

```
Public Class Producto
    Public Property Vprecio As String
        Get
            Return Vprecio
        End Get
        Set(ByVal value As String)
            Vprecio = value
        End Set
    End Property
    Public Overrides Function PrecioDescuento() As Double
        Return Me.Precio * 0.9
    End Function
End Class
Public Class Producto
    Inherits Producto
    Public Overrides Function PrecioDescuento() As Double
        Return Me.Precio * 0.7
    End Function
End Class
Public Class Auto
    Private Vpatente As String
    Private Vcolor As String
    Public Property Vpatente As String
        Get
            Return Vpatente
        End Get
        Set(ByVal value As String)
            Vpatente = value
        End Set
    End Property
    Public Overrides ReadOnly Property DatosAuto() As String
        Get
            Return Me.Vpatente & ", " & Me.Vcolor
        End Get
    End Property
    End Class
    Public Class Auto
        Inherits Auto
        Public Overrides ReadOnly Property DatosAuto() As String
            Get
                Return "Los datos del auto son:"
                "Patente:" & vbTab & MyBase.DatosAuto
                "Color:" & vbTab & MyBase.DatosAuto.Split("")(1)
            End Get
        End Property
    End Class
    Public MustInherit Class Alumno
        Public MustOverride Function TasaAlumno() As Double
    End Class
    Public Class AlumnoUniversitario
        Inherits Alumno
        Public Overrides Function TasaAlumno() As Double
            Dim T As Double
            Dim C As Integer
            T = 1000000 + Integer.Tan((C - 1) * 1000000)
            Return T
        End Function
    End Class

```



Universidad Abierta
Interamericana



ORIENTACIÓN A OBJETOS

TEORÍA Y PRÁCTICA

ORIENTACIÓN A OBJETOS

TEORÍA Y PRÁCTICA

**GRADY BOOCH
DARÍO CARDACCI**

Cardacci, Darío
Orientación a objetos : teoría y práctica / Darío Cardacci y Grady Booch. - 1a ed. - Buenos Aires : Pearson Education, 2013.
388 p. : il. ; 18x23 cm.
E-book
ISBN 978-987-615-377-5
1. Informática. 2. Programación. 3. Software. I. Booch, Grady II. Título CDD 005.3

Gerente Editorial de Educación Superior, Pearson Latinoamérica: Marisa de Anta •

Gerente de Contenidos y Tecnología Educativa: Gonzalo Miranda • **Editora:** Victoria Villalba •

Diseño de portada y de interiores: Eclipse Gráfica Creativa SRL • **Edición y corrección:** Paula Budris •

Versión digital: Patricio Szczygiel y Victoria Villalba

Orientación a objetos. Teoría y práctica

Autores • Grady Booch y Darío Cardacci

ISBN versión impresa • 978-987-615-356-0

ISBN versión digital • 978-987-615-377-5

Primera edición • 2013

© Pearson Education SA • 2013

Av. Belgrano 615, piso 11, C1092AAG

Ciudad Autónoma de Buenos Aires, Argentina

La obra original, *Object-Oriented Analysis and Design with Applications. Second Edition*, de Grady Booch, fue publicada originalmente en inglés por Addison-Wesley Professional, Copyright © 1994, ISBN 978-0-8053-5340-2.

Las partes 1 a 4, el glosario y el vocabulario técnico-bilingüe incorporados en el presente libro fueron recuperados, con autorización, de su versión en español, *Análisis y diseño orientado a objetos con aplicaciones*, de Grady Booch, editada por Addison-Wesley Iberoamericana S.A., México, Copyright © 1998, ISBN 978-968-444-352-5.

Visual Studio 2012, Visual Basic, .NET y Visual Basic.NET son marcas registradas de Microsoft.

Queda hecho el depósito que marca la ley 11.723.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

Índice

Carta de la Universidad.....	11
Prefacio	13
Introducción.....	15
<hr/>	
PRIMERA SECCIÓN: CONCEPTOS por Grady Booch	
1 Complejidad.....	21
1.1 La complejidad inherente al software	21
Las propiedades de los sistemas de software simples y complejos.....	21
Por qué el software es complejo de forma innata.....	22
Las consecuencias de la complejidad ilimitada.....	26
1.2 La estructura de los sistemas complejos	26
Ejemplos de sistemas complejos.....	26
Los cinco atributos de un sistema complejo.....	29
Complejidad organizada y desorganizada.....	31
1.3 Imponiendo orden al caos	34
El papel (rol) de la descomposición	34
El papel (rol) de la abstracción	38
El papel (rol) de la jerarquía	39
1.4 Del diseño de sistemas complejos	39
La ingeniería como ciencia y como arte.....	39
El significado del diseño.....	40
2 El modelo de objetos.....	47
2.1 La evolución del modelo de objetos	47
Tendencias en ingeniería del software	47
Fundamentos del modelo de objetos.....	54
POO, DOO y AOO	55

2.2 Elementos del modelo de objetos.....	60
Tipos de paradigmas de programación.....	60
Abstracción	61
Encapsulamiento.....	69
Modularidad.....	74
Jerarquía	79
Tipos (tipificación)	85
Concurrencia	93
Persistencia.....	95
2.3. Aplicación del modelo de objetos	97
Beneficios del modelo de objetos.....	97
Aplicaciones del modelo de objetos.....	98
Problemas planteados	99
3 Clases y objetos	105
3.1 La naturaleza de los objetos	105
Qué es y qué no es un objeto.....	105
Estado	107
Comportamiento	110
Identidad	115
3.2. Relaciones entre objetos.....	122
Tipos de relaciones.....	122
Enlaces.....	123
Agregación	127
3.3. La naturaleza de una clase	128
Qué es y qué no es una clase.....	128
Interfaz e implementación	129
Ciclo de vida de las clases.....	131
3.4 Relaciones entre clases.....	131
Tipos de relaciones.....	131
Asociación	133
Herencia	135
Agregación	155
Uso	156
Instanciación (creación de instancias)	157
Metaclases	160

3.5 La interacción entre clases y objetos	162
Relaciones entre clases y objetos.....	162
El papel de clases y objetos en análisis y diseño	162
3.6 De la construcción de clases y objetos de calidad.....	163
Medida de la calidad de una abstracción.....	163
Selección de operaciones.....	165
Elección de relaciones.....	167
Elección de implementaciones	168
4 Clasificación.....	175
 4.1 La importancia de una clasificación correcta	175
Clasificación y diseño orientado a objetos.....	175
La dificultad de la clasificación.....	176
 4.2 Identificando clases y objetos	179
Enfoques clásicos y modernos	179
Análisis orientado a objetos	184
 4.3 Abstracciones y mecanismos clave	191
Identificación de las abstracciones clave.....	191
Identificación de mecanismos.....	193
<hr/>	
SEGUNDA SECCIÓN: LA IMPLEMENTACIÓN por Dario Cardacci	
Consideraciones tecnológicas de la implementación.....	203
De la teoría a la práctica.....	203
La notación utilizada	203
El lenguaje de programación	204
5 Clases	205
 5.1 Creación de una clase	205
 5.2 Tipos de clases.....	205
5.2.1 Ámbitos de las clases	206
 5.3. Estructura de una clase	209
5.3.1. Variables internas de la clase (campos – variables miembro)	209
5.3.2. Acceso a las características de la clase (propiedades – getters y setters)	210
5.3.3. Acciones de la clase (métodos)	218
5.3.4. Constructores	221
5.3.5. Destructores.....	222
5.3.6. Eventos	225

5.4. Relaciones entre clases.....	231
5.4.1. Herencia.....	231
5.4.1.1. Tipos de Herencia.....	233
5.4.1.2. Sobrecarga.....	234
5.4.1.3. Sobrescritura.....	235
5.4.1.4. Polimorfismo.....	239
5.4.2. Agregación	243
5.4.3. Asociación	251
5.4.4. Uso.....	253
5.5. Interfaces.....	255
Ejemplo de herencia múltiple de interfaces:	261
Ejemplo sobre cómo funciona el tipado por interfaz:	262
Ejemplo de interfaz anidada:	267
5.5.1 La interfaz IComparable	269
5.5.2 La interfaz IComparer	274
5.5.3 La interfaz IClonable	282
5.5.4 Las interfaces IEnumerable e IEnumerator	291
5.6. Tipos	295
Tipos genéricos.....	296
5.7. Comunicación entre aplicaciones distribuidas	300
La aplicación cliente.....	323
6 Introducción a la arquitectura de software	333
Capa de datos.....	337
La clase VaComando	349
La clase VaDatosParametros	351
Capa de interfaces	352
Capa de estructura	352
Capa de lógica.....	354
Capa de vista	355
Conclusiones	368
Glosario	369
Vocabulario técnico bilingüe	381

Carta de la Universidad

La Universidad Abierta Interamericana, en su rol de institución formadora de hombres y mujeres que ejerzan sus profesiones en un marco ético, de valores y en busca de la excelencia, ha participado en la gestión de la presente obra con los objetivos de consolidar y fortalecer la relación existente entre el mundo académico y el editorial, así como también el de poner a disposición de los alumnos y de los lectores en general un material que se ajusta a los requerimientos actuales de quienes abordan temas relacionados con la orientación a objetos.

Esta obra combinada de dos autores de jerarquía ha permitido que se vea reflejado en este libro un equilibrado tratamiento de los conceptos teóricos referidos a la orientación a objetos, aportados por Grady Booch, más la adaptación práctica que Darío Cardacci, titular de la asignatura Programación Orientada a Objetos en la Facultad de Tecnología Informática, ha sabido plasmar para que los contenidos se ajusten a los lineamientos utilizados en su cátedra.

En nuestra visión sobre la formación de profesionales que piensan como hombres de acción y actúan como hombres pensantes, contar con bibliografía que refuerce esa forma de abordaje se torna muy significativo para el actuar cotidiano de los docentes.

La obra ha logrado recopilar, ordenar y sistematizar las experiencias y prácticas que se producen cotidianamente en las aulas con el objetivo de resumirlas y adaptarlas para un mejor aprovechamiento por parte del lector.

Cabe mencionar que el presente material no intenta agotar un tema tan amplio como la orientación a objetos, pero consideramos que se puede transformar en una herramienta estratégica fundamental para todos aquellos que desean comenzar a conocer este apasionante mundo y esta peculiar forma de desarrollar software.

Esperamos que nuestros alumnos se sientan a gusto y puedan aprovechar estas páginas, ya que se han desarrollado pensando en ellos y en la manera en la que les podría resultar más valioso este material.

Estamos seguros de que sabrán valorar el esfuerzo hecho por ambas instituciones para acercarles un libro a la medida de sus expectativas y las necesidades de la asignatura donde se gestó.

Dr. Marcelo De Vincenzi
Vicerrector de Gestión y Evaluación
Decano de la Facultad de Tecnología Informática
Universidad Abierta Interamericana

Prefacio

En diversos aspectos de la vida podemos observar cómo buenas ideas se desvanecen si no poseen una buena implementación en la práctica. Se las cataloga como aspectos teóricos o filosóficos, pero no como prácticas posibles de ejecutar cotidianamente.

El desarrollo de software se ha constituido a nivel global como una de las industrias más importantes y con una expectativa de crecimiento difícil de estimar. En un mundo cada vez más conectado con los procesos productivos y de gestión, *virtualizándose* a un ritmo que se acelera con el paso del tiempo, es fundamental poder desarrollar software de calidad.

Si bien existen muchas metodologías conviviendo al mismo tiempo, considero que la orientación a objetos, por su trayectoria y proyección, merece un lugar propio.

Cuando nos referimos a orientación a objetos, en términos generales se produce una disociación entre lo que el modelo plantea y la aplicación del mismo.

Creemos que esta bibliografía colabora para que se pueda relacionar el marco teórico conceptual con la implementación práctica, aspectos que conforman extremos sobre un mismo eje de trabajo al desarrollar software.

Durante los últimos 20 años ejercí la docencia, algo por lo que estoy sumamente agradecido, ya que sin ella no hubiese podido investigar y aprender cosas nuevas cada día. He podido observar que en innumerables ocasiones, en función de las características de los alumnos, tuvimos que adaptar contenidos de los libros, complementándolos con prácticas que cubrieran las necesidades puntuales de una asignatura o exemplificando aspectos que eran bien abordados pero tal vez un tanto abstractos para ser comprendidos al momento de desarrollarlos. Todo esto lo hicimos para salvar la brecha entre la teoría y la práctica. Es así como, con el paso del tiempo, fuimos acumulando material y experiencia en cómo enseñar y transmitir algunos temas de estudio.

La propuesta de poder compilar una parte de ese material para que acompañe a los estudiantes desde la bibliografía fue sumamente significativa y atractiva, no solo por los contenidos en sí, sino porque sería un libro combinado.

Al elegir coautor, me sedujo la idea de que fuera Grady Booch, quien ha tenido la deferencia de permitirme trabajar con su libro y sus conceptos. Sus conocimientos sobre el tema y el enfoque que le imprime me resultan muy adecuados para que los estudiantes puedan trabajar con ellos. Llevar a la práctica sus conceptos ha representado una gran responsabilidad.

La parte práctica de este libro ha sido el resultado de innumerables charlas con colegas universitarios, profesionales del ámbito con quienes interactúo y alumnos en debates durante las jornadas universitarias.

En estas páginas se condensan la experiencia y el esfuerzo de intentar transmitir de manera concreta la orientación a objetos, pero sobre todo la esperanza de que a los lectores les permita acercarse desde la práctica a un tema apasionante para el desarrollo de software.

Fueron muchas las personas que me acompañaron en este proyecto, un profundo agradecimiento a todas ellas. Quisiera mencionar a quienes tuvieron una incidencia directa sobre él. La Facultad de Tecnología Informática de la Universidad Abierta Interamericana y sus autoridades, que con Pearson han hecho este proyecto posible. Al Dr. Marcelo De Vincenzi, quien ha impulsado este trabajo incansablemente; a mis colegas Pablo Vilaboa, Carlos Neil, Leonardo Ghigliani, Luis Perdomo y Diego Barnech, quienes me han brindado sus conocimientos y tiempo para poder realizar el libro; a mis alumnos Matías Teragni, Darío García y Nicolás Aguirrezzarobe, quienes brindaron su visión sobre el abordaje de los prácticos; y a todos aquellos maestros que la vida me ha puesto en el camino para enseñarme, guiarme y orientarme en la construcción del conocimiento.

En un párrafo muy especial quiero agradecerle a mi familia; Rosario y Antonio, mis padres, pues de ellos recibí siempre sabiduría; Cecilia, mi compañera incondicional, sin dudas pocas cosas podría realizar sin su apoyo, amor y comprensión; y María Sol, Ezequiel y Martín, las estrellas que alegran mi vida. ¡Gracias por compartirme sus tiempos!

Darío Cardacci

Introducción

Por Darío Cardacci

La presente obra se ha constituido con la idea de ser utilizada como material universitario de apoyo a cursos de programación orientada a objetos.

Se ha recuperado una parte teórica de Grady Booch,¹ que abarca los elementos más importantes del modelo orientado a objetos y se complementa con ejemplos que acompañan cada concepto tratado.

El lenguaje de programación Visual Basic .NET y la herramienta de desarrollo Visual Studio 2012 que se utilizan fueron seleccionados en función de las prácticas realizadas en el curso de programación orientada a objetos que dio origen a la segunda sección del presente libro. Esta singularidad no representa mayores inconvenientes al momento de escribir los ejemplos en otros lenguajes de programación, siempre que estos posean capacidades para crear clases y generar objetos. Por razones referidas a la extensión del libro, los ejemplos fueron desarrollados solo en un lenguaje. Con el conocimiento apropiado de la sintaxis del lenguaje deseado, le proponemos al lector que, con la ayuda de su instructor, lo reescriban. Esto seguramente les permitirá llegar a la conclusión de que el lenguaje de programación no es más que una herramienta y lo importante es la manera en que se implementan los conceptos que le dan vida al modelo orientado a objetos. En adelante expresaremos OO para referirnos a orientación a objetos o el modelo orientado a objetos indistintamente.

En particular, el lenguaje de programación seleccionado en este material se utiliza por su sencillez sintáctica y semántica, lo cual es altamente beneficioso para los alumnos que se inician en el apasionante mundo de la orientación a objetos. Esto se ha hecho sin desconocer los riesgos que se corren ya que potencialmente, tanto el lenguaje como la herramienta permiten hacer cosas que escapan a la propuesta de la OO. Dejo en manos del estudiante y del instructor que lo acompañe la responsabilidad de ajustarse a los ejemplos propuestos para no apartarse del modelo.

Como el lector podrá observar, a lo largo de la bibliografía los ejemplos se desarrollan de manera incremental, comenzando por aspectos sumamente básicos hasta alcanzar niveles medios

¹ Las partes 1 a 4, el glosario y el vocabulario técnico-bilingüe incorporados en el presente libro fueron recuperados, con autorización, de su versión en español, *Análisis y diseño orientado a objetos con aplicaciones*, de Grady Booch, editada por Addison Wesley Iberoamericana S.A., México, © 1998, ISBN 978-968-444-352-5. La obra original, *Object-Oriented Analysis and Design with Applications. Second Edition*, de Grady Booch, fue publicada originalmente en inglés por Addison-Wesley Professional, Copyright © 1994, 978-0-8053-5340-2.

de complejidad. Entiendo que esta forma de abordaje colaborará para lograr una mejor comprensión de los temas. Sobre algunos puntos se ha sido reiterativo intencionalmente; esto no es casualidad, sino la voluntad de poner énfasis en esos conceptos para que se puedan fijar profundamente debido a su importancia conceptual.

