 22(12).
- Gwinn, J. Febrero 1992. Object-Oriented Programs in Realtime. *SIGPLAN Notices* vol. 27(2).
- Harrison, W., Shilling, J., and Sweeney, P. Octubre 1989. Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm. *SIGPLAN Notices* vol. 24(10).
- Hekmatpour, A., Orailoglu, A., and Chau, P. Abril 1991. Hierarchical Modeling of the VLSI Design Process. *IEEE Expert* vol. 6(2).
- Hollowell, G. Noviembre 1991. Leading the U.S. Semiconductor Manufacturing Industry Toward an Object-Oriented Technology Standard. *Hotline on Object-Oriented Technology* vol. 3(1).
- Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., and Doyle, K. Septiembre 1988. Fabrik: A Visual Programming Environment. *SIGPLAN Notices* vol. 23(11).
- Jacky, J., and Kalet, I. Noviembre 1986. An Object-Oriented Approach to a Large Scientific Application. *SIGPLAN Notices* vol. 21(11).
- Jacobson, I. Enero 1993. Is Object Technology Software's Industrial Platform? *IEEE Software* vol. 10(1).
- Jerrell, M. Octubre 1989. Function Minimization and Automatic Differentiation using C++. *SIGPLAN Notices* vol. 24(10).
- Johnson, R., and Foote, B. Junio/Julio 1988. Designing Reusable Classes. *Journal of Object-Oriented Programming* vol. 1(2).
- Jones, M., and Rashid, R. Noviembre 1986. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. *SIGPLAN Notices* vol. 21(11).
- Jurgen, R. Mayo 1991. Smart Cars and Highways Go Global. *IEEE Spectrum* vol. 28(5).
- Kamath, Y., and Smith, J. Noviembre/Diciembre 1992. Experiences in C++ and O-O Design. *Journal of Object-Oriented Programming* vol. 5(7).
- Kay, A., and Goldberg, A. Marzo 1977. Personal Dynamic Media. *IEEE Computer*.
- Kerr, R., and Percival, D. Octubre 1987. Use of Object-Oriented Programming in a Time Series Analysis System. *SIGPLAN Notices* vol. 22(12).
- Kiyooka, G. Diciembre 1992. Object-Oriented DLLs. *Byte* vol. 17(14).
- Kozaczynski, W., and Kuntzmann-Combelle, A. Enero 1993. What it Takes to Make O-O Work. *IEEE Software* vol. 10(1).
- Krueger, C. Junio 1992. Software Reuse. *ACM Computing Surveys* vol. 24(2).
- Kuhl, F. 1988. *Object-Oriented Design for a Workstation for Air Traffic Control*. McLean, VA: The MITRE Corporation.
- LaPolla, M. 1988. *On the Classification of Object-Oriented Design: The Object-Oriented Design of the AirLand Battle Management Menu System*. Austin, TX: Lockheed Software Technology Center.
- Lea, D. 12 Agosto 1988. *User's Guide to GNU C++ Library*. Cambridge, MA: Free Software Foundation.
— 1988. The GNU C++ Library. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Leathers, B. Julio 1990. Cognos and Eiffel A Cautionary Tale. *Hotline on Object-Oriented Technology* vol. 1(9).
- Ledbetter, L., and Cox, B. Junio 1985. Software-ICs. *Byte* vol. 10(6).
- Lee, K., and Rissman, M. Febrero 1989. *An Object-Oriented Solution Example: A Flight Simulator Electrical System*. Pittsburgh, PA: Software Engineering Institute.

- Lee, K., Rissman, M., D'Ippolito, R., Plinta, C., and Van Scy, R. Diciembre 1987. *An OOD Paradigm for Flight Simulators*, Report CMU/SEI-87-TR-43. Pittsburgh, PA: Software Engineering Institute.
- Levy, P. 1987. *Implementing Systems Software in Ada*. Mountain View, CA: Rational.
- Lewis, J., Henry, S., Kafura, D., and Shulman, R. Julio/Agosto 1992. On the Relationship Between the Object-Oriented Paradigm and Software Reuse: An Empirical Investigation. *Journal of Object-Oriented Programming* vol. 5(4).
- Linton, M., Vlissides, J., and Calder, P. Febrero 1989. Composing User Interfaces with InterViews, *IEEE Computer* vol. 22(2).
- Liu, L., and Horowitz, E. Febrero 1989. Object Database Support for a Software Project Management Environment. *SIGPLAN Notices* vol. 24(2).
- Locke, D., and Goodenough, J. 1988. *A Practical Application of the Ceiling Protocol in a Real-Time System*, Report CMU/SEI-88-SR-3. Pittsburgh, PA: Software Engineering Institute.
- Love, T. 1993. *Object-Lessons*. New York, New York: SIGS Publications.
- Lu, Cary. Diciembre 1992. Objects For End Users. *Byte* vol. 17(14).
- Madany, P., Leyens, D., Russo, V., and Campbell, R. 1988. A C++ Class Hierarchy for Building UNIX-like File Systems. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Madduri, H., Raeuchle, T., and Silverman, J. 1987. *Object-Oriented Programming for Fault-Tolerant Distributed Systems*. Golden Valley, MN: Honeywell Computer Science Center.
- Maloney, J., Borning, A., and Freeman-Benson, B. Octubre 1989. Constraint Technology for User Interface Construction in ThingLab II. *SIGPLAN Notices* vol. 24(10).
- McDonald, J. Octubre 1989. Object-Oriented Programming for Linear Algebra. *SIGPLAN Notices* vol. 24(10).
- Mentor's Lessons in the School of Hard Knocks. Enero 25, 1993. *Business Week*.
- Meyrowitz, N. Noviembre 1986. Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework, *SIGPLAN Notices* vol. 21(11).
- Miller, M., Cunningham, H., Lee, C., and Vegdahl, S. Noviembre 1986. The Application Accelerator Illustration System. *SIGPLAN Notices* vol. 21(11).
- Mohan, L., and Kashyap, R. Mayo 1988. An Object-Oriented Knowledge Representation for Spatial Information. *IEEE Transactions on Software Engineering* vol. 14(5).
- Morgan, T., and Davis, J. Marzo 1991. Large-Scale Object Systems Development. *Hotline on Object-Oriented Technology* vol. 2(5).
- Mraz, R. Diciembre 1986. *Performance Evaluation of Parallel Branch and Bound Search with the Intel iPSC Hypercube Computer*. Wright-Patterson Air Force Base, Ohio: Air Force Institute of Technology.
- Muller, H., Rose, J., Kempf, J., and Stansbury, T. Octubre 1989. The Use of Multimethods and Method Combination in a CLOS-Based Window Interface. *SIGPLAN Notices* vol. 24(10).
- Murphy, E. Diciembre 1988. All Aboard for Solid State. *IEEE Spectrum* vol. 25(13).
- Nelson, J. Septiembre 1992. Applying Object-Oriented Analysis and Design. *Communications of the ACM* vol. 35(9).
- NeXT Embraces a New Way of Programming. 25 de Noviembre 1988. *Science* vol. 242.
- Orden, E. 1987. Application Talk. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1(1). Everette, WA: Object-Oriented Programming for Smalltalk Application Developers Association.

- Orfali, R., and Harkey, D. 1992. *Client/Server Programming with OS/2*. New York, New York: Van Nostrand Reinhold.
- Oshima, M., and Shirai, Y. Julio 1983. Object Recognition Using Three-Dimensional Information. *IEEE Transactions on Pattern Analysis and Machine Intelligence* vol. 5(4).
- Page, T., Berson, S., Cheng, W., and Muntz, R. Octubre 1989. An Object-Oriented Modeling Environment. *SIGPLAN Notices* vol. 24(10).
- Pashtan, A. 1982. Object-Oriented Operating Systems: An Emerging Design Methodology. *Proceedings of the ACM '82 Conference*. New York, NY: Association of Computing Machinery.
- Piersol, K. Noviembre 1986. Object-Oriented Spreadsheets: The Analytic Spreadsheet Package. *SIGPLAN Notices* vol. 21(11).
- Pinson, L., and Wiener, R. 1990. *Applications of Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley Publishing Company.
- Pittman, M. Enero 1993. Lessons Learned in Managing Object-Oriented Development. *IEEE Software* vol. 10(1).
- Plinta, C., Lee, K., and Rissman, M. 29 de Marzo 1989. A Model Solution for C3I: Message Translation and Validation. Pittsburgh, PA: Software Engineering Institute.
- Pope, S. Abril/Mayo 1988. Building Smalltalk-80-based Computer Music Tools. *Journal of Object-Oriented Programming* vol. 1(1).
- Raghavan, R. 1990. *Taming Windows 3.0 and DOS Using C++*. Lake Oswego, Oregon: Wyatt Software.
- Rockwell International. 1989. *Rockwell Advanced Railroad Electronic Systems*. Cedar Rapids, IA.
- Rombach, D. Marzo 1990. Design Measurement: Some Lessons Learned. *IEEE Software* vol. 7(2).
- Rubin, K., Jones, P., Mitchell, C., and Goldstein, T. Septiembre 1988. A Smalltalk Implementation of an Intelligent Operator's Associate. *SIGPLAN Notices* vol. 23(11).
- Rubin, R., Walker, J., and Golin, E. Octubre 1990. Early Experience with the Visual Programmer's WorkBench. *IEEE Transactions on Software Engineering* vol. 16(10).
- Rusconi, E., and Fraley, R. 1983. ID: An Intelligent Information Dictionary System, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Russo, V., Johnston, G., and Campbell, R. Septiembre 1988. Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. *SIGPLAN Notices* vol. 23(11).
- Sampson, J., and Womble, B. 1988. *SEND: Simulation Environment for Network Design*. Dallas, TX: Southern Methodist University.
- Santori, M. Agosto 1990. An Instrument that Isn't Really. *IEEE Spectrum* vol. 27(8).
- Scaletti, C., and Johnson, R. Septiembre 1988. An Interactive Environment for Object-Oriented Music Composition and Sound Synthesis. *SIGPLAN Notices* vol. 23(11).
- Schindler, J., and Joy, S. Febrero 1992. *An Introduction to Object Technology at Liberty Mutual*. Liberty Mutual Information Systems Research and Development.
- Schoen, E., Smith, R., and Buchanan, B. Diciembre 1988. Design of Knowledge-Based Systems with a Knowledge-Based Assistant. *IEEE Transactions on Software Engineering* vol. 14(12).
- Schulert, A., and Erf, K. 1988. Open Dialogue: Using an Extensible Retained Object Workspace to Support a UIIMS. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.

- Scott, R., Reddy, P., Edwards, R., and Campbell, D. 1988. GPIO: Extensible Objects for Electronic Design. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Smith, R., Barth, P., and Young, R. 1987. A Substrate for Object-Oriented Interface Design. *Research Directions in Object-Oriented Programming*. Cambridge, MA: The MIT Press.
- Smith, R., Dinitz, R., and Barth, P. Noviembre 1986. Impulse-86: A Substrate for Object-Oriented Interface Design. *SIGPLAN Notices* vol. 21(11).
- Sneed, H., and Gawron, W. 1993. The Use of the Entity/Relationship Model as a Schema for Organizing the Data Processing Activities at the Bavarian Motor Works, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Snodgrass, R. 1987. An Object-Oriented Command Language, in *Object-Oriented Computing Implementations* vol. 2. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Software Made Simple. 30 de Septiembre, 1991. *Business Week*.
- Sridhar, S. Septiembre 1988. Configuring Stand-Alone Smalltalk-80 Applications. *SIGPLAN Notices* vol. 23(11).
- Stadel, M. Enero 1991. Object-Oriented Programming Techniques to Replace Software Components on the Fly in a Running Program. *SIGPLAN Notices* vol. 26(1).
- Stevens, A. 1992. *C++ Database Development*. New York, New York: MIS Press.
- Stokes, R. 1988. Prototyping Database Applications with a Hybrid of C++ and 4GL. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Szcur, M., and Miller, P. Septiembre 1988. Transportable Applications Environment(TAE) PLUS: Experiences in Objectively Modernizing a User Interface Environment. *SIGPLAN Notices* vol. 23(11).
- Szekely, P., and Myers, B. Septiembre 1988. A User Interface Toolkit Based on Graphical Objects and Constraints. *SIGPLAN Notices* vol. 23(11).
- Tanner, J. 1 Abril 1986. *Fault Tree Analysis in an Object-Oriented Environment*. Mountain View, CA: IntelliCorp.
- Taylor, D. 1992. *Object-Oriented Information Systems*. New York, New York John Wiley and Sons.
- Temte, M. Noviembre/Diciembre 1984. Object-Oriented Design and Ballistics Software. *Ada Letters* vol. 4(3).
- Tripathi, A. and Aksit, M. Noviembre/Diciembre 1988. Communication, Scheduling, and Resource Management in SINA. *Journal of Object-Oriented Programming* vol. 1(4).
- Tripathi, A., Ghonami, A., and Schmitz, T. 1987. Object Management in the NEXUS Distributed Operating System. *Proceedings of the Thirty-second IEEE Computer Society International Conference*. New York, NY: Computer Society Press of the IEEE.
- Ursprung, P. and Zehnder, C. 1983. HIQUEL: An Interactive Query Language to Define and Use Hierarchies, in *Entity-relationship Approach to Software Engineering*. Ed. C. Davis et. al. Amsterdam, The Netherlands: Elsevier Science.
- Van der Meulen, P. Octubre 1987. INSIST: Interactive Simulation in Smalltalk. *SIGPLAN Notices* vol. 22(12).
- Vernon, V. Septiembre/Octubre 1989. The Forest for the Tress. *Programmer's Journal* vol. 7(5).
- Vilot, M. Fall 1990. Using Object-Oriented Desing and C++. *The C++ Journal*, vol. 1(1).
- Vines, D. and King, T. 1987. *Experiences in Building a Prototype Object-Oriented Framework in Ada*. Minneapolis, MN: Honeywell.

- Vlissides, J. and Linton, M. 1988. Applying Object-Oriented Design to Structured Graphics. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Volz, R. Mudge, T., and Gal, D. 1987. Using Ada as a Programming Language for Robot-Based Manufacturing Cells, in *Object-Oriented Computing: Concepts* vol. 1. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Walther, S. and Peskin, R. Octubre 1989. Strategies for Scientific Prototyping in Smalltalk. *SIGPLAN Notices* vol. 24(10).
- Wasserman, A. and Pircher, P. Enero 1991. Object-Oriented Structured Desing and C++. *Computer Language* vol. 8(1).
- Weinand, A., Gamma, E., and Marty, R. Septiembre 1988. ET++-An Object-Oriented Application Framework in C++. *SIGPLAN Notices* vol. 23(11).
- Welch, B. Julio/Agosto 1991. Securities Objects-The Complexity. *Object Magazine* vol. 1(2).
- White, S. Octubre 1986. Panel Problem: Software Controller for an Oil Hot Water Heating System. *Proceedings of COMPSAC*. New York, NY: Computer Society Press of the IEEE.
- Wirfs-Brock, R. Septiembre 1988. An Integrated Color Smalltalk-80 System. *SIGPLAN Notices* vol. 23(11).
- Octubre 1991. Object-Oriented Frameworks. *American Programmer* vol. 4(10).
- WOSA Extensions for Financial Services*. Diciembre 1992. Banking Systems Vendor Council.
- Wu, P. Enero 1992. An Object-Oriented Specification for a Compiler. *SIGPLAN Notices* vol. 27(1).
- Yoshida, N. and Hino, K. Septiembre 1988. An Object-Oriented Framework of Pattern Recognition. *SIGPLAN Notices* vol. 23(11).
- Yoshida, T. and Tokoro, M. 31 Marzo 1986. *Distributed Queueing Network Simulation: An Application of a Concurrent Object-Oriented Language*. Yokohama, Japan: Keio University.
- Young, R. Octubre 1987. An Object-Oriented Framework for Interactive Data Graphics. *SIGPLAN Notices* vol. 22(12).

D. Arquitecturas orientadas a objetos

- Athas, W. and Seitz, C. Agosto 1988. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer* vol. 21(8).
- Dahlby, S., Henry, G. Reynolds, D., and Taylor, P. 1982. The IBM System/38: A High Level Machine, in *Computer Structures: Principles and Examples*. Ed. G. Bell and A. Newell. New York, NY: McGraw-Hill.
- Dally, W. and Kajiyama, J. Marzo 1985. An Object-Oriented Architecture. *SIGARCH Newsletter* vol. 13(3).
- Fabry, R. 1987. Capability-Based Addressing, in *Object Oriented Computing: Implementations*, vol. 2. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Flynn, M. Octubre 1980. Directions and Issues in Architecture and Language. *IEEE Computer* vol. 13(10).
- Harland, D. and Beloff, B. Diciembre 1986. Microcoding an Object-Oriented Instruction Set. *Computer Architecture News* vol. 14(5).

- Hills, D. 1985. *The Connection Machine*. Cambridge, Massachusetts: The MIT Press.
- Iliffe, J. 1982. *Advanced Computer Desing*. London, England: Prentice/Hall International.
- Intel. 1981. *iAPX 432 Object Primer*. Santa Clara, CA.
- Ishikawa, Y., and Tokoro, M. Marzo 1984. The Desing of an Objetc-Oriented Architecture. *SIGARCH Newsletter* vol. 12(3).
- Kavi, K. and Chen, D. 1987. Architectural Support for Object-Oriented Languages. *Proceedings of the Thirty-second IEEE Computer Society International Conference*. New York, NY: Computer Society Press of the IEEE.
- Lahtinen, P. Septiembre/Octubre 1982. A Machine Architecture for Ada. *Ada Letters* vol. 2(2).
- Lampson, B. and Pier, K. Enero 1981. A Processor for a High-Performance Personal Computer, in *The Dorado: A High Performance Personal Computer*, Report CSL-81-1. Palo Alto, CA: Xerox Palo Alto Research Center.
- Langdon, G. 1982. *Computer Desing*. San José, CA: Computeach Press.
- Levy, H. 1984. *Capability-Based Computer Systems*. Bedford, MA: Digital Press.
- Lewis, D., Galloway, D., Francis, R., and Thomson, B. Noviembre 1986. Swamp: A fast Processor for Smalltalk-80. *SIGPLAN Notices* vol. 21(11).
- Mashburn, H. 1982. The C.mmp/Hydra Project: An Architectural Overview, in *Computer Structures: Principles and Examples*. ed. G. Bell and A. Newell. New York, NY: McGraw-Hill.
- Myers, G. 1982. *Advances in Computer Architecture*, Second Edition. New York, NY: John Wiley and Sons.
- Rattner, J. 1982. Hardware/Software Cooperation in the iAPX-432. *Proceedings of the Symposium on Architectural Support for Programming Longuages and Operating Systems*. New York, NY: Association of Computing Machinery.
- Rose, J. Septiembre 1988. Fast Dispatch Mechanisms for Stock Hardware. *SIGPLAN Notices* vol. 23(11).
- Samples, D., Ungar, D., and Hilfinger, P. Noviembre 1986. SOAR: Smalltalk Without Bytecodes. *SIGPLAN Notices* vol. 21(22).
- Soltis, R. and Hoffman, R. 1987. Design Considerations for the IBM System/38, in *Object-Oriented Computing Implementations* vol. 2. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Thacker, C., McCreight, E., Lampson, B., Sproull, R., and Boggs, D. Agosto 1979. *Alto: A Personal Computer*, Report CSL-79-11. Palo Alto, CA: Xerox Palo Alto Research Center.
- Ungar, D. 1987. *The Design and Evaluation of a High-Performance Smalltalk System*. Cambridge, MA: The MIT Press.
- Ungar, D. and Patterson, D. Enero 1987. What Price Smalltalk? *IEEE Computer* vol. 20(1).
- Ungar, D., Blau, R., Foley, P., Samples, D. and Patterson, D. Marzo 1984. Architecture of SOAR: Smalltalk on a RICS. *SIGARCH Newsletter* vol. 23(3).
- Wah, B. and Li, G. Abril 1986. Survey on Special Purpose Computer Architectures for AI. *SIGART Newsletter*, no. 96.
- Wulf, W. Enero 1980. Trends in the Desing and Implementation of Programming Languages. *IEEE Computer* vol. 13(1).
- Wulf, W., Levin, R. and Harbison, S. 1981. *HIDRA/C.mmp: An Experimental Computer System*. New York, NY: McGraw-Hill.

E. Bases de datos orientadas a objetos

- Alford, M. 1983. Derivation of Element-Relation-Attribute Database Requirements by Decomposition of System Functions, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam. The Netherlands: Elsevier Science.
- Andleigh, P. and Gretzinger, M. 1982. *Distributed Object-Oriented, Data-Systems Design*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Atkinson, M., Bailey, P., Chisholm, K., Cockshott, P., and Morrison, R. 1983. An Approach to Persistent Programming. *The Computer Journal* vol. 26(4).
- Alkinson, M. and Buneman, P. Junio 1987. Types and Persistence in Database Programming Languages. *ACM Computing Surveys* vol. 19(2).
- Althinson, M. and Morrison, R. Octubre 1985. Procedures as Persistent Data Objets. *ACM Transactions on Programming Languages and Systems* vol. 7(4).
- Atwood, T. Febrero 1991. Object-Oriented Databases. *IEEE Spectrum* vol. 28(2).
- Bachman, C. 1983. The Structuring Capabilities of the Molecular Data Model, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Batini, C. and Lenzerini, M. 1983. A Methodology for Data Schema Integration in the Entity-Relationship Model, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevire Science.
- Beech, D. 1987. Groundwork for an Object Database Model, In *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Septiembre 1988. Intensional Concepts in an Object Database Model. *SIGPLAN Notices* vol. 23(11).
- Bertino, E. 1983. Distributed Database Design Using the Entity-Relationship Model, in *Entity-Relationship Approach to Sofware Engineering*. Ed. C. Davis et al. Amsterdam. The Netherlands: Elsevier Science.
- Blackwell, P., Jajodia, S., and Ng, P. 1983: A View of Database Management Systems as Abstract Data Types, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam. The Netherlands: Elsevier Science.
- Bloom, T. Octubre 1987. Issues in the Desing of Objetc-Oriented Database Programming Languages. *SIGPLAN Notices*, vol. 22(12).
- Bobrow, D., Fogelson, D., and Miller, M. 1987. Definition Groups: Making Sources into First-class Objects, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Brathwaite, K. 1983. An Implementation of a Data Dictionary to Support Databases Designed Using the Entity-Relationship(E-R) Approach, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam. The Netherlands: Elsevier Science.
- Breazeal, J., Blattner, M., and Burton, H. 28 Marzo 1986. *Data Standardization Through the Use of Data Abstraction*. Livermore, CA: Lawrence Livermore National Laboratory.
- Brodie, M. 1984. On the Development of Data Models, in *On Conceptual Modeling: Perspectives from Artificial Intelligence. Databases, and Programming Languages*. Ed. M. Brodie, J. Mylopoulos and J. Schmidt. New York, NY: Springer-Verlag.
- Brodie, M. and Ridjanovic, D. 1984. On the Design and Specification of Database Transactions. *On Conceptual Modeling: Perspectives from Artificial Intelligence, Da-*

- tabases, and Programming Languages.* ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.
- Butterworth, P., Otis, A and Stein J. Octubre 1991. The GemStone Object Database Management System. *Communications of the ACM* vol. 34(19).
- Carlson, C. and Arora, A. 1983. UPM: A Formal Tool for Expressing Database Update Semantics, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam. The Netherlands: Elsevier Science.
- Casanova, M. 1983. Designing Entity-Relationship Schemes for Conventional Information Systems, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Cattell, R. 1991. *Object Data Management*. Reading. Massachusetts: Addison-Wesley Publishing Company.
- Mayo 1983. *Design and Implementation of a Relationship-Entity-Datum Data Model*. Report CSL-83-4. Palo Alto. CA: Xeros Palo Alto Research Center.
- Chen, P. 1983. ER-A Historical Perspective and Future Directions, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Marzo 1976. The Entity-Relationship Model-Toward a Unified View of Data. *ACM Transactions on Database Systems* vol. 1(1).
- Claybrook, B. Claybrook, A., and Williams, J. Enero 1985. Defining Database Views as Data Abstractions. *IEEE Transactions on Software Engineering* vol. SE-11(1).
- D'Cunha, A. and Radhakrishnan, T. 1983. Applications of E-R Concepts to Data Administration, *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam. The Netherlads: Elsevier Science.
- Date, C. 1981, 1983. *An Introduction to Database Systems*. Reading. MA: Addison-Wesley.
- 1986. *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley.
- 1987. *The Guide to the SQL Standard*. Reading, MA: Addison-Wesley.
- Duhl, J. and Damon, C. Septiembre 1988. A Performance Comparison of Object and Relational Database Using the Sun Benchmark. *SIGPLAN Notices* vol. 23(11).
- Harland, D. and Beloff, B. Abril 1987. OBJEKT-A Persistent Object Store with an Integrated Garbage Collector. *SIGPLAN Notices* vol. 22(4).
- Hawryszkiewycz, L. 1984. *Database Analysis and Desing*. Chicago, IL: Science Research Associates.
- Higa, K., Morrison, M., Morrison, J. and Sheng. O. Junio 1992. An Object-Oriented Methodology for Knowledge Base/Database Coupling. *Communications of the ACM* vol. 35(6).
- Hull, R. and King, R. Septiembre 1987. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys* vol. 19(3).
- Jajodia, S., Ng, P., and Springsteel, F. 1983. On Universal and Representative Instances for Inconsistent Databases, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Ketabchi, M. and Berzns, V. Enero 1988. Mathematical Model of Composite Objects and Its Application for Organizing Engineering Databases. *IEEE Transactions on Software Engineering* vol. 14(1).
- Ketabchi, M. and Wiens, R. 1987. Implementation of Persistent Multi-User Object-Oriented Systems. *Proceedings of the Thirty-Second IEEE Computer Society International Conference*. New York, NY: Computer Society Press of the IEEE.
- Khoshafian, S., and Abnous, R. 1990. *Object-Orientation: Concepts, Languages, Databases, User Interfaces*. New York: John Wiley and Sons.