La segunda sección comienza con las consideraciones tecnológicas de la implementación, La segunda sección comienza con las consideraciones tecnológicas de la implementación, donde se plantean los detalles referidos a la notación y a las herramientas tecnológicas utilizadas. Al inicio del capítulo 5 se tratan los aspectos relacionados con las clases, cómo se crean, los ámbitos para definir su visibilidad, su estructura interna en términos de campos, propiedades, métodos y eventos, los constructores y destructores y la manera en que se pueden relacionar. En particular, se le dedica un espacio importante a la relación de herencia y al comportamiento polimórfico que se puede generar a partir de ella. La relación de agregación merece también un exquisito tratamiento por sus implicancias. A esta altura, el lector podrá comenzar a desarrollar sus propias clases y relacionarlas. Luego se abordan los temas de interfaces y tipos. Aquí se comprenderá cómo crear interfaces personalizadas y también cómo aprovechar de la mejor manera las provistas por el framework .NET. En este mismo punto se trabaja el concepto de “tipo”, un aspecto fundamental para obtener lo mejor del modelo OO. Teniendo un manejo fluido de lo tratado, el lector podrá potenciar sus desarrollos OO y, al mismo tiempo, dotarlos de flexibilidad y extensibilidad. Para finalizar el capítulo se hace foco en el manejo de socket. Esto nos habilitará a construir aplicaciones distribuidas y conectarlas, permitiendo que los programas posean la capacidad de intercambiar datos.

A modo de cierre e integración de los contenidos tratados, en el capítulo 6 se realiza una introducción a la programación en capas. Esta introducción intenta presentar la manera de estructurar un desarrollo para que sea robusto y que las piezas de software que lo constituyen sean independientes y reutilizables, otorgándole al producto final niveles aceptables de acoplamiento y cohesión. Cabe destacar que este material no intenta abarcar en su totalidad los aspectos que propone la OO. Simplemente, sobre aquellos que han sido abordados, presentar ejemplos sencillos y eficaces para su implementación práctica.

Finalmente, es oportuno destacar que la obra logra amalgamar teoría y práctica sobre una columna vertebral: la orientación a objetos. Esto le otorga al lector la posibilidad de desplazarse a gusto desde los conceptos hacia la implementación y viceversa, haciendo énfasis en los aspectos que más deseé desarrollar. Es muy importante que cuando avance sobre la segunda sección del libro, el lector no dude en buscar referencias y releer la primera parte si percibe la implementación como algo muy abstracto o sin posibilidades de asimilarse con los conceptos tratados anteriormente.

PRIMERA SECCIÓN

Conceptos

Por Grady Booch

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.

1

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 piensa 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 conforme 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. También existen 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 correcto funcionamiento. 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 conllevan 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, costo, 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 problema. 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, capacitándolos 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

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

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 solo 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 arriba.¹ 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

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 en 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*.

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.

Las consecuencias de la complejidad ilimitada

“Cuanto más complejo sea el sistema, más abierto está al derrumbamiento total” [5]. Un constructor raramente pensaría 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 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

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.*).

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 solo 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 solo 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 solo 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 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. Solo 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 solo 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, observe 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, 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 general 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 solo 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 solo unas pocas clases diferentes de subsistemas en varias combinaciones y disposiciones” [11].

En otras palabras, los sistemas complejos tienen patrones comunes. Estos patrones pueden llevar 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, solo 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*). Se llama a esas jerarquías *estructura de clases* y *estructura de objetos*, respectivamente.³

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

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.

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 manifestase 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: solo a título muy excepcional se encuentran sistemas de software que se entreguen a tiempo, 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 factor 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,

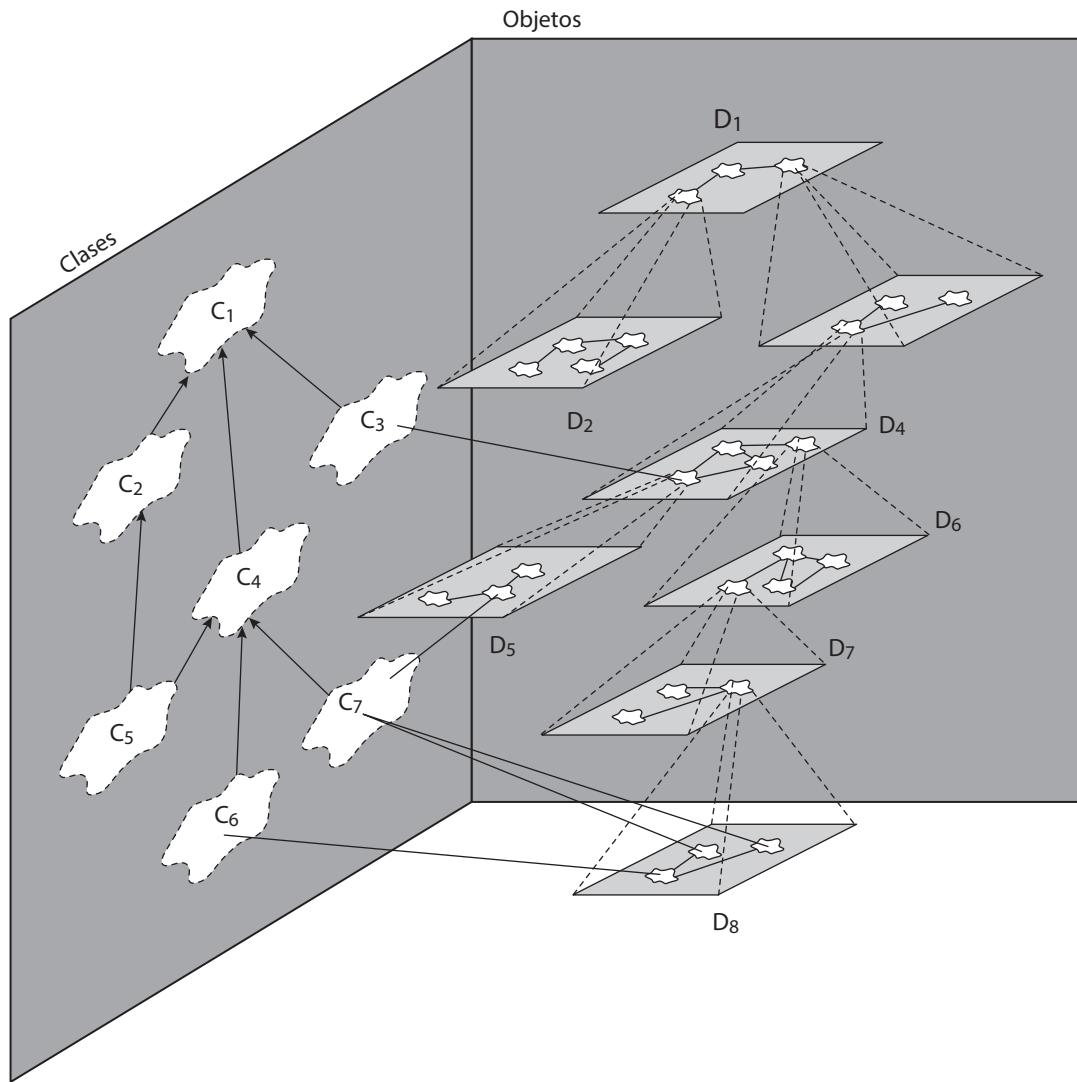


Figura 1.1. La forma canónica de un sistema complejo.

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

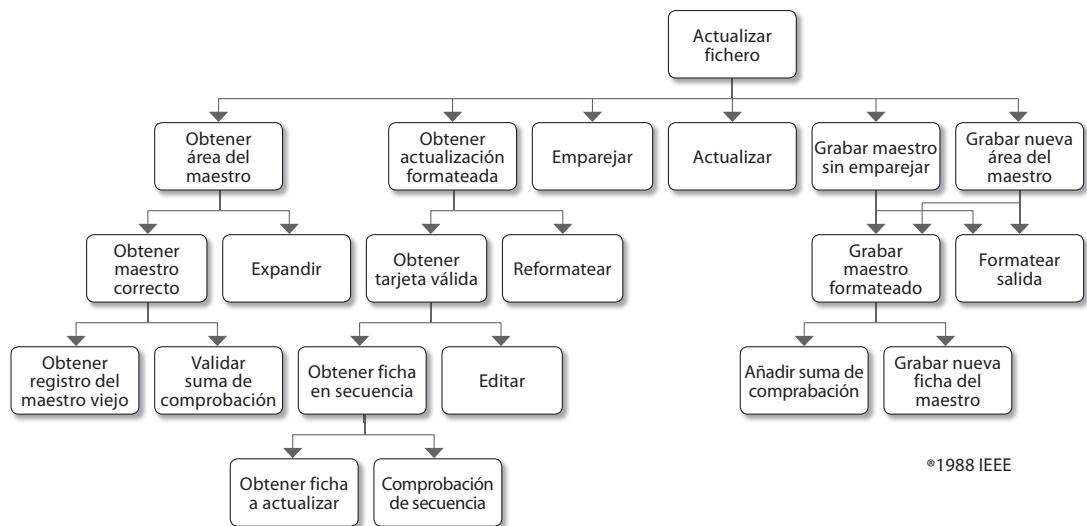
El papel (rol) de la descomposición

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



©1988 IEEE

Figura 1.2. Descomposición algorítmica.

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 perpendiculares.⁴ 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 general o filosófica. Los métodos son importantes por varias

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” [19].

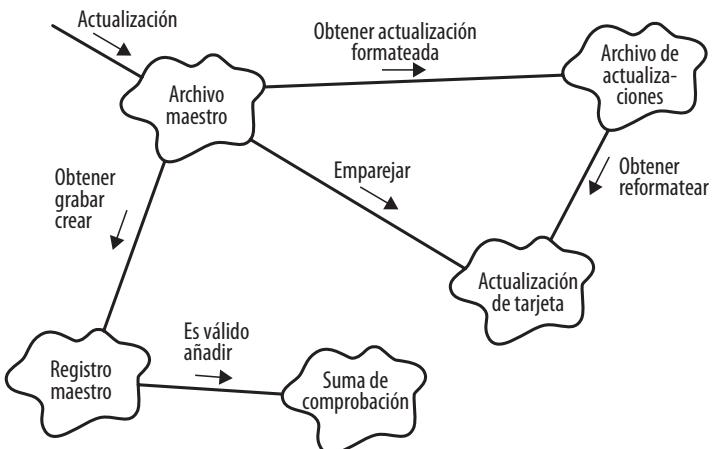


Figura 1.3. Descomposición orientada a objetos.

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 costos 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” [20]. 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 [21] y Yau y Tsai [22], y en una exhaustiva inspección por Teledyne Brown Engineering [23]. 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 [24]:

- 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 [25], Myers [26] y Page-Jones [27]. Los fundamentos de este método se derivan del trabajo de Wirth [28, 29] y Dahl, Dijkstra y Hoare [30]; se encuentra una importante variación del diseño estructurado en el método de Mills, Linger y Hevner [31]. 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 [32, 33] y los métodos de Warnier y Orr [34]. 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 el capítulo 2, 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.

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 solo 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 Simon llamaba “formas intermedias estables”) como fundamento sobre el que construir nuevos sistemas complejos. En presencia de una gran biblioteca de componentes de 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 solo 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 costo, o las herramientas disponibles para realizar el diseño” [39].

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” [40]. 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 [41]. 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” [42] 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.

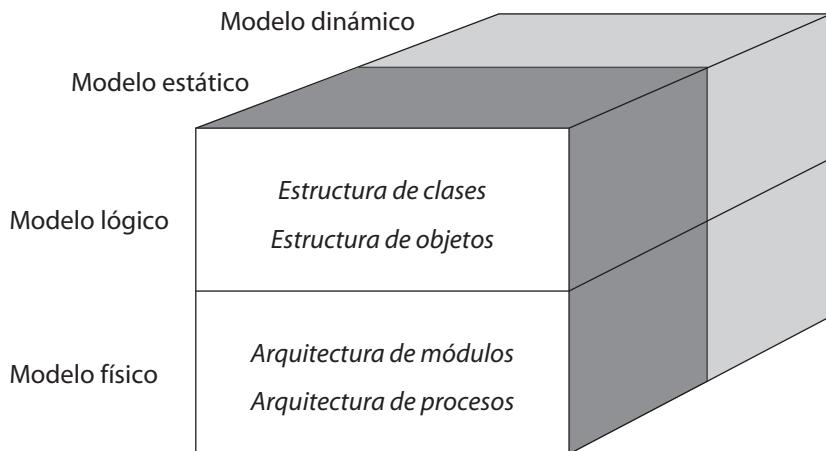


Figura 1.4. Los modelos del desarrollo orientado a objetos.

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 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.

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 e 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*. NewYork, 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. *IEEE 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] Langdon, G. 1982. *Computer Design*. San Jose, CA: Computeach Press, p. 6.
- [20] 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.

- [21] Peters. *Software Design*.
- [22] Yau, S. and Tsai, J. June 1986. A Survey of Software Design Techniques. *IEEE Transactions on Software Engineering* vol. SE-12(6).
- [23] Teledyne Brown Engineering. *Software Methodology Catalog*. Report MC87-COMM/ADP-0036. October 1987. Tinton Falls, NJ.
- [24] Sommerville, I. 1985. *Software Engineering*. Second Edition. Workingham, England: Addison-Wesley, p. 68.
- [25] Yourdon, E. and Constantine, L. 1979. *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall.
- [26] Myers, G. 1978. *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold.
- [27] Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press.
- [28] Wirth, N. January 1983. Program Development by Stepwise Refinement. *Communications of the ACM* vol. 26(1).
- [29] Wirth, N. 1986. *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall.
- [30] Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming*. London, England: Academic Press.
- [31] Mills, H., Linger, R., and Hevner, A. 1986. *Principles of Information System Design and Analysis*. Orlando, FL: Academic Press.
- [32] Jackson, M. 1975. *Principles of Program Design*. Orlando, FL: Academic Press.
- [33] Jackson, M. 1983. *System Development*. Englewood Cliffs, NJ: Prentice-Hall.
- [34] Orr, K. 1971. *Structured Systems Development*. New York, NY: Yourdon Press.
- [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] Mostow, J. Spring 1985. Toward Better Models of the Design Process. *AI Magazine* vol. 6(1), p. 44.
- [40] Stroustrup, B. 1991. *The C++ Programming Language*, Second Edition. Reading, MA: Addison-Wesley, p. 366.*
- [41] Eastman, N. 1984. Software Engineering and Technology. *Technical Directions* vol. 10(1): Bethesda, MD: IBM Federal Systems Division, p. 5.
- [42] Brooks. No Silver Bullet, p. 10.

* Existe versión en español por Addison-Wesley Iberoamericana.

2

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 de 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].

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 costo 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 frenesí 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. Solo 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, 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.

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

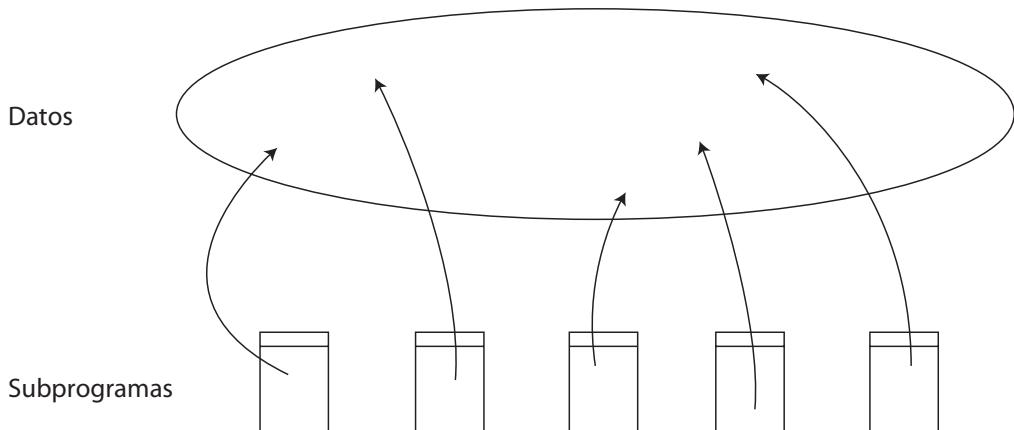


Figura 2.1. La topología de los lenguajes de programación de primera generación y principios de la segunda.

relativamente plana, consistente solo 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 [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.

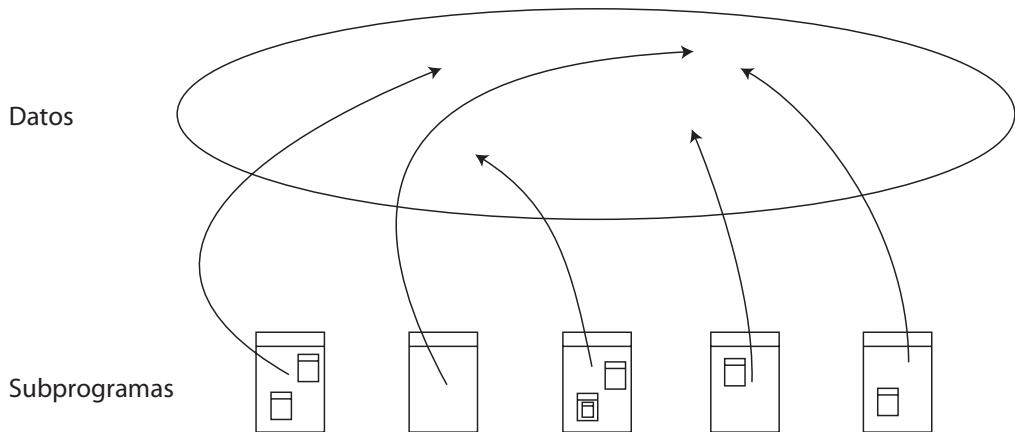


Figura 2.2. La topología de los lenguajes de programación de finales de la segunda generación y principios de la tercera.

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 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 cambiasean 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 las 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

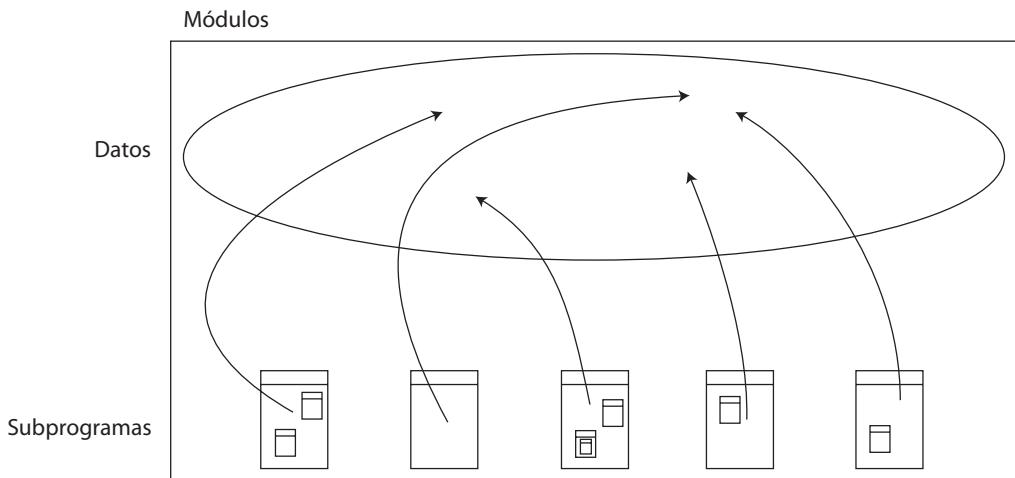


Figura 2.3. La topología de los lenguajes de programación de finales de la tercera generación.

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 solo 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 adecuada para la descripción de operaciones abstractas, pero no es particularmente buena para la descripción de objetos abstractos. Este 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 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 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

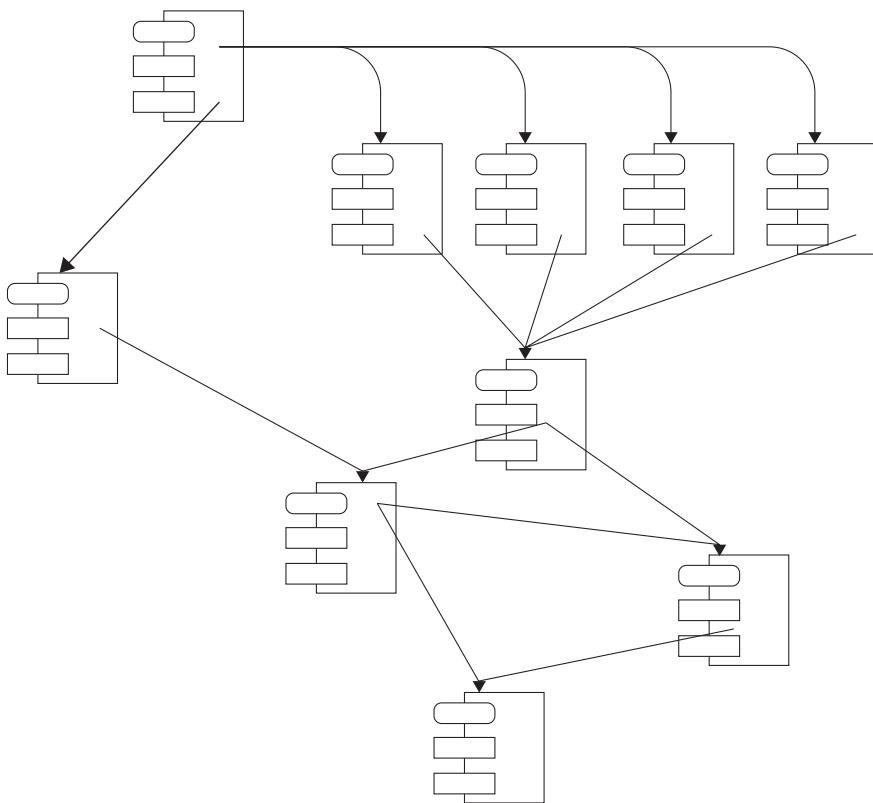


Figura 2.4. La topología de las aplicaciones de tamaño pequeño a moderado que utilizan lenguajes basados en objetos y orientados a objetos.

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

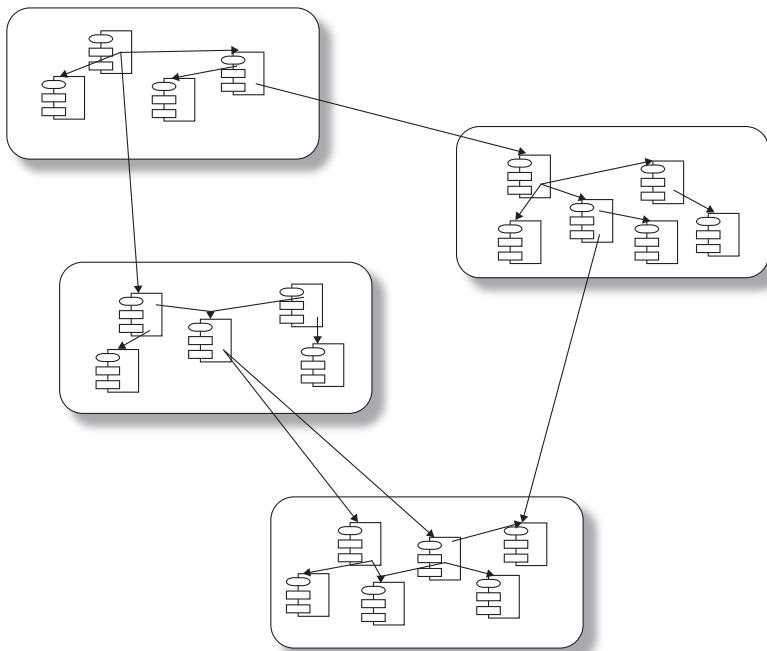


Figura 2.5. La topología de las aplicaciones a gran escala que utilizan lenguajes basados en objetos y orientados a objetos.

determinada para ver su implantación, se desvela un nuevo conjunto de abstracciones cooperando. Esta 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 solo de la programación orientada a objetos. Por contra, el modelo de objetos ha demostrado ser un concepto unificador en la informática, aplicable no solo 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 solo en los principios del diseño estructurado. 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 solo 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 solo 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. Ofrézcase una taladradora eléctrica a un carpintero que no sabe nada de la electricidad, y la utilizará como un martillo. Acabará 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 programador 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 solo puede cambiar de estado, actuar, ser manipulado o permanecer en relación con otros objetos de maneras apropiadas para ese objeto. Dicho

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.*).

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 solo se desplaza arriba y abajo por su hueco... Cualquier simulación de un ascensor debe incorporar estos invariantes, porque son intrínsecos al concepto de ascensor” [9].

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” [10]. 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 [11].

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 [12]. 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 [13]. 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 [14], SWARD [15], el Intel 432 [16], el COM de Caltech [17], el IBM System/38 [18], 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 [19]. 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 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 adelante, la investigación en lenguajes dio 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 a Pascal ha llevado a la aparición de Object Pascal, Eiffel y Ada. Además, hay muchos dialectos de Lisp que incorporan las características orientadas a objetos de Simula y Smalltalk, como Flavors, LOOPS y, más recientemente, Common Lisp Object System (CLOS).

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.

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

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]. 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 solo 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 orientado 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” [32]. 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 solo si satisface los siguientes requisitos:

- Soporta objetos que son abstracciones de datos con una 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]” [33].

Para un lenguaje, 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 [34], Yourdon [35] y Gane y Sarson [36], con extensiones para tiempo real de Ward y Mellor [37] y de Hatley y Pirbhai [38], 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 solo 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” [39]. 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” [40]. 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 intensivo. 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.

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” [41]. 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” [42]. Berzins, Gray y Naumann recomiendan que “un concepto merece el calificativo de abstracción solo si se puede describir, comprender y analizar independientemente del mecanismo que vaya a utilizarse eventualmente para realizarlo” [43]. 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* [44] que se consigue aplicando el principio de mínimo compromiso, mediante el cual la interfaz de un objeto muestra su comportamiento esencial, y nada más [45]. 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” [46]. 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 [47]: 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).

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 [48].

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

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

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* surgieron de tres culturas de programación diferentes (Ada, Smalltalk y C++). 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 estas 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 significativas 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, grava 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 jardinero 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 solo 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. En C++, podrían escribirse las siguientes declaraciones⁴ que plasman la abstracción descrita de un sensor de temperatura:

```
// 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:
    ...
};

}
```

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.*).

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 problema.⁵ **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.

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:

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.

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.*).

```
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 `SensorTemperaturaActivo` 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.

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 `SensorTemperaturaActivo` es función de sus aspectos internos, y no es de la incumbencia de los clientes externos. Estos 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 esta 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 de la 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 comportamiento.⁷ Las decisiones de diseño 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;
```

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).

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 iluminacion;
    pH acidez;
    Concentracion concentracion;
};
```

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 sobreescrbían 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 solo en las responsabilidades de la clase, no su representación.

Encapsulamiento

El significado del encapsulamiento.⁸ 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” [49]. 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” [50].

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.

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 solo de un esquema que denota la vista lógica de los mismos [51]. 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” [52]. En la práctica, esto significa que cada clase debe tener dos partes: una interfaz y una implementación. La *interfaz* de una clase captura solo 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. La interfaz de una clase es el único lugar en el que se declaran todas las suposiciones que un 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:

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

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 la interfaz contractual de una abstracción y su implantación.

Britton y Parnas llaman a esos elementos encapsulados los “secretos” de una abstracción [53].

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 solo 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:  
    ...  
};
```

Esta 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 de 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 la interfaz de esta clase; solo se precisaría modificar su implantación. A causa de las reglas sobre obsolescencia de C++, posiblemente habría que recomilar 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 abierta.⁹ 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 solo 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 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 refieran 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,

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.*).

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 pública la interfaz de una clase, solo se documenten los nombres de sus funciones genéricas, y el hecho de que un slot tenga funciones que acceden a él no se revele [54]. 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 solo para la propia clase y sus subclases. C++ también soporta la noción de “amigas” (*friends*): clases colaboradoras que, solo 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 solo 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” [55]. 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 de cara a la comprensión del programa” [56]. 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” [57]. La mayoría de los lenguajes que soportan el módulo como un concepto adicional distinguen también entre la 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 las 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 la interfaz y la implantación de un módulo. Las dependencias entre units pueden declararse solamente en la 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” [58].

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, es frecuente tener

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

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 solo constituye una pesadilla a la hora de documentar, sino que es terriblemente difícil para cualquier usuario encontrar las clases que necesita. Además, cuando 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 [59]. 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 estos 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 estos; [y] la facilidad de realizar un cambio en el diseño debería guardar una relación razonable con la probabilidad de que ese cambio fuese necesario” [60]. Hay un límite pragmático a esas líneas maestras. En la práctica, el coste de recompilar el cuerpo de un módulo es relativamente pequeño: solo hay que recompilar esa unidad y volver a enlazar la aplicación con el montador de enlaces. Sin embargo, el costo de recompilar la *interfaz* de un módulo es relativamente alto. Especialmente en lenguajes con comprobación estricta de tipos, hay que recompilar la interfaz del módulo, su cuerpo, y todos los demás módulos que dependen de esa 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 una simple interfaz de un módulo puede dar lugar a varios minutos, si no horas, de recomplilación. Obviamente, un jefe de desarrollo no puede permitirse habitualmente el dejar que suceda con frecuencia un “big bang” de recomplilaciones masivas. Por esta razón, la interfaz de un módulo debería ser tan estrecha como fuese posible, siempre y cuando se satisfagan las necesidades de todos los módulos que la 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 la 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 este” [61]. 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 sinérgicos. 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 estos de forma que se minimicen las interfaces entre distintas partes de la organización de desarrollo. Por norma, los diseñadores experimentados se responsabilizan de las 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 las interfaces entre los módulos, tal y como se haya acordado entre las distintas compañías. El cambio de estas interfaces suele conllevar mucho llanto y rechinazos 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 solo 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 organizados 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 una interfaz gráfica de usuario (IGU) innovadora. 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 (`tiposcul.h`, `excep.h` y `acciones.h`), en cuya interfaz hay que confiar.

Las implantaciones de estas clases de planes de cultivo aparecen en la implantación de este módulo, en un fichero llamado (por convenio) `plancult.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 la interfaz de `plancult.h`, así como de ficheros que encapsulen ciertas 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 la 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” [62].

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 abstracciones 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, esta 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 `repProduccion`) 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`.

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 puede 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” [63].

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 la cual se ocultan los métodos y el estado; la herencia requiere abrir esta interfaz en cierto grado y puede permitir el acceso a los métodos y al estado sin abstracción” [64]. 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” [65]. 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, C++ ofrece quizás la mayor flexibilidad. Específicamente, la interfaz de una clase puede tener tres partes: partes *private*, que declaran miembros accesibles solo a la propia clase, partes *protected*, que declaran miembros accesibles solo 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 solo para la propia clase y sus subclases. Por el contrario, todos los miembros declarados en la parte `private` son accesibles solo 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 {
public:
    FlorAditiva(Dia momentoFlorecimiento, Dia momentoSiembra);
    virtual ~FlorAditiva();

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

protected:
    ...
};
```

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.

```
};

class FrutaHortalizaAditiva {
public:

    FrutaHortalizaAditiva(Dia momentoRecolección);
    virtual ~FrutaHortalizaAditiva();

    Dia momentoRecolección() 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 introduce 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 (requeríendose 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 estas, se habla a menudo de *niveles de abstracción*, un concepto descrito por primera vez por Dijkstra [66]. 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 *Flo*r está a nivel más alto de abstracción que una clase *Planta*. En términos 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” [67]. Para nuestros propósitos, se utilizarán los términos *tipo* y *clase* de manera intercambiable.¹² 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 los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse solo 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 [68].

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 idea de congruencia es central a la noción de tipos. Por ejemplo, considérense las unidades de medida en física [70]. 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 aun 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 esta 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 la interfaz de una clase base requieren la recompilación de todas las subclases. Además, en ausencia de clases parametrizadas 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 de C aplicado a 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 solo 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 lenguaje,¹³ pero puede lograrse un efecto similar en la práctica, definiendo una operación en la clase base que retoma una cadena o tipo enumerado que identifica la clase concreta del objeto. En general, sin embargo, la identificación de tipos en ejecución debería utilizarse solo cuando hay una razón de peso, porque puede representar un debilitamiento del encapsulamiento. Como se verá en la 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 “rebanada”). 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:

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

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

Solo 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 tipos 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” [71].

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” [72]. 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 que 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 miembro:¹⁴

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

Llamar a la operación `equilibrarNiveles` con instancias de `TanqueAlmacen` 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 solo 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 [73].

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.

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

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 más adelante, 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 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 solo 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 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, Windows/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” [74]. 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” [75]. 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” [76].

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 [77]. El objeto es un concepto que unifica 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 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 la 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.

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

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. Este es el punto en el que las ideas de abstracción, encapsulamiento y concurrencia interactúan. En presencia de la concurrencia, no es 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.
- Datos que existen entre ejecuciones de un programa.
- Datos que existen entre varias versiones de un programa.
- Datos que sobreviven al programa” [78].

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 [79].

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

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 una interfaz orientada a objetos, a través de la 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.

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 [80]. 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.

La persistencia abarca algo más que la mera duración de los datos. En las bases de datos orientadas a objetos, no solo 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.

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 estructurado, 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++” [81]. 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 solo de software, sino de diseños enteros, conduciendo a la creación de marcos de desarrollo de aplicaciones reutilizables [82]. Se ha encontrado que los sistemas orientados a objetos son frecuentemente más pequeños que sus implantaciones equivalentes no orientadas a objetos. Esto no solo significa escribir y mantener menos código, sino que la mayor reutilización del software también se traduce en beneficios de costo 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 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.

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.

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” [83].

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.

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?

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 la 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 Hilfinger [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], Wirsfs-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 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] Jones, A. 1979. The Object Model: A Conceptual Tool for Structuring Software. *Operating Systems*. New York, NY: Springer-Verlag, p. 8.
- [10] Yonezawa, A. and Tokoro, M. 1987. Object-Oriented Concurrent Programming: An Introduction, in *Object-Oriented Concurrent Programming*. Cambridge, MA: The MIT Press, p. 2.
- [11] Levy, H. 1984. *Capability-Based Computer Systems*. Bedford, MA: Digital Press, p. 13.
- [12] Ramamoorthy, C. and Sheu, P. Fall 1988. Object-Oriented Systems. *IEEE Expert* vol. 3(3), p. 14.
- [13] Myers, G. 1982. *Advances in Computer Architecture*. Second Edition. New York, NY: John Wiley and Sons, p. 58.
- [14] Levy. *Capability-Based Computer*.
- [15] Kavi, K. and Chen, D. 1987. Architectural Support for Object-Oriented Languages. *Proceedings of the Thirty-second IEEE Computer Society International Conference*. IEEE.
- [16] *iAPX 432 Object Primer*. 1981. Santa Clara, CA: Intel Corporation.
- [17] Dally, W. J. and Kajiyama, J. T. March 1985. An Object-Oriented Architecture. *SIGARCH Newsletter* vol. 13(3).
- [18] Dahlby, S., Henry, G., Reynolds, D., and Taylor, P. 1982. The IBM System/38: A High Level Machine, in *Computer Structures: Principles and Examples*. New York, NY: McGraw-Hill.
- [19] Dijkstra, E. May 1968. The Structure of the "THE" Multiprogramming System. *Communications of the ACM* vol. 11(5).
- [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 Development 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] Stroustrup, B. May 1988. What Is Object-Oriented programming? *IEEE Software* vol. 5(3), p. 10.
- [33] Cardelli, L. and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. December 1985. *ACM Computing Surveys* vol. 17(4), p. 481.
- [34] DeMarco, T. 1979. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice-Hall.
- [35] Yourdon, E. 1989. Modern Structured Analysis. Englewood Cliffs, NJ: Prentice-Hall.
- [36] Gane, C. and Sarson, T. 1979. *Structured Systems Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- [37] Ward. P. and Mellor, S. 1985. *Structured Development for Real-Time Systems*. Englewood Cliffs, NJ: Yourdon Press.
- [38] Hatley, D. and Pirbhai, I. 1988. *Strategies for Real-Time System Specification*. New York, NY: Dorset House.
- [39] Jenkins, M. and Glasgow, J. January 1986. Programming Styles in Nial. *IEEE Software* vol. 3(1), p. 48.
- [40] Bobrow, D. and Stefik, M. February 1986. Perspectives on Artificial Intelligence Programming. *Science* vol. 231, p. 951.
- [41] Dahl, O., Dijkstra, E., and Hoare, C. A. R. 1972. *Structured Programming*. London, England: Academic Press, p. 83.
- [42] Shaw. M. October 1984. Abstraction Techniques in Modern Programming Languages. *IEEE Software* vol. 1(4), p. 10.
- [43] Berzins, V., Gray, M., and Naumann, D. May 1986. Abstraction-Based Software Development. *Communications of the ACM* vol. 29(5), p. 403.
- [44] Abelson, H. and Sussman, G. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press, p. 126.
- [45] Ibid., p. 132.
- [46] 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.
- [47] Meyer, B. 1988. *Object-Oriented Software Construction*. New York, NY: Prentice Hall.
- [48] Wirfs-Brock, R. and Wilkerson, B. October 1989. Object-Oriented Design: A Responsibility-Driven Approach. *SIGPLAN Notices* vol. 24(10).
- [49] 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.
- [50] Gannon, J., Hamlet, R., and Mills, H. July 1987. Theory of Modules. *IEEE Transactions on Software Engineering* vol. SE-13(7), p. 820.