- Kim, W. and Lochovsky, K. 1998. *Object-Oriented Concepts, Databases, and Applications*. Reading. MA: Addison-Wesley.
- Kim, W., Ballou, N., Chou, H., Garze, J., Woelk, D. and Banerjee, J. Septiembre 1988. Integrating an Object-Oriented Programming System with a Database System. *SIGPLAN Notices* vol. 23(11).
- Kim, W., Banerjee, J., Chou, H., Garza, J., and Woelk, D. Octubre 1987. Composite Object Support in an Object-Oriented Database System. *SIGPLAN Notices* vol. 22(12).
- Kung, C. Object Subclass Hierarchy in SQL: A Simple Approach. *Communications of the ACM* vol. 33(7).
- Laenens, E. and Vermeir, D. Agosto 1988. An Overview of OOPS+, An Object-Oriented Database Programming Language. *Proceedings of ECCOP'88; European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. Octubre 1991. The ObjectStore Database System. *Communications of the ACM* vol. 34(10).
- Larson, J. and Dwyer, P. 1983. Defining External Schemas for an Entity-Relationship Database, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Maier, D. and Stein, J. 1987. Development and Implementation of an Object-Oriented DBMS, in *Research Directions in Object-Oriented Programming*. ed. B. Schriener and P. Wegner. Cambridge, MA: The MIT Press.
- Margrave, G., Lusk, E., and Overbeek, R. 1983. Tools for the Creation of IMS Database Designs from Entity-Relationship Diagrams, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Mark, L., and Poussopoulos, N. 1983. Integration of Data, Schema, and Meta-schema in the Context of Self-documenting Data Models, in *Entity-Relationship Approach to Software Engineering* ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Markowitz, V. and Makowsky, J. Agosto 1990. Identifying Extended Entity-Relationship Object Structures in Relational Schemas. *IEEE Transactions on Software Engineering* vol. 16(8).
- Marti, R. 1983. Integrating Database and Program Descriptions Using an ER Data Dictionary, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Merrow, T. and Laursen, J. Octubre 1987. A Pragmatic System for Shared Persistent Objects. *SIGPLAN Notices* vol. 22(12).
- Mitchell, J. and Wegbreit, B. 1977. Schemes: A High-Level Data Structuring Concept, in *Current Trends in Programming Methodology: Data Structuring* vol. 4. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Morrison, R., Atkinson, M., Brown, A., and Dearle, A. Abril 1988. Bindings in Persistent Programming Languages. *SIGPLAN Notices* vol. 23(4).
- Moss, E., Herlihy, M., and Zdonik, S. Septiembre 1988. *Object-Oriented Databases, Course Notes*. San Diego, CA: OOPSLA'88.
- Moss, J. Agosto 1992. Working with Persistent Objects To Swizzle or Not to Swizzle. *IEEE Transactions on Software Engineering* vol. 18(8).
- Nastos, M. Enero 1998. Databases, Etc. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1(2). Everett, WA: Object Oriented Programming for Smalltalk Application Developers Association.

- Navathe, S. and Cheng, A. 1983. A Methodology for Database Schema Mapping from Extended Entity Relationship Models into the Hierarchical Model, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Ontologic. 1987. *Vbase Technical Overview*. Billerica, MA.
- Oracle. 1989. *Oracle for Macintosh: References, Versión 1.1*. Belmont, CA.
- Penny, J. and Stein, J. Octubre 1987. Class Modification in the GemStone Object-Oriente DBMS. *SIGPLAN Notices* vol. 22(12).
- Peterson, R. 1987. Object-Oriented Database Design, in *Object-Oriented Computing: Implementations* vol. 2. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Premperlani, W., Blaha, M., Rumbaugh, J., and Varwig, T. Noviembre 1990. An Object-Oriented Relational Database. *Communications of the ACM* vol. 33(11).
- Sakai, H. 1983. Entity-Relationship Approach to Logical Database Design, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Skarra, A. and Zdonik, S. 1987. Type Evolution in a Object-Oriented Database, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Skarra, A. and Zdonik, S. Noviembre 1986. The Management of Changing Types in an Object-Oriented Database. *SIGPLAN Notices* vol. 21(11).
- Smith, D. and Smith, J. 1980. Conceptual Database Design, in *Tutorial on Software Design Techniques*, Third Edition. ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Smith, J. and Smith, D. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems* vol. 2(2).
- Smith, K. and Zdonik, S. Octubre 1987. Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems. *SIGPLAN Notices* vol. 22(12).
- Stein, J. Marzo 1988. Object-Oriented Programming and Database Design. *Dr. Dobb's Journal* vol. 13(3).
- Marzo 1988. Object-Oriented Programming and Database Design. *Dr. Dobb's Journal of Software Tools for the Professional Programmer*, no. 137.
- Teorey, Y., Yang, D., and Fry, J. Junio 1986. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Surveys* vol. 18(2).
- Thuraisingham, M. Octubre 1989. Mandatory Security in Object-Oriented Database Systems. *SIGPLAN Notices* vol. 24(10).
- Veloso, P. and Furtado, A. 1983. View Constructs for the Specification and Design of External Schemas, in *Entity-Relationship Approach to Software Engineering*. ed C. davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Wiebe, D. Noviembre 1986. A Distributed Repository for Immutable Persistent Objects. *SIGPLAN Notices* vol. 21(11).
- Wiederhold, G. Diciembre 1986. Views, Objects, and Databases. *IEEE Computer* vol. 19(12).
- Wile, D. and Allard, D. Mayo 1982. Worlds: an Organizing Structure for Objects-bases. *SIGPLAN Notices* vol. 19(5).
- Zdonik, S. and Maier, D. 1990. *Readings in Object-Oriented Database Systems*. San Mateo, CA: Morgan Kaufmann.

Zhang, Z. and Mendelzon, A. 1983. A Graphical Query Language for Entity-Relationship Databases, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

F. Diseño orientado a objetos

- Abbott, R. Agosto 1987. Knowledge Abstraction. *Communications of the ACM* vol. 30(8).
- Noviembre 1983. Program Design by Informal English Descriptions. *Communications of the ACM* vol. 26(11).
- Ackroyd, M. and Daum, D. 1991. Graphical Notation for Object-Oriented Design and Programming. *Journal of Object-Oriented Programming* vol. 3(5).
- Alabios, B. Septiembre 1988. Transformation of Data Flow Analysis Models to Object-Oriented Design. *SIGPLAN Notices* vol. 23(11).
- Arnold, P., Bodoff, S., Coleman, D., Gilchrist, H., and Hayes, F. Junio 1991. *An Evaluation of Five Object-Oriented Development Methods*. Bristol, England: Hewlett-Packard Laboratories.
- Bear, S., Allen, P., Coleman, D., and Hayes, F. Graphical Specification of Object-Oriented Systems. *Object-Oriented Programming Systems, Languages, and Applications*. Ottawa, Canada: OOPSLA'90.
- Beck, K. and Cunningham, W. Octubre 1989. A Laboratory for Teaching Object-Oriented Thinking. *SIGPLAN Notices* vol. 24(10).
- Berard, E. 1986. *An Object-Oriented Design Handbook*. Rockville, MD: EVB Software Engineering.
- Berzins, V., Gray, M., and Naumann, D. Mayo 1986. Abstraction-Based Software Development. *Communications of the ACM* vol. 29(5).
- Blaha, M. Abril 1988. Relational Database Design Using an Object-Oriented Methodology. *Communications of the ACM* vol. 31(4).
- Booch, G. Septiembre 1981. Describing Software Design in Ada. *SIGPLAN Notices* vol. 16(9).
- Marzo/Abril 1982. Object-Oriented Design. *Ada Letters* vol. 1(3).
- Febrero 1986. Object-Oriented Development. *IEEE Transactions on Software Engineering* vol. 12(2).
- 1987. *On the Concepts of Object-Oriented Design*. Denver, CO: Rational.
- Verano de 1989. What Is and What Isn't Object-Oriented Design. *American Programmer* vol. 2(7-8).
- Booch, G. and Vilot, M. Object-Oriented Design. *The C++ Report*.
- Booch, G., Jacobson, I., and Kert, N. Septiembre 1988. *Specification and Design Methodologies in Support of Object-Oriented Programming. Course Notes*. San Diego, CA: OOPSLA'88.
- Bowles, A. Noviembre/Diciembre 1991. Evolution Vs Revolution: Should Structured Methods Be Objectified? *Object Magazine* vol. 1(4).
- Boyd, S. Julio/Agosto 1987. Object-Oriented Design and PALMELATM. *Ada Letters*, vol. 7(4).
- Bril, R., deBunje, T., and Ouvry, A. Octubre 1991. *Development of SCORE: Towards the Industrialization of an Object-Oriented Method using the Formal Design Language COLD-I as Notation*. Eindhoven, The Netherlands: Philips Research Labora-

- tories. Brookman, D. Noviembre/Diciembre 1991. SA/SD versus OOD. *Ada Letters* vol. XI(9).
- Bruno, G. and Balsamo, A. Noviembre 1986. Petri Net-Based Object-Oriented Modeling of Distributed Systems. *SIGPLAN Notices* vol. 21(11).
- Buhr, R. 1984. *System Design with Ada*. Englewood Cliffs, NJ: Prentice-Hall.
- 22 de Agosto 1988. *Machine Charts for Visual Prototyping in System Design*. SCE Report 88-2. Ottawa, Canada: Carleton University.
- 14 de Septiembre 1988. *Visual Prototyping in System Design*. SCE Report 88-14. Ottawa, Canada: Carleton University.
- 1989. *System Design with Machine Charts: A CAD Approach with Ada Examples*. Englewood Cliffs, NJ: Prentice-Hall.
- Buhr, R., Karam, G., Haves, C., and Woodside, M. Marzo 1989. Software CAD: A Revolutionary Approach. *IEEE Transactions on Software Engineering* vol. 15(3).
- Bulman, D. Agosto 1989. An Object-Based Development Model. *Computer Language* vol. 6(8).
- Cherry, G. 1987. *PAMELA 2: An Ada-Based Object-Oriented Design Method*. Reston, VA: Thought**Tools.
- 1990. *Software Construction by Object-Oriented Pictures*, Canandaigua, NY: Thought**Tools.
- Clark, R. Junio 1987. Designing Concurrent Objects. *Ada Letters* vol. 7(6).
- Coad, P. Septiembre 1991. OOD Criteria. *Journal of Object-Oriented Programming* vol.(5).
- Coleman, D., Hayes, F., and Bear, S. Diciembre 1990. *Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design*. Bristol, England: Hewlett-Packard Laboratories.
- Comer, E. Julio 1989. *Ada Box Structure Methodology Handbook*. Melbourne, FL: Software Productivity Solutions.
- Constantine, L. Verano 1989. Object-Oriented and Structured Methods: Towards Integration. *American Programmer* vol. 2(7-8).
- CRI, CISI Ingenierie, and Matra. 20 de Junio 1987. *HOOD: Hierarchical Object-Oriented Design*. Paris, France.
- Cribbs, J., Moon, S., and Roe, C. 1992. *An Evaluation of Object-Oriented Analysis and Design Methodologies*. Raleigh, North Carolina: Alcatel Network Systems.
- Cunningham, W. and Beck, K. Noviembre 1986. A Diagram for Object-Oriented Programs. *SIGPLAN Notices* vol. 21(11).
- Davis, N., Irving, M., and Lee, J. *The Evolution of Object-Oriented Design from Concept to Method*. 1988. Surrey, United Kingdom: Logica Space and Defence Systems Limited.
- Dean, H. Mayo 1991. Object-Oriented Design Using Message Flow Decomposition, *Journal of Object-Oriented Programming* vol. 4(2).
- deChampeaux, D., Balzer, B., Bulman, D., Culver-Lozo, K., Jacobson, I., Mellor, S. *The Object-Oriented Software Development Process*. Vancouver, Canada: OOPSLA'92.
- deChampeaux, D., Lea, D., and Faure, P. *The Process of Object-Oriented Design*. Vancouver, Canada: OOPSLA'92.
- Edwards, J. and Henderson-Sellers, B. Noviembre 1991. *A Graphical Notation for Object-Oriented Analysis and Design*. New South Wales, Australia University of New South Wales.
- Felsinger, R. 1987a. *Integrating Object-Oriented Design, Structured Analysis/Structured Design, and Ada for Real-time Systems*. Mt. Pleasant, SC.

- 1987b. *Object-Oriented Design, Course Notes*. Torrance, CA: Data Processing Management Association.
- Fichman, R. and Kemerer, C. Octubre 1992. Object-Oriented and Conventional Analysis and Design Methodologies. *IEEE Computer* vol. 25(10).
- Firesmith, D. Mayo 6, 1986. *Object-Oriented Development*. Fort Wayne, Indiana: Magnavox Electronic Systems Co.
- 1993. *Object-Oriented Requirements Analysis and Logical Design*. New York, New York: John Wiley and Sons.
- Fowler, M. 1992. *A Comparison of Object-Oriented Analysis and Design Methods*. Vancouver, Canada: OOPSLA'92.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1993. A Catalog of Object-Oriented Design Patterns. Cupertino, California: Taligent.
- Gane, C. Verano 1989. Object-Oriented Data/Process Modeling. *American Programmer* vol. 2(7-8).
- Giddings, R. Mayo 1984. Accommodating Uncertainty in Software Design. *Communications of the ACM* vol. 27(5).
- Gomaa, H. Septiembre 1984. A Software Design Method for Real-Time Systems. *Communications of the ACM* vol. 27(9).
- Gossain, S. and Anderson, B. *An Iterative Design Model for Reusable Objects*. Ottawa, Canada: OOPSLA'90.
- Gouda, M., Han, Y., Jensen, E., Johnson, W., and Kain, R. Noviembre 1977. Towards a Methodology of Distributed Computer System Design. *Sixth Texas Conference on Computing Systems*. New York, NY: Association of Computing Machinery.
- Graham, I. 1991. *Object-Oriented Methods*. Workingham, England: Addison-Wesley Publishing Company.
- Grosch, J. Diciembre 1983. Type Derivation Graphs - A Way to Visualize the Type Building Possibilities of Programming Languages. *SIGPLAN Notices* vol. 18(12).
- Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* vol. 8.
- Mayo 1988. On Visual Formalism. *Communications of the ACM* vol. 31(5).
- Henderson-Sellers, B. 1992. *A Book of Object-Oriented Knowledge*. Englewood Cliffs, New Jersey: Prentice Hall.
- Inwood, C. 1992. Analysis versus Design: Is there a Difference? *The C++ Journal* vol. 2(1).
- Jackson, M. Verano 1989. Object-Oriented Software. *American Programmer* vol. 2(7-8).
- Jacobson, I. Agosto 1985. *Concepts for Modeling Large Real-Time Systems*. Academic dissertation. Stockholm, Sweden: Royal Institute of Technology, Department of Computer Science.
- Octubre 1987. Object-Oriented Development in an Industrial Environment. *SIGPLAN Notices* vol. 22(12).
- Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. 1992. *Object-Oriented Software Engineering*. Workingham, England: Addison-Wesley Publishing Company.
- Jamsa, K. Enero 1984. Object-Oriented Design vs. Structured Design - A Student's Perspective. *Software Engineering Notes* vol. 9(1).
- Johnson, R. and Russo, V. Mayo 1991. *Reusing Object-Oriented Designs*. Urbana, Illinois: University of Illinois.
- Jones, A. 1979. The Object Model: A Conceptual Tool for Structuring Software, in *Operating Systems*. ed. R. Bayer et. al. New York, NY: Springer-Verlag.
- Kadie, C. 1986. *Refinement Through Classes: A Development Methodology for Object Oriented Languages*. Urbana, IL: University of Illinois.

- Kaplan, S. and Johnson, R. 21 Julio 1986. *Designing and Implementing for Reuse*. Urbana, IL: University of Illinois, Department of Computer Science.
- Kay, A. Agosto 1969. *The Reactive Engine*. Salt Lake City, Utah: The University of Utah, Department of Computer Science.
- Kelly, J. 1986. A Comparison of Four Design Methods for Real-Time Systems. *Proceedings of the Ninth International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.
- Kent, W. 1983. Fact-Based Data Analysis and Design, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Kim, J. and Lerch, J. 1992. *Towards a Model of Cognitive Processes in Logical Design: Comparing Object-Oriented and Traditional Functional Decomposition Software Methodologies*. Pittsburgh, Pennsylvania: Carnegie Mellon University.
- Ladden, R. Julio 1988. A Survey of Issues to Be Considered in the Development of an Object-Oriented Development Methodology for Ada. *Software Engineering Notes* vol. 13(3).
- Lieberherr, K. and Riel, A. Octubre 1989. Contributions to Teaching Object-Oriented Design and Programming. *SIGPLAN Notices* vol. 24(10).
- Liskov, B. 1980. A Design Methodology for Reliable Software Systems, in *Tutorial on Software Design Techniques*. Third Edition. ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Lorenz, M. 1993. *Object-Oriented Software Development*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Mannino, P. Abril 1987. A Presentation and Comparison of Four Information System Development Methodologies. *Software Engineering Notes* vol. 12(2).
- Martin, B. 1993. *Designing Object-Oriented C++ Applications Using the Booch Method*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Masiero, P. and Germano, F. Julio 1988. JSD As an Object-Oriented Design Method. *Software Engineering Notes* vol. 13(3).
- Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall.
- Meyer, B. Marzo 1987. Reusability: The Case for Object-Oriented Design. *IEEE Software* vol. 4(2).
- 1989. From Structured Programming to Object-Oriented Design: The Road to Eiffel. *Structured Programming* vol. 10(1).
- Mills, H. Junio 1988. Stepwise Refinement and Verification in Box-Structured Systems. *IEEE Computer* vol. 21(6).
- Mills, H., Linger, R., and Hevner, A. 1986. *Principles of Information System Design and Analysis*. Orlando, FL: Academic Press.
- Minkowitz, C. and Henderson, P. Marzo 1987. *Object-Oriented Programming of Discrete Event Simulation Using Petri Nets*. Stirling, Scotland: University of Stirling.
- Monarchi, D. and Puhr, G. Septiembre 1992. A Research Typology for Object-Oriented Analysis and Design. *Communications of the ACM* vol. 35(9).
- Mostow, J. Primavera 1985. Toward Better Models of the Design Process. *AI Magazine* vol. 6(1).
- Moulin, B. 1983. The Use of EPAS/IPSO Approach for Integrating Entity Relationship Concepts and Software Engineering Techniques, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Mullin, M. 1989. *Object-Oriented Program Design with Examples in C++*. Reading, MA: Addison-Wesley.

- Nielsen, K. and Shumate, K. Agosto 1987. Designing Large Real-Time Systems with Ada. *Communications of the ACM* vol. 30(8).
- Nielsen, K. Marzo 1988. *An Object-Oriented Design Methodology for Real-Time Systems in Ada*. San Diego, CA: Hughes Aircraft Company.
- Nies, S. 1986. The Ada Object-Oriented Approach. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.
- Ossher, H. 1987. A Mechanism for Specifying the Structure of Large, Layered, Systems, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Page-Jones, M., Constantine, L., and Weiss, S. Octubre 1990. Modeling Object-Oriented Systems: The Uniform Object Notation. *Computer Language* vol. 7(10).
- Parnas, D. 1979. On the Criteria to be Used in Decomposing Systems into Modules. *Classics in Software Engineering*, ed. E. Yourdon. New York, NY: Yourdon Press.
- Parnas, D., Clements, P., and Weiss, D. Marzo 1985. The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering* vol. SE-11(3).
- Pasik, A. and Schor, M. Enero 1984. Object-Centered Representation and Reasoning. *SIGART Newsletter*, no. 87.
- Rajlich, V. and Silva, J. 1987. *Two Object-Oriented Decomposition Methods*. Detroit, Michigan: Wayne State University.
- Ramamoorthy, C. and Sheu, P. Otoño 1988. Object-Oriented Systems. *IEEE Expert* vol. 3(3).
- Reenskaug, T. Agosto 1981. User-Oriented Descriptions of Smalltalk Systems. *Byte* vol. 6(8).
- Reiss, S. 1987. An Object-Oriented Framework for Conceptual Programming, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Richter, C. Agosto 1986. An Assessment of Structured Analysis and Structured Design. *Software Engineering Notes* vol. 11(4).
- Rine, D. Octubre 1987. A Common Error in the Object Structure of Object-Oriented Methods. *Software Engineering Notes* vol. 12(4).
- Rosenberg, D. and Jennett, P. Julio 1992. Object-Oriented Analysis and Design Methods. *Frameworks* vol. 6(4).
- Ross, R. 1987. *Entity Modeling: Techniques and Application*. Boston, MA: Database Research Group.
- Rosson, M. and Gold, E. Octubre 1989. Problem-Solution Mapping in Object-Oriented Design. *SIGPLAN Notices* vol. 24(10).
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Sahraoui, A. 1987. *Towards a Design Approach Methodology Combining OOP and Petri Nets for Software Production*. Toulouse, France: Laboratoire d'Automatique et d'analyses des systemes du C.N.R.S.
- Sakai, H. 1983. A Method for Entity-Relationship Behavior Modeling, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam. The Netherlands. Elsevier Science.
- Seidewitz, E. Mayo 1985. *Object Diagrams*. Greenbelt, MD: NASA Goddard Space Flight Center.
- Seidewitz, E. and Stark, M. 1986. Towards a General Object-Oriented Software Development Methodology. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.