- [51] Date, C. 1986. *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley, p. 180.
- [52] Liskov, B. May 1988. Data Abstraction and Hierarchy. *SIGPLAN Notices* vol. 23(5), p. 19.
- [53] Britton, K. and Parnas, D. December 8, 1981. *A-7E Software Module Guide*. Washington, D.C.: Naval Research Laboratory, Report 4702, p. 24.
- [54] Gabriel, R. 1990. Private communication.
- [55] Stroustrup, B. 1988. Private communication.
- [56] Myers, G. 1978. *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold, p. 21.
- [57] Liskov, B. 1980. A Design Methodology for Reliable Software Systems, in *Tutorial on Software Design Techniques*. Third Edition. New York, NY: IEEE Computer society, p. 66.
- [58] Zelkowitz, M. June 1978. Perspectives on Software Engineering. *ACM Computing Surveys* vol. 10(2), p. 20.
- [59] 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.
- [60] Britton and Parnas. *A-7E Software*, p. 2.
- [61] 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.
- [62] Meyer, *Object-Oriented Software Construction*, p. 47.
- [63] Cox, B. 1986. *Object-Oriented Software Construction*, p. 47.
- [64] Danforth, S. and Tomlinson, C. March 1988. Type Theories and Object-Oriented Programming. *ACM Computing Surveys* vol. 20(1), p. 34.
- [65] Liskov. 1988, p. 23.
- [66] As quoted in Liskov. 1980, p. 67.
- [67] 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.
- [68] Wegner, P. October 1987. Dimensions of Object-Based Language Design. *SIGPLAN Notices* vol. 22(12), p. 171.
- [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] Stroustrup, B. 1992. Private communication.
- [71] Tesler, L. August 1981. The Smalltalk Environment. *Byte* vol. 6(8), p. 142.
- [72] Borning and Ingalls. Type Declaration, p. 133.
- [73] Thomas, D. March 1989. What's in an Object? *Byte* vol. 14(3), p. 232.
- [74] Lim, J. and Johnson, R. April 1989. The Heart of Object-Oriented Concurrent Programming. *SIGPLAN Notices* vol. 24(4), p. 165.
- [75] Ibid., p. 165.
- [76] 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.
- [77] Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming. April 1989. *SIGPLAN Notices* vol. 24(4), p. 1.

- [78] 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.
- [79] Khoshafian, S. and Copeland, G. November 1986. Object Identity. *SIGPLAN Notices* vol. 21(11), p. 409.
- [80] *Vbase Technical Overview*. September 1987. Billerica, MA: Ontologic, p. 4.
- [81] Stroustrup, B. November 1987. Possible Directions for C++. *Proceedings of the USENIX C++ Workshop*. Santa Fe, NM, p. 14.
- [82] Meyer. *Object-Oriented Software Construction*, p. 30-31.
- [83] Robson, D. August 1981. Object-Oriented Software Systems. *Byte* vol. 6(8), p. 74.

3

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 solo definiciones informales de estos dos elementos, este capítulo se vuelve en 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 definición.¹ 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

¹ Esto es cierto solo 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.

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.

Estado

Semántica. Considérese una máquina expendedora de refrescos. El comportamiento usual de esos 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 “Solo el dinero exacto” e introduce dinero extra. La mayoría de las máquinas son “hostiles al usuario”; se tragará 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 aun 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. Esta 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 estos 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 tenga 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 específica.² Para declarar objetos de esta clase, se escribe:

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.

```
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 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 razones.³ 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 recomilar 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. Solo clientes especiales (es decir, subclases de esta clase) tienen permiso para modificar los valores de estas propiedades. Más aun, solo estos clientes especiales pueden modificar o recuperar el salario de un empleado, mientras que los clientes externos no pueden.

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 solo 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 (punteros) colgadas. En sistemas de producción, es preferible usar una clase parametrizada de cadenas de longitud variable. Además, las clases son más simples que la `struct` de C envueltas en sintaxis de C++; como se explica en el capítulo 4, la clasificación requiere una atención deliberada hacia estructura y comportamiento comunes.

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 refinarla 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 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 de 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` solo 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 objeto.⁴ 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`). El estilo que se propone es definir una clase separada que actúa como el agente responsable de iterar a lo largo de las colas.

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].

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

En lenguajes orientados a objetos puros como Smalltalk, solo pueden declararse las operaciones como métodos, ya que el lenguaje no permite declarar procedimientos o funciones separados de ninguna clase. 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 clases. Los subprogramas libres están típicamente agrupados según las clases sobre las que se los ha construido; por tanto, se llama a tales colecciones de subprogramas libres *utilidades de clase*. Por ejemplo, dada la declaración 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 clase `Cola`.

Forma parte del estilo habitual de C++ (y de Smalltalk) recoger todos los subprogramas libres relacionados lógicamente y declararlos como parte de una clase 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 clases diferentes, y no se deriva ningún beneficio especial al declarar esa operación en una clase 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.

Con frecuencia, se inicia 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.

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, solo 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 comportamiento 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`. Solo `elemento2` y `elemento3` apuntan realmente a objetos `ElementoPantalla` precisos (porque solo 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; solo puede hacerse referencia a esos objetos 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->mover(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.

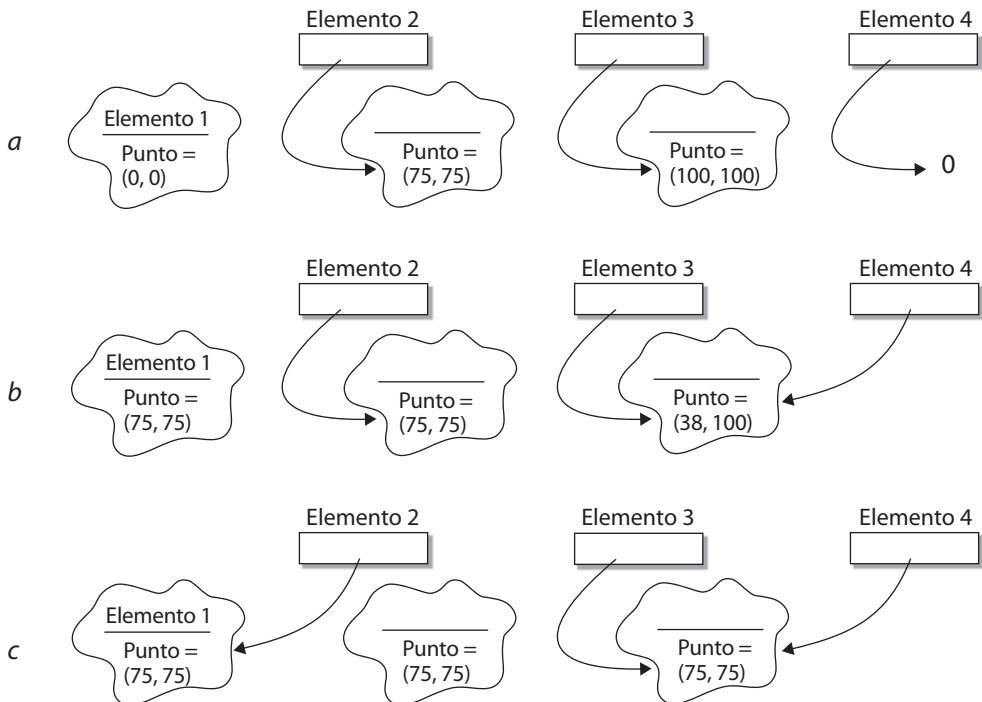


Figura 3.1. Identidad de los objetos.

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 desastrosas.⁵

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 original.⁶ 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 solo 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

⁵ Considérense 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++.

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 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 solo 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, solo 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 ElementoPantallas&) 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 sobrescribir 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]. Solo 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 explícitamente como parte de la destrucción.

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 estos. 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 nombra 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`. Solo 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` solo 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.⁸ 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 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; solo 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.

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.

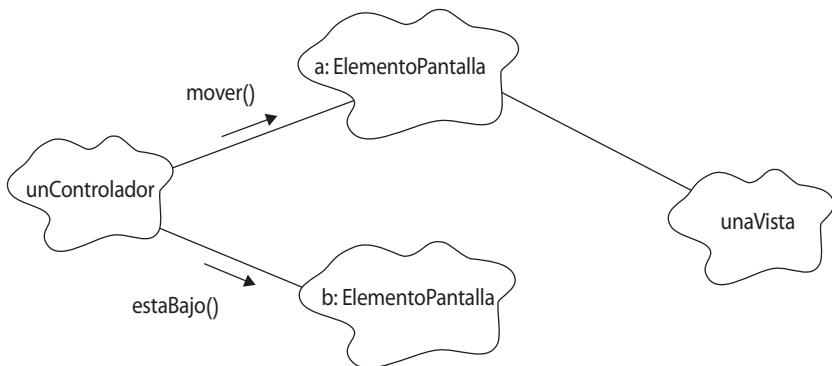


Figura 3.2. Enlaces.

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, esta es solo 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. 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 solo 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.

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 solo si este conocimiento es parte del estado de `h`.

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.

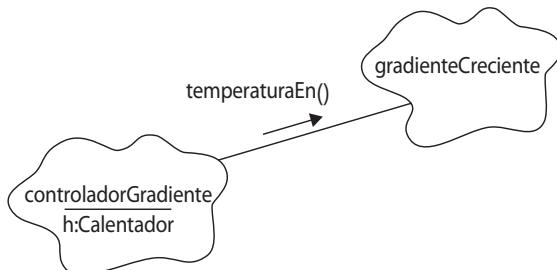


Figura 3.3. Agregación.

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 solo una abstracción, la “esencia” de un objeto. 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 clase.¹⁰ 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]. Aquí, en cambio, se llama a tales grupos una *categoría de clases*.

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

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 una interfaz grande es mala práctica, porque la mayoría de los clientes querrán referenciar solo un pequeño subconjunto de los servicios proporcionados. Es más, el cambio de una parte de una 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; solo los aplaza.

relacionados. Así, una clase sirve como una especie de contrato que vincula a una abstracción y todos sus clientes. Capturando estas decisiones en la 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. La *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. Esta 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, constantes, 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 la interfaz de la misma.

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

- **Public (pública)** Una declaración accesible a todos los clientes.
- **Protected (protegida)** Una declaración accesible solo a la propia clase, sus subclases, y sus clases amigas (friends).
- **Private (privada)** Una declaración accesible solo 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 de la 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 distintas.¹¹ 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 de la 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 de la interfaz de una clase (aunque sea una parte no pública) y no de su implementación. Por razones

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.

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 elemento1;
```

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 solo 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. 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. Analógicamente, 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).
- Metaclase.

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].

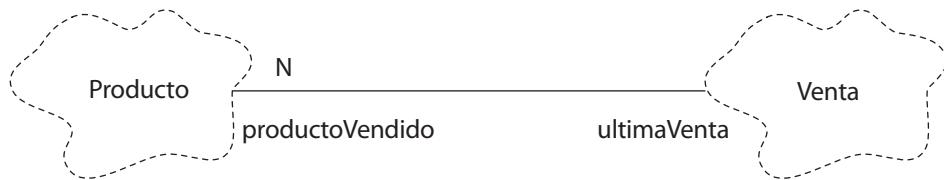


Figura 3.4. Asociación.

De estos seis tipos diferentes de relaciones entre clases, las asociaciones son el más general, pero también el de mayor debilidad semántica. 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 menudo 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 instancia que, al igual que la herencia, soportan un tipo de generalización, aunque de forma completamente diferente. Las relaciones de metaclasa son bastante distintas y solo las soportan explícitamente lenguajes como Smalltalk y CLOS. Básicamente, una metaclasa 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 ve 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 solo 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 (solo 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 solo es necesario identificar esas dependencias. Mediante la creación de asociaciones, se llega a plasmar 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. 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 aun, 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 recomilar 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 sistema. 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 generalizada. 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 distintas para estos dos tipos de clientes [27]. En particular, se desea exponer a los clientes instancia solo los comportamientos visibles exteriormente, pero se necesita exponer las funciones de asistencia y las representaciones solamente a los clientes subclase. Esta 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` incluyen 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.

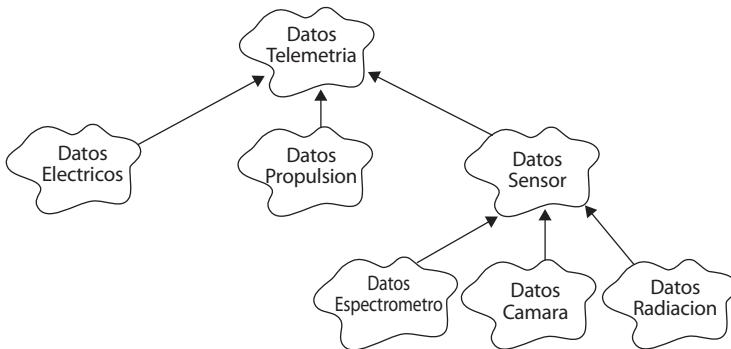


Figura 3.5. Herencia simple.

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(telemetry, 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 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 solo 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 programación 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

¹³ Esta 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.

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. Esta 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.

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 superclase.¹⁴

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`.

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 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 `Circulo`, `Triangulo` y `Rectangulo`. `Rectangulo` tiene también una subclase, llamada `RectanguloSolido`. 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 solo las subclases saben cómo dibujarse y moverse a sí mismas.

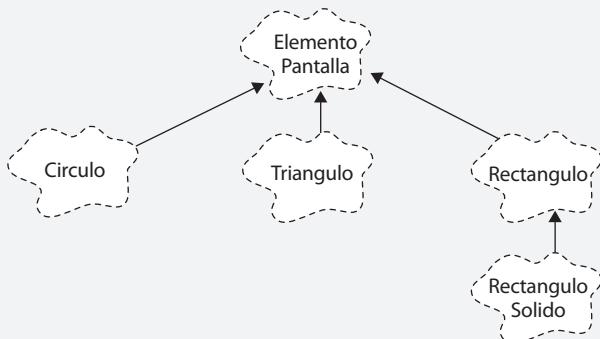


Figura 3.6. Diagrama de clases de ElementoPantalla.

La clase `Circulo` 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 `Rectangulo` 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 `RectanguloSolido` hereda 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 contener 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

posición 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 *viable* 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 solo tres o cuatro referencias a memoria más que una llamada a función normal; la herencia múltiple añade solo 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; // electricos 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 electricos
```

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 solo 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 conversiones 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,

* También denominada moldeado (*N. del T.*).

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 tipo.¹⁵ 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 solo 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 la interfaz externa de la clase sigue siendo la misma” [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 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 la misma interfaz (y, por tanto, la misma estructura y comportamiento). Alternativamente, en la declaración de una clase, se puede afirmar que una superclase es `private`,

¹⁵ Algunas extensiones propuestas a C++ para identificación de tipos en tiempo de ejecución ayudarán a mitigar este problema.

lo que significa que la estructura y comportamiento de la superclase son compartidos, pero la subclase no es un subtipo de la superclase.¹⁶ 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 la misma 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 solo un subtipo restringido, mientras que un tipo 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

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.

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衍生 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, se debe contemplar la herencia múltiple.¹⁷ 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`.

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...
```

¹⁷ De hecho, esta 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 solo donde se deseé.

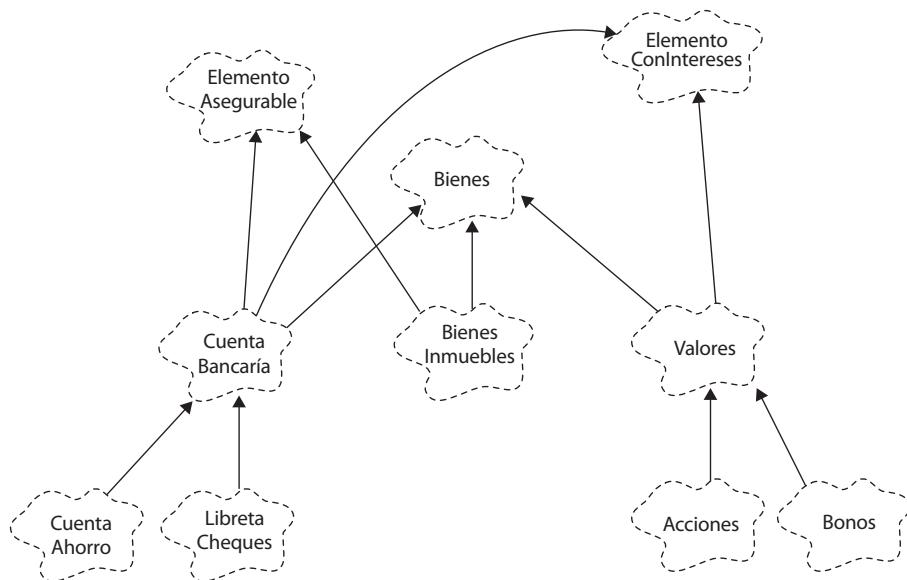


Figura 3.7. Herencia múltiple.