- Agosto 1986. *General Object-Oriented Software Development*, Report SEL-86-002. Greenbelt, MD: NASA Goddard Space Flight Center.
- Julio/Agosto 1987. Towards a General Object-Oriented Design Methodology: *Ada Letters* vol. 7(4).
- 1988. *An Introduction to General Object-Oriented Software Development*. Rockville, MD: Millennium Systems.
- Shilling, J. and Sweeney, P. Octubre 1989. Three Steps to Views: Extending the Object-Oriented Paradigm. *SIGPLAN Notices* vol 24(10).
- Shlaer, S., Mellor, S., and Hywari, W. 1990 *OODLE: A Language-Independent Notation for Object-Oriented Design*. Berkeley, California: Project Technology, California.
- Shumate, K. 1987. *Layered Virtual Machine/Object-Oriented Design*. San Diego, CA: Hughes Aircraft Company.
- Smith, M. and Tockey, S. 1988. *An Integrated Approach to Software Requirements Definition Using Objects* Seattle, WA: Boeing Commercial Airplane Support Division.
- Solsi, S., and Jones, E. Marzo/Abril 1991. Simple Yet Complete Heuristics for Transforming Data Flow Diagrams into Booch Style Diagrams. *Ada Letters* vol. XI(2).
- Song, X. Mayo 1992. *Comparing Software Design Methodologies Through Process Modeling*. Irvine, California: University of California.
- Stark, M. Abril 1986. *Abstraction Analysis: From Structured Analysis to Object-Oriented Design*. Greenbelt, MD: NASA Goddard Space Flight Center.
- Strom, R. Octubre 1986. A Comparison of the Object-Oriented and Process Paradigms. *SIGPLAN Notices* vol. 21(10).
- Teledyne Brown Engineering. Octubre 1987. *Software Methodology Catalog*, Report MC87-COMM/ADP-0036. Tinton Falls, NJ.
- The Fusion Object-Oriented Analysis and Design Method*. Mayo 1992. Bristol, England: Hewlett Packard Laboratories.
- Thomas, D. Mayo/Junio 1989. In Search of an Object-Oriented Development Process. *Journal of Object-Oriented Programming* vol. 2(1).
- Wahl, S. 13 Diciembre 1988. Introduction to Object-Oriented Software. *C++ Tutorial Program of the USENIX Conference* Denver, CO: USENIX Association.
- Walters, N. Julio/Agosto 1991. An Ada Object-Based Analysis and Design Approach. *Ada Letters* vol. XI(5).
- Wasserman, T., Pircher, P., and Muller, R. Diciembre 1988. *An Object-Oriented Structured Design Method for Code Generation*. San Francisco, CA: Interactive Development Environments.
- Verano 1989. Concepts of Object-Oriented Structured Design. *American Programmer* vol. 2(7-8).
- Marzo 1990. The Object-Oriented Structured Design Notation for Software Design Representation. *IEEE Computer* vol. 23(3).
- Webster, D. Diciembre 1988. Mapping the Design Information Representation Terrain. *IEEE Spectrum* vol. 21(12).
- Williams, L. 1986. *The Object Model in Software Engineering*. Boulder, CO: Software Engineering Research.
- Wirfs-Brock, R. and Wilkerson, B. Octubre 1989. Object-Oriented Design: A Responsibility-Driven Approach. *SIGPLAN Notices* vol. 24(10).
- Wirfs-Brock, R., Wilkerson, B., and Wiener, L. 1990. *Designing Object-Oriented Software* Englewood Cliffs, New Jersey: Prentice Hall.
- Xong, X. and Osterweil, L. Junio 1992. *A Detailed Objective Comparison and Integration of Two Object-Oriented Design Methodologies*. Irvine, California: University of California.

- Yau, S. and Tsai, J. Junio 1986. A Survey of Software Design Techniques. *IEEE Transactions on Software Engineering* vol. SE-12(6).
- Zachmand, J. 1987. A Framework for information Systems Architecture. *IBM Systems Journal* vol. 26(3).
- Zimmerman, R. 1983. Phases, Methods, and Tools - A Triad of System Development, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.

G. Programación orientada a objetos

- Ada and C++: *Business Case Analysis*. Julio 1991. Washington, D.C.: Deputy Assistant Secretary of the Air Force.
- Adams, S. Julio 1986. MetaMethods: The MVC Paradigm. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1(4). Everette, WA: Object-Oriented Programming for Smalltalk Applications Developers Association.
- Agha, G. Octubre 1986. An Overview of Actor Languages. *SIGPLAN Notices* vol. 21(10).
- 1988. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: The MIT Press.
- Agha, G., and Hewitt, C. 1987. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming, in *Research Directions in Object-Oriented Programming*. ed. B. Schriever and P. Wegner. Cambridge, MA: The MIT Press.
- Aksit, M. and Tripathi, A. Septiembre 1988. Data Abstraction Mechanisms in Sina/st. *SIGPLAN Notices* vol. 23(11).
- Albano, A. Junio 1983. Type Hierarchies and Semantic Data Models. *SIGPLAN Notices* vol. 18(6).
- Almes, G., Black, A., Lazowska, E., and Noe, J. Enero 1985. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering* vol. SE-11(1).
- Alpert, S., Woyak, S., Shrobe, H. and Arowood, L. Diciembre 1990. Object-Oriented Programming in AI. *IEEE Expert* vol. 5(6).
- Althoff, J. Agosto 1981. Building Data Structures in the Smalltalk-80 System. *Byte* vol. 6(8).
- Ambler, A. 1980. Gypsy: A Language for Specification and Implementation of Verifiable Programs, in *Programming Language Design*. ed. A Wasserman. New York, NY: Computer Society Press.
- America, P. 1987. POOL-T: A Parallel Object-Oriented Language, in *Object-Oriented Concurrent Programming*. ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Apple Computer. 1989. *MacApp: The Expandable Macintosh Application*, version 2.0B9. Cupertino, CA.
- *Macintosh Programmer's Workshop Pascal 3.0 Reference*. Cupertino, CA.
- AT&T Bell Laboratories. 1989. *UNIX System V ATT C++ Language System, Release 2.0 Library Manual*. Murray Hill, NJ.
- *UNIX System V ATT C++ Language System, Release 2.0 Product Reference Manual*. Murray Hill, NJ.
- *UNIX System V ATT C++ Language System, Release 2.0 Release Notes*. Murray Hill, NJ.
- *UNIX System V ATT C++ Language System, Release 2.0 Selected Readings*. Murray Hill, NJ.

- Attardi, G. 1987. Concurrent Strategy Execution in Omega, in *Object-Oriented Concurrent Programming*. ed. Yonezawa and M. Tokoro, Cambridge, MA: The MIT Press.
- Bach, I. Noviembre/Diciembre 1982. On the Type Concept of Ada. *Ada Letters* vol. II(3).
- Badrinath, B. and Ramamritham, K. Mayo 1988. Synchronizing Transactions on Objects. *IEEE Transactions on Computers* vol. 37(5).
- Ballard, M., Maier, D., and Wirsfs-Brock, A. Noviembre 1986. QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods. *SIGPLAN Notices* vol. 21(11).
- Beaudet, P. and Jenkins, M. Junio 1988. Simulating the Object-Oriented Paradigm in Nial. *SIGPLAN Notices* vol. 23(6).
- Bennett, J. Octubre 1987. The Design and Implementation of Distributed Smalltalk. *SIGPLAN Notices* vol. 22(12).
- Bergin, J. and Greenfield, S. Marzo 1988. What Does Modula-2 Need to Fully Support Object-Oriented Programming? *SIGPLAN Notices* vol. 23(3).
- Bhaskar, K. Octubre 1983. How Object-Oriented Is Your System? *SIGPLAN Notices* vol. 18(10).
- Birman, K., Joseph, T., Raeuchle, T., and Abbadi, A. Junio 1985. Implementing Fault-tolerant Distributed Objects. *IEEE Transactions on Software Engineering* vol. SE-11(6).
- Birtwistle, G., Dahl, O.-J., Myhrhaug, B., and Nygard, K. 1979. *Simula begin*. Lund, Sweden: Studentlitteratur.
- Black, A., Hutchinson, N., Jul, E., and Levy, H. Noviembre 1986. Object Structure in the Emerald System. *SIGPLAN Notices* vol. 21(11).
- Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. Julio 1986. *Distribution and Abstract Types in Emerald*. Report 86-02-04. Seattle, WA: University of Washington.
- Blaschek, G. 1989. Implementation of Objects in Modula-2. *Structured Programming* vol. 10(3).
- Blaschek, G., Pomberger, G., and Stritzinger, A. 1989. A Comparison of Object-Oriented Programming Languages. *Structured Programming* vol. 10(4).
- Block, F. and Chan, N. Octubre 1989. An Extended Frame Language. *SIGPLAN Notices* vol. 24(10).
- Bobrow, D. Noviembre 1984. *If Prolog Is the Answer, What Is the Questions?* Palo Alto, California. Xerox Palo Alto Research Center.
- 1985. An Overview of KRL, a Knowledge Representation Language, in *Readings in Knowledge Representation*, ed. R. Brachman and H. Levesque. Los Altos, CA: Morgan Kaufmann.
- Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., and Moon, D. Septiembre 1988. Common Lisp Object System Specification X3J13 Document 88-002R. *SIGPLAN Notices* vol. 23.
- Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. Agosto 1985. *COMMONLOOPPS: Merging Common Lisp and Object-Oriented Programming*. Report ISL-85-8. Palo Alto, CA: Xerox Palo Alto Research Center, Intelligent Systems Laboratory.
- Borgida, A. Enero 1985. Features of Languages for the Development of Information Systems at the Conceptual Level. *IEEE Software* vol. 2(1).
- Octubre 1986. Exceptions in Object-Oriented Languages, *SIGPLAN Notices* vol. 21(10).
- Borning, A. and Ingalls, D. 1982a. A Type Declaration and Inference System for Smalltalk. Palo Alto CA: Xerox Palo Alto Research Center.

- 1982b. Multiple Inheritance in Smalltalk-80. *Proceedings of the National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI.
- Bos, J. Septiembre 1987. PCOL - A Protocol-Constrained Object Language. *SIGPLAN Notices* vol. 22(9).
- Briot, J. and Cointe, P. Octubre 1989. Programming with Explicit Metaclasses in Smalltalk. *SIGPLAN Notices* vol. 24(10).
- Buzzard, G. and Mudge, T. 1987. Object-Based Computing and the Ada Programming Language, in *Object-Oriented Computing: Concepts* vol. 1. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Canning, P., Cook, W., Hill, W., and Olthoff, W. Octubre 1989. Interfaces for Strongly-Typed Object-Oriented Programming. *SIGPLAN Notices* vol. 24(10).
- Caudill, P. and Wirfs-Brock, A. Noviembre 1986. A Third Generation Smalltalk-80 Implementation. *SIGPLAN Notices* vol. 21(11).
- Chambers, C., Ungar, D., and Lee, E. Octubre 1989. An Efficient Implementation of Self, A Dynamically-Typed Object-Oriented Language Based on Prototypes. *SIGPLAN Notices* vol. 24(10).
- Chang, S. 1990. *Visual Languages and Visual Programming*. New York, New York: Plenum Press.
- Chin, R. and Chanson, S. Marzo 1991. Distributed Object-base Programming Systems. *ACM Computing Surveys* vol. 23(1).
- Clark, K. Diciembre 1988. PARLOG and Its Application. *IEEE Transactions on Software Engineering* vol. 14(12).
- Cleaveland, C. 1980. Programming Languages Considered as Abstract Data Types. *Communications of the ACM*.
- Coad, P. and Nicola, J. 1993. *Object-Oriented Programming*. Englewood Cliffs, New Jersey: Youndorn Press.
- Cointe, P. Octubre 1987. Metaclasses Are First Class: the ObjVlisp Model. *SIGPLAN Notices* vol. 22(12).
- Connor, R., Dearle A., Morrison, R., and Brown, A. Octubre 1989. An object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance. *SIGPLAN Notices* vol. 24(10).
- Conroy, T. and Pelegri-Llopert, E. 1983. An Assessment of Method-lookup Caches for Smalltalk-80 Implementations, in *Smalltalk-80: Bits of History, Words of Advice*. ed. G. Krasner. Reading, MA: Addison-Wesley.
- Coplien, J. 1992. *Advanced C++ Programming Styles and Idioms*. Reading, Massachusetts: Addison-Wesley Publishing Company.
- Corradi, A. and Leonardi, L. Diciembre 1988. The Role of Opaque Types in Building Abstractions. *SIGPLAN Notices* vol. 23(12).
- Cox, B. 1986. *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley.
- Cox, B. Enero 1983. The Object-Oriented Pre-compiler. *SIGPLAN Notices* vol. 18(1).
- Enero 1984. Message/Object Programming: An Evolutionary Change in Programming Technology. *IEEE Software* vol. 1(1).
- Febrero/Marzo 1984. Object-Oriented Programming: A Power Tool for Software Craftsmen. *Unix Review*.
- Octubre/Noviembre 1983. Object-Oriented Programming in C. *Unix Review*.
- Cox, B. and hunt, B. Agosto 1986. Objects, Icons, and Software-ICs. *Byte* vol. 11(8).
- Cox, P. and Pietrzykowski, T. Marzo 1989. *Prograph: A Pictorial View of Object-Oriented Programming*. Nova Scotia, Canada: Technical University of Nova Scotia.

- deJong, P. Octubre 1986. Compilation into Actors. *SIGPLAN Notices* vol. 21(10).
- Deutsch, P. Agosto 1981. Building Control Structures in the Smalltalk-80 System. *Byte* vol. 6(8).
- 1983. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*. New York, NY: Association of Computing Machinery.
- Dewhurst, S. and Stark, K. 1989. *Programming in C++*. Englewood Cliffs, NJ: Prentice Hall.
- Diederich, J. and Milton, J. Mayo 1987. Experimental Prototyping in Smalltalk. *IEEE Software* vol. 4(3).
- Dixon, R., McKee, T., Schweizer, P., and Vaughn, M. 1989. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. *SIGPLAN Notices* vol. 24(10).
- Dony, C. Agosto 1988. An Object-Oriented Exception Handling System for an Object-Oriented Language. *Proceeding of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Duff, C. Agosto 1986. Designing an Efficient Language. *Byte* vol. 11(8).
- Dussud, P. Octubre 1989. TICLOS: An Implementation of CLOS for the Explorer Family. *SIGPLAN Notices* vol. 24(10).
- Eccles, J. 1988. Porting from Common Lisp with Flavors to C++. *Proceedings of USE-NIX C++ Conference*. Berkeley, CA: USENIX Association.
- Edelson, D. Septiembre 1987. How Objective Mechanisms Facilitate the Development of Large Software Systems in Three Programming Languages. *SIGPLAN Notices* vol. 22(9).
- Ellis, M. and Stroustrup, B. 1990. *The Annotated C++ Reference Manual*. Reading, Massachusetts: Addison-Wesley Publishing Company.*
- Endres, T. Mayo 1985. Clascal - An Object-Oriented Pascal. *Computer Language* vol. 2(5).
- Entsminger, G. 1990. *The Tao of Objects*. Redwood City, California: M & T Books.
- Filman, R. Octubre 1987. Retrofitting Objects. *SIGPLAN Notices* vol. 22(12).
- Finzer, W. and Gould, L. Junio 1984. Programming by Rehearsal. *Byte* vol. 9(6).
- Foote, B. and Johnson, R. Octubre 1989. Reflective Facilities in Smalltalk-80. *SIGPLAN Notices* vol. 24(10).
- Freeman-Benson, B. Octubre 1989. A Module Mechanism for Constraints in Smalltalk. *SIGPLAN Notices* vol. 24(10).
- Fukunaga, K. and Jirose, S. Noviembre 1986. An Experience with a Prolog-Based Object-Oriented Language. *SIGPLAN Notices* vol. 21(11).
- Gabriel, R., White, J., and Bobrow, D. Septiembre 1991. CLOS Integrating Object-Oriented and Functional Programming. *Communications of the ACM* vol. 34(9).
- Goldberg, A. Agosto 1981. Introducing the Smalltalk-80 System. *Byte* vol. 6(8).
- Goldberg, A. Septiembre 1988. Programmer as Reader. *IEEE Software* vol. 4(5).
- Goldberg, A. and Kay, A. Marzo 1976. *Smalltalk-72 Instruction Manual*. Palo Alto, CA: Xerox Palo Alto Research Center.
- 1977. *Methods for Teaching the Programming Language Smalltalk*, Report SSL 77-2. Palo Alto, CA: Xerox Palo Alto Research Center.
- Goldberg, A. and Pope, S. Verano 1989. Object-Oriented Programming Is Not Enough. *American Programmer* vol. 2(7-8).
- Goldberg, A. and Robson, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Reading, MA: Addison-Wesley.

* Existe versión en español por Addison-Wesley/Díaz de Santos.

- 1989. *Smalltalk-80: The Language*. Reading, MA: Addison-Wesley.
- Goldberg, A. and Ross, J. Agosto 1981. Is the Smalltalk-80 System for Children? *Byte* vol. 6(8).
- Goldstein, T. Mayo 1989. The Object-Oriented Programmer. *The C++ Report* vol. 1(5).
- Gonsalves, G. and Silvestri, A. Diciembre 1986. Programming in Smalltalk-80: Observations and Remarks from the Newly Initiated. *SIGPLAN Notices* vol. 21(12).
- Gorlen, K. 1989. An Introduction to C++, in *UNIX System V ATT C++ Language System, Release 2.0 Selected Readings*. 1989. Murray Hill, NJ: ATT Bell Laboratories.
- Gorlen, K., Orlow, S., and Plexico, P. 1990. *Data Abstraction and Object-Oriented Programming in C++*. New York, NY: John Wiley and Sons.
- Gougen, J. and Meseguer, J. 1987. Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: THE MIT Press.
- Graube, N. Agosto 1988. Reflexive Architeture: From ObjVLisp to CLOS. *Procedings of ECCOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Grogono, P. Noviembre 1989. Polymorphism and Type Checking in Object-Oriented Languages. *SIGPLAN Notices* vol. 24(11).
- 1991. Issues in the Design of an Object-Oriented Programming Language. *Structured Programming* vol. 12(1).
- Hagmann, R. 1983. Preferred Classes. A Proposal for Faster Smalltalk-80 Execution, in *Smalltalk-80: Bits of History, Words of Advice*. ed. G. Krasner. Reading, MA: Addison-Wesley.
- Hailpern, B. and Nguyen, V. 1987. A model for Object-Based Inheritance, in *Research Directions in Object Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Halbert, D. and O'Brien, P. Septiembre 1988. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software* vol. 4(5).
- Halstead, R. 1987. Object Management on Distributed Systems, in *Object-Oriented Computing: Implementations* vol. 2. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Harland, D., Szypkowski, M., and Wainwright, J. Octubre 1985. An Alternative View of Polymorphism. *SIGPLAN Notices* vol. 20(10).
- Handler, J. Octubre 1986. Enhancement for Multiple Inheritance. *SIGPLAN Notices* vol. 21(10).
- Hines, T. and Unger, E. 1986. *Conceptual Object-Oriented Programming*. Manhattan, Kansas: Kansas State University.
- Ingalls, D. The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, New York, NY: Association of Computing Machinery.
- Agosto 1981a. Design Principles Behind Smalltalk. *Byte* vol. 6(8).
- Agosto 1981b. The Smalltalk Graphics Kernel. *Byte* vol. 6(8).
- 1983. The Evolution of the Smalltalk Virtual Machine, in *Smalltalk-80: Bits of History, Words of Advice*. ed. G. Krasner. Reading, MA: Addison-Wesley.
- Noviembre 1986. A Simple Technique for Handling Multiple Polymorphism. *SIGPLAN Notices* vol. 21(11).
- Ishikawa, Y. and Tokoro, M. 1987. Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation, in *Object-Oriented Concurrent Programming*. ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.

- Jackson, M. Mayo 1988. Objects and Other Subjects. *SIGPLAN Notices* vol. 23(5).
- Jacky, J. and Kalet, I. Septiembre 1987. An Object-Oriented Programming Discipline for Standard Pascal. *Communications of the ACM* vol. 30(9).
- Jacobson, I. Noviembre 1986. Language Support for Changeable, Large, Real-Time Systems. *SIGPLAN Notices* vol. 21(11).
- Jeffery, D. Febrero 1989. Object-Oriented Programming in ANSI C. *Computer Language*.
- Jenkins, M. and Glasgow, J. Enero 1986. Programming Styles in Nial. *IEEE Software* vol. 3(1).
- Johnson, R. Noviembre 1986. Type-Checking Smalltalk. *SIGPLAN Notices* 21(11).
- Johnson, R., Graver, J., and Zurawski, L. Septiembre 1988. TS: An Optimizing Compiler for Smalltalk. *SIGPLAN Notices* vol. 23(11).
- Kaehler, T. and Patterson, D. 1986. *A Taste of Smalltalk*. New York, NY: W. W. Norton.
- Agosto 1986. A Small Taste of Smalltalk. *Byte* vol. 11(8).
- Kaehler, T. Noviembre 1986. Virtual Memory on a Narrow Machine for an Object-Oriented Language. *SIGPLAN Notices* vol. 21(11).
- Kahn, K., Tribble, E., Miller, M., and Bobrow, D. 1987. Vulcan: Logical Concurrent Objects, in *Research Directions in Object-Oriented Programming*, ed. B. Schriener and P. Wegner. Cambridge, MA: The MIT Press.
- Noviembre 1986. Objects in Concurrent Logic Programming Languages. *SIGPLAN Notices* vol. 21(11).
- Kaiser, G. and Garlan, D. Octubre 1987. MELDing Data Flow and Object-Oriented Programming. *SIGPLAN Notices* 22(12).
- Kalme, C. 27 Marzo 1986. *Object-Oriented Programming: A Rule-Based Perspective*. Los Angeles, CA: Inference Corporation.
- Kay, A. *New Directions for Novice Programming in the 1980s* Palo Alto, CA: Xerox Palo Alto Research Center.
- Keene, S. 1989. *Object-Oriented Programming in Common Lisp*. Reading, MA: Addison-Wesley.
- Kelly, K., Rischer, R., Pleasant, M., Steiner, D., McGrew, C., Rowe, J., and Rubin, M. 30 Marzo 1987. *Textual Representations of Object-Oriented Programs for Future Programmers*. Palo Alto, CA: Xerox AI Systems.
- Kempf, R. Octubre 1987. Teaching Object-Oriented Programming with the KEE System. *SIGPLAN Notices* vol. 22(12).
- Kempf, J., Harris, W., D'Souza, R., and Snyder, A. Octubre 1987. Experience with CommonLoops. *SIGPLAN Notices* vol. 22(12).
- Khoshafian, S. and Copeland G. Noviembre 1986. Object Identity. *SIGPLAN Notices* vol. 21(11).
- Kiczales, G., Rivieres, J., and Bobrow, D. 1991. *The Art of the Metaobject Protocol*. Cambridge, Massachusetts: The MIT Press.
- Kilian, M. Abril 1987. *An Overview of the Trellis/Owl Compiler*. Hudson, MA: Digital Equipment Corporation.
- Kimminau, D. and Seagren, M. 1987. *Comparison of Two Prototype Developments Using Object-Based Programming*. Naperville, IL: AT&Bell Laboratories.
- Knowledge Systems Corporation. 1987. *PluggableGauges Version 1.0 User Manual*. Cary, NC.
- Knudsen, J. and Madsen, O. Agosto 1988. Teaching Object-Oriented Programming Is More than Teaching Object-Oriented Programming Languages. *Proceedings of*

- ECCOP'88: European Conference on Object-Oriented Programming.* New York, NY: Springer-Verlag.
- Knudsen, J. Agosto 1988. Name Collision in Multiple Classification Hierarchies. *Proceedings of ECCOP'88: European Conference on Object-Oriented Programming.* New York, NY: Springer-Verlag.
- Korson, T. and McGregor, J. Septiembre 1990. Understanding Object-Oriented: A Unifying Paradigm. *Communications of the ACM* vol. 33(9).
- Koshmann, T. and Evens, M. Julio 1988. Bridging the Gap Between Object-Oriented and Logic Programming. *IEEE Software* vol. 5(4).
- Koskimies, K. and Paakki, J. Julio 1987. TOOLS: A Unifying Approach to Object-Oriented Language Interpretation. *SIGPLAN Notices* vol. 22(7).
- Krasner, G. Agosto 1981. The Smalltalk-80 Virtual Machine. *Byte* vol. 6(8).
- ed. 1983. *Smalltalk-80: Bits of History, Words of Advice.* Reading MA: Addison-Wesley.
- Krasner, G. and Pope, S. Agosto/Septiembre 1988. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* vol. 1(3).
- Kristensen, B., Madsen, O., Moller-Pedersen, B., and Nygaard, K. 1987. The BETA Programming Language, in *Research Direction in Object-Oriented Programming.* ed. B. Schriever and P. Wegner. Cambridge, MA: The MIT Press.
- LaLonde, W. Abril 1989. Designing Families of Data Types Using Exemplars. *ACM Transactions on Programming Languages and Systems* vol. 11(2).
- LaLonde, W., Thomas, D., and Pugh, J. Noviembre 1986. An Exemplar Based Smalltalk. *SIGPLAN Notices* vol. 21(11).
- LaLonde, W. and Pugh, J. 1990. *Inside Smalltalk, Volumes 1 and 2.* Englewood Cliffs, New Jersey: Prentice Hall.
- Lang, K. and Peralmutter, B. Noviembre 1986. Oaklisp: a Object-Oriented Scheme with First Class Types. *SIGPLAN Notices* vol. 21(11).
- Laursen, J. and Atkinson, R. Octubre 1987. Opus: A Smalltalk Production System. *SIGPLAN Notices* vol. 22(12).
- Lieberherr, K., and Holland, I. Marzo 1989. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices* vol. 24(3).
- Septiembre 1989. Assuring Good Style for Object-Oriented Programs. *IEEE Software* vol. 6(5).
- Lieberherr, K., Holland, I., Lee, G., and Riel, A. Junio 1988. An Objective Sense of Style. *IEEE Computer* vol. 21(6).
- Lieberman, H. Noviembre 1986. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *SIGPLAN Notices* vol. 21(11).
- 1987. Concurrent Object-Oriented Programming in Act 1, in *Object-Oriented Concurrent Programming.* ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Lieberman, H., Stein L., and Ungar, D. Mayo 1988. Of Types and Prototypes: The Treaty of Orlando. *SIGPLAN Notices* vol. 23(5).
- Lim, J. and Johnson, R. Abril 1989. The Heart of Object-Oriented Concurrent Programming. *SIGPLAN Notices* vol. 24(4).
- Linowes, J. Agosto 1988. It's and Attitude. *Byte* vol. 13(8).
- Lippman, S. 1991. *C++ Primer*, Second Edition. Reading, Massachusetts: Addison-Wesley Publishing Company.
- Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, C., Scheiffer, R., and Snyder, R. 1981. *CLU Reference Manual.* New York, NY: Springer-Verlag.

- Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. 1980. Abstraction Mechanisms in CLU, in *Programming Language Design*. ed. A. Wasserman. New York, NY: Computer Society Press.
- Liu, C. Marzo 1991. On the Object-Orientedness of C++. *SIGPLAN Notices* 26(3).
- Lujun, S. and Zhongxiu. Agosto 1987. An Object-Oriented Programming Language for Developing Distributed Software. *SIGPLAN Notices* vol. 22(8).
- MacLennan, B. 1987. Values and Objects in Programming Lenguajes, in *Object-Oriented Computing: Concepts* vol. 1. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Madsen, O. 1987. Block Structure and Object-Oriented Languages, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Madsen, O. and Moller-Pedersen, B. Agosto 1988. What Object-Oriented Programming Mayo Be-And What It Does Not Have To Be. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Madsen, O. and Moller-Pedersen, B. Octubre 1989. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. *SIGPLAN Notices* vol. 24(10).
- Manci, D. 1990. *Use of Metrics to Evaluate C++*. Liberty Corner, New Jersey: AT&T Bell Laboratories.
- Mannino, M., Choi, I., and Batory, D. Noviembre 1990. The Object-Oriented Functional Data Language. *IEEE Transactions on Software Engineering* vol. 16(11).
- Marcus, R. Noviembre 1985. Generalized Inheritance. *SIGPLAN Notices* vol. 20(11).
- Markowitz, V. and Raz, Y. 1983. Eroll: An Entity-Relationship. Role-Oriented Query Language, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam. The Netherlands: Elsevier Science.
- Masini, G., Napoli, A., Colnet, D., Leonard, D., and Tompre, K. 1991. *Object-Oriented Languages*. London, England: Academic Press.
- Mellender, F. Octubre 1988. An Integration of Logic and Object-Oriented Programming. *SIGPLAN Notices* vol. 23(10).
- Methfessel, R. Abril 1987. Implementing and Access and Object-Oriented Paradigm in a Language That Supports Neither. *SIGPLAN Notices* vol. 22(4).
- Meyer, B. Noviembre 1986. Genericity versos Inheritance. *SIGPLAN Notices* 21(11).
- Febrero 1987. Eiffel: Programming for Reusability and Extendability. *SIGPLAN Notices* vol. 22(2).
- Noviembre/Diciembre 1988. Harnessing Multiple Inheritance. *Journal of Object-Oriented Programming* vol. 1(4).
- Micallef, J. Abril/Mayo 1988. Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages. *Journal of Object-Oriented Programming* vol. 1(1).
- Microsoft C++ Tutorial*. 1992. Redmond, Washington: Microsoft Corporation.
- Microsoft Windows Guide to Programming*. 1992. Redmond, Washington: Microsoft Corporation.
- Minsky, N. and Rozenshtein, D. Octubre 1987. A Law-Based Approach to Object-Oriented Programming. *SIGPLAN Notices* vol. 22(12).
- Octubre 1989. Controllable Delegation: An Exercise in Law-Governed Systems. *SIGPLAN Notices* vol. 24(10).
- Miranda, E. Octubre 1987. BrouHaHa-A Portable Smalltalk Interpreter. *SIGPLAN Notices* vol. 22(12).
- Mittal, S., Bobrow, D., and Kahn, K. Noviembre 1986. Virtual Copies: At the Boundary Between Classes and Instances. *SIGPLAN Notices* vol. 21(11).

- Moon, D. Noviembre 1986. Object-Oriented Programming with Flavors. *SIGPLAN Notices* vol. 21(11).
- Morrison, R., Dearle, A., Connor, R., and Brown, A. Julio 1991. An Ad Hoc Approach to the Implementation of Polymorphism. *ACM Transactions on Programming Languages and Systems* vol. 13(3).
- Mossenbock, H. and Templ, J. 1989. Object Oberon-A Modest Object-Oriented Language. *Structured Programming* vol. 10(4).
- Mudge, T. Marzo 1985. Object-Based Computing and the Ada Language. *IEEE Computer* vol. 18(3).
- Murray, R. 1990. *C++ Tactics*. Liberty Corner, New Jersey: AT&T Bell Laboratories.
- Nelson, M. Octubre 1991. Concurrency and Object-Oriented Programming. *SIGPLAN Notices* vol. 26(10).
- Nierstrasz, O. Octubre 1987. Active Objects in Hybrid. *SIGPLAN Notices* vol. 22(12).
- Novak, G. Junio 1983. Data Abstraction in GLIPS. *SIGPLAN Notices* vol. 18(6).
- Fall 1983. GLISP: A Lisp-Based Programming System with Data Abstraction. *Ai Magazine* vol. 4(3).
- Nygaard, K. Octubre 1986. Basic Concepts in Object-Oriented Programming. *SIGPLAN Notices* vol. 21(10).
- Nygaard, K. and Dahl, O-J. 1981. The Development of the Simula Languages, in *History of Programming Languages*. ed. R. Wexelblat. New York, NY: Academic Press.
- O'Brien, P. 15 Noviembre 1985. *Trellis Object-Based Environment: Language Tutorial*. Hudson, MA: Digital Equipment Corporation.
- O'Grady, F. Julio/Agosto 1990. Is There life After COBOL?. *American Programmer* vol. 3(7-8).
- Object-Oriented Programming Workshop. Octubre 1986. *SIGPLAN Notices* vol. 21(10).
- Olthoff, W. 1986. *Augmentation of Object-Oriented Programming by Concepts of Abstract Data Type Theory: The ModPascal Experience*. Kaiserslautern, West Germany: University of Kaiserslautern.
- Osterbye, K. Junio/Julio 1988. Active Objects: An Access-Oriented Framework for Object-Oriented Languages. *Journal of Object-Oriented Programming* vol. 1(2).
- Paepcke, A. Octubre 1989. PCLOS: A Critical Review. *SIGPLAN Notices* vol. 24(10).
- Parc Place Systems. 1988. *The Smalltalk-80 Programming System Version VI 2.3*. Palo Alto, CA.
- Pascoe, G. Agosto 1986. Elements of Object-Oriented Programming. *Byte* vol. 11(8).
- Noviembre 1986. Encapsulators: A New Software Paradigm in Smalltalk-80. *SIGPLAN Notices* vol. 21(11).
- Perez, E. Septiembre/Octubre 1988. Simulating Inheritance with Ada. *Ada Letters* vol. 8(7).
- Peterson, G. ed. 1987. *Object-Oriented Computing Concepts*. New York, NY: Computer Society Press of the IEEE.
- Pinson, L. and Wiener, R. 1988. *An Introduction to Object-Oriented Programming and Smalltalk*. Reading, MA: Addison-Wesley.
- Pohl, I. 1989. *C++ for C Programmers*. Redwood City, CA: Benjamin/Cummings.
- Pokkunuri, B. Noviembre 1989. Object-Oriented Programming. *SIGPLAN Notices* vol. 24(11).
- Ponder, C. and Bush, B. Junio 1992. Polymorphism Considered Harmful. *SIGPLAN Notices* vol. 27(6).
- Pountain, D. Agosto 1986. Object-Oriented FORTH. *Byte* vol. 11(8).
- Proceedings of ECOOP'88: European Conference on Object-Oriented Programming. Agosto 1988. New York, NY: Springer-Verlag.

- Proceedings of OOPSLA'86: Object-Oriented Programming Systems, Languages, and Applications.* Noviembre 1986. *SIGPLAN Notices* vol. 21(11).
- Proceedings of OOPSLA'87: Object-Oriented Programming Systems, Languages, and Applications.* Octubre 1987. *SIGPLAN Notices* vol. 22(12).
- Proceedings of OOPSLA'88: Object-Oriented Programming Systems, Languages, and Applications.* Septiembre 1988. *SIGPLAN Notices* vol. 23(11).
- Proceedings of OOPSLA'89: Object-Oriented Programming Systems, Languages, and Applications,* Octubre 1989. *SIGPLAN Notices* vol. 24(10).
- Proceedings of OOPSLA'90: Object-Oriented Programming Systems, Languages, and Applications.* Octubre 1990. *SIGPLAN Notices* vol. 25(10).
- Proceedings of OOPSLA'91: Object-Oriented Programming Systems, Languages, and Applications.* Noviembre 1991. *SIGPLAN Notices* vol. 26(11).
- Proceedings of OOPSLA'92: Object-Oriented Programming Systems, Languages, and Applications.* Octubre 1992. *SIGPLAN Notices* vol. 27(10).
- Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming.* Abril 1989. *SIGPLAN Notices* vol. 24(4).
- Proceedings of the USENIX Association C++ Workshop.* Noviembre 1987. Berkeley, CA: USENIX Association.
- Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modeling.* 1980. *SIGPLAN Notices* vol. 16(1).
- Pugh, J. Marzo 1984. Actors - The Stage is Set. *SIGPLAN Notices* vol. 19(3).
- Rathke, C. 1986. *ObjTalk: Repräsentation von Wissen in einer objektorientierten Sprache.* Stuttgart, West Germany: Institut für Informatik der Universität Stuttgart.
- Rentsch, T. Septiembre 1982. Object-Oriented Programming. *SIGPLAN Notices*, vol. 17(12).
- Retting, M., Morgan, T., Jacobs, J., and Wimberly, D. Enero 1989. Object-Oriented Programming in AI. *AI Expert*.
- Robson, D. Agosto 1981. Object-Oriented Software Systems. *Byte* vol. 6(8).
- Rumbaugh, J. Octubre 1987. Relations as Semantic Construct in an Object-Oriented Language. *SIGPLAN Notices* vol. 22(12).
- Russo, V. and Kaplan, S. 1988. A C++ Interpreter for Scheme. *Proceedings of USENIX C++ Conference.* Berkeley, CA: USENIX Association.
- Sakkinen, M. Agosto 1988. On the Darker Side of C++. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming.* New York, NY: Springer-Verlag.
- Diciembre 1988. Comments on "the Law of Demeter" and C++. *SIGPLAN Notices*, vol. 23(12).
- Saltzer, J. 1979. Naming and Binding of Objects, in *Operating Systems*. ed. R. Bayer et al. New York, NY: Springer-Verlag.
- Sandberg, D. Noviembre 1986. An Alternative To Subclassing. *SIGPLAN Notices* vol. 21(11).
- Octubre 1988. Smalltalk and Exploratory Programming. *SIGPLAN Notices* vol. 23(10).
- Saunders, J. Marzo/Abril 1989. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming* vol. 1(6).
- Schaffert, C., Cooper, T. and Wilpolt, C. Noviembre 25, 1985. *Trellis Object-Based Environment: Language Reference Manual.* Hudson, MA: Digital Equipment Corporation.
- Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. Noviembre 1986. An Introduction to Trellis/Owl. *SIGPLAN Notices* vol. 21(11).

- Schmucker, K. 1986a. MacApp: An Application Framework. *Byte* vol. 11(8).
- 1986b. Object-Oriented Languages for the Macintosh. *Byte* vol. 11(8).
- 1986c. *Object-Oriented Programming for the Macintosh*. Hasbrouk Heights, NJ: Hayden.
- Schrivener, B. and Wegner, P. eds. 1987. *Research Directions in Object-Oriented Programming*. Cambridge, MA: The MIT Press.
- Seidewitz, E. Marzo/Abril 1992. Object-Oriented Programming in Smalltalk and Ada. *SIGPLAN Notices* vol. XII(2).
- Octubre 1987. Object-Oriented Programming in Smalltalk and Ada. *SIGPLAN Notices* vol. 22(12).
- Shafer, D. 1988. *HyperTalk Programming*. Indianapolis, IN: Hayden.
- Shah, A. Rumbaugh, J., Hamel, J., and Borsari, R. Octubre 1989. DSM: An Object-Relationship Modeling Language. *SIGPLAN Notices* vol. 24(10).
- Shammas, N. Octubre 1988. Smalltalk a la C. *Byte* vol. 13(10).
- Shan, Y. Octubre 1989. An Event-Driven Model-View-Controller Framework for Smalltalk. *SIGPLAN Notices* vol. 24(10).
- Shapiro, J. 1991. *A C++ Toolkit*. Englewood Cliffs, New Jersey Prentice-Hall.
- Shaw, M. 1981. *ALPHARD: Form and Content*. New York, NY: Springer-Verlag.
- Shibayama, E. Septiembre 1988. How to Invent Distributed Implementation Schemes of an Object-Based Concurrent Language - A Transformational Approach. *SIGPLAN Notices* vol. 23(11).
- Shibayama, E. and Yonezawa, A. 1987. Distributed Computing in ABCL/1, in *Object-Oriented Concurrent Programming*. ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Shopiro, J. 13 Diciembre 1988. Programming Techniques with C++. *Tutorial Program of the USENIX Conference*. Denver, CO: USENIX Association.
- Diciembre 1989. An Example of Multiple Inheritance in C++: A Model of the Iostream Library. *SIGPLAN Notices* vol. 24(12).
- Simonian, R. and Crone, M. Noviembre/Diciembre 1988. InnovAda: True Object-Oriented Programming in Ada. *Journal of Object-Oriented Programming* vol. 1(4).
- Snyder, A. Febrero 1985. *Object-Oriented Programming for Common Lisp*. Report ATC-85-1. Palo Alto, CA: Hewlett-Packard.
- Noviembre 1986. Encapsulation and Inheritance in Object-Oriented Programming Language. *SIGPLAN Notices* vol. 21(11).
- 1987. Inheritance and the Development on Encapsulated Software Components, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Enero 1993. The Essence of Objects: Concepts and Terms. *IEEE Software* vol. 10(1).
- Software Productivity Solutions. 1988. *Classical-Ada User Manual*. Melbourne, FL.
- Stankovic, J. Abril 1982. Software Communication Mechanism: Procedure Calls Versus Messages. *IEEE Computer* vol. 15(4).
- Stefik, M. and Bobrow, D. Invierno 1986. Object-Oriented Programming: Themes and Variations, *AI Magazine* vol. 6(4).
- Stefik, M., Bobrow, D., Mittal, S., and Conway, L. Fall 1983, Knowledge Programming in Loops. *AI Magazine* vol. 4(3).
- Stein, L. Octubre 1987. Delegation Is Inheritance. *SIGPLAN Notices* vol. 22(12).
- Stroustrup, B. Enero 1982. Classes: An Abstract Data Type Facility for the C Language. *SIGPLAN Notices* vol. 17(1).
- Octubre 1986. An Overview of C++. *SIGPLAN Notices* vol. 21(10).

- 1987. The Evolution of C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, NM: USENIX Association.
- Noviembre 1987. Possible Directions for C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, NM: USENIX Association.
- 1988. Parameterized Types for C++. *Proceedings of USENIX C++ Conference*. Berkeley, CA: USENIX Association.
- Mayo 1988. What Is Object-Oriented Programming? *IEEE Software* vol. 5(3).
- Agosto 1988. A Better C? *Byte* vol. 13(8).
- 1991. *The C++ Programming Language*, Second Edition. Reading, Massachusetts: Addison-Wesley Publishing Company.*
- Suzuki, N. 1981. Inferring Types in Smalltalk. *Proceedings of the Eighth Annual Symposium of ACM Principles of Programming Languages*. New York, NY: Association of Computing Machinery.
- Suzuki, N. and Terada, M. 1983. Creating Efficient Systems for Object-Oriented Languages. *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*. New York, NY: Association of Computing Machinery.
- Symposium on Actor Languages. Octubre 1980. *Creative Computing*.
- Tektronix, 1988. Modular Smalltalk.
- Tesler, L. Agosto 1986. Programming Experiences. *Bytes* vol. 11(8).
- The Smalltalk-80 System. Agosto 1981. *Byte* vol. 6(8).
- Thomas, D. Marzo 1989. What's in an Object? *Byte* vol. 14(3).
- Tieman, M. 1 Mayo 1988. *User's Guide to GNU C++*. Cambridge, MA: Free Software Foundation.
- Tokoro, M. and Ishikawa, Y. Octubre 1986. Concurrent Programming in Orient84/K: An Object-Oriented Knowledge Representation Language. *SIGPLAN Notices* vol. 21(10).
- Touati, H. Mayo 1987. Is Ada an Object-Oriented Programming Language? *SIGPLAN Notices* vol. 22(5).
- Touretzky, D. 1986. *The Mathematics of Inheritance Systems*. Los Altos, California: Morgan Kaufman Publishers.
- Tripathi, A. and Berge, E. An Implementation of the Object-Oriented Concurrent Programming Language SINA. *Software - Practice and Experience* vol. 19(3).
- U. S. Department of Defense. Febrero 1983. *Reference Manual for the Ada Programming Language*. Washington, D.C.: Ada Joint Program Office.
- Ungar, D. Septiembre 1988. Are Classes Obsolete? *SIGPLAN Notices* vol. 23(11).
- Ungar, D. and Smith, R. Octubre 1987. Self: The Power of Simplicity. *SIGPLAN Notices* vol. 22(12).
- van den Bos, J. and Laffra, C. Octubre 1989. PROCOL: A Parallel Object Language with Protocols. *SIGPLAN Notices* vol. 24(10).
- Vaucher, J., Lapalme, G., and Malenfant, J. Agosto 1988. SCOOP: Structured Concurrent Object-Oriented Prolog. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Warren, S. and Abbe, D. Mayo 1980. Presenting Rosetta Smalltalk. *Datamation*.
- Watanabe, T. and Yonezawa, A. Septiembre 1988. Reflection in an Object-Oriented Concurrent Language. *SIGPLAN Notices* vol. 23(11).
- Wegner, P. Octubre 1987. Dimensions of Object-Based Language Design. *SIGPLAN Notices* vol. 22(12).

* Existe versión en español por Addison-Wesley Iberoamericana.

- Enero 1988. Workshop on Object-Oriented Programming at ECOOP 1987. *SIGPLAN Notices* vol. 23(1).
- Agosto 1990. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger* vol. 1(1).
- Octubre 1992. Dimensions of Object-Oriented Modeling. *IEEE Computer* vol. 25(10).
- Wiener, R. Junio 1987. Object-Oriented Programming in C++ - A Case Study. *SIGPLAN Notices* vol. 22(6).
- Williams, G. Verano 1989. Designing the Future: The Power of Object-Oriented Programming. *American Programmer* vol. 2(7-8).
- Wilson, R. 1 Noviembre 1987. Object-Oriented Languages Reorient Programming Techniques. *Computer Design* vol. 26(20).
- Winblad, A., Edwards, S., and King, D. 1990. *Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley Publishing Company.
- Winston, P. and Horn, B. 1989. *Lisp*. Third Edition. Reading, MA: Addison-Wesley.
- Wirfs-Brock, R. and Wilkerson, B. Septiembre 1988. An Overview of Modular Smalltalk. *SIGPLAN Notices* vol. 23(11).
- Wirth, N. Junio 1987. Extensions of Record Types. *SIGCSE Bulletin* vol. 19(2).
- Julio 1988a. From Modula to Oberon. *Software - Practice and Experience* vol. 18(7).
- Julio 1988b. The Programming Language Oberon. *Software - Practice and Experience* vol. 18(7).
- Wolf, W. Septiembre 1989. A Practical Comparison of Two Object-Oriented Languages. *IEEE Software* vol. 6(5).
- Yokote, Y. and Tokoro, M. Noviembre 1986. The Design and Implementation of Concurrent Smalltalk. *SIGPLAN Notices* vol. 21(11).
- Octubre 1987. Experience and Evolution of Concurrent Smalltalk. *SIGPLAN Notices* vol. 22(12).
- Yonezawa, A. and Tokoro, M. eds. 1987. *Object-Oriented Concurrent Programming*. Cambridge, MA: The MIT Press.
- Yonezawa, A., Briot, J., and Shibayama, E. Noviembre 1986. Object-Oriented Concurrent Programming in ABCL71. *SIGPLAN Notices* vol. 21(11).
- Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y. 1987. Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, in *Object-Oriented Concurrent Programming*. ed. Yonezawa and M. Tokoro. Cambridge, MA: The MIT Press.
- Yourdon, E. Febrero 1990. Object-Oriented COBOL. *American Programmer* vol. 3(2).
- Enero 1992. Modeling Magic. *American Programmer* vol. 5(1).
- Zave, P. Septiembre 1989. A Compositional Approach to Multiparadigm Programming. *IEEE Software* vol. 6(5).

H. Ingeniería del software

- Abdel-Hamid, T. and Madnick, S. 1991. *Software Project Dynamics*. Englewood Cliffs, New Jersey Prentice Hall.
- Abelson, H. and Sussman, G. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press.
- Andrews, D. and Leventhal, N. 1993. *FUSION: Integration IE, CASE, and JAD: A Handbook for Reengineering the Systems Organization*. Englewood Cliffs, New Jersey: Yourdon Press.

- Appleton, D. 15 Enero 1986. Very Large Projects. *Datamation*.
- Aron, J. 1974a. *The Program Development Process: The Programming Team*. Vol. 1. Reading, MA: Addison-Wesley.
- 1974b. *The Program Development Process. The Programming Team*. Vol. 2. Reading, MA: Addison-Wesley.
- Babich, W. 1986. *Software Configuration Management*. Reading, Massachusetts: Addison-Wesley Publishing Company.
- Ben-Ari, M. 1982. *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice-Hall.
- Berard, E. 1993. *Essays on Object-Oriented Software Engineering*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Berson, A. 1992. Client/Server Architecture. New York, NY: McGraw-Hill.
- Berzins, V. and Luqi. 1991 *Software Engineering with Abstractions*. Reading, Massachusetts: Addison-Wesley Publishing Company.
- Biggerstaff, T. and Perlis, A. 1989. *Software Reusability*. New York, New York: ACM Press.
- Bisant, D., and Lyle, J. Octubre 1989. A Two-Person Inspection Method to Improve Programming Productivity. *IEEE Transactions on Software Engineering* vol. 15(10).
- Bischofberger, W. and Keller, R. 1989 Enhancing the Software Life Cycle by Prototyping. *Structured Programming*.
- Bloom, P. Abril 1993. Trends in Client-Server/Cooperative Processing Application Development Tools. *American Programmer*, Arlington MA: Cutter Information Corporation.
- Boar, B. 1984. *Application Prototyping*. New York, New York: John Wiley and Sons.
- Boehm, B. Agosto 1986. A Spiral Model of Software Development and Enhancement. *Software Engineering Notes* vol. 11(4).
- Septiembre 1992. Risk Control. *American Programmer* vol. 5(7).
- Boehm, B. and Papaccio, P. 1988. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering* vol. 4(10).
- Boehm-Davis, D. and Ross, L. Octubre 1984. *Approaches to Structuring the Software Development Process*, Report GEC/DIS/TR-84-B1V-1. Arlington, VA: General Electric.
- Booch, G. 1986. *Software Engineering with Ada*. Menlo Park, CA: Benjamin/Cummings.
- Brooks, F. 1975. *The Mythical Man-Month*. Reading, MA: Addison-Wesley.
- Abril 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* vol. 20(4).
- Charette, R. 1989. *Software Engineering Risk Analysis and Management*. New York, New York: McGraw-Hill Book Company.
- Chidamber, S. and Kemerer, C. *Towards a Metrics Suite for Object-Oriented Design*. Phoenix, Arizona: OOPSLA'91.
- 1993. *A Metrics Suite for Object-Oriented Design*. Cambridge, Massachusetts: MIT Sloan School of Management.
- Chmura, L., Norcio, A., and Wicinski, T. Julio 1990. Evaluating Software Design Processes by Analyzing Change Date Over Time. *IEEE Transactions on Software Engineering* vol. 16(7).
- Cox, B. Noviembre 1990. Planning the Software Industrial Revolution. *IEEE Software* vol. 7(6).
- Curtis, B. 17 Mayo 1989.....*But You Have To Understand, This Isn't The Way We De-*

- velop Software At Our Company. MCC Technical Report Number STP-203-89. Austin, TX: Microelectronics and Computer Technology Corporation.
- Curtis, B., Kellner, M., and Over, J. Septiembre 1992. Process Modeling. *Communications of the ACM* vol. 35(9).
- Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming*. London, England: Academic Press.
- Davis, A. 1990. *Software Requirements: Analysis and Specification*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Davis, A., Bersoff, E., and Comer, E. Octubre 1988. A Strategy for Comparing Alternative Software Development Life Cycle Models. *IEEE Transactions on Software Engineering* vol. 14(10).
- Davis, C., Jajodia, S., Ng, P., and Yeh, R. eds. 1983. *Entity-Relationship Approach to Software Engineering*. Amsterdam, The Netherlands: Elsevier Science.
- DeMarco, T. and Lister, T. 1987. *Peopleware*. New York, NY: Dorset House.
- DeRemer, F. and Kron, H. 1980. Programming-in-the-Large versus Programming-in-the-Small. *Tutorial on Software Design Techniques*. Third Edition. ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Dewire, D. 1992. *Client/Server Computing*. New York, NY: McGraw-Hill.
- Dijkstra, E. 1979. Programming Considered as a Human Activity, in *Classics in Software Engineering*. ed. E. Yourdon. New York, NY: Yourdon Press.
- 1982. *Selected Writings on Computing: A Personal Perspective*. New York, NY: Springer-Verlag.
- Dowson, M. Agosto 1986. The Structure of the Software Process. *Software Engineering Notes* vol. 11(4).
- Dowson, M., Nejmech, B., and Riddle, W. Febrero 1990. *Software Engineering Practices in Europe, Japan, and the U.S.* Boulder, Colorado Software Design and Analysis.
- Dreger, B. 1989. *Function Point Analysis*. Englewood Cliffs, New Jersey: Prentice Hall.
- Eastman, N. 1984. Software Engineering and Technology. *Technical Directions* vol. 10(1). Bethesda, MD: IBM Federal Systems Division.
- Fagan, M. Junio 1976. *Design and Code Inspection and Process Control in the Development of Programs*. IBM-TR-00.73.
- Foster, C. 1981. *Real-Time Programming*. Reading, MA: Addison-Wesley.
- Freedman, D. Febrero 1992. The Devil Is in the Details Everything Important Must be Reviewed. *American Programmer* vol. 5(2).
- Freeman, P. 1975. *Software Systems Principles*. Chicago, IL: Science Research Associates.
- Freeman, P. and Wasserman, A. eds. 1983. *Tutorial on Software Design Techniques*. Fourth Edition. New York, NY: Computer Society Press of the IEEE.
- Gehani, N. and McGetrick, A. 1986. *Software Specification Techniques*. Reading, Massachusetts: Addison-Wesley Publishing Company.
- Gilb, T. 1988. *Principles of Software Engineering Management*. Reading, Massachusetts Addison-Wesley Publishing Company.
- Glass, R. 1982. *Modern Programming Practices: A Report from Industry*. Englewood Cliffs, NJ: Prentice-Hall.
- 1983. *Real-Time Software*. Englewood Cliffs, NJ: Prentice-Hall.
- 1991. *Software Conflict*. Englewood Cliffs, New Jersey: Yourdon Press.
- Goldberg, A. and Rubin, K. 1992. *Tutorial on Object-Oriented Project Management*. Vancouver, Canada: OOPSLA'92.
- Guengerich, S. 1992. *Downsizing Information Systems*. Carmel, Indiana: Sams

- Guindon, R., Krasner, H., and Curtis, B. 1987. *Breakdowns and Processes During the Earley Activities of Software Design by Professionals. Empirical Studies of Programmers, Second Workshop*. Norwood, New Jersey: Ablex Publishing Company.
- Guttman, M. and Matthews, J. Noviembre/Diciembre 1992. Managing a Large Project. *Object Magazine* vol. 2(4).
- Hansen, P. 1977. *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall.
- Henderson-Sellers, B. and Edwards, J. Septiembre 1990. The Object-Oriented Systems Lifecycle. *Communications of the ACM* vol. 33(9).
- Hoare, C. Abril 1984. Programming: Sorcery or Science? *IEEE Software* vol. 1(2).
- Holt, R., Lazowska, E., Graham, G. and Scott, M. 1978. *Structured Concurrent Programming*. Reading, MA: Addison-Wesley.
- Humphrey, W. 1988. Characterizing the Software Development Process: A Maturity Framework. *IEEE Software* vol. 5(2).
- 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley.
- Jackson, M. 1975. *Principles of Program Desing*. Orlando, FL: Academic Press.
- 1983. *System Development*. Englewood Cliffs, NJ: Prentice-Hall.
- Jensen, R. and Tonies, C. 1979. *Software Engineering*. Englewood Cliffs, NJ: Prentice-Hall.
- Jones, C. Septiembre 1984. Reusability in Programming: A survey of the State of the Art. *IEEE Transactions on Software Engineering* vol. SE-10(5).
- Septiembre 1992. Risky Business: The Most Common Software Risks. *American Programmer* vol. 5(7).
- Karam, G. and Casselman, R. Febrero 1993. A Cataloging Framework for Software Development Methods. *IEEE Computer*.
- Kishida, K., Teramoto, M. Torri, K., and Urano, Y. Septiembre 1988. Quality Assurance Technology in Japan. *IEEE Software* vol. 4(5).
- Lammers, S. 1986. *Programmer at Work*. Redmond, WA: Microsoft Press.
- Laranjeira, L. Mayo 1990. Software Size Estimation of Object-Oriented Systems. *IEEE Transactions on Software Engineering* vol. 16(5).
- Ledgard, H. Verano 1985. Programmers: The Amateur vs. the Professional. *Abacus* vol. 2(4).
- Lejter, M., Myers, S., and Reiss, S. Diciembre 1992. Support for Maintaining Object-Oriented Programs. *IEEE Transactions on Software Engineering* vol. 18(12).
- Linger, R. and Mills, H. 1977. On the Devolument of Large Reliable Programs, in *Current Trends in Programming Methodology: Software Specification and Design* vol. 1, ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Linger, R., Mills, H., and Witt, B. 1979. *Structured Programming: Theory and Practice*. Reading, MA: Addison-Wesley.
- Liskov, B. and Guttag, J. 1986. *Abstraction and Spectification in Program Development*. Cambridge, MA: The MIT Press.
- Lorin, H. 1972. *Parallelism in Hardware and Software*. Englewood Cliffs, NJ: Prentice-Hall.
- Luqi. Agosto 1990. A Graph Model for Software Evolution. *IEEE Transactions on Software Engineering* vol. 16(8).
- Mayo 1990. Software Evolution Through Rapid Prototyping. *IEEE Computer* vol. 22(5).
- Martin, J. and McClure, C. 1988. *Structured Techniques. The Basis for CASE*. Englewood Cliffs, NJ: Prentice-Hall.

- Mascot, Version 3.1, The Official Handbook of.* Junio 1987. London, England: Crown Copyright.
- Matsubara, T. Julio/Agosto 1990. Bringing up Software Designers. *American Programmer* vol. 3(7-8).
- McCabe, T. and Butler, C. Diciembre 1989. Design Complexity Measurement and Testing. *Communications of the ACM* vol. 32(12).
- Mellichamp, D. 1983. *Real-Time Computing*. New York, NY: Van Nostrand Reinhold.
- Mills, H. Noviembre 1986. Structured Programming: Retrospect and Prospect. *IEEE Software* vol. 3(6).
- Mills, J. Julio 1985. A Pragmatic View of the System Architect. *Communications of the ACM* vol. 28(7).
- Mimno, P. Abril 1993. Client-Server Computing. *American Programmer*, Arlington MA: Cutter Information Corporation.
- Mullin, M. 1990. *Rapid Prototyping for Object-Oriented Systems*. Reading, Massachusetts Addison-Wesley Publishing Company.
- Munck, R. 1985. Toward Large Software Systems That Work. *Proceedings of the AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference*. Menlo Park, CA: AIAA.
- Myers, G. 1978. *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold.
- Newport, J. 28 Abril 1986. A Growing Gap in Software. *Fortune*.
- Ng, P. and Yeh, R. 1990. *Modern Software Engineering*. New York, New York: Van Nostrand Reinhold.
- Office of the Under Secretary of Defense for Acquisition. Septiembre 1987. *Report of the Defense Science Board Task Force on Military Software*. Washington, D.C.
- Oman, P. and Lewis, T. 1990. *Milestones in Software Evolution*. Los Alamitos, California: Computer Society Press of the IEEE.
- Orr, K. 1971. *Structured Systems Development*. New York, NY: Yourdon Press.
- Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press.
- Parnas, D. Diciembre 1985. Software Aspects of Strategic Defense Systems. *Communications of the ACM* vol. 28(12).
- Julio 1985a. Why Conventional Software Development Does Not Produce Reliable Programs. *Software Aspects of Strategic Defense Systems*. Report DCS-47-IR. Victoria, Canada: University of Victoria.
 - Julio 1985b. Why Software is Unreliable. *Sotfware Aspects of Strategic Defense Systems*, Report DCS-47-IR. Victoria, Canada: University of Victoria.
- Parnas, D. and Clements, P. 1986. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering* vol. SE-12(2).
- Peters, L. 1981. *Software Design*. New York, NY: Yourdon Press.
- Pressman, R. 1988. *Making Software Happen*. Englewood Cliffs, New Jersey: Prentice Hall.
- 1992. *Software Engineering: A Practitioners's Approach*, Third Edition. New York, New York: McGraw-Hill Book Company.
- Rakos, J. 1990. *Software Project Management for Small to Medium Sized Projects*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Ramamoorthy, C., Garg, V. and Prakask, A. Julio 1986. Programming in the Large. *IEEE Transactions on Software Engineering* vol. SE-12(7).
- Rechtin, E. Octubre 1992. The Art of Systems Architecting. *IEEE Spectrum* vol. 29(10).
- Retting, M. Octubre 1990. Software Teams. *Communications of the ACM* vol. 33(10).
- Ross, D., Goodenough, J., and Irvine, C. 1980. Software Engineering: Process, Princi-

- ples, and Goals. *Tutorial on Software Design Techniques*. Third Edition. ed. P. Freeman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Rubinstein, R. and Hersh, H. 1984. *The Human Factor*. Burlington, Massachusetts Digital Press.
- Schulmeyer, G. and McManus, J. 1992. *Handbook of Software Quality Assurance*, Second Edition. New York, New York: Van Nostrand Reinhold.
- Shaw, M. Noviembre 1990. Prospects for an Engineering Discipline of Software. *IEEE Software* vol. 7(6).
- Smith, M. and Robson, D. Junio 1992. A Framework for Testing Object-Oriented programs. *Journal of Object-Oriented Programming* vol. 5(3).
- Software Process Workshop. Mayo 1988. *SIGSOFT Software Engineering Notes* vol. 14(4).
- Sommerville, I. 1989. *Software Engineering*. Third Edition. Wokingham, England: Addison-Wesley.
- Song, X., and Osterweil, L. 1993. *Executing an Iterative Design Process*. Irvine, California: University of California.
- Spector, A. and Gifford, D. Abril 1986. Computer Science Perspective of Bridge Desing. *Communications of the ACM* vol. 29(4).
- Stevens, W., Myers, G., and Constantine, L. 1979. Structured Design, in *Classics in Software Engineering*. ed. E. Yourdon. New York, NY: Yourdon Press.
- Symons, C. 1988. Function Point Analysis: Difficulties and Improvements. *IEEE Transactions on Software Engineering* vol. (14)1.
- Taylor, D. 1990. *Object-Oriented Tecnology A Manager's Guide*. Alameda, California: Servio Corporation.
- The Software Trap: Automate-Or Else. 9 Mayo 1988. *Business Week*.
- Thomsett R. Julio/Agosto 1990. Effective Project Teams. *American Programmer* vol. 3(7-8).
- Junio 1991. Managing Superlarge Projects: A Contingency Approach. *American Programmer* vol. 4(6).
- U. S. Departament of Defense. 30 Julio 1982. *Report of the DoD Joint Service Task Force on Software Problems*. Washington, D.C.
- van Genuchten, M. Junio 1991. Why is Software Late? An Empirical Study of Reasons for Delay in Software Development. *IEEE Transactions on Software Engineering* vol. 17(6).
- Vick, C. and Ramamoorthy, C. 1984. *Software Engineering*. New York, NY: Van Nostrand Reinhold.
- Vonk, R. 1990. *Prototyping*. Englewood Cliffs, NJ: Prentice-Hall.
- Walsh, J. *Preliminary Defect Data from the Iterative Development of a Large C++ Program*. Vancouver, Canada: OOPSLA'92.
- Enero 1993. *Software Quality in an Iterative Object-Oriented Development Paradigm*. Santa Clara, California: Rational.
- Ward, M. 1990. *Software that Works*. San Diego, California: Academic Press.
- Ward, P. and Mellor, S. 1985. *Structured Development for Real-Time Systems: Introduction and Tools*. Englewood Cliffs, NJ: Yourdon Press.
- Wegner, P. 1980. *Research Directions in Software Technology*. Cambridge, MA: The MIT Press.
- Julio 1984. Capital-intensive Software Technology. *IEEE Software* vol. 1(3).
- Weinberg, G. 1988. *Understanding the Professional Programmer*. New York, New York: Dorset House Publishing.

- Weinberg, G. and Freedman, D. 1990. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. New York, New York: Dorset House.
- Whitten, N. 1990. *Managing Software Development Projects*. New York, New York: John Wiley and Sons.
- Wilde, N. and Huiitt, R. Diciembre 1992. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering* vol. 18(2).
- Wilde, N., Matthews, P., and Huiitt, R. Enero 1993. Maintaining Object-Oriented Software. *IEEE Software* vol. 10(1).
- Wirth, N. 1986. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall.
- Workshop on Software Configuration Management. Noviembre 1989. *SIGSOFT Software Engineering Notes* vol. 17(7).
- Yamaura, T. Enero 1992. Standing Naked in the Snow. *American Programmer* vol. 5(1).
- Yeh, R. ed. 1977. *Current Trends in Programming Methodology: Software Specification and Desing*. Englewood Cliffs, NJ: Prentice-Hall.
- Yourdon, E. 1975. *Techniques of Program Structure and Design*. Englewood Cliffs, NJ: Prentice-Hall.
- 1979. ed. *Classics in Software Engineering*. New York, NY: Yourdon Press.
- 1989a. *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- 1989b. *Structured Walkthrougs*. Englewood Cliffs, NJ: Prentice-Hall.
- Agosto 1989c. The Year of the Object. *Computer Language* vol. 6(8).
- Verano 1989d. Object-Oriented Observations. *American Programmer* vol. 2(7-8).
- Yourdon, E and Constantine, L. 1979. *Structured Desing*. Englewood Cliffs, NJ: Prentice-Hall.
- Zahniseer, R. Julio/Agosto 1990. Building Software In Groups. *American Programmer* vol. 3(7-8).
- Zave, P. Febrero 1984. The Operational versus the Conventional Approach to Software Development. *Communications of the ACM* vol. 27(2).
- Zelkowitz, M. Junio 1978. Perspective on Software Engineering. *ACM Computing Surveys* vol. 10(2).

I. Referencias especiales

- Alexander, C. 1979. *The Timeless Way of Building*. New York, New York: Oxford University Press.
- DeGrace, P. and Stahl, L. 1990. *Wicked Problems, Righteous Solutions*. Englewood Cliffs, New Jersey: Yourdon Press.
- Fukuyama, F. 1992. *The End of History and the Last Man*. New York: The Free Press.
- Gall, J. 1986. *Systemantics: How Systems Really Work and How They Fail*. Second Edition. Ann Arbor, MI: The General Systemantics Press.
- Gleick, J. 1987. *Chaos*. New York, NY: Penguin Books.
- Heckbert, P. 1988. Ray Tracing Jell-O Brand Gelatin. *Communications of the ACM* vol. 31(2).
- Heinlein, R. 1966. *The Moon Is a Harsh Mistress*. New York, NY: The Berkeley Publishing Group.
- Hofstadter, D. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York, NY: Vintage Books.

- Inside Macintosh Volumes 1-5.* 1988. Reading, MA: Addison-Wesley.
- Kawasaki, G. 1990. *The Macintosh Way.* Glenview, Illinois Scott, Foresman and Company.
- Lakoff, G. and Johnson, M. 1980. *Metaphors We Live By.* Chicago, Illinois: The University of Chicago Press.
- Lammers, S. 1986. *Programmers at Work.* Bellevue, Washington, Microsoft Press.
- Meyer, C. and Matyas. 1982. *Cryptography.* New York, NY: John Wiley and Sons.
- Parker, T. 1983. *Rules of Thumb.* Boston, Massachusetts: Houghton Mifflin Company.
- Peter, L. 1986. *The Peter Pyramid.* New York, NY: William Morrow.
- Petroski, H. 1985. *To Engineer Is Human.* New York, NY: St. Martin's Press.
- Rand, Ayn. 1979. *Introduction to Objectivist Epistemology.* New York, NY: New American Library.
- Reti, L. 1988. *The Unknown Leonard.* New York, New York: Abradale Press.
- Sears, F., Zemansky, M., and Young, H. 1987. *University Physics.* Seventh edition. Reading, MA: Addison-Wesley.
- vonOech, R. 1990. *A Whack on the Side of the Head.* New York, New York: Warner Book, Incorporated.
- Wagner, J. 1986. *The Search for Signs of Intelligent Life in the Universe.* New York, NY: Harper and Row.
- Whitehead, A. 1958. *An Introduction to Mathematics.* New York: NY: Oxford University Press.

J. Teoría

- Aho, A., Hopcroft, J., and Ullman, J. 1974. *The Design and Analysis of Computer Programs.* Reading, MA: Addison-Wesley.
- Almarode, J. Octubre 1989. Rule-Based Delegation for Prototype. *SIGPLAN Notices* vol. 24(10).
- Appelbe, W. and Ravn, A. Abril 1984. Encapsulation Constructs in Systems Programming Languages. *ACM Transactions on Programming Languages and Systems* vol. 6(2).
- Averill, E. Abril 1982. Theory of Design and Its Relationship to Capacity Measurement. *Proceedings of the Fourth Annual International Conference on Computer Capacity Management.* San Francisco, CA: Association of Computing Machinery.
- Barr, A. and Feigenbaum, E. 1981. *The Handbook of Artificial Intelligence.* Los Altos, CA: William Kaufmann.
- Bastani, F. and Iyengar, S. Marzo 1987. The Effect of Data Structures on the Logical Complexity of Programs. *Communications of the ACM* vol. 30(3).
- Bastani, F., Hilal, W., and Sitharama, S. Octubre 1987. Efficient Abstract Data Type Components for Distributed and Parallel Systems. *IEEE Computer* vol. 20(10).
- Balkhouche, B. and Urban, J. Mayo 1986. Direct Implementation of Abstract Data Types from Abstract Specifications. *IEEE Transactions on Software Engineering* vol. SE-12(5).
- Bensley, E., Brando, T., and Prelle, M. Septiembre 1988. An Execution Model for Distributed Object-Oriented Computation. *SIGPLAN Notices* vol. 23(11).
- Berztiss, A. 1980. Data Abstraction, Controlled Iteration, and Communicating Processes. *Communications of the ACM.*

- Bishop, J. 1986. *Data Abstraction in Programming Languages*. Wokingham, England: Addison-Wesley.
- Boehm, H., Demers, A., and Donahue, J. Octubre 1980. *An Informal Description of Russell*. Technical Report TR 80-430. Ithaca, NY: Cornell University.
- Borning, A., Duisberg, R., Freemann-Benson, B., Kramer, A., and Woolf, M. Octubre 1987. Constraint Hierarchies. *SIGPLAN Notices* vol. 22(12).
- Boute, R. Enero 1988. Systems Semantics: Principles, Applications, and Implementation. *ACM Transactions on Programming Languages and Systems* vol. 10(1).
- Brachman, R. Octubre 1983. What Is-a Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Computer* vol. 16(10).
- Brachman, R. and Levesque, H. eds. 1985. *Readings in Knowledge Representation*. Los Altos, CA: Morgan Kaufmann.
- Brooks, R. Abril 1987. *Intelligence without Representation*. Cambridge, Massachusetts MIT Artificial Intelligence Laboratory.
- Bruce, K. and Wegner, P. Octubre 1986. An Algebraic Model of Subtypes in Object-Oriented Languages. *SIGPLAN Notices* vol. 21(10).
- Card, S., Moran, T., and Newell, A. 1983. *The Psychology of Human-Computer Interaction*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.
- Cardelli, L. and Wegner, P. Diciembre 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys* vol. 17(4).
- Claybrook, B. and Wyckof, M. 1980. Module: an Encapsulation Mechanism for Specifying and Implementing Abstract Data Types. *Communications of the ACM*.
- Cline, A. and Rich, E. Diciembre 1983. *Building and Evaluating Abstract Data Types*, Report TR-83-26. Austin, TX: University of Texas, Department of Computer Sciences.
- Cohen, A. Enero 1984. Data Abstraction, Data Encapsulation, and Object-Oriented Programming. *SIGPLAN Notices* vol. 19(1).
- Cohen, N. Noviembre/Diciembre 1985. Tasks as Abstraction Mechanisms. *Adda Letters* vol. 5(3-6).
- Cohen, P. and Loiselle, C. Agosto 1988. Beyond ISA: Structures for Plausible Inference in Semantic Nets. *Proceedings of the Seventh National Conference on Artificial Intelligence*. Saint Paul, MN: American Association for Artificial Intelligence.
- Collins, W. 1992. *Data Structures: An Object-Oriented Approach*. Reading, Massachusetts Addison-Wesley Publishing Company.
- Cook, W. and Palsberg, J. Octubre 1989. A Denotational Semantics of Inheritance and Its Correctness. *SIGPLAN Notices* vol. 24(10).
- Courtois, P., Heymans, F., and Parnas, D. Octubre 1971. Concurrent Control with «Readers» and «Writers». *Communications of the ACM* vol. 14(10).
- Danforth, S. and Tomlinson, C. Marzo 1988. Type Theories and Object-Oriented Programming. *ACM Computing Surveys* vol. 20(1).
- Demers, A., Donahue, J., and Skinner, G. Data Types as Values: Polymorphism, Type-Checking, Encapsulation. *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*. New York, NY: Association of Computing Machinery.
- Dennis, J. and Van Horn, E. Marzo 1966. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM* vol. 9(3).
- Donahue, J. and Demers, A. Julio 1985. Data Types Are Values. *ACM Transactions on Programming Languages and Systems* vol. 7(3).
- Eckart, J. Abril 1987. Iteration and Abstract Data Types. *SIGPLAN Notices* vol. 22(4).