A continuación se tienen varias clases intermedias, cada una de ellas 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 especialmente si es herencia múltiple, es una tarea difícil. Como se explica en el capítulo 4, este 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? Esta 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. Este 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. Este 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 FondoInversion : 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. Este es el enfoque de Smalltalk y Eiffel (donde Eiffel permite nuevamente renombrar para eliminar la ambigüedad de las referencias duplicadas). Segundo, se puede permitir la duplicación de superclases, pero requerir el uso de nombres plenamente calificados para referirse a los miembros de una copia específica. Este es uno de los

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.

enfoques adoptados por C++. Tercero, se pueden tratar las referencias múltiples a la misma clase como denotando a la misma clase. Este 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 Handler: “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 clase.¹⁹ 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 solo métodos `:before` y `:after` para aumentar el comportamiento de los métodos primarios existentes.

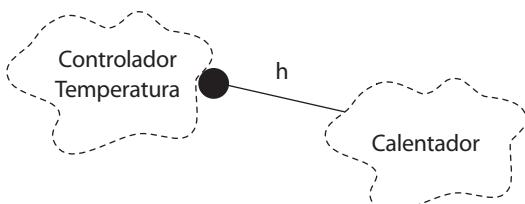


Figura 3.8. Agregación.

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. Este 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 solo 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 solo 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 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 siguiente:²⁰

```
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

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.

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 solo 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 esta: si y solo 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 momento se representó mediante una relación “de uso” entre sus clases correspondientes, `ControladorTemperatura` y `GradienteTemperatura`:

```
class ControladorTemperatura {
public:
```

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.

```

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`.

Clients 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 3.9.²²

En realidad, una clase puede utilizar a otra de diversas formas. En el ejemplo, el `ControladorTemperatura` usa a la `GradienteTemperatura` en la signatura de su interfaz. El `ControladorTemperatura` podría usar también a otra clase como `Pronosticador` en su implementación de la función miembro `planificar`. Esta no es una afirmación de una relación todo/parte: la instancia de la clase `Pronosticador` solamente es utilizada por la instancia de `ControladorTemperatura`, 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 solo a la interfaz public del proveedor. A veces, por razones tácticas, hay que romper el encapsulamiento de estas abstracciones, y este 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 soportan genericidad.

²² Como se afirmó anteriormente, una relación “de uso” cíclica es equivalente a una asociación, aunque la afirmación inversa no es necesariamente cierta.

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 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. 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.

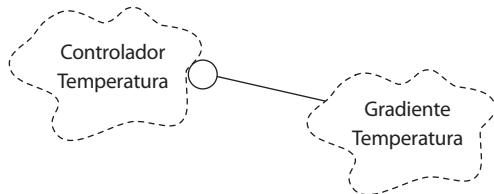


Figura 3.9. La relación “de uso”.

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. Este 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 solo 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. 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 llenados) 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.

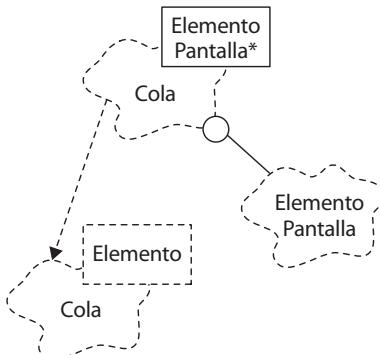


Figura 3.10. *Instanciación.*

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 instancia. 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 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

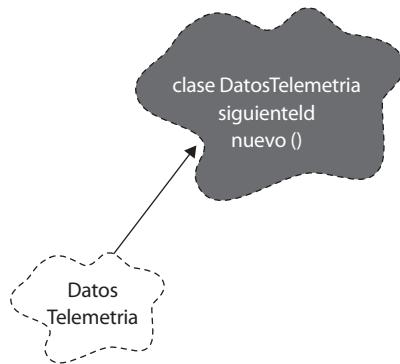


Figura 3.11. Metaclasses.

de una clase? La respuesta es: simplemente, una metaclasa. Dicho de otra forma, una *metaclasa* es una clase cuyas instancias son, ellas mismas, clases. Los lenguajes como Smalltalk y CLOS soportan el concepto de metaclasa directamente; C++ no. Realmente, la idea de metaclasa 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 metaclasses haciendo notar que “en un sistema bajo desarrollo, una clase proporciona al programador una 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 metaclasa 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 metaclasa [50]. Por convenio, una metaclasa 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 metaclasa de **DatosTelemetria**. 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 metaclasses 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 metaclasas. 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 metaclasa 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. Análogamente, 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 propia 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 lo 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. 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 contrapunto 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 la interfaz de la clase o módulo captura todas las características significativas de la abstracción. Mientras la suficiencia implica una interfaz mínima, una interfaz completa es aquella que cubre todos los aspectos de la abstracción. Una clase o módulo completo es aquel cuya 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 solo 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 solo accediendo a la representación interna. Una operación que podría implantarse sobre operaciones primitivas existentes, pero a un costo de recursos computacionales significativamente mayor, es también candidata para su inclusión como operación primitiva.

Selección de operaciones

Semántica funcional. El desarrollo de la 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 esta 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 una interfaz más simple pero métodos más grandes y complicados; distribuir un comportamiento entre varios métodos lleva a una interfaz más complicada, 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:

- | | |
|---|--|
| ■ Reutilización (<i>reusabilidad</i>) | ¿Sería este comportamiento más útil en más de un contexto? |
| ■ Complejidad | ¿Qué grado de dificultad plantea el implementar este comportamiento? |
| ■ Aplicabilidad | ¿Qué relevancia tiene este comportamiento para el tipo en el que podría ubicarse? |
| ■ Conocimiento de la implementación | ¿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 concurrencia,²³ 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 solo 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.

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++.

■ De intervalo	Igual que el síncrono, excepto en que el emisor esperará a que el receptor esté listo solo 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 solo cuando se haya decidido ya sobre la semántica funcional de dicha operación.

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 solo donde sus ámbitos se superpongan y solo donde estén garantizados los derechos de acceso (por ejemplo, las partes privadas de una clase son accesibles solo 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 solo 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 communalidad, 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

Solo 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 cuyas interfaces son prácticamente idénticas, 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 [61]? 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 competitores 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 costo 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” [62].

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 satisfaga 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 Lorensen, 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] Stroustrup, 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., Szyplewski, 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] Stroustrup, B. 1988. Private communications.
- [35] Stroustrup, 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] Handler, 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.

4

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 de 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 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 primitivas 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 software 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 las interfaces gráficas 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. Solo 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 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]. Solo 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, solo 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 alta 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 apor taron 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 nues tros 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 significativa. 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 de pendiendo 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.

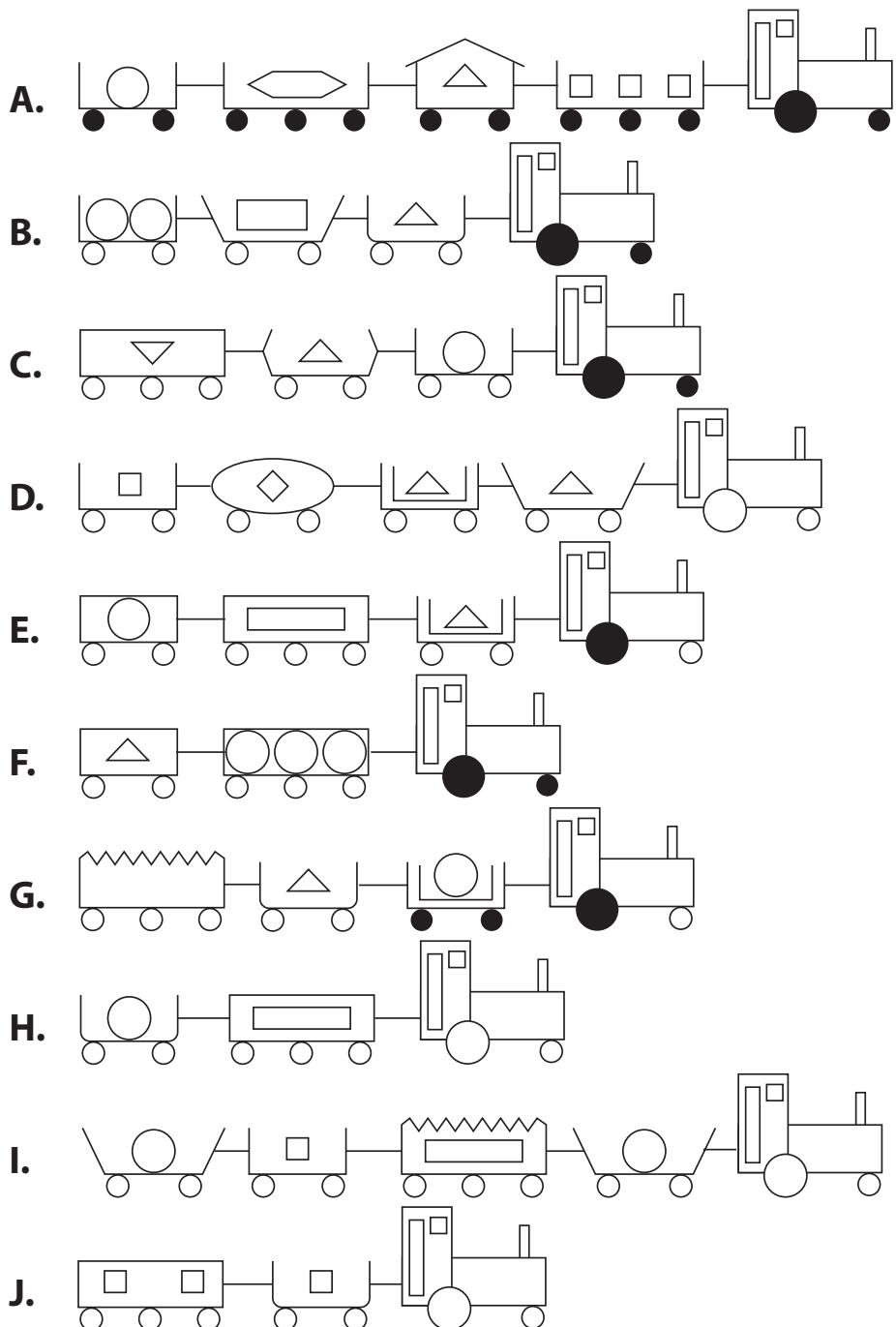


Figura 4.1. Un problema de clasificación.

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. Esta 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 solo 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 vuelan, 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]. Estos 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”. Este 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.

¹ En el original en inglés *Conceptual clustering* (N. del T.).

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... 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]. Esta 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 solo 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 compartan 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 compartan 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, estos 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 requiere.²

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]:

- Cosas tangibles Coches, datos de telemetría, sensores de presión.
- Papeles (roles) Madre, profesor, político.
- Eventos Aterrizaje, interrupción, petición.
- Interacciones Préstamo, reunión, intersección.

Desde la perspectiva del modelado de bases de datos, Ross ofrece una lista similar [33]:

- Personas Humanos que llevan a cabo alguna función.
- Lugares Áreas reservadas para personas o cosas.
- Cosas Objetos físicos, o grupos de objetos, que son tangibles.

² La notación y proceso descritos en este libro son perfectamente aplicables a las fases tradicionales del desarrollo de análisis y diseño.

■ Organizaciones	Colecciones formalmente organizadas de personas, recursos, instalaciones y posibilidades que tienen una misión definida, cuya existencia es, en gran medida, independiente de los individuos.
■ Conceptos	Principios o ideas no tangibles <i>per se</i> , utilizados para organizar o llevar cuenta de actividades de negocios y/o comunicaciones.
■ Eventos	Cosas que suceden, habitualmente a alguna otra cosa, en una fecha y hora concretas, o como pasos dentro de una secuencia ordenada.

Coad y Yourdon sugieren otro conjunto más de fuentes de objetos potenciales [34]:

■ Estructuras	Relaciones “de clases” y “de partes”.
■ Otros sistemas	Sistemas externos con los que la aplicación interactúa.
■ Dispositivos	Dispositivos con los que la aplicación interactúa.
■ Eventos recordados	Sucesos históricos que hay que registrar.
■ Papeles desempeñados	Los diferentes papeles que juegan los usuarios en su interacción con la aplicación.
■ Posiciones	Ubicaciones físicas, oficinas y lugares importantes para la aplicación.
■ Unidades de organización	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 objetos.³ 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” [35]. 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.

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 [36].

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. Estos 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 relación 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 aviación 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

* *Business function* en el original en inglés (*N. del T.*).

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 objetos 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 en 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. 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 escenarios.⁴ 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 3x5,⁵ 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 español.⁶ 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

⁴ CRC viene de Clases/Responsabilidades/Colaboradores.

⁵ Si el presupuesto de desarrollo puede afrontarlo, mejor comprar fichas de 5x7. Las tarjetas con líneas son elegantes, un puñado de tarjetas de colores muestra que uno es un desarrollador muy cool.

⁶ Descripción informal en inglés, en el original (*N. del T.*).

representan operaciones candidatas sobre ellos. Esta técnica se presta a la automatización, y 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 solo 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 caer 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 predisposto hacia la descomposición algorítmica. Esta 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 sean,⁷ 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 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

⁷ Las razones políticas o históricas no son respetables en absoluto.

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, solo 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. 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 determinada 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 partición. 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 granularidad* [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 español,⁸ 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.

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`.

⁸ Prosa en inglés en el original (*N. del T.*).

- 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 colaborasen 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 por completo 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 costo).

El mecanismo que elige un desarrollador entre un conjunto de alternativas es frecuentemente el resultado de otros de factores, como costo, fiabilidad, facilidad de fabricación y seguridad.

Del mismo modo que es una falta de educación el que un cliente viole la 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 solo 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, la interfaz de una clase individual es más bien una decisión de diseño táctica. Estas decisiones estratégicas 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 lenguaje.⁹ 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.

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. 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. Analogamente, 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 gráficas 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

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 aun, como a un intruso, que no merece respeto alguno.

* *Idiom* en el original en inglés (*N. del T.*).

** *Framework* en el original en inglés (*N. del T.*).

(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 gráfica de usuario orientada 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 la 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 explorado 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.

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 aun 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.
- [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] Wirfs-Brock, R., Wilkerson, B., and Wiener, L. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey: Yourdon Press.
- [36] Shlaer, S. and Mellor, S. 1992. *Object Lifecycles: Modeling the World in States*. Englewood 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*. Englewood 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: Adison-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).

SEGUNDA SECCIÓN

La implementación

Por Darío Cardacci

Consideraciones tecnológicas de la implementación

La implementación práctica de los conceptos teóricos tratados en la primera parte del libro se han adaptado para que el lector que posee recientes conocimientos sobre orientación a objetos pueda ir avanzando en su lectura y observando gradualmente cada concepto en la práctica. En algunos casos, se es reiterativo intencionalmente por considerar que el concepto es de suma importancia y amerita que sea apropiado de manera significativa por el lector.

Debido a que el texto fue pensado para la formación de personas del ámbito universitario, se ha seleccionado para los ejemplos de código un lenguaje de programación sencillo de comprender, Visual Basic.Net. El entorno de desarrollo utilizado es Visual Studio 2012.

De la teoría a la práctica

Como resulta natural en muchos aspectos del conocimiento, y orientación a objetos no es la excepción, existen pautas teóricas abstractas que no siempre se implementan fácilmente en la práctica o bien la tecnología seleccionada no facilita su uso.

En nuestro caso, estas pautas conforman solo un pequeño grupo de conceptos. Se recomienda que si el texto está siendo utilizado en un ámbito universitario, el profesor a cargo los elabore y enriquezca con el objetivo de complementar los códigos que están expuestos en esta sección práctica.

Como el lector notará, se ha intentado reducir la escritura al mínimo posible y en su lugar exemplificar con código, pues la idea final es que se visualicen los elementos de la teoría de objetos como elementos programables que conformen el código fuente de nuestra aplicación.

La notación utilizada

Para campos relacionados con características de las clases, tendrán antepuesta la letra “V” mayúscula seguida del nombre elegido en minúscula. Si el nombre elegido posee más de una palabra, desde la segunda en adelante se coloca la primera letra en mayúscula. Por ejemplo Vnombre, VdiaNacimiento.

Para parámetros de funciones y procedimientos, se utilizará como primer carácter la letra “p” minúscula seguida del nombre elegido. Si el nombre elegido posee más de una palabra, cada una de ellas se coloca con la primera letra en mayúscula. Por ejemplo pNombre, pDiaNacimiento.

Cuando en el nombre de una clase, objeto o algunos de sus miembros corresponda colocar una tilde según las reglas del idioma utilizado, se evitará su uso tanto si el nombre está siendo utilizado dentro de los párrafos explicativos del libro como si está en un ejemplo de código fuente.

El lenguaje de programación

La tecnología utilizada en los desarrollos prácticos es .NET, la suite de desarrollo es Visual Studio 2012 y el lenguaje de programación Visual Basic.NET. Al momento de seleccionar este conjunto de elementos para trabajar, no se realizaron juicios de valor sobre aspectos que hacen a su performance o difusión, simplemente se utilizaron estos debido a las características que definen el curso para el cual fue diseñado. No obstante, con un mínimo esfuerzo usted podrá transcribir este código al lenguaje que le resulte más familiar.

5

Clases

Como hemos visto en la teoría, el concepto de clase es altamente relevante dentro del modelo orientado a objetos. Es por ello que será una de las primeras cosas que aprenderemos a construir en la práctica.

5.1 Creación de una clase

Para crear una clase, utilizamos la palabra clave `Class ... End Class`.

```
Class Persona  
End Class
```

Dentro de este bloque, se colocará toda la información que constituye la estructura y el comportamiento de la clase que se esté construyendo, en este caso, la clase `Persona`. En nuestra implementación práctica, cabe aclarar que cuando mencionamos comportamiento nos referimos al conjunto de *acciones y reacciones* que poseen los objetos resultantes de instanciar la clase.

5.2 Tipos de clases

Existen tres tipos de clase que podemos reconocer: *clases abstractas*, *clases selladas* y *clases concretas*.

Una *clase abstracta* es aquella que puede heredarle a una o más subclases, pero no puede ser instanciada. Esto trae aparejado que nunca podamos tener objetos de clases abstractas. En general, son creadas para contener definiciones (estructura y/o comportamiento) que deseamos que futuras subclases posean. Para lograrlo, se deberá utilizar una característica denominada *herencia*. Esto permite que ese conjunto de subclases tenga en común lo que la superclase ha definido.