- Embley, D. and Woodfield, S. 1988. Assessing the Quality of Abstract Data Types Written in Ada. *Proceedings of the 10th International Conference on Software Engineering*. New York, NY: Computer Society Press of the IEEE.
- Ferber, J. Octubre 1989. Computational Reflection in Class-Based Object-Oriented Languages. *SIGPLAN Notices* vol. 24(10).
- Fisher, J. and Gipson, D. Noviembre 1992. *In Search of Elegance*. Computer Language vol. 9(11).
- Gannon, J., Hamlet, R., and Mills, H. Julio 1987. Theory of Modules. *IEEE Transactions on Software Engineering* vol. SE-13(7).
- Gannon, J., McMullin, P., and Hamlet, R. Julio 1981. Data Abstraction Implementation, Specification, and Testing, *ACM Transactions on Programming Languages and Systems* vol. 3(3).
- Gardner, M. Mayo/Junio 1984. When to Use Private Types. *Ada Letters* vol. 3(6).
- Goguen, J., Thatcher, J., and Wagner, E. 1977. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, in *Current Trends in Programming Methodology: Data Structuring* vol. 4. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Goldberg, D. 1989. *Genetic Algorithms*. Reading, Massachusetts. Addison-Wesley Publishing Company.
- Graube, N. Octubre 1989. Metaclass Compatibility. *SIGPLAN Notices* vol. 24(10).
- Gries, D. and Prins, J. Julio 1985. A New Notion of Encapsulation. *SIGPLAN Notices* vol. 20(7).
- Grogono, P. and Bennett, A. Noviembre 1989. Polymorphism and Type Checking in Object-Oriented Languages. *SIGPLAN Notices* vol. 24(11).
- Guttag, J. 1980. Abstract Data Types and the Development of Data Structures, in *Programming Language Design*. ed. A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Hammons, C. and Dobbs, P. Mayo/Junio 1985. Coupling, Cohesion, and Package Unity in Ada. *Ada Letters* vol. 4(6).
- Harel, D. and Kahana, C. Octubre 1992. On Statecharts with Overlapping. *ACM Transactions on Software Engineering and Methodology* vol. 1(4).
- Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politis, M., Sherman, R., Shtull-Trauring, S., and Trakhtenbrot, M. Abril 1990. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering* vol. 16(4).
- Harrison, G. and Liu, D. Julio/Agosto 1986. Generic Implementations Via Analogies in the Ada Programming Language. *Ada Letters* vol. 6(4).
- Hayes, P. 1981. The Logic of Frames, in *Readings in Artificial Intelligence*. ed. B. Webber and N. Nilson. Palo Alto, CA: Tioga.
- Hayes-Roth, F. Julio 1985. A Blackboard Architecture for Control. *Artificial Intelligence* vol. 26(3).
- Hayes-Roth, F., Waterman, D., and Lenat, D. 1983. *Building Expert Systems*. Reading, MA: Addison-Wesley.
- Haynes, C. and Friedman, D. Octubre 1987. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages and Systems* vol. 9(4).
- Henderson, P. Febrero 1986. Functional Programming, Formal Specification, and Rapid Prototyping. *IEEE Transactions on Software Engineering* vol. SE-12(2).
- Herlihy, M. and Liskov, B. Octubre 1982. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems* vol. 4(4).

- Hesselink, W. Enero 1988. A Mathematical Approach to Nondeterminism in Data Types. *ACM Transactions on Programming Languages and Systems* vol. 10(1).
- Hibbard, P., Hisgen, A., Rosenbers, J., Shaw, M., and Sherman, M. 1981. *Studies in Ada Style*. New York, NY: Springer-Verlag.
- Hilfinger, P. 1982. *Abstraction Mechanisms and Language Design*. Cambridge, MA: The MIT Press.
- Hoare, C. Octubre 1974. Monitors: An Operating System Structuring Concept. *Communications of the ACM* vol. 17(10).
- Hoare, C. 1985. *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice/Hall International.
- Hogg, J. and Weiser, S. Octubre 1987. OTM: Applying Objects to Tasks. *SIGPLAN Notices* vol. 22(12).
- Jajodia, S. and Ng, P. 1983. On Representation of Relational Structures by Entity-Relationship Diagrams, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Johnson, C., 1986. Some Design Constraints Required for the Assembly of Software Components: The Incorporation of Atomic Abstract Types into Generically Structured Abstract Types. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.
- Kernighan, B. and Plauger, P. 1981. *Software Tools in Pascal*. Reading, MA: Addison-Wesley.
- Knight, B. 1983. A Mathematical Basis for Entity Analysis, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam. The Netherlands: Elsevier Science.
- Knuth, D. 1973. *The Art of Computer Programming*, Vol— 1-3. Reading, MA: Addison-Wesley.
- Kosko, B. 1992. *Neural Networks and Fuzzy Systems*. Englewood Cliffs, New Jersey: Prentice-Hall Incorporated.
- LaLonde, W. and Pugh, J. Agosto 1985. Specialization, Generalization, and Inheritance: Teaching Objectives Beyond Data Structures and Data Types. *SIGPLAN Notices* vol. 20(8).
- Leeson, J. and Spear, M. Marzo 1987. Type-Independent Modules: The Preferred Approach to Generic ADTs in Modula-2. *SIGPLAN Notices* vol. 22(3).
- Lenzerini, M. and Santucci, G. 1993. Cardinality Constraints in the Entity-Relationship Model, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Levesque, H. Julio 1984. Foundations of a Functional Approach to Knowledge Representation. *Artificial Intelligence* vol. 23(2).
- Lindgreen, P. 1983. Entity Sets and Their Description, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Lins, C. 1989. A First Look at Literate Programming. *Structured Programming*.
- Liskov, B. Mayo 1988. Data Abstraction and Hierarchy. *SIGPLAN Notices* vol. 23(5).
- 1980. Programming with Abstract Data Types, in *Programming Language Design*. ed. A. Wasserman, New York, NY: Computer Society Press of the IEEE.
- Liskov, B. and Scheifler, R. Julio 1983. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems* vol. 5(3).

- Liskov, B. and Zilles, S. 1977. An Introduction to Formal Specifications of Data Abstractions, in *Current Trends in Programming Methodology: Software Specification and Desing* vol. 1. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Lowry, M. and McCartney. 1991. *Automating Software Design*. Cambridge, Massachusetts: The MIT Press.
- Lucco, S. Octubre 1987. Parallel Programming in a Virtual Object Space. *SIGPLAN Notices* vol. 22(12).
- Maes, P. Octubre 1987. Concepts and Experiments in Computational Reflection. *SIGPLAN Notices* vol. 22(12).
- Mark, L. 1983. What is the Binary Relationship Approach?, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Markowitz, V. and Raz, Y. 1983. A Modified Relational Algebra and Its Use in an Entity-Relationship Environment, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Matsuoka, S. and Kawai, S. Septiembre 1988. Using Tuple Space Communication in Distributed Object-Oriented Languages. *SIGPLAN Notices* vol. 23(11).
- McAllester, D. and Zabih, F. Noviembre 1986. Boolean Classes. *SIGPLAN Notices* vol. 21(11).
- McCullough, P. Octubre 1987. Transparent Forwarding: First Steps. *SIGPLAN Notices* vol. 22(12).
- Merlin, P. and Bochmann, G. Enero 1983. On the Construction of Submodule Specifications and Communication Protocols. *ACM Transactions on Programming Languages and Systems* vol. 5(1).
- Meyer, B. 1987. *Programming as Contracting*, Report TR-EI-12/CO. Goleta, CA: Interactive Software Engineering.
- Octubre 1992. Applying «Design by Contract». *IEEE Computer* vol. 25(10).
- Minoura, T. and Iyengar, S. Enero 1989. Data and Time Abstraction Techniques for Multilevel Concurrent Systems. *IEEE Transactions of Software Engineering* vol. 15(1).
- Murata, T. 1984. Modeling and Analysis of Concurrent Systems, in *Software Engineering*. ed. C. Vick and C. Ramamoorthy. New York, NY: Van Nostrand Reinhold.
- Mylopoulos, J. and Levesque, H. 1984. An Overview of Knowledge Representation. *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.
- Nakano, R. 1983. Integrity Checking in a Logic-Oriented ER Model, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Newton, M. and Watkins, J. Noviembre/Diciembre 1988. The Combination of Logic and Objects for Knowledge Representation. *Journal of Object-Oriented Programming* vol. 1(4).
- Nii, P. Verano 1986. Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Magazine* vol. 7(2).
- Ohori, A. and Buneman, P. Octubre 1989. Static Type Inference for Parametric Classes. *SIGPLAN Notices* vol. 24(10).
- Pagan, F. 1981. *Formal Specification of Programming Languages*. Englewood Cliffs, NJ: Prentice-Hall.
- Parent, C. and Spaccapieta, S. Julio 1985. An Algebra for a General Entity-Relationship Model. *IEEE Transactions of Software Engineering* vol. SE-11(7).

- Parnas, D. 1977. The Influence of Software Structure on Reliability, in *Current Trends in Programming Methodology: Software Specification and Design* vol. 1. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- 1980. Designing Software for Ease of Extension and Contraction, in *Tutorial on Software Design Techniques*. Third Edition. ed. P. Freedman and A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Parnas, D., Clements, P., and Weiss, D. 1983. Enhancing Reusability with Information Hiding. *Proceedings of the Workshop on Reusability in Programming*, Stratford, CT: ITT Programming.
- Pattee, H. 1973. *Hierarchy Theory*. New York, NY: George Braziller.
- Peckham, J. and Maryanski, F. Septiembre 1988. Semantic Data Models. *ACM Computing Surveys* vol. 20(3).
- Pedersen, C. Octubre 1989. Extending Ordinary Inheritance Schemes to Include Generalization. *SIGPLAN Notices* vol. 24(10).
- Peterson, J. Septiembre 1977. Petri Nets. *Computing Surveys* vol. 9(3).
- Reed, D. Septiembre 1978. Naming and Synchronization in a Decentralized Computer System. Cambridge, MA: The MIT Press.
- Rich, C. and Wills, L. Enero 1990. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software* vol. 7(1).
- Robinson, L. and Levitt, K. 1977. Proof Techniques for Hierarchically Structured Programs, in *Current Trends in Programming Methodology: Program Validation* vol. 2. ed. R. Yeh. Englewood Cliffs, NJ: Prentice-Hall.
- Ross, D. Julio/Agosto 1986. Classifying Ada Packages. *Ada Letters* vol. 6(4).
- Ruane, L. Enero 1984. Abstract Data Types in Assembly Language Programming. *SIGPLAN Notices* vol. 19(1).
- Rumbaugh, J. Septiembre 1988. Controlling Propagation of Operations Using Attributes on Relations. *SIGPLAN Notices* vol. 23(11).
- Sedgewich, R. 1983. *Algorithms*. Reading, MA: Addison-Wesley.
- Shankar, K. 1984. Data Design: Types, Structures, and Abstractions, in *Software Engineering*. ed. C. Vick and C. Ramamoorthy. New York, NY: Van Nostrand Reinhold.
- Shaw, M. 1984. The Impact of Modeling and Abstraction Concerns on Modern Programming Languages. *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.
- Octubre 1984. Abstraction Techniques in Modern Programming Languages. *IEEE Software* vol. 1(4).
- Mayo 1989. Larger Scale Systems Require Higher-Level Abstractions. *SIGSOFT Engineering Notes* vol 14(3).
- Shaw, M., Feldman, G., Fitzgerald, R., Hilfinger, P., Kimura, I., London, R., Rosenberg, J., and Wulf, W. 1981. Validating the Utility of Abstraction Techniques, in *ALPHARD: Form and Content*. ed. M. Shaw. New York, NY: Springer-Verlag.
- Shaw, M., Wulf, W., and London, R. 1981. Abstraction and Verification in ALPHARD: Iteration and Generators, in *ALPHARD: Form and Content*. ed. M. Shaw. New York, NY: Springer-Verlag.
- Sherman, M., Hisgen A., and Rosenberg, J. 1982. A Methodology for Programming Abstract Data Types in Ada. *Proceedings of the AdaTEC Conference on Ada*. New York, NY: Association of Computing Machinery.
- Siegel, J. Abril 1988. Twisty Little Passages. *HOOPLA: Hooray for Object-Oriented Programming Languages* vol. 1(3). Everett, WA: Object Oriented Programming for Smalltalk Application Developers Association.

- Stefik, M., Bobrow, D., and Kahn, K. Enero 1986. Integrating Access-Oriented Programming into a Multiparadigm Environment. *IEEE Software* vol. 3(1).
- Strom, R. and Yemini, S. Enero 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering* vol. SE-12(1).
- Stubbs, D. and Webre, N. 1985. *Data Structures with Abstract Data Types and Pascal*. Monterey, CA: Brooks/Cole.
- Swaine, M. Junio 1988. Programming Paradigms. *Dr. Dobb's Journal of Software Tools*, no. 140.
- Tabourier, Y. 1983. Further Development of the Occurrences Structure Concept: The EROS Approach, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Tanenbaum, A. 1981. *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall.
- Tenenbaum, A. and Augenstein, M. 1981. *Data Structures Using Pascal*. Englewood Cliffs, NJ: Prentice-Hall.
- Throelli, L. Octubre 1987. Modules and Type Checking in PL/LL. *SIGPLAN Notices* vol. 22(12).
- Tomlinson, C. and Singh, V. Octubre 1989. Inheritance and Synchronization with Enabled-sets. *SIGPLAN Notices* vol. 24(10).
- Toy, W. 1984. Hardware/Software Tradeoffs, in *Software Engineering*. ed. C. Vick and C. Ramamoorthy. New York, NY: Van Nostrand Reinhold.
- Vegdahl, S. Noviembre 1986. Moving Structures between Smalltalk Images. *SIGPLAN Notices* vol. 21(11).
- Walters, N. Octubre 1992. Using Hared Statecharts to Model Object-Oriented Behavior. *SIGSOFT Notices* vol. 17(4).
- Wasserman, A. 1980. Introduction to Data Types, in *Programming Language Design*. ed. A. Wasserman. New York, NY: Computer Society Press of the IEEE.
- Weber, H. and Ehring, H. Julio 1986. Specification of Modular Systems. *IEEE Transactions on Software Engineering* vol. SE-12(7).
- Wegner, P. 6 Junio 1981. *The Ada Programming Language and Environment*. Unpublished draft.
- Wegner, P. 1987. On the Unification of Data and Program Abstraction in Ada, in *Object-Oriented Computing: Concepts* vol. 1. ed. G. Peterson. New York, NY: Computer Society Press of the IEEE.
- Wegner, P. 1987. The Object-Oriented Classification Paradigm, in *Research Directions in Object-Oriented Programming*. ed. B. Schriver and P. Wegner. Cambridge, MA: The MIT Press.
- Wegner, P. and Zdonik, S. Agosto 1988. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Weihl, W. and Liskov, B. Abril 1985. Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems* vol. 7(2).
- Weinberg, G. 1971. *The Psychology of Computer Programming*. New York, New York, Van Nostrand Reinhold Company.
- Weller, D. and York, B. Mayo 1984. A Relational Representation of an Abstract Type System. *IEEE Transactions on Software Engineering* vol. SE-10(3).
- White, J. Julio 1983. On the Multiple Implementation of Abstract Data Types within a Computation. *IEEE Transactions on Software Engineering* vol. SE-9(4).
- Wirth, N. Diciembre 1974. On the Composition of Well-structured Programs. *Computing Surveys* vol. 6(4).

- Enero 1983. Program Development by Stepwise Refinement. *Communications of the ACM* vol. 26(1).
- 1986. *Algorithms and Data Structures*, Second Edition. Englewood Cliffs, NJ: Prentice-Hall.
- Abril 1988. Type Extensions. *ACM Transactions on Programming Languages and Systems* vol. 10(2).
- Wolf, A., Clarke, L., and Wileden, J. Abril 1988. A Model of Visibility Control. *IEEE Transactions on Software Engineering* vol. 14(4).
- Woods, W. Octubre 1983. What's Important About Knowledge Representation? *IEEE Computer* vol. 16(10).
- Zilles, S. 1984. Types, Algebras, and Modeling, in *On Conceptual Modeling: Perspectives from Artificial Intelligence Databases, and Programming Languages*. ed. M. Brodie, J. Mylopoulos, and J. Schmidt. New York, NY: Springer-Verlag.
- Zippel, R. Junio 1983. Capsules. *SIGPLAN Notices* vol. 18(6).

K. Herramientas y entornos

- Andrews, T. and Harris, C. 1987. Combining Language and Database Advances in an Object-Oriented Development Environment. Billerica, MA: Ontologic.
- Corradi, A. and Leonardi, L. 1986. *An Environment Bases on Parallel Objects*. Bologna, Italy: Universita' di Bologna.
- Deutsch, P. and Taft, E. Junio 1980. *Requirements for an Experimental Programming Environment*, Report CSL-80-10. Palo Alto, CA: Xerox Palo Alto Research Center.
- Diederich, J. and Milton, J. Octubre 1987. An Object-Oriented Design System Shell. *SIGPLAN Notices* vol. 22(12).
- Durant, D., Carlson, G., and Yao, P. 1987. *Programmer's Guide to Windows*. Berkeley, CA: Sybex.
- Erman, L., Lark, J., and Hayes-Roth, F. Diciembre 1988. ABE: An Environment for Engineering Intelligent Systems. *IEEE Transactions on Software Engineering* vol. 14(12).
- Ferrel, P. and Meyer, R. Octubre 1989. Vamp: The Aldus Application Framework. *SIGPLAN Notices* vol. 24(10).
- Fischer, H. and Martin, D. 1987. *Integrating Ada Design Graphics into the Ada Software Development Process*. Encino, CA: Mark V Business Systems.
- Goldberg, A. 1984a. *Smalltalk-80: The Interactive Programming Environment*. Reading, MA: Addison-Wesley.
- 1984b. The Influence of an Object-Oriented Language on the Programming Environment, in *Interactive Programming Environments*. ed. B. Barstow. New York, NY: McGraw-Hill.
- Goldstein, I. and Bobrow, D. Marzo 1981. *An Experimental Description-Based Programming Environment*, Report CSL-81-3. Palo Alto, CA: Xerox Palo Alto Research Center.
- Gorlen, K. Mayo 1986. *Object-Oriented Program Support*, Bethesda, MD: National Institute of Health.
- Hecht, A. and Simmons, A. 1986. Integrating Automated Structured Analysis and Design with Ada Programming Support Environments. *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*. Houston, TX: NASA Lyndon B. Johnson Space Center.

- Hedin, G. and Magnusson B. Agosto 1988. The Mjolner Environment: Direct Interaction with Abstractions. *Proceedings of ECOOP'88: European Conference on Object-Oriented Programming*. New York, NY: Springer-Verlag.
- Hudson, S. and King, R. Junio 1988. The Cactic Project: Database Support for Software Environments. *IEEE Transactions on Software Engineering* vol. 14(6).
- International Business Machines. Abril 1988. *Operating System/2 Seminar Proceedings, IBM OS/2 Standard Edition Version 1.1, IBM Operating System/2 Update, Presentation Manager*. Boca Raton, FL.
- Kant, E. 26 Marzo 1987. *Interactive Problem Solving with a Task Configuration and Control System*. Ridgefield, CT: Schlumberger-Doll Research.
- Kleyn, M. and Gingrich, P. Septiembre 1988. GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views. *SIGPLAN Notices* vol. 23(11).
- Laff, M. and Hailpern, B. Julio 1985. SW-2 - An Object-Bases Programming Environment. *SIGPLAN Notices* vol. 20(7).
- MacLenna, B. Julio 1985. A Simple Software Environment Based on Objects and Relations. *SIGPLAN Notices* vol. 20(7).
- Marques, J. and Guedes, P. Octubre 1989. Extending the Operating System to Support an Object-Oriented Environment. *SIGPLAN Notices* vol. 24(10).
- Minsky, N. and Rozenshtein, D. Febrero 1988. A Software Development Environment for Law-Governed Systems. *SIGPLAN Notices* vol. 24(2).
- Moreau, D. and Dominick, W. 1987. *Object-Oriented Graphical Information Systems: Research Plan and Evaluation Metrics*. Lafayette, LA: University of Southwestern Louisiana, Center for Advanced Computer Studies.
- Nakata, S. and Yamazak, G. 1983. ISMOS: A System Based on the E-R Model and its Application to Database-Oriented Tool Generation, in *Entity-Relationship Approach to Software Engineering*. ed. C. Davis et al. Amsterdam, The Netherlands: Elsevier Science.
- Nye, A. 1989. *Xlib Programming Manual for Version 11*. Newton, MA: O'Reilly and Associates.
- O'Brien, P., Halbert, D., and Kilian, M. Octubre 1987. The Trellis Programming Environment. *SIGPLAN Notices* vol. 22(12).
- Open Look Graphical User Interface Functional Specification*. 1990. Reading, MA: Addison-Wesley.
- OSF/Motif Style Guide, Version 1.0*. 1989. Cambridge, MA: Open Software Foundation.
- Penedo, M., Ploedereder, E., and Thomas, I. Febrero 1988. Object Management Issues for Software Engineering Environments. *SIGPLAN Notices* vol. 24(2).
- Reenskaug, T. and Skaar, A. Octubre 1989. An Environment for Literature Smalltalk Programming. *SIGPLAN Notices* vol. 24(10).
- Rosenplatt, W., Wileden, J., and Wolf, A. Octubre 1989. OROS: Toward a Type Model for Software Development Environments. *SIGPLAN Notices* vol. 24(10).
- Russo, V. and Campbell, R. Octubre 1989. Virtual Memory and Backing Storage Management in Multiprocessor Operating System Using Object-Oriented Design Techniques. *SIGPLAN Notices* vol. 24(10).
- Scheifler, R. and Gettys, J. 1986. The X Window System. *ACM Transactions on Graphics* vol. 63.
- Schwan, K. and Matthews, J. Julio 1986. Graphical Views of Parallel Programs. *Software Engineering Notes* vol. 11(3).
- Shear, D. 8 Diciembre 1988. CASE Shows Promise but Confusion Still Exist. *EDN* vol. 33(25).