En el resto del texto utilizaremos como sinónimos superclase y clase base. Para crear una clase abstracta se utiliza **MustInherit**.

```
MustInherit Class Persona
```

```
End Class
```

Una *clase sellada* es aquella que puede instanciarse pero no se podrá heredar desde ella. Se construyen cuando se desea que una clase no pueda ser extendida por medio de la especialización. Se construye con **NotInheritable**.

```
NotInheritable Class Persona
```

```
End Class
```

Una *clase concreta* es aquella que puede ser instanciada o heredada. Estas clases son las clases más comunes y por ello no se utiliza ninguna palabra o instrucción en particular para crearlas.

```
Class Persona
```

```
End Class
```

Al desarrollar un sistema de información, cada clase representará un aspecto a considerar y estarán conviviendo en mayor o menor medida los tres tipos de clases.

5.2.1 Ámbitos de las clases

Consideramos como ámbito de un elemento a aquello que determina quién puede referenciarlo directamente. Debemos entender que una referencia directa es aquella que no recurre a técnicas de importación para darle visibilidad al elemento.

Aquí analizaremos el ámbito que pueden tener las clases, los cuales determinarán la visibilidad de las mismas (desde dónde se puede tener acceso a ellas). Los ámbitos que analizaremos también son aplicables a los miembros que esta posee (sus elementos internos), aunque algunos ámbitos en particular, según se avance con los ejemplos, será más natural verlos aplicados a los miembros que a las mismas clases.

Los niveles de acceso de una clase pueden ser:

- **Public** (Público)
- **Protected** (Protegido)
- **Friend** (Amigo)
- **Protected Friend** (Protegido y Amigo)
- **Private** (Privado)

Public. Se utiliza para declarar una clase que sea accesible desde cualquier lugar del proyecto actual o desde un proyecto que refiera al proyecto actual. La forma de declarar una clase con esta visibilidad es:

```
Public Class Persona
```

```
End Class
```

Protected. Al utilizarlo en la declaración de una clase, se logra que la misma sea accesible desde dentro de sí misma o desde una subclase de ella. Se utiliza para la declaración de miembros de la clase. Para aplicarlo a nivel de clase, en general se justifica hacerlo cuando existen clases anidadas.

```
Public Class Persona
```

```
    Protected Class Alumno
```

```
End Class
```

```
End Class
```

En el código anterior, se puede observar que la clase **Alumno** está declarada dentro de la clase **Persona**, a esto se lo denomina clase anidada, tema que profundizaremos más adelante. Aquí nos incumbe el efecto que causa colocar una clase con **Protected**. En este caso, la clase **Alumno** tendrá visibilidad desde la clase **Persona** o desde cualquier subclase que esta posea.

Como el **Protected** lo podemos utilizar a nivel de clase (dentro de una clase), el siguiente fragmento de código ocasionaría el siguiente error “Error: **Protected** solo puede ser usado dentro de una clase”.

```
Protected Class Alumno
```

```
End Class
```

Friend. Permite que la clase posea visibilidad desde el ensamblado donde esta se encuentra. Cabe aclarar que en esta tecnología, un ensamblado es la menor unidad ejecutable. Los ensamblados en general son archivos que se obtienen luego de la compilación del código fuente y normalmente poseen extensión .exe o .dll. A ellos pueden estar asociados aspectos referidos al versionamiento y la seguridad, entre otros.

```
Friend Class Persona
```

```
End Class
```

Protected Friend. Reúne las características enunciadas oportunamente para cada una de esas características (**Protected** y **Friend**) por separado. La visibilidad alcanza a la propia clase, sus clases derivadas y el ensamblado en el cual esta se encuentra.

```
Protected Friend Class Persona
```

```
End Class
```

Private. Si se utiliza en la declaración de una clase, se logra que la misma sea accesible solo desde dentro de quien la contiene. En general, se utiliza para la declaración de miembros de la clase. Se justifica utilizarlo a nivel de clase cuando existen clases anidadas.

```
Public Class Persona
```

```
    Private Class Alumno
```

```
    End Class
```

```
End Class
```

En este caso, la clase **Alumno** tendrá visibilidad desde la clase **Persona** solamente.

Como el **Private** lo podemos utilizar a nivel de clase, el siguiente fragmento de código occasionaría el siguiente error “Error: **Private** debe estar dentro de otro tipo”.

```
Private Class Alumno
```

```
End Class
```

5.3. Estructura de una clase

Se denomina estructura interna de una clase a todos los elementos que esta posee. Se debe tener presente que en un modelo orientado a objetos, las clases brindan una visión netamente estática. Esta visión estática estará compuesta por las características que la definen y los comportamientos que podrán adoptar sus instancias. Efectuada la aclaración precedente, dos elementos muy importantes definen la estructura interna de la clase: los correspondientes a su estructura estática, conformados por las características cualitativas y cuantitativas, y los correspondientes a su estructura dinámica, compuestos por sus acciones y reacciones. Este conjunto de elementos definen a la clase y en general una gran cantidad de ellos son los que tendrán visibilidad externa a través de su interfaz. También dentro de la clase nos encontraremos con variables, que serán las encargadas de mantener los valores que las instancias de estas clases (objetos) adoptarán en distintos momentos de su ciclo de vida (estados). Estas variables internas, también denominadas campos o variables miembro, son privadas, ya que se deberá respetar el criterio de encapsulamiento. Esto es muy importante en orientación a objetos debido a que por cada característica definida, hemos de prever como será leída y como será escrita, con todas las rigurosidades de validación que esta deba considerar. En definitiva, una clase debe ser visualizada como una especificación estática y sus instancias (objetos) son las que manifestarán dinámicamente lo que ellas definen.

5.3.1. Variables internas de la clase (campos – variables miembro)

Las variables internas de una clase o campos determinan, en primer lugar, la estructura real que esta poseerá, sus características cualitativas y cuantitativas. También pueden existir variables internas que solo sean utilizadas para cálculos intermedios y no necesariamente constituyan aspectos relevantes referidos a las características que la clase posee.

```
Public Class Persona
    'Campos de la Clase
    Dim Vnombre As String
    Dim Vapellido As String
    Dim Vedad As Integer
End Class
```

En este ejemplo, se puede observar que los campos definidos están precedidos por la instrucción `Dim`. Esto causa que los campos sean privados de la clase, por lo que serán visibles solo desde dentro de ella misma.

5.3.2. Acceso a las características de la clase (propiedades – getters y setters)

Los campos de una clase que representan características de la misma necesitan ser accedidos con el objetivo de ser leídos o escritos. Esto se puede hacer de dos maneras. En la primera, para leer el campo se utiliza una función que retorne el valor que posee y para escribirlo un procedimiento con parámetro que reciba el valor que se le colocará. La segunda es usando el concepto de propiedades (*properties*) que nos provee esta tecnología. Es una forma mucho más versátil de realizar esta tarea y será la que utilizaremos en la mayoría de los casos. No obstante, exemplificaremos ambas para que se pueda observar claramente cómo implementar la solución por cualquiera de las dos vías.

El siguiente código muestra como, a partir de la función `GetApellido`, se retorna el valor del campo `Vapellido` y con el procedimiento `SetApellido` se le asigna un valor.

```
Public Class Persona
    'Campos de la Clase
    Dim Vapellido
    Public Function GetApellido() As String
        Return Vapellido
    End Function
    Public Sub SetApellido(pApellido)
        Vapellido = pApellido
    End Sub
End Class
```

Si creamos un objeto del tipo `persona` y luego le solicitamos que nos muestre su interfaz colocándole un punto, se podrán observar `GetApellido` y `SetApellido` para ser utilizados. En la figura 5.1 se puede observar lo expresado.

Esta forma es muy utilizada por muchos lenguajes de programación, pero a continuación analizaremos el uso de las *properties*.

Existen cinco tipos de propiedades, que podemos enumerar como:

1. De escritura y lectura.
2. De solo lectura.
3. De solo escritura.
4. Con parámetros.
5. Por defecto.

Los cinco casos vienen provistos de una estructura que les permite realizar la escritura y/o la lectura de campos según corresponda. En general, se desarrollará una propiedad por cada característica de la clase, la cual poseerá un campo para almacenar el valor de esta cuando el objeto esté en ejecución.

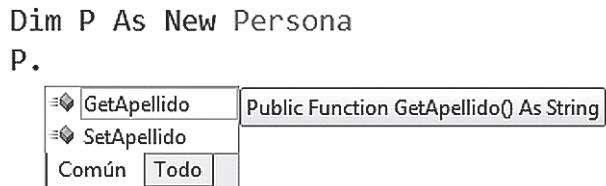


Figura 5.1.

Las propiedades son públicas por defecto y funcionan a nivel de clase. Básicamente, se trata de crear una propiedad y darle un nombre. Luego se programa el **Set** para determinar qué código se ejecutará al momento de ingresar un valor y el **Get** para cuando lo leamos. El **Set** posee un parámetro que se utiliza para recibir el valor ingresado.

A continuación, se abordará cada uno de los cinco tipos de propiedades planteados:

1. Propiedad de escritura y lectura

Una propiedad de lectura y escritura nos permite almacenar o leer el valor de un campo, de un objeto resultante de instanciar una clase. El bloque de código que se debe utilizar es:

```

Public Property NombreDeLaPropiedad() As Tipo
    Get
        ...
    End Get
    Set(ByVal value As Tipo)
        ...
    End Set
End Property

```

En el siguiente código se puede observar que el bloque **Set** recibe un dato del tipo **String** en el parámetro **value**. El dato recibido será el que se guarde en el campo que se designe para almacenar el estado de la característica que se maneja con esta propiedad.

El bloque **Get** simplemente retorna el contenido almacenado en el campo. En los bloques **Get** y **Set** se pueden colocar todas las líneas de código que sean necesarias para manipular el valor de ingreso o retorno, y estas pueden estar relacionadas con validaciones, cálculos y otras operaciones que el programador deba desarrollar.

Para utilizar esta propiedad (la cual estará definida dentro de una clase), necesitamos una instancia de ella (un objeto).

Desarrollemos un ejemplo asumiendo que nuestra propiedad **Color** está dentro de la clase **Auto**, el código quedaría como el siguiente.

```
Public Class Auto
    Private Vcolor As String
    Public Property Color() As String
        Get
            Return Vcolor
        End Get
        Set(ByVal value As String)
            Vcolor = value
        End Set
    End Property
End Class
```

Para poder cargar un valor, tendríamos que tener un objeto del tipo Auto:

```
Dim XA as New Auto
XA.Color = "Rojo"
```

```
Private Vcolor As String
Public Property Color() As String
    Get
        Return Vcolor
    End Get
    Set(ByVal value As String)
        Vcolor = value
    End Set
End Property
```

Para leer el valor almacenado a través de la propiedad Color hacemos:

```
Dim XA as New Auto
XA.Color = "Rojo"
Msgbox (XA.Color)
```

```
Private Vcolor As String
Public Property Color() As String
    Get
        Return Vcolor
    End Get
    Set(ByVal value As String)
        Vcolor = value
    End Set
End Property
```

2. Propiedad de solo lectura

Este tipo de propiedades se caracteriza por no permitir el ingreso de valores. Se utilizan generalmente cuando se da la situación en donde una característica que forma parte del estado del objeto se genera internamente por medio de cálculos y no se necesita ingresar el valor desde el exterior. En este caso, no tiene sentido que la propiedad permita ingresar valores.

Supongamos que desea conocer la fecha y hora en que un objeto ha sido instanciado, este es un valor que se genera internamente en el objeto y se observa la situación enunciada (no tiene sentido ingresar el valor desde el exterior del objeto). Para poder construir esto, se captura la información deseada (fecha y hora de instantiación) y se almacena en un campo para que pueda ser consultada en cualquier momento. En este caso, solo se necesita el bloque Get de la propiedad. Esto se puede lograr gracias a la palabra reservada `ReadOnly`. Al colocarla, solo se programa el Get. Si intentamos agregarle un Set dará un error. Si bien más adelante se abordará en detalle para qué se utiliza el procedimiento `New`, para comprender mejor el ejemplo siguiente estableceremos que el mismo se ejecuta al instanciarse la clase y crearse el objeto. Se puede observar que al suceder esto en el campo `VfechaHora`, se guarda la fecha y hora actual con la función `Now`.

```
Public Class Space
    Dim VfechaHora As Date
    Sub New()
        VfechaHora = Now
    End Sub
    Public ReadOnly Property FechaHoraInstancia() As Date
        Get
            Return VfechaHora
        End Get
    End Property
End Class
```

La forma de aprovechar esta propiedad sería programando lo siguiente:

```
Dim XS As New Space
MsgBox(XS.FechaHoraInstancia)

Public Class Space
    Dim VfechaHora As Date
    Sub New()
        VfechaHora = Now
    End Sub
    Public ReadOnly Property FechaHoraInstancia() As Date
        Get
            Return VfechaHora
        End Get
    End Property
End Class
```

3. Propiedad de solo escritura

Este tipo de propiedades se caracteriza por no permitir la lectura de valores. Se utilizan generalmente cuando una característica, que forma parte del estado del objeto, se carga desde el exterior del objeto y no se provee la posibilidad de leer el valor desde la interfaz del mismo. Supongamos que deseamos ingresar el título de una persona para que acompañe al nombre y apellido cuando lo retorne. Una vez ingresado, consideramos que leerlo por si solo no tiene sentido. En este caso, se necesita el bloque `Set` únicamente. La construcción se logra gracias a la palabra reservada `WriteOnly`. Al colocarla, solo se programa el `Set`. Si intentamos agregarle un `Get`, dará un error. En el ejemplo siguiente se ha construido una propiedad de solo escritura denominada `Titulo`. El valor ingresado a través de ella se almacena en el campo `Vtitulo` del objeto. Luego, el método `IngresaNombreyApellido` permite ingresar ambos valores y los almacena en los campos `Vnombre` y `Vapellido`. Por último, el método `ObtenerNombreyApellido` concatena los tres valores (`Titulo`, `NOMBRE` y `Apellido`) y los retorna. Se puede observar claramente que el valor ingresado por medio de la propiedad `Titulo` no ha sido necesario leerlo a través de ella pues se utiliza como parte de una cadena que se retorna por medio de otro método.

```

Dim XP As New Persona
XP.Titulo = "Sr."
XP.IngresaNombreyApellido _ 
    (InputBox("Ingrese el Nombre: "), _
     InputBox("Ingrese el Apellido: "))
MsgBox(XP.ObtenerNombreyApellido)

```

```

Public Class Persona
    Private Vtitulo As String
    'Propiedad de solo escritura
    Public WriteOnly Property Titulo() As String
        Set(ByVal value As String)
            Vtitulo = value
        End Set
    End Property
    'Declaración de Campos
    Dim Vnombre As String
    Dim Vapellido As String
    'Método con dos parámetros
    Public Sub IngresaNombreyApellido _ 
        (ByVal QueNombre As String, _
         ByVal QueApellido As String)
        Vnombre = QueNombre
        Vapellido = QueApellido
    End Sub
    'Función que retorna un String
    Public Function ObtenerNombreyApellido() As String
        Return Vtitulo & " " & Vnombre & " " & Vapellido
    End Function
End Class

```

4. Propiedades con parámetros

Las propiedades con parámetros otorgan la posibilidad de que, además del valor que ingresamos en la propiedad, podamos ingresar un valor o varios por los parámetros que se le coloquen. Por el resto, su funcionamiento es similar a lo que hemos visto hasta el momento.

Su estructura es la siguiente:

```

Public Property NombreDeLaPropiedad _ 
    (ByVal/ByRef parámetro1 As Tipo, _
     ByVal/Byref parámetro2 As Tipo, ...) As Tipo

    Get
        ...
    End Get
    Set(ByVal value As Tipo)

```

```
...
End Set
End Property
```

Los tipos de los parámetros se pueden definir por valor o referencia de acuerdo a las necesidades. Llevados estos conceptos a un ejemplo, a continuación podemos observar cómo se pueden manejar tres direcciones con la misma propiedad `Direccion`.

```
Dim Vdireccion(2) As String
Public Property Direccion (ByVal pIndex As Integer) As String
    Get
        If pIndex >= 0 And pIndex <= 2 Then
            Return Vdireccion(pIndex)
        End If
    End Get
    Set(ByVal value As String)
        If pIndex >= 0 And pIndex <= 2 Then
            Vdireccion(pIndex) = value
        End If
    End Set
End Property
```

Para lograrlo, se ha utilizado para el campo que almacena el valor un vector en lugar de una variable, y en la propiedad un parámetro denominado `pIndex`, que servirá para ingresar el índice que permita posicionarse en el vector.

Ahora, con estos ajustes se puede observar que logramos guardar hasta tres direcciones. Se debe tener presente que para utilizarlo se deben pasar: la dirección a cargar, que ingresará por el parámetro `value` del `Set`, y el índice que entrará por el parámetro `pIndex` de la propiedad.

```
Dim MiPersona As New Persona
MiPersona.Direccion(0) = _
<--> InputBox("Ingrese la dirección: ", , "4678-3465")<-->

Dim Vdireccion(2) As String
Public Property Telefono (ByVal pIndex As Integer) As String
    Get
        If pIndex >= 0 And pIndex <= 2 Then
            Return Vdireccion(pIndex)
        End If
    End Get
    Set(ByVal value As String)
        If pIndex >= 0 And pIndex <= 2 Then
            Vdireccion(pIndex) = value
        End If
    End Set
End Property
```

5. Propiedades por defecto

Las propiedades por defecto son propiedades que al ser utilizadas, podemos obviar escribir su nombre. Una clase solo puede tener definida una propiedad por defecto. Una propiedad para poder ser construida de esta manera, además, debe ser una propiedad con parámetros (al menos uno).