- Sun Microsystems. 29 Marzo 1987. *NeWS Technical Overview* Mountain View, CA.
- Tarumi, H., Agusa, K., and Ohno, Y. 1988. A Programming Environment Supporting Reuse of Object-Oriente Software. *Proceedings of the 10th International Conference on Software Engineering*, New York, NY: Computer Society Press of the IEEE.
- Taylor, R., Belz, F., Clarke, L., Osterweil, L., Selby, R., Wileden, J., Wolf, A., and Young, M. Febrero 1988. Foundations for the Arcadia Environment. *SIGPLAN Notices* vol. 24(2).
- Tesler, L. Agosto 1981. The Smalltalk Environment. *Byte* vol. 6(8).
- Vines, D. and King, T. 1988. *Gaia: An Object-Oriented Framework for an Ada Environment*. Minneapolis, MN: Honeywell.
- Weinand, A., Gamma, E., and Marty, R. 1989. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming* vol. 10(2).
- Wiorkowski, G. and Kull, D. 1988. *DB2 Design and Development Guide*. Reading, MA: Addison-Wesley.

Vocabulario técnico bilingüe¹

TÉRMINO ORIGINAL EN INGLÉS	TÉRMINO USADO EN EL LIBRO	VOCABLOS ALTERNATIVOS DE USO COMÚN
INF	INF	—
abstract class	clase abstracta	—
abstraction	abstracción	—
action	acción	—
adornment	marca	adorno
agent	agente	—
aggregation	agregación	—
ALU	unidad aritmético-lógica	ALU, unidad lógico-aritmética
animation	animación	—
applicability	aplicabilidad	facilidad de aplicación
architectural	arquitectónico	arquitectural
assembly language	lenguaje ensamblador	lenguaje assembler
assert	aserción	aserto, afirmación, declaración
association	asociación	—
asynchronous	asíncrono	asincrónico
balking	abandono inmediato	detención, abandono brusco
base class	clase base	superclase
behaviour	comportamiento	conducta
bidding	ligadura	enlace
boolean	booleano	lógico
boundary	frontera	límite
breakdown	derrumbamiento	caída, interrupción, fallo, corte
browser	hojeador	examinador, navegador
buried pointer	puntero escondido	puntero oculto, enterrado, enmascarado
bus	bus	—
by value	por valor	—
by reference	por referencia	—
callback	función callback	función de retorno
cardinality	cardinalidad	multiplicidad
casting	conversión forzada	moldeado
categorization	categorización	por categorías
class utility	utilidad de clase	utilidad clase
class	clase	—
cliente/supplier	cliente/proveedor	cliente/servidor

¹ Ante la falta de un lenguaje estandarizado en castellano para las ciencias de la computación se ha elaborado el presente vocabulario con la traducción que hemos dado en este libro a los principales términos de la versión original en inglés, así como vocablos alternativos de uso común en España y América latina. Esta labor se verá compensada por el servicio que pueda prestar al lector. (N. del E.)

<i>TÉRMINO ORIGINAL EN INGLÉS</i>	<i>TÉRMINO USADO EN EL LIBRO</i>	<i>VOCABLOS ALTERNATIVOS DE USO COMÚN</i>
cluster	agrupamiento, agrupar	racimo, apiñar
clustering	agrupamiento	apiñamiento
cognitive science	ciencia cognitiva	ciencia del conocimiento
cohesion	cohesión	—
coincidental abstraction	abstracción de coincidencia	abstracción coincidental
completeness	compleción	plenitud, completitud, completud
computer	computador	ordenador
conceptualization	conceptualización	—
concurrency	conurrencia	paralelismo
conformance	congruencia	conformismo, conformidad
constraint	restricción	limitación
constructor	constructor	—
container	contenedor	<i>container</i> , envase, recipiente, depósito
containment	contención	—
contract model	modelo contractual	modelo de contrato
copy constructor	constructor de copia	—
coupling	acoplamiento	—
crash	estallar	romper
chunk	bloque	trozo
dangling reference	referencia colgada	referencia suspendida
data-driven	dirigido por los datos	—
deadlock	bloqueo entre procesos	abrazo mortal, interbloqueo
debug	depurar	poner a punto
debugging	depuración	puesta a punto
deep copy	copia profunda	copia honda
delegation	delegación	—
deque	cola doble	<i>deque</i>
dereferencing	derreferenciar	desreferenciar, indireccional
descomposable	descomponible	desdoblable
destructor	destructor	—
dispatch	seleccionar	distribuir, lanzar
domain analysis	análisis del dominio	—
drive	unidad	—
dynamic binding	ligadura dinámica	ligadora posterior
early binding	ligadura temprana	ligadura estática
encapsulate	encapsular	—
encapsulation	encapsulamiento	encapsulación
entity-relationship	entidad-relación	entidad-interrelación
event	evento	suceso
event-dispatching	selección de eventos	selección de sucesos
evolutionary	evolutiva	<i>evolucionaria</i>
friend	amiga	<i>friend</i>
file	archivo	fichero

<i>TÉRMINO ORIGINAL EN INGLÉS</i>	<i>TÉRMINO USADO EN EL LIBRO</i>	<i>VOCABLOS ALTERNATIVOS DE USO COMÚN</i>
flag	indicador	señalizador
floating-point	punto flotante	coma flotante
frame	marco	cuadro, <i>frame</i>
framework	marco de referencia	marco de trabajo
free subprogram	subprograma libre	—
garbage collection	recolección de basura	información inservible
generalization	generalización	—
generic function	función genérica	—
genericity	genericidad	—
GUI	interfaz gráfico de usuario, IGU	GUI
hard drive	unidad duro	—
header file	archivo de cabecera	archivo de encabezamiento
heap	montículo, heap	montón, cúmulo
hierarchy	jerarquía	—
hypermedia	hipermedia	hypermedia, hipermedios
I/O	E/S	I/O
icon	ícono	—
identity	identidad	—
imperative language	lenguaje imperativo	—
implementation	implementación	implantación, realización, instrumentación
indexable	indexable	indizable
indexed	indexado	indizado
information hiding	ocultación de la información	ocultamiento de la información
initializing	inicialización	iniciación
instance	instancia	ejemplar, ejemplo, caso
instance variable	variable de instancia	variable instancia
instantiation	instanciación	—
interaction	interacción	—
interface	el interfaz	la interfaz
invariant	invariante	—
is a	es un	es-un-tipo-de
iterator	iterador	—
key	clave	llave
label	etiqueta	—
late binding	ligadura tardía	—
lattice	trama	retícula
layer	capa	estrato
legacy	legado	herencia
library	biblioteca	biblioteca
client	cliente	—
link	enlace	—
linkages	enlaces	enlazados
lookup	búsqueda	consulta
macro	macro	—

<i>TÉRMINO ORIGINAL EN INGLÉS</i>	<i>TÉRMINO USADO EN EL LIBRO</i>	<i>VOCABLOS ALTERNATIVOS DE USO COMÚN</i>
maintenance	mantenimiento	—
many-to-many	muchas-a-muchas	—
mapping	correspondencia	<i>mapeado</i>
matching	emparejamiento	correspondencia, concordancia
memory-mapped	correspondencia de memoria	<i>mapeado</i> de memoria
message passing	paso de mensajes	—
metaclass	metaclase	—
methodologist	diseñador de metodología	metodologista, metodólogo
metrics	métricas	—
milestones	hitos	mojones
modelling	modelado	modelación
modifier	modificador	—
module	module	—
multiple inheritance	herencia múltiple	—
nesting	anidamiento	encajamiento
network	en red	—
non preemptive	no desplazante	no prioritario, no apropiativo
object	objeto	—
object-based	basado en objetos	basado en objeto
object model	modelo de objeto	modelo objeto
object-oriented	orientado a objetos	orientado a/al objeto
one-to-many	una-a-muchas	—
OOA	AOO	OOA
OOD	DOO	OOD
OOP	POO	OOP
overloading	sobrecarga	—
override	redefinir	reemplazar, suplantar, sustituir
packaging	empaqueamiento	—
parameterized class	clase paramétrica	—
parametric polymorphism	polimorfismo paramétrico	—
parent/child	padre/hijo	—
part of	parte de	—
pattern	patrón	modelo, plantilla
peer-to-peer	hermano a hermano	par a par
persistence	persistencia	—
pointer	puntero	apuntador
polymorphism	polimorfismo	—
portability	portabilidad	transportabilidad
postcondition	postcondición	—
precondition	precondición	—
preservation	conservación	preservación
primary memory	memoria principal	memoria primaria
primitiveness	ser primitivo	<i>primitividad</i>
processor	procesador	—
programming-in-the-large	programación al por mayor	programación en gran escala

<i>TÉRMINO ORIGINAL EN INGLÉS</i>	<i>TÉRMINO USADO EN EL LIBRO</i>	<i>VOCABLOS ALTERNATIVOS DE USO COMÚN</i>
protocol	protocolo	—
pure virtual function	función virtual pura	—
query	consulta	—
queue	cola	—
recognition	reconocimiento	—
redefining	redefinición	—
relational	relacional	—
relationship	relación	interrelación
repository	depósito	almacén, repositorio
requirement	requisito	requerimiento
responsibility	responsabilidad	—
reusability	reutilización	reusabilidad
reuse	reutilizar	reusar
reusing	reutilización	reusación
rol	rol, papel	—
run time	ejecución	tiempo de ejecución
sallow copy	copia superficial	—
scaling down	escalado descendente	—
scaling up	escalado ascendente	—
scenario	escenario	—
scenario planning	planificación del escenario	—
script	guión	escritura, manuscrito
scaling	escalado	escalamiento
schedule	planificar	—
selector	selección	—
semantics	semántica	—
send	enviar	transmitir
seniority	antigüedad	senioridad
separate compilation	compilación separada	compilación independiente
server	servidor	—
signature	presentación, firma	firmatura
single inheritance	herencia simple	—
slot	slot, ranura	abertura
software engineering	ingeniería del software	ingeniería de software
sorting	ordenación	clasificación
specification	especificación	—
specialization	especialización	—
spreadsheet	hoja de cálculo	—
stack	pila	stack
state transition	diagrama de transición de estados	—
static binding	ligadura estática	enlace, ligado estático
store	almacenar	guardar
storyboarding	narración de sucesos	—
stream	flujo	corriente

<i>TÉRMINO ORIGINAL EN INGLÉS</i>	<i>TÉRMINO USADO EN EL LIBRO</i>	<i>VOCABLOS ALTERNATIVOS DE USO COMÚN</i>
stress testers	analizadores de rendimiento	probadores de rendimiento
strong typing	comprobación estricta de tipos	tipeado fuerte, tipado fuerte
structural sharing	compartición estructural	—
subclass	subclase	clase derivada
subsystem	subsistema	—
superclass	superclase	—
superstate	superestado	—
supplier	proveedor	servidor
synchronous	síncrono	sincrónico
synegistic	sinérgico	sinergístico
task	tarea	—
template	plantilla	patrón, modelo
thread	hiló	hebra
thread of control	hiló de control	hebra de control
timeout	de intervalo	espera
timer	temporizador	reloj
to instantiate	instanciar	crear instancias
top-down	descendente	arriba-abajo
topology	topología	—
triggering	disparo	activación
type cast	ahormado de tipos, conversión forzosa de tipos	moldes
type coercion	coerción	—
type checking	comprobación de tipos	verificación de tipos
type-safe	seguros respecto al tipo	tipos seguros
typing	tipos, tipificación	tipado, tipeado
unrestrained	ilimitada	no restringida
use-case	casos de uso	—
using	uso	—
view	vista	visión
weak typing	comprobación débil de tipos	tipificación débil
whole/part	todo/parte	<i>whole/part</i>
workstation	estación de trabajo	<i>workstation</i>

Índice analítico

- 1FN, 452
2FN, 453
3FN, 453
4GL, 469
Abbott, R., 182
abstracción, 14, 22, 23, 31, 45, 46, 67
 activa, 51
 búsqueda, 185
 calidad de, 155
 clave, 185
 de coincidencia, 47
 de entidad, 47
 ejemplos de, 49
 orientada a la lógica, 45
 orientada a objetos, 45
 orientada a procedimientos, 45
 orientada a reglas, 45
 orientada a restricciones, 45
 pasiva, 51
refinamiento, 185
regla práctica, 106
semántica dinámica, 48
semántica estática, 48
significado de, 46
tipos de, 45, 47
virtual, 47
vista externa, 57
vista interna, 57
acción, 230-2
acoplamiento, 156
actividad, 232
actor, 561
Actors, 83
Ada, 21, 33, 37, 83, 159, 273
Adams, S., 289
aditivo, 561
administrador del sistema, 313
adquisición de datos, 337
agente, 115, 424
agregación, 72, 118, 124, 146, 392
 cíclica, 148
 ícono, 180
 y contención, 119
 y herencia, 142
 y propiedad, 73
agrupación, 22, 315
agrupamiento conceptual, 173, 176
Albrecht, A., 288
Alexander, C., 27, 163
ALGOL, 32
algoritmo, 283, 423
alias, 109
Alphard, 33, 41
amigo, 60, 220, 427
análisis, 17, 21, 44, 178, 277, 287, 338, 376, 434, 474, 510
 de casos de uso, 181
 de dominios, 180, 289, 376
 de esquemas, 399
 del comportamiento, 179, 271
 estructurado, 183
 horizontal de dominios, 180
 orientado a objetos, 21, 26, 32, 44, 178
 vertical de dominios, 180
analista, 313
Andert, G. 294
aplicación
 adquisición de datos, 337
 computación cliente/servidor, 433
 dirección y control, 511
 inteligencia artificial, 473
 marco de referencia, 375
Aquino, T., 174, 192
Aristóteles, 174, 191
arquitecto, 269, 274, 311
 del proyecto, 311
 del software, 269, 311, 312
arquitectura, 16, 279, 291, 294, 317, 358, 399, 436, 482
 atributos de, 264
 de las galaxias, 12
 de las instituciones sociales, 12
 de las plantas, 10
 de los animales, 10
 de los sistemas complejos, 13
 del computador personal, 9
 estándares, 441
 orientada a objetos, 41
arquitectura
 cliente/servidor, 458

- arquitectura (*cont.*)
de módulo, 561
de procesos, 561
de von Neumann, 41
orientada a objetos, 40
asegurar la calidad, 264, 313, 318
aserción, 405
asignación, 110, 400, 410
de nombres, 186, 240
de recursos, 310
asociación, 124
atribuida, 224, 451
especificación, 279
ícono, 208
recorrido, 125, 148
refinamiento, 278, 280
reflexiva, 208
aspectos prácticos, 305
atomicidad, 464
atributo, 206
derivado, 224
- bala de plata, 25
basado en objetos, 33, 36, 43
base de datos, 273, 326, 433
en red, 449
jerárquica, 449
orientada a objetos, 41, 85, 326, 449, 455
relacional, 326, 448-50, 456
- Beck, K., 182
beneficios del modelo de objetos, 86, 328
Berard, E., 302
bibliotecario, 325
bibliotecario de clases, 84, 325
big bang (explosión), 63, 314
Biggerstaff, T., 430
bloqueo, 415
Boehm, B., 268
Booch lite, 200
Boyle, R., 170
Brooks, F., 3, 5, 27, 264
- C++, 21, 33, 37, 49, 59, 60, 75, 140, 152, 273
calidad, 155, 158, 162, 318
cambio, 296-9, 438, 506, 534
de escala, 259
campo, 562
caos, 3, 7
capa, 562
- cardinalidad, 126, 208
categoría de clases, 210, 276, 315, 386
ejemplos de, 212
ícono, 210
categorías, 168
categorización clásica, 173
caza de errores, 319
centro de control, 250
ciclo de vida,
en cascada, 266
iterativo e incremental, 265
- clase, 42
«de uso», 124, 148-50
abstracta, 207, 386-7
acoplamiento, 156
aditiva, 71, 78, 145, 392, 427
agregación, 124, 146
agrupación (cluster), 315
anidamiento, 187, 217
asociación, 124
base, 131
calidad de, 155
ciclo de vida, 123
clave, 221
cliente, 131
cohesión, 156
colección, 363
completud, 156
concreta, 130
conflicto de granularidad, 186
contenedor, 562
descubrimiento, 23, 168
diseño aislado de, 273-5
herencia, 124, 142
hoja, 130
ícono, 206
identificación, 167, 176, 269
implantación, 281
instanciación, 150
invención, 23, 168
metaclase, 124, 153
métodos, 320
naturaleza de, 120
nombrar, 186-7
papel 221
parametrizada, 152, 213
polimorfismo, 132
promoción, 186
relaciones, 123, 277

- clase (*cont.*)
 respuesta, 320
 restricción, 222
 semántica, 273
 ser primitiva, 156
 soporte, 394, 419
 suficiencia, 156
 versus objetos, 120, 154
 versus tipo, 73
 y análisis, 155
 y diseño, 155
clasificación, 23, 167, 279
 dificultad de, 169
 ejemplos de, 169
 importancia de, 167
 incremental, 171
 inteligente, 168, 171
 naturaleza incremental, 171
clave, 221
cliente, 47, 52, 149, 241
CLOS, 21, 33, 37, 41, 60, 141, 144-6
CLU, 33, 41
Coad, P., 179, 289
COBOL, 20, 32, 34, 43
Coggins, J., 382
cohesión, 64, 156
colaboración, 278-9
Coleman, D., 230
colisión de nombres, 144
compartición estructural, 110
compilador incremental, 324
complejidad, 158
 arbitraria, 5
 atributos de, 13
 combatir a, 46
 de un proceso, 5, 6
 del software, 3, 5
 desorganizada, 14, 17
 dominar, 17
 ejemplos de, 9
 esencial, 5
 espacial, 158
 evolución de, 14
 límites de, 16
 organizada, 14, 17
 sin restricciones, 9
 temporal, 158
 versus simplicidad, 6
completud, 156-7
componente, 13
 primitivo, 13
comportamiento, 4, 18, 47, 55, 95, 101, 116, 131, 179, 280, 288, 289
 de sistemas discretos, 5, 7
 ejemplos de, 102
 emergente, 11
 versus forma, 287
 y estado, 101
comprobación,
 estricta de tipos, 74, 77-80, 562, 563
computación cliente/servidor, 434, 439
conceptualización, 285
conurrencia, 31, 45, 411
 ejemplos de, 83
 significado de, 81
conexión, 256
conflicto de granularidad, 186
conservación, 6, 300
consistente respecto al tipo, 79
Constantine, I., 20
constructor, 54, 103
contención, 119, 147-8
 física, 119, 147, 220
 por referencia, 147, 220
 por valor, 147, 220
 y agregación, 119
contenedor, 118, 422
 heterogéneo, 152
 homogéneo, 152
 control de acceso, 563
 de exportación, 217
 de versiones, 315
conversión, 139
 de tipos, 38, 139
conversión forzada de tipos, 38, 139
copia, 110, 400, 410
 profunda, 111
 superficial, 111
Coplien, J., 189
CORBA, 440
cosa de otro mundo, 437
creatividad, 267, 286, 300
crisis del software, 9
Cunningham, W., 182
Curtis, B., 267
Chidamber, C., 320

- Da Vinci, L., 4
Dahl, O., 21
Darwin, C., 169
decisión de diseño, 563
 estratégica, 188, 265, 563
 táctica, 188, 265, 292, 563
DeChampeaux, D., 288
delegación, 124
DeMarco, T., 44
Demeter, 160
Demócrata, 19
densidad de defectos, 319
dependencia semántica, 126
depósito, 270
depuración de patrones, 276
desadaptación de impedancias, 5
desarrollo incremental, 265, 285, 294
desarrollo orientado a objetos,
 beneficios de, 328
 modelos de, 25, 200
 parte más difícil, 167
 riesgos de, 328
Descartes, 42, 174, 192
descomposición, 17, 24
 algorítmica, 17
 orientada a objetos, 18, 21, 43
descubrimiento, 23, 167, 185, 270
destructo, 54, 103
Deutsch, P., 73
diagrama,
 de clases, 203, 209, 212, 277, 346, 352, 360,
 364, 385-6, 392, 399, 403, 422, 426, 446,
 451, 463, 465, 486, 491, 498, 522, 525, 528
 de interacción, 204, 273, 349, 351, 408, 443,
 445, 515
 de módulos, 204, 277, 404, 531
 de objetos, 203, 204, 239, 255, 273, 277, 413,
 415, 459, 502, 523, 526
 de procesos, 204, 436, 518
 de transición de estados, 204, 229, 356, 494
diagramas de clases,
 ejemplos de, 252, 253
diccionario de datos, 270, 272, 278, 282
Dijkstra, E., 17, 21, 41
dirección y control, 509
diseño, 20, 277, 291, 358, 382, 457, 482, 519
 aislado de clases, 273-5
 alternativas, 364
 base de datos, 448
 dirigido por los datos, 20
 estructurado, 18, 20
 global circular, 564
 integridad, 433
 orientado a objetos, 26, 31, 43
 significado de, 24
 disparador, 235, 461
 dispositivo, 256
 documentación, 283, 321
 Donne, J., 52
 economía, 338, 509
 de expresión, 13
 eficiencia, 330
 Eiffel, 33, 37, 41, 74, 144, 152
 ejemplar, 124, 164
 elaboración, 108
 empaquetamiento, 163
 encapsulación inteligente, 59
 encapsulamiento, 31, 45
 ejemplos de, 56
 inteligente, 59
 significado de, 54
 enlace, 113, 243
 ícono, 240, 241
 intercomponentes, 13
 intracomponentes, 13
 entrenamiento, 326
 epistemología objetivista, 41
 escenario, 269, 272, 273, 278, 279, 287, 308, 349,
 442, 515
 primario, 289, 442
 secundario, 289, 442
espacio de estados, 564
especialización, 69, 125, 279
especie, 170
especificación, 225, 239, 281
 de clases, 226
 de módulos, 254
 de operación, 228
 de procesos, 228, 258
estado, 96
 acción, 234
 anidado, 236
 discreto, 7
 ejemplos de, 99
 historia, 238
 ícono, 230
 ortogonal, 238

- estado (*cont.*)
 semántica dinámica, 98
 semántica estática, 98
 y comportamiento, 101
estándares, 440, 441
estilo de programación, 44
estimación del riesgo, 291, 293
estructura, 131, 279, 378
 de clases, 15, 22, 43
 de objetos, 15, 22, 43
evento, 232-5
evolución, 6, 14, 294, 300, 364, 418, 468, 500,
 529
excepción, 380, 403
experto del dominio, 181, 274
exploración, 286

familia, 386, 388
 de clases, 386, 388
fichas CRC, 182, 271, 274, 290
Flavors, 33, 41, 145
Flex, 41
flexibilidad ante cambios, 21, 87
Flowmatic, 32
flujo de datos, 244
forma, 287
 intermedia estable, 24, 87
 limitada, 387
 no limitada, 387
 normal, 452
FORTRAN, 20, 32, 34
fuente de conocimiento, 480
función, 565
 aplicar (apply), 407
 de retorno (callback), 51, 83
 genérica, 565
 miembro, 38, 48, 101
 no miembro, 80, 104
 virtual, 131, 383
 virtual pura, 131

Gall, J., 14
Gane, C., 44
generador de aplicaciones, 469
generalización, 69, 125, 279
genericidad, 151
gestión, 286
 del almacenamiento, 112, 380, 399, 403
 de cambios, 296

de configuraciones, 283, 315
de versiones, 314, 315, 468
del riesgo, 306
Goldberg, A., 164, 179, 289
guión, 249

Harel, D., 230
heap, 112
Heráclito, 19
herencia, 42, 66, 127, 275
herencia múltiple, 69-72, 141
 piedra de toque, 129, 142
 sobreutilización, 72
 y colisión de nombres, 144
herencia repetida, 144
herencia simple, 66-9, 129
 anchura, 320
 árbol, 131
 bosque, 131
 ícono, 208
 para aumentar, 130
 para restringir, 130
 profundidad, 320
 semántica de, 140
 y agregación, 142
 y colisiones de nombres, 144
 y polimorfismo, 134
 y tipos, 134
herramienta, 25, 322-5, 378, 422
 papel de, 204
heurísticos, 275
hilo de control, 8, 82, 118, 159, 380
historia, 238
hitos, 272, 276, 281, 283, 287, 290, 293, 299, 301
Hoare, T., 21
Hofstadter, D., 192
hojeador, 323
Humphrey, W., 267, 303

identidad, 106
 ejemplos de, 106
 única, 109
identificación, 167, 172, 269
 de tipos en tiempo de ejecución, 79, 140
IGU, 325, 440, 466
igualdad, 110, 401, 410
implantación, 55, 121, 217, 385
 calidad, 161
Ingalls, D., 113

- ingeniería, 23
 directa, 565
 inversa, 565
 del software, 263
 tendencias, 32
ingeniero de reutilización, 313
inspección, 318
instancia, 42, 131
instanciación, 124, 150-1
integración, 314, 500
inteligencia artificial, 42, 473
interfaz, 55, 62, 121, 418
 de usuario, 438
 private, 122
 protectec, 122
 public, 122
internacionalización, 352
interrupciones, 84
invariante, 48
invención, 23, 168, 185, 271
invocación de un método, 135
IPL V, 32
iteración, 380, 399, 407
iterador, 103, 409
 activo, 407
 pasivo, 407

Jackson, M., 21
Jacobson, I., 181, 248, 289, 423
jefe de integración, 313
 de proyecto, 312
 de subsistema, 311-13
jerarquía, 13-14, 22, 45, 66
 «de partes», 13, 66
 «de tipos», 15, 42, 72
 «ejemplos de», 69
 «significado de», 66, 119
 «todo-parte», 13, 118, 119

Knuth, D., 376

Lakeoff, G., 177, 192
LAN, 437
lanzar (excepción), 404, 405
Lavoisier, 170
Lefrancois, G., 537
lenguaje,
 basado en objetos, 33, 36, 42-4
 de cuarta generación, 469

funcional, 407
generaciones de, 32
orientado a objetos, 33, 36, 37, 41
sin tipos, 80
topología, 34-38
Ley de Demeter, 160
librería de clases, 381
 de tareas ATT, 84, 159
ligadura, 282
 dinámica, 80
 estática, 80
 tardía, 80
 y polimorfismo, 134
Linneo, C., 169
Lippman, S., 103
Liskov, B., 55, 61, 69
Lisp, 32
lista de precedencia de clases, 145
lista-guion, 300
Locke, 174
LOOPS, 33, 41

llamada a procedimiento remoto, 463

MacApp, 467
macroproceso, 268, 283
mantenimiento, 6, 300, 373, 426, 470, 504
marco, 359, 360
marco de referencia, 190, 375, 418, 429, 478-9
mecanismo, 13, 22, 161, 189, 467
 de almacenamiento, 400
 de bloqueo, 415
 de interfaz de usuario, 370
 de persistencia, 427
 de suposición, 502
 de transacción, 463
 de visualización, 369, 528
IGU, 467
MVC, 114, 190
sensor, 366
mecanismo SQL, 459
 adquisición de datos, 528
 búsqueda, 187
 excepción, 403
 gestión del almacenamiento, 399
 iteración, 407
 marco, 360
 paso de mensajes, 518
 sincronización, 411

- Mellor, S., 178, 287
Mendeleiev, 170
mensaje, 241-2
Mesa, 41
metaclase, 124, 153, 214
 ícono, 214
método, 19, 25, 26, 38, 48, 101, 104
 invocación, 135-8
 de grano fino, 158
métricas, 155, 272, 276, 281, 283, 287, 290, 293,
 299, 301, 318
Meyer, B., 164
microorganización, 389
micropoproceso, 268, 269
Miller, G., 17, 22, 28
Mills, H., 21
Minsky, H., 42, 335
modelo,
 conectividad, 259
 del desarrollo orientado a objetos, 25, 200
 físico, 203
 importancia de, 24
 lógico, 203
 semántica dinámica, 204
 semántica estática, 204
 y vista, 202
modelo de bucle de eventos, 468
modelo de callback de eventos, 468
modelo de contratos, 47, 121
modelo de objetos, 16, 32, 38, 44
 aplicaciones de, 88
 beneficios de, 86
 elementos de, 44
 fundamentos de, 38, 40
modelo físico, 203
modelo lógico, 203
modificador, 54, 103
modismo, 189
Modula, 41
modularidad, 31, 45
 ejemplos de, 65
 significado de, 60
modularización, 61, 62
modularización inteligente, 63
módulo, 62
 dependencia, 250
 ícono, 251
monitor, 415
monomorfismo, 81, 133
muchos a muchos, 126
multitarea, 82
navegación, 125, 148
Neighbors, J., 180
nivel de abstracción, 72
no determinista, 17
normalización, 452
nota, 224
notación, 25, 199, 323
 adaptación, 254, 257
 Booch lite, 200
 cambio de escala, 259
np-completo, 446
Object Management Group, 90, 440
Object Pascal, 21, 37, 41, 60
objeto, 18, 38, 40, 42, 97
 activo, 106, 115, 246
 agente, 115
 agregación, 113
 alias, 110
 asignación, 110
 calidad de, 155
 clave, 244
 como máquina, 105
 comportamiento de, 101
 control, 423, 497
 creación, 112
 destrucción, 112
 difuso, 97
 ejemplos de, 115
 enlace, 113, 114
 estado, 98
 ícono, 239
 identidad de, 106
 identificación, 269, 270
 igualdad, 110, 111
 implantación, 281
 naturaleza de, 95
 papel, 104, 115, 244
 pasivo, 106
 relaciones, 113, 277
 representación, 59, 122
 responsabilidades, 48, 104
 restricción, 244
 secuencial, 118
 semántica, 272
 servidor, 115

- objeto (*cont.*)
 sincronización, 118
 síncrono, 118
 tangible, 96
 versus clase, 120, 154
 vigilado, 118
 visibilidad, 117
 y análisis, 155
 y diseño, 155
ocultación, 60
 de información, 54
OLE, 441
OMG, 90, 440
Open Look, 466
operación, 48, 101-104, 274
 abstracta, 567
 calidad de, 157
 constructor, 54, 103
 de clase, 567
 de grano fino, 158
 destructor, 54, 103
 iterador, 103
 modificador, 54, 103
 selector, 54, 103
orientado a objetos, 38
OS/2, 83
OSI, 440
- Page-Jones, M., 20, 294
papel, 104, 221
 del equipo de desarrollo, 310
parametrización de tipos, 152
Parnas, D., 63, 268, 303
partición, 567
Pascal, 32, 43
paso de mensajes, 101, 114, 118, 246, 518
 asíncrono, 159
 con abandono inmediato, 159
 con cuenta atrás, 159
 con retorno inmediato, 159
 síncrono, 159
patrón, 14, 189, 275, 276, 312, 379, 380
pérdida de memoria, 109, 110
persistencia, 31, 45, 84, 427
 en el espacio, 86
 en el tiempo, 86
Petroski, H., 23, 197
Piaget, J., 174, 193
PL/1, 32
- planificación, 257, 258, 294, 307
arquitectura, 292
de escenarios, 289, 290
de procesos, 257
de tareas, 306-307
de versiones, 292, 293, 364
desplazante, 258
ejecutiva, 258
manual, 258
Platón, 174, 191
polimorfismo, 81
 múltiple, 145
 paramétrico, 133
polimorfismo simple, 132
 ad hoc, 133
 piedra de toque, 133
 y herencia, 134
 y ligadura tardía, 134
 y sentencia switch, 133
 y sobrecarga, 133
política, 385
 de gestión de memoria, 112
 táctica, 291
POSIX, 440
postcondición, 48
precondición, 48
Presentation Manager, 467
presupuesto de tiempo, 247
principio de mínimo compromiso, 47
private, 122, 217
procesador, 255, 257
proceso, 25
 de diseño racional, 267
 ligero, 82
proceso pesado, 82
 complejidad de, 5
 ligero, 82
 macro, 268, 283
 madurez, 267
 micro, 268, 269, 295
 pesado, 82
 racional, 267
programación al por mayor, 38
 orientada a objetos, 42
programador de aplicaciones, 311-13
promoción de clases, 186
propiedad, 73, 98, 219
 estática, 219
 virtual, 219

- protected, 122, 217
protocolo, 48, 273, 274, 381, 393, 420
 de realización (commit), 464
prototipo, 285, 295, 347
 de comportamiento, 295
proveedor, 149, 241
prueba, 316
 de subsistema, 316
 del sistema, 316
 unitaria, 316
public, 122, 217
puntero tapado, 125
punto funcional, 180, 288, 293
- Rand, A., 42,
recorrido, 283, 308, 318
referencia, 147, 395
 versus valor, 395
referencia colgante, 109
relación, 277
 calidad, 160
 cliente/proveedor, 118
 de antigüedad, 113
 de hermano a hermano, 118
 entre clases, 123
 entre objetos, 113
 interesante, 277
 todo parte, 118
 de uso, 124, 148, 149, 208
 padre/hijo, 113
Rentsch, T., 32
representación, 59, 122, 161, 281, 282
requisitos, 5, 289, 475, 506, 511
responsabilidad, 48, 104, 272-5
responsable de herramientas, 313
restricción, 222
 temporal, 236
reusabilidad, 158
reutilización, 317, 340, 375
revisión, 311
riesgo, 307, 329
 de eficiencia, 330
Ross, D., 178
Rubin, K., 164, 179, 289
Rumbaugh, J., 206, 224, 230, 248, 456
- secuencial, 387
seguro respecto al tipo, 78, 140, 150
Seidewitz, E., 113, 184
- selector, 54, 103
semáforo, 415
semántica, 238
 funcional, 157
 tiempo y espacio, 158, 391
sentencia switch, 133
separación de intereses, 8, 114
ser primitivo, 156
servicio, 568
servidor, 47, 51, 568
Shaw, 35, 42, 46, 91
Shlaer, 178
Simon, H., 13, 23, 27, 192
Simula, 32, 41, 96
sincronización, 118, 246, 380, 411
síncrono, 118, 388, 412, 414
sistema,
 abierto, 437
 análogo, 8
 casi descomponible, 13
 con éxito, 16, 263
 concurrente, 387
 continuo, 8
 de gestión de información, 438
 descomponible, 43
 discreto, 7
 distribuido, 86
 en caída libre, 265
 evolución de, 14
 forma canónica, 14
 secuencial, 387
 operativo, 41
 operativo orientado a objetos, 41
 reactivo, 568
 transformacional, 568
sistemas legados, 325
Smalltalk, 21, 33, 37, 41, 73, 83, 114, 141, 159, 210, 273
sobrecarga, 133
software,
 como una inversión de capital, 6
 complejidad inherente, 3, 5
 conservación, 6
 de dimensión industrial, 4, 32, 266
 evolución, 6
 flexibilidad de, 7, 6
 mantenimiento, 6
software de dimensión industrial, 4, 32
SQL, 453, 463

- SQL3, 440
storyboarding (narración de sucesos), 273
Stroustrup, B., 24, 43, 90, 151, 164, 264, 380
subclase, 69, 71, 128, 131, 397
subestado, 237
subprograma libre, 80, 104
subsistema, 352, 315, 316, 531
 ícono, 251
suficiencia, 156
superclase, 69, 71, 129, 131, 397
superestado, 237
suposición, 403

tabla, 449
tasa de descubrimiento de errores, 299, 318
taxonomía, 169
teoría de prototipos, 173, 176
Tesler, L., 80
tiempo real, 326, 387
tipo,
 abstracto de datos, 43
 conversión, 139
 definido por el usuario, 395
 parametrizado, 140
 predefinido, 395
 versus clase, 73
todo-parte, 118, 124, 126
Tomlin, L., 1
TOOLS, 90
topología, 34-8
trama de herencias, 71, 462
transacción, 463
transferencia tecnológica, 327
transición,
 de estados, 231
 condicional, 234
 ícono, 231
transiciones de estado condicionales, 234
Trellis/Owl, 73

tubería (pipe), 463

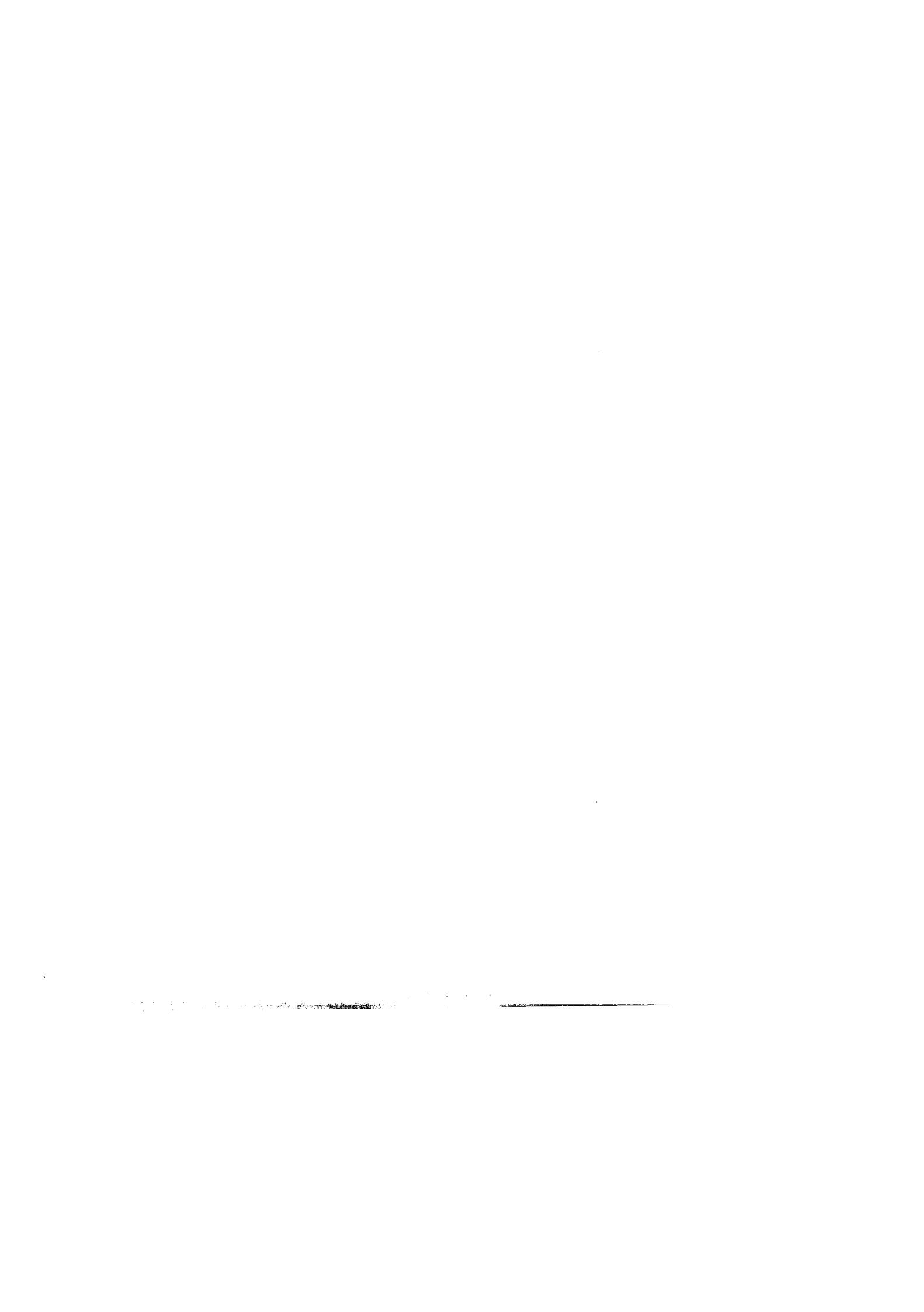
UNIX, 82
uno a muchos, 126
uno a uno, 126
uso de tipos, 31, 45, 73
 beneficios de, 80
 débil, 74
 estricto, 74, 77
 ligadura dinámica, 80
 ligadura estática, 80
 versus herencia, 134
utilidad de clase, 104
 ícono, 216

valor, 147, 395
 versus referencia, 395
versión, 291, 294, 299, 315
vigilado, 118, 388, 412, 414
visibilidad, 117, 161, 244
 compartida, 161
visión arquitectónica, 264
vista, 25, 200, 202
 externa, 57
 interna, 57
vocabulario, 188

Ward, P., 44, 183
Wegner, P., 32, 43, 133
Weinberg, P., 201
Windows, 467
Wirfs-Brock, R., 105, 179, 289, 418
Wirth, N., 21, 163
Wright, F., 4
Wulf, W., 22

X Windows, 466

Yourdon, E., 20, 44, 179



La Tarjeta de Referencia Booch de Rational (*Rational Booch Reference Card*) es un glosario práctico de la notación completa de Booch. Si desea una copia gratis contacte con Rational en cualquiera de los siguientes números de teléfono, o bien por Internet con booch-card@rational.com

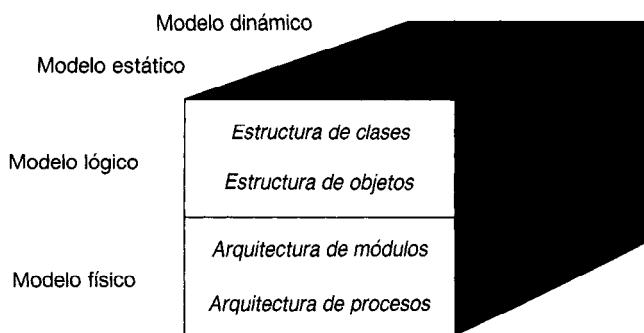


Santa Clara, CA 800-767-3237 o 408-496-3600

Francia +33-1-47-17-41-77, Alemania +49-89-797-021, Reino Unido +44-273-204733, Suecia +46-8-761-0600,
Australia +61-3-521-0507, Taiwan +886-2-720-1938, Italia -39-2-264-0107, España +34-1-279-72-56

Modelos de Análisis y Diseño Orientado a Objetos

Múltiple soporte, vistas interrelacionadas de un sistema de desarrollo.



El Proceso de Desarrollo Orientado a Objetos

Soporta el desarrollo iterativo e incremental de un sistema.

Macroproceso

- *Establecer los requisitos centrales (modelo conceptual).*
- *Desarrollar un modelo del comportamiento deseado (análisis).*
- *Crear una arquitectura (diseño).*
- *Transformar la implementación (evolución).*
- *Gestionar la evolución posterior a la entrega (mantenimiento).*

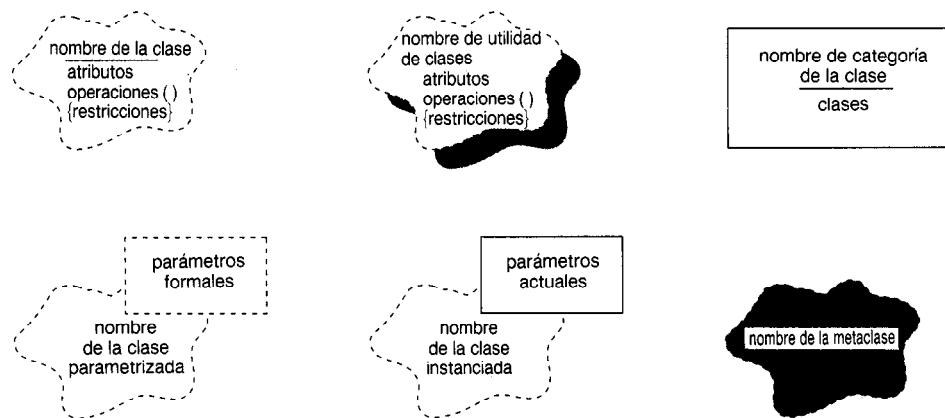
Microproceso

- *Identificar clases y objetos a un nivel dado de abstracción.*
- *Identificar la semántica de estas clases y objetos.*
- *Identificar las relaciones entre estas clases y objetos.*
- *Especificación del interfaz y después la implementación de estas clases y objetos.*

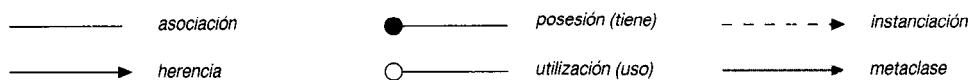
Diagrama de Clases

Muestra la existencia de clases y sus relaciones en la vista lógica de un sistema.

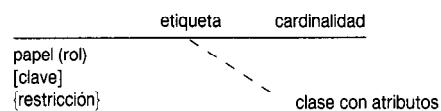
Iconos de clases



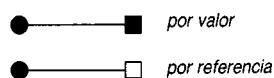
Relaciones de clases



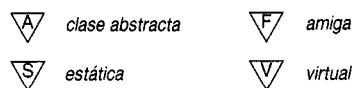
Marcas de las relaciones



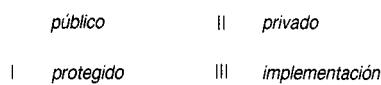
Marcas de contención



Propiedades



Control de exportación



Anidamiento



Notas

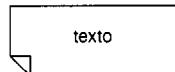


Diagrama de Módulos

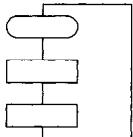
Muestra la asignación de clases y objetos a módulos en la vista física de un sistema.

Iconos de los módulos

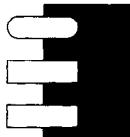
programa principal



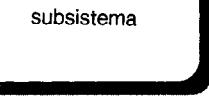
especificación



cuerpo



subsistema



Dependencia



Diagrama de Procesos

Muestra la asociación de procesos a procesadores en la vista física de un sistema.

Iconos



proceso 1
proceso 2
...
proceso 3



Conexión rótulo

Diagrama de Métricas

Medida de la bondad.

Calidad del sistema

- Estabilidad.
- Densidad de defectos.
- Tasa de descubrimiento de defectos.

Tamaño/Complejidad

- Número de clases.
- Clases por categoría.

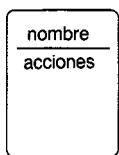
Calidad de la clase

- Número de operaciones.
- Figura de tramas de herencias.
- Número de hijos.
- Acoplamiento/Cohesion.
- Respuesta.
- Primitiva/suficiente/completa.

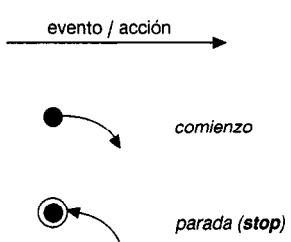
Diagrama de Transición de Estados

Muestra la existencia de clases y sus relaciones en la vista lógica de un sistema.

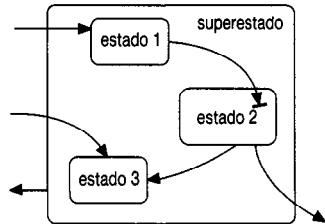
Icono de estado



Transiciones entre estados



Anidamiento



History



Diagrama de Objetos

Muestra la existencia de objetos y sus relaciones en la vista lógica de un sistema.

Icono de un objeto



Sincronismo

- simple
- ⤠ sincrónico
- ⤠ abandono inmediato
- ⤠ intervalo (de espera)
- ⤠ asincrónico

Visibilidad

- [G] global
- [P] parámetro
- [F] campo
- [L] local

Enlace

orden: mensaje
objeto/valor →

papel (rol)
[clave]
{restricción}

Diagrama de Interacción

Traza de la ejecución de un escenario.

Guion