Si tomamos el ejemplo anterior, podríamos transformar la propiedad `Direccion` en una propiedad por defecto agregando la palabra `Default`.

```

Dim MiPersona As New Persona
MiPersona(0) = _

InputBox("Ingrese la dirección: ", , "4678-3465")>

Dim Vtelefono(2) As String
Default Public Property Telefono (ByVal pIndex As Integer) As String
Get
    If Index >= 0 And Index <= 2 Then
        Return Vtelefono(Index)
    End If
End Get
Set(ByVal value As String)
    If Index >= 0 And Index <= 2 Then
        Vtelefono(Index) = value
    End If
End Set
End Property

```

Como puede observarse en el código anterior, al momento de aprovechar lo construido no hace falta utilizar la forma `MiPersona.Teléfono(0)`. Al ser una propiedad por defecto, basta con colocar `MiPersona(0)`, omitiendo el nombre de la propiedad. Esta última forma puede confundirse con un vector, por lo cual se recomienda utilizarla con precaución.

5.3.3. Acciones de la clase (métodos)

Los *métodos* definidos en las clases se construyen por medio de las funciones y los procedimientos que esta posee internamente. Al igual que otros elementos de las clases, los métodos poseen un ámbito. En particular, por ahora nos interesa identificar los ámbitos *privado* (`dim` o `private`) y *público* (`public`). Si una función o procedimiento posee ámbito privado, no se observará en la interfaz de los objetos de la clase que lo implementa. Esto ocasiona que los elementos externos a la clase no tengan visibilidad de esos procedimientos y funciones. En general, denominamos *métodos* a las funciones o procedimientos públicos que figuran en la interfaz, mientras que a los privados los denominamos *funciones* o *procedimientos* de la clase.

La manera de construir un método sustentado en un procedimiento es colocando:

```
Public Sub NombreCompleto()
...
End Sub
```

Las líneas de código que se coloquen dentro del bloque definido por `Sub` y `End Sub` serán las que se ejecuten cuando se invoque al procedimiento.

El procedimiento anterior no posee parámetros pero podría tenerlos. Un procedimiento similar al anterior pero con parámetros podría ser:

```
Public Sub NombreCompleto _
    (ByVal QueNombre as String, ByVal QueApellido as String )
    Dim VnombreCompleto as String
    VnombreCompleto = QueNombre & " " & QueApellido
    MsgBox (VnombreCompleto)
End Sub
```

Se observa que el procedimiento `NombreCompleto` posee dos parámetros: uno para que se pueda ingresar el nombre y otro para el apellido. Ambos son por valor (`ByVal`), y se podría haber declarado por referencia (`ByRef`). Esto hubiese tenido como efecto que se pase la dirección de memoria donde está almacenado el nombre y el apellido en lugar de sus valores correspondientes. Un procedimiento se caracteriza por no retornar valor, y en este caso particular lo que hace es guardar el nombre en la variable local `VnombreCompleto` que es de tipo `String`, concatenándole un espacio y luego el apellido. La última línea de código lo muestra en una caja de mensajes. Si a este procedimiento le hubieran pasado como nombre “Juan” y como apellido “Garcia”, al momento de mostrarlo se vería “Juan Garcia”.

Las funciones se crean colocando:

```
Public Function NombreCompleto () As String
...
End Function
```

Como se puede observar, las palabras `Function` ... `End Function` determinan el bloque de código de la función (donde se colocan las instrucciones a ser ejecutadas). Como peculiaridad, vemos que la función posee al final de su firma un tipo de retorno que en este caso particular es `String`. Ese tipo de retorno determina de qué tipo de datos será lo que la función retorna.

Las funciones, al igual que los procedimientos, pueden o no tener parámetros, y estos ser `ByVal` o `ByRef` según corresponda.

```
Public Function NombreCompleto _  
    (ByVal QueNombre as String, ByVal QueApellido as String ) As String  
    Return QueNombre & " " & QueApellido  
End Function
```

Para comprender mejor como se construyen procedimientos y funciones, se aplicarán estos conceptos a la clase Persona en un ejemplo.

La clase Persona posee dos campos denominados Vnombre y Vapellido. Se creará un procedimiento denominado IngresaNombreyApellido para ingresar el nombre y el apellido de una persona y una función denominada ObtenerNombreyApellido que los retorne concatenados.

```
Public Class Persona  
    'Declaración de Campos  
    Dim Vnombre As String  
    Dim Vapellido As String  
    'Método con dos parámetros  
    Public Sub IngresaNombreyApellido _  
        (ByVal pNombre As String, _  
         ByVal pApellido As String)  
        Vnombre = pNombre  
        Vapellido = pApellido  
    End Sub  
    'Función que retorna un String  
    Public Function ObtenerNombreyApellido() As String  
        Return Vnombre & " " & Vapellido  
    End Function  
End Class
```

Al instanciar la Clase Persona como se observa más abajo, si se coloca P (nombre de la variable que apunta al objeto resultante de instanciar la clase persona) seguido de un punto, se verá la interfaz. En este caso particular, se pueden observar los métodos IngresaNombreyApellido y ObtenerNombreyApellido.

```
Public Class Form1  
    Private Sub Button1_Click _  
        (sender As System.Object, e As System.EventArgs) _  
        Handles Button1.Click  
        'Se declara la variable P y se instancia una Persona  
        Dim P As New Persona  
        'Se pasa el nombre y apellido al método por medio de InputBox  
        P.IngresaNombreyApellido _  
        (InputBox("Ingrese el nombre: "), _
```

```
    InputBox("Ingrese el apellido: ")
    'Se muestra en un MsgBox lo que retorna el método ObtenerNombreyApellido
    MsgBox(P.ObtenerNombreyApellido)
End Sub
End Class
```

5.3.4. Constructores

El constructor es un procedimiento cuya peculiaridad es ser el primer procedimiento que se ejecuta cuando se instancia un objeto de la clase que lo posee.

Su forma es:

```
Sub New
...
End Sub
```

Los constructores pueden poseer varios parámetros o ninguno. Son muy útiles cuando se desea inicializar el estado de un objeto ni bien se crea. Cuando poseen parámetros, se les pueden pasar valores desde el exterior que sirvan para inicializar el estado o lograr un comportamiento peculiar en ese preciso momento inicial. En el ejemplo siguiente, podremos observar que tanto Persona como Auto poseen constructores. En el caso de Persona, el constructor no posee parámetros y lo que hace es inicializar el estado del objeto al momento que es instanciado, otorgándole a los campos Vnombre y Vapellido valores de cadena vacía. En el caso de la clase Auto, que posee un parámetro, cuando se instancia el objeto se le está pasando lo que se ingresa por medio del InputBox (en este caso la patente), y ese valor inicializará el campo Vpatente.

```

Public Class Form1
    Private Sub Button1_Click _
        (sender As System.Object, e As System.EventArgs)
        Handles Button1.Click
        Dim P As New Persona
        Dim A As New Auto(InputBox("Ingrese la patente: "))
    End Sub
End Class
Public Class Persona
    Dim Vnombre As String
    Dim Vapellido As String
    Sub New()
        Vnombre = ""
        Vapellido = ""
    End Sub
End Class
Public Class Auto
    Dim Vpatente As String
    Sub New(ByVal pPatente As String)
        Vpatente = pPatente
    End Sub
End Class

```

5.3.5. Destructores

Un destructor es un procedimiento que se ejecuta justo antes de eliminar el objeto. En la tecnología que estamos trabajando, el concepto básico de destructor se maneja a través de dos procedimientos denominados `Finalize` y `Dispose`. Para entender mejor por qué es así, veamos cómo se maneja la memoria administrada de nuestro equipo (Heap). Para administrar la memoria eficientemente existe el `Garbage Colector` (GC). Él es el encargado de liberar memoria cuando esta es escasa, colocando como memoria disponible a aquella que posee objetos que ya no se utilizan, detectando esta situación debido a que esos objetos no tienen referencias que los apunten. Funciona automáticamente aunque se lo puede invocar manualmente. Esto último no es recomendable debido a la gran cantidad de recursos que insume su ejecución.

Los objetos pertenecen por defecto a una generación. Esta indica cuánto tiempo de vida tienen. Al funcionar el recolector de basura, los objetos que no son borrados (sobreviven) se elevan a la generación siguiente, esta puede oscilar entre 0 y 2. De esta forma, se puede saber cuáles objetos han sobrevivido más, asumiendo que esto se debe a que son más utilizados. Este aspecto, que apunta sin lugar a dudas a la optimización del proceso, puede acarrear un problema. Este radica en que el tiempo que transcurre entre que un objeto ha dejado de usarse y que el Garbage Colector lo recolecte de la memoria no es necesariamente poco. Este desfasaje en el tiempo que transcurre entre que se deja de utilizar y su destrucción, puede aparejar que un recurso

compartido quede tomado por un objeto que ya no lo necesita, pues probablemente él esté en desuso en la memoria a la espera de que Garbage Colector determine que hace falta espacio y decida borrarlo, mientras que otro que lo desea utilizar no podrá hacerlo. Esta situación resulta no solo poco deseable sino inaceptable. Es por ello que utilizamos los dos procedimientos que mencionamos anteriormente. El primero, `Finalize`, asociado a Garbage Colector, será el último procedimiento que se ejecute antes de que Garbage Colector mate al objeto; y el segundo, `Dispose`, está asociado al programador para que este lo use cuando lo necesite y de esta manera no tenga que depender de que Garbage Colector se ejecute. En ambos procedimientos, se coloca código o se llaman rutinas siempre relacionadas con acciones que deseamos que ocurran al finalizar el uso de un objeto. Por ejemplo, en las relaciones de agregación con contención física, el objeto que agrega, crea y agrega al resto de los objetos (los agregados) en lo que denominamos *constructor*, al finalizar el ciclo de vida del objeto que agrega debe matar a los objetos que creó y agregó, y esto se realiza en un procedimiento denominado *destructor* que es el último procedimiento que se ejecuta antes de que sea eliminado.

El siguiente código muestra como se puede programar `Finalize`.

```
Protected Overrides Sub Finalize()
    'Este procedimiento es disparado por el GC
    Console.WriteLine("La persona " & Me.Apellido & ", " & Me.Nombre & _
        " está siendo borrada")
End Sub
```

Si observamos atentamente la firma de este procedimiento, vemos que su ámbito es `Protected` y está sobrescribiendo (`Overridable`) la definición original. Esto se debe a que este procedimiento se hereda de `Object`. Es importante respetar la firma del mismo si no dará un error. Más adelante, trataremos en profundidad la *sobrescritura de métodos*.

Una buena manera de balancear el uso de `Dispose` y `Finalize` es usando ambos. En el ejemplo siguiente, se puede observar como se prevé un bloque de código que hace que ambos coexistan.

```
Public Class PuebaDispose
    Implements IDisposable
    Private disposedValue As Boolean ' To detect redundant call IDisposable
    Protected Overridable Sub Dispose(disposing As Boolean)
        If Not Me.disposedValue Then
            If disposing Then
                ' TODO: dispose managed state (managed objects).
            End If
                ' TODO: free unmanaged resources (unmanaged objects) and
                ' override Finalize() below.
                ' TODO: set large fields to null.
    End Sub
```

```

    End If
    Me.disposedValue = True
End Sub
    ' TODO: override Finalize() only if Dispose(ByVal
    ' disposing As Boolean) above has code to free unmanaged
    ' resources.
    'Protected Overrides Sub Finalize()
    ' Do not change this code. Put cleanup code in
    ' Dispose(ByVal disposing As Boolean) above.
    '   Dispose(False)
    ' MyBase.Finalize()
    'End Sub

    ' This code added by Visual Basic to correctly implement
    ' the disposable pattern.
Public Sub Dispose() Implements IDisposable.Dispose
    ' Do not change this code. Put cleanup code in
    ' Dispose(ByVal disposing As Boolean) above.
    Dispose(True)
    GC.SuppressFinalize(Me)
End Sub
End Class

```

Por haber utilizado la interfaz `IDisposable`, el procedimiento implementado originalmente (`Dispose`) básicamente delega la ejecución a una sobrecarga del método que posee un parámetro de tipo `Boolean`, denominado `disposing`. Además, este procedimiento posee una variable llamada `disposedValue`, que oficia de variable semáforo para saber si ya se ha ejecutado este procedimiento y evitar que se ejecute varias veces. Si esto ocurriese, ciertas líneas de código podrían dar error ya que existen elementos en programación que si son destruidos, borrados o cerrados y se vuelven a ejecutar, nos conducirán a un error.

El valor `boolean` recibido en `disposing` se utiliza para diferenciar entre el tratamiento de los objetos administrados y los no administrados, pudiendo discriminar cuándo deseó trabajar sobre un grupo u otro.

También existe la posibilidad de querer borrar los recursos no administrados cuando Garbage Collector se ejecute, en ese caso habría que sobrescribir el procedimiento `Finalize` y como puede observarse en el código, él llama al procedimiento `Dispose` pasándole un parámetro `False` para que evite ejecutar las líneas que tratan a los objetos administrados.

En resumen, podrían darse los siguientes escenarios:

1. Que el programador invoque `Dispose` y no se encuentre sobrescrito `Finalize`: la primera vez, se ejecutará el código que maneja a los objetos administrados y no administrados. La segunda, saltará todo pues `disposedValue` será `True`.
2. Que el programador invoque `Dispose` con `True` antes de que Garbage Collector mate al objeto y se encuentre sobrescrito `Finalize`: la primera vez, se ejecutará el código que

maneja a los objetos administrados y no administrados. La segunda, saltará todo pues `disposedValue` será `True`. Cuando Garbage Collector se ejecute, `Finalize` será ignorado pues previamente se había ejecutado el código `GC.SuppressFinalize(Me)`. Esto ocasiona que Garbage Collector ignore a ese objeto para la ejecución de su `Finalize`.

3. Que el programador invoque `Dispose` con `True` después de que Garbage Collector mate al objeto y se encuentre sobrescrito `Finalize`: la primera vez, se ejecutará el código que maneja a los objetos no administrados. Intentar ejecutar `Dispose` después de que esto haya ocurrido genera un error de referencia nula, pues el objeto ya fue sacado de la memoria por parte de Garbage Collector.

Como puede apreciarse, se puede discriminar con bastante flexibilidad respecto a qué y cómo se desean borrar los objetos administrados y los no administrados.

5.3.6. Eventos

Los eventos pueden ser considerados como las reacciones de los objetos a estímulos externos. Cabe destacar que en la teoría pura de objetos no se mencionan los eventos, pues se considera que el comportamiento definido en una clase, y en consecuencia el que tendrán sus instancias, queda circunscripto al conjunto de acciones que posea. En la tecnología que se está utilizando podemos ampliar el concepto y considerar que el comportamiento de un objeto está dado por el conjunto de métodos y eventos que este posee. Dicho en otras palabras, las acciones y reacciones que posee un objeto conforman su comportamiento.

Los eventos son un aporte muy importante porque permiten programar bloques de código para que se ejecuten cuando ocurra un determinado suceso. Esto permite trabajar con un cierto asincronismo ya que no se debe recurrir periódicamente a la indagación para determinar la ocurrencia del suceso sino todo lo contrario. Cuando ocurra el suceso, provocará que se desencadene la ejecución del bloque de código con el que está asociado el evento. Hace un tiempo, la inclusión del concepto de eventos fue tan importante que dio lugar a un tipo de programación denominada “programación orientada a eventos”.

Para crear un evento hay que desarrollar tres cosas:

1. La declaración del evento en la clase.
2. Lo que le debe ocurrir (suceso) a un objeto de la clase para que el evento acontezca.
3. Asociar el evento a un procedimiento del ámbito donde el objeto que lo posee está operando.

La forma de declaración de un evento estándar es:

```
Public Event Mievento(ByVal sender As Object, ByVal e As EventArgs)
```

Se coloca el ámbito (`Public`), luego la palabra `Event` seguida del nombre del evento (`Mievento`) elegido por el programador y, si está construido de acuerdo a las buenas prácticas de programación, dos parámetros. El primer parámetro se denomina `sender` del tipo `Object` (con lo cual podrá ir potencialmente cualquier tipo de objeto por ser todos derivados de `Object`), y llevará al objeto responsable de que el evento haya ocurrido. El segundo parámetro es `e`, del tipo `EventArgs`, y llevará toda la información adicional que el suceso necesite. Más adelante se verá como especializar al objeto de tipo `EventArgs` para que se ajuste a cualquier necesidad. La forma estándar para hacer que un suceso ocurra es con la palabra `RaiseEvent` y el nombre del evento. Además, se deben colocar los dos parámetros indicados anteriormente. Si en el segundo parámetro se envía `Nothing`, significa que no se desea enviar ninguna información adicional al evento.

```
RaiseEvent Mievento(Me, Nothing)
```

Supongamos para nuestro ejemplo que tenemos la clase `persona` y que esta posee la capacidad de enviar y recibir cartas. Se desea que cada vez que reciba una carta se desencadene un evento. Para ello seguiremos los pasos 1, 2 y 3 mencionados anteriormente.

1. Declaramos el evento.

```
Public Event EntroUnaCarta _  
    (ByVal sender As Persona, ByVal e As MisArgumentosEventArgs)
```

Como se puede observar, el nombre de nuestro evento es `EntroUnaCarta` y posee los dos parámetros que explicamos anteriormente, con la peculiaridad que el tipo del segundo parámetro es `MisArgumentosEventArgs`. La funcionalidad es la misma pero más especializada. Lo abordaremos unos puntos más adelante.

2. Establecemos cuándo se debe ejecutar (“desencadenar”) el evento.

Esto se programa en el lugar que representa lo que le deberá ocurrir al objeto para que el evento se ejecute. En nuestro ejemplo, será la recepción de una nueva carta. En este caso particular, asumimos que cuando se recibe una carta se ejecuta un procedimiento denominado `RecibirCarta` de la clase `persona`.

El siguiente código muestra cómo se debe disparar el evento `EntroUnaCarta`.

```
RaiseEvent EntroUnaCarta(Me, New MisArgumentosEventArgs)
```

Podemos observar que el primer parámetro representa al objeto en sí mismo (`Me`), mientras que el segundo es una especialización de `EventArgs` (`MisArgumentosEventArgs`).

`RaiseEvent` se debe utilizar en el lugar deseado, como se comentó anteriormente cuando entra una carta. Para nuestro caso, el procedimiento `RecibirCarta` de la clase `Persona`.

```
Public Sub RecibirCarta(ByVal QueCarta As Carta)
    VMisCartas.Add(QueCarta)
    RaiseEvent EntroUnaCarta(Me, New MisArgumentosEventArgs)
End Sub
```

3. Usamos el evento creado.

El tercer paso es usar el evento creado. Para ello se observan varios detalles. El primero es que la variable `P10` del tipo `Persona` está declarada a nivel de clase con la palabra `WithEvents`. Esto provoca que para esa instancia de `Persona`, estén habilitados los eventos.

```
Dim WithEvents P10 As persona = New Persona
```

El evento se puede utilizar asociándolo a un procedimiento (`P10_EntroUnaCarta`) por medio de `Handles`. Para que se comprenda mejor su funcionamiento, primero se analizará el código de la clase `MisArgumentosEventArgs`, ya que el segundo parámetro del procedimiento es de este tipo.

La clase `MisArgumentosEventArgs` hereda de `System.EventArgs` y por relación “es-un” (herencia), es un argumento de evento. Posee un constructor que carga la fecha y la hora del sistema en el campo `VfechaHora` al momento de instanciarse el objeto y una función denominada `FechaHoraCorreo` que permite consultar ese valor.

```
Public Class MisArgumentosEventArgs
    Inherits System.EventArgs
    Private VfechaHora As Date
    Sub New()
        VfechaHora = Now
    End Sub
    Public Function FechaHoraCorreo() As Date
        Return VfechaHora
    End Function
End Class
```

Ahora, podemos comprender la mecánica de la asociación del evento `P10.EintroUnacarta` con el procedimiento `P10_EntroUnaCarta` a través de `Handles`. Dentro del procedimiento, se coloca

el código que se desea ejecutar cuando se produzca el suceso (la llegada de una carta) en el objeto P10 y se provoque que el evento EntroUnaCarta de ese objeto se dispare (RaiseEvent...) y ocurra la ejecución del procedimiento P10_EntroUnaCarta.

```
Private Sub P10_EntroUnaCarta(ByVal sender As persona, _
                               ByVal e As MisArgumentosEventArgs) _
                               Handles P10.EntroUnaCarta
    MsgBox("A: " & sender.Nombre & " le entró un correo ." & _
           Constants.vbCrLf & "En la fecha: " & e.FechaHoraCorreo & _
           Constants.vbCrLf & "Enviado por: " & _
           Sender.MisCartas.Item(sender.MisCartas.Count - 1).Remitente.Nombre & _
           Constants.vbCrLf & "Que dice: " & _
           Sender.MisCartas.Item(sender.MisCartas.Count - 1).Mensaje)
End Sub
```

El código completo del ejemplo quedaría como el siguiente:

```
Public Class Form1
    Dim WithEvents P10 As Persona = New Persona
    Dim P11 As New Persona
    Private Sub Button1_Click(ByVal sender As System.Object, _
                           ByVal e As System.EventArgs) _
                           Handles Button1.Click
        Dim C As New Carta
        P10.Nombre = "María"
        P11.Nombre = "Juan"
        C.Remitente = P11
        C.Destinatario = P10
        C.Mensaje = "TE INVITO AL CURSO DE POO QUE ES MUY INTERESANTE"
        P11.EnviarCarta(C)
    End Sub
    Private Sub P10_EntroUnaCarta(ByVal sender As Persona, _
                               ByVal e As MisArgumentosEventArgs) _
                               Handles P10.EntroUnaCarta
        MsgBox("A: " & sender.Nombre & " le entró un correo ." & _
               Constants.vbCrLf & "En la fecha: " & e.FechaHoraCorreo & _
               Constants.vbCrLf & "Enviado por: " & _
               sender.MisCartas.Item(sender.MisCartas.Count - 1) _
               .Remitente.Nombre & Constants.vbCrLf & "Que dice: " & _
               sender.MisCartas.Item(sender.MisCartas.Count - 1).Mensaje)
    End Sub
End Class
Public Class Persona
    Public Event EntroUnaCarta(ByVal sender As Persona, _
                               ByVal e As MisArgumentosEventArgs)
```

```

Private Vnombre As String
Public Property Nombre() As String
    Get
        Return Vnombre
    End Get
    Set(ByVal value As String)
        Vnombre = value
    End Set
End Property
Private VmisCartas As New List(Of Carta)
Public ReadOnly Property MisCartas() As List(Of Carta)
    Get
        Return VmisCartas
    End Get
End Property
Public Sub EnviarCarta(ByVal pQueCarta As Carta)
    pQueCarta.Destinatario.RecibirCarta(pQueCarta)
End Sub
Public Sub RecibirCarta(ByVal pQueCarta As Carta)
    VmisCartas.Add(pQueCarta)
    RaiseEvent EntroUnaCarta(Me, New MisArgumentosEventArgs)
End Sub
End Class

```

```

Public Class Carta
    Private Vdestinatario As Persona
    Public Property Destinatario() As persona
        Get
            Return Vdestinatario
        End Get
        Set(ByVal value As persona)
            Vdestinatario = value
        End Set
    End Property

    Private VRemitente As persona
    Public Property Remitente() As persona
        Get
            Return VRemitente
        End Get
        Set(ByVal value As persona)
            VRemitente = value
        End Set
    End Property
    Private VMensaje As String
    Public Property Mensaje() As String
        Get
            Return VMensaje
        End Get

```

```

        Set(ByVal value As String)
            VMensaje = value
        End Set
    End Property
End Class
Public Class MisArgumentosEventArgs
    Inherits System.EventArgs
    Private VfechaHora As Date
    Sub New()
        VfechaHora = Now
    End Sub
    Public Function FechaHoraCorreo() As Date
        Return VfechaHora
    End Function
End Class

```

En el ejemplo anterior, se puede observar que existen dos instancias de persona, P10 (María) y P11 (Juan), y que también tenemos una carta C. La carta posee un remitente que es P11 y un destinatario que es P10, además de un mensaje que expresa “TE INVITO AL CURSO DE POO QUE ES MUY INTERESANTE”.

P11 le envía una carta C al destinatario P10 por medio del método `EnviarCarta`.

P11.EnviarCarta(C)

Al ejecutarse el método que resuelve la solicitud de este mensaje en P11, el siguiente código entra en acción.

```

Public Sub EnviarCarta(ByVal pQueCarta As Carta)
    pQueCarta.Destinatario.RecibirCarta(pQueCarta)
End Sub

```

Se toma el destinatario que posee la carta que llegó a `pQueCarta` y se le envía un mensaje `RecibirCarta` con la carta como parámetro. Debemos recordar que el destinatario de la carta es P10 y en él se ejecuta el siguiente código.

```

Public Sub RecibirCarta(ByVal pQueCarta As Carta)
    VmisCartas.Add(pQueCarta)
    RaiseEvent EntrouUnaCarta(Me, New MisArgumentosEventArgs)
End Sub

```

El código del procedimiento `RecibirCarta` agrega la carta que recibe a la lista de cartas `VmisCartas` y se desencadena el evento `EntroUnaCarta` con `RaiseEvent`. Allí se envía como objeto responsable del desencadenamiento en el primer parámetro el objeto `P10`, colocando `Me` y una instancia de `MisArgumentosEventArgs`, que es la especialización de argumento de eventos que realizamos.

Al ocurrir esto, donde está instanciado `P10` se manifiesta el evento `EntroUnaCarta` y allí llega `sender` (`P10`) y el argumento de evento `e` (una instancia de `MisArgumentosEventArgs`).

Este evento está asociado al procedimiento `P10_EntroUnaCarta` por medio de `Handles`, lo que produce que ese procedimiento se ejecute.

Para terminar con este tema, solo expresaremos que `sender` también es útil cuando un procedimiento está relacionado por medio de `Handles` con más de un *evento* del mismo objeto o de distintos objetos para discriminar cuál es el objeto responsable de que el procedimiento se esté ejecutando.

5.4. Relaciones entre clases

Como se ha destacado en el abordaje teórico de esta bibliografía, las *clases* se relacionan con el objetivo de poder conformar el modelo de representación abstracta de un dominio de problema real. En definitiva, son la esencia y el punto de partida de todo el modelo orientado a objetos.

En particular, se profundizarán algunas relaciones por sus implicancias y por ser las más utilizadas en la práctica.

5.4.1. Herencia

Para comenzar a tratar este tema, se establecerán como sinónimos:

- a) *Superclase*, clase base, clase de orden superior.
- b) *Subclase*, clase derivada, clase de orden inferior.

La herencia es una de las relaciones que pueden darse entre clases. Este tipo de relación permite que una *subclase* posea (reciba) todo lo que una *superclase* (de orden superior o superclase) le hereda (propiedades, métodos, eventos, etcétera). Cabe aclarar que en la tecnología que usamos para la práctica, los constructores no se heredan, por lo que hay que programarlos en cada clase. Esta forma de relación permite generar jerarquías del tipo *Generalización – Especialización*. Una de las características más interesantes de la herencia es que se reutiliza el código ya programado. Otra característica es que la subclase también será del tipo de la superclase, consecuencia directa de la relación que existe entre la herencia y la relación conceptual que plantea el modelo orientado a objetos “es-un” (este tema se abordará con mayor profundidad en el punto que se analizan los *tipos*). En el código siguiente se observa una herencia. La clase `Persona` implementa las propiedades `Nombre` y `Apellido` y oficia de superclase o clase base ya

que le hereda a Alumno. La clase Alumno recibe todo lo que la clase Persona posee (salvo los constructores, que en este caso no existen). Luego, la clase Alumno se especializa agregando Legajo a su propia característica y VerDatosAlumno a su comportamiento. La jerarquía creada por la herencia es multinivel, no existe una cantidad fija de niveles sobre los que se puede heredar, eso queda a criterio del desarrollador.

```
Public Class Form1
    Private Sub Button1_Click _
        (sender As System.Object, e As System.EventArgs) Handles Button1.Click
        Dim A1 As New Alumno
        A1.Nombre = InputBox("Ingrese el nombre: ")
        A1.Apellido = InputBox("Ingrese el apellido: ")
        A1.Legajo = InputBox("Ingrese el legajo: ")
        MsgBox(A1.VerDatosAlumno)
    End Sub
End Class
Public Class Persona
    Private Vnombre As String
    Public Property Nombre() As String
        Get
            Return Vnombre
        End Get
        Set(ByVal value As String)
            Vnombre = value
        End Set
    End Property
    Private Vapellido As String
    Public Property Apellido() As String
        Get
            Return Vapellido
        End Get
        Set(ByVal value As String)
            Vapellido = value
        End Set
    End Property
End Class
Public Class Alumno
    Inherits Persona
    Private Vlegajo As String
    Public Property Legajo() As String
        Get
            Return Vlegajo
        End Get
        Set(ByVal value As String)
            Vlegajo = value
        End Set
    End Property
    Public Function VerDatosAlumno() As String
```

```

Return "Legajo: " & vbTab & Me.Legajo & vbCrLf & _
"Apellido: " & vbTab & Me.Apellido & ", " & vbCrLf & _
"Nombre: " & vbTab & Me.Nombre
End Function
End Class

```

5.4.1.1. Tipos de Herencia

Existen dos tipos de herencia que se denominan herencia simple y herencia múltiple. La *herencia simple* se produce cuando una sola superclase le hereda estructura y comportamiento a una o más subclases. Si en lugar de esto una o más subclases heredan estructura y comportamiento de más de una superclase, estamos en presencia de *herencia múltiple*.

En la tecnología que utilizamos para exemplificar estos conceptos, solo podremos usar la herencia simple ya que no contempla la herencia múltiple. Existen controversias sobre la utilidad de la herencia múltiple debido a que hay opiniones encontradas sobre sus beneficios. Están quienes defienden la postura de que los problemas que resuelve la herencia múltiple se pueden resolver con otros mecanismos, como la utilización de herencia simple y la implementación de interfaces, entre otras formas.

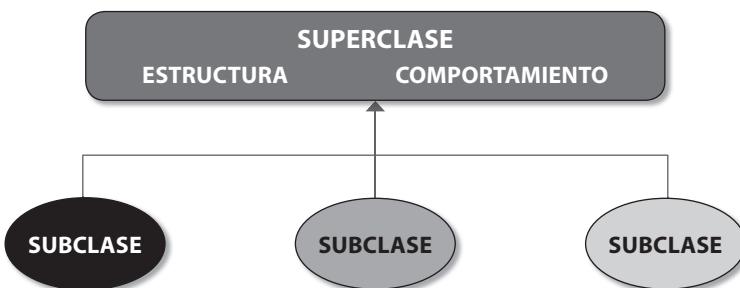


Figura 5.2. Esquema de herencia simple.

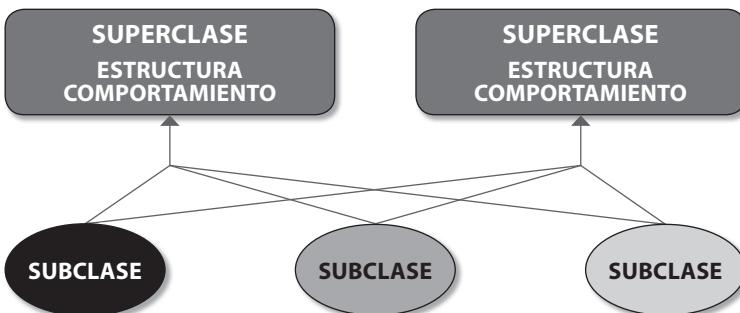


Figura 5.3. Esquema de herencia múltiple.

5.4.1.2. Sobrecarga

Se denomina sobrecarga cuando se tiene varios métodos con igual nombre y distinta firma. Se denomina firma de un método al conjunto **Nombre del método + cantidad de parámetros + tipo de cada parámetro**. El tipo de valor devuelto en caso de que el método sea una función no interviene en la determinación de la firma. A continuación, un ejemplo donde puede observarse claramente un método sobrecargado denominado NombreCompleto en la clase Persona.

```
Public Class Persona
    Private Vnombre As String
    Public Property Nombre() As String
        Get
            Return Vnombre
        End Get
        Set(ByVal value As String)
            Vnombre = value
        End Set
    End Property
    Private Vapellido As String
    Public Property Apellido() As String
        Get
            Return Vapellido
        End Get
        Set(ByVal value As String)
            Vapellido = value
        End Set
    End Property
    Private VfechaNacimiento As Date
    Public Property FechaNacimiento() As Date
        Get
            Return VfechaNacimiento
        End Get
        Set(ByVal value As Date)
            VfechaNacimiento = value
        End Set
    End Property
    Public Function NombreCompleto() As String
        Return Me.Nombre & ", " & Me.Apellido
    End Function
    Public Function NombreCompleto(ByVal titulo As String) As String
        Return titulo & " " & Me.NombreCompleto
    End Function
End Class
```

5.4.1.3. Sobrescritura

La sobrescritura de miembros permite trabajar sobre un elemento heredado cuando no resulta útil la implementación que posee, se desea refinar la implementación o bien ha sido heredado desde una clase abstracta siendo un método virtual.

Para que un miembro se pueda sobrescribir, debe estar definido como sobrescribible con la palabra **Overridable** (su contexto de aplicación son procedimientos, funciones y propiedades).

Si un método heredado está definido como **Overridable**, entonces en la subclase se puede sobrescribir (redefinir su código) con **Overrides**. Veamos algunos ejemplos de sobrescritura.

En el siguiente código se puede observar cómo se sobrescribe el cálculo del precio de descuento que es de un 10% en la clase **Producto** y luego en **ProductoA** pasó a ser del 30%.

```
Public Class Producto
    Private Vprecio As String
    Public Property Precio() As String
        Get
            Return Vprecio
        End Get
        Set(ByVal value As String)
            Vprecio = value
        End Set
    End Property
    Public Overridable Function PrecioDescuento() As Double
        Return Me.Precio * 0.9
    End Function
End Class

Public Class ProductoA
    Inherits Producto
    Public Overrides Function PrecioDescuento() As Double
        Return Me.Precio * 0.7
    End Function
End Class
```

En el siguiente ejemplo se puede observar la sobrescritura de un procedimiento. En la clase **Servicio**, el método **HoraInstanciacion** muestra la fecha y hora de instanciación. La clase **ServicioA** hereda de **Servicio** y sobrescribe el método **HoraInstanciacion**, donde solo se muestra la hora de instanciación de los objetos que sean instancias de **ServicioA**.

```
Public Class Servicio
    Protected Vhora As Date
    Sub New()
        Vhora = Now
```

```

    End Sub
    Public Overridable Sub HoraInstanciacion()
        MsgBox(Me.Vhora)
    End Sub
End Class
Public Class ServicioA
    Inherits Servicio
    Sub New()
        Vhora = Now
    End Sub
    Public Overrides Sub HoraInstanciacion()
        MsgBox(Me.Vhora.TimeOfDay.ToString.Substring(0, 8))
    End Sub
End Class

```

En el siguiente ejemplo, se puede observar cómo sobrescribir una propiedad.

```

Public Class Auto
    Private Vpatente As String
    Public WriteOnly Property Patente() As String
        Set(ByVal value As String)
            Vpatente = value
        End Set
    End Property
    Private Vcolor As System.Drawing.Color
    Public WriteOnly Property Color() As System.Drawing.Color
        Set(ByVal value As System.Drawing.Color)
            Vcolor = value
        End Set
    End Property
    Public Overridable ReadOnly Property DatosAuto() As String
        Get
            Return Me.Vpatente & ", " & Me.Vcolor.ToString
        End Get
    End Property
End Class
Public Class AutoA
    Inherits Auto
    Public Overrides ReadOnly Property DatosAuto As String
        Get
            Return "Los datos del auto son:" & vbCrLf & _
                "Patente:" & vbTab & MyBase.DatosAuto.Split(",")(0) & vbCrLf & _
                "Color:" & vbTab & MyBase.DatosAuto.Split(",")(1)
        End Get
    End Property
End Class

```

Básicamente, en el ejemplo anterior la sobrescritura está formateando de distinta manera el texto de salida de la propiedad `DatosAuto` con los datos de la patente y el color. El texto resultante se vería como en la figura 5.4.

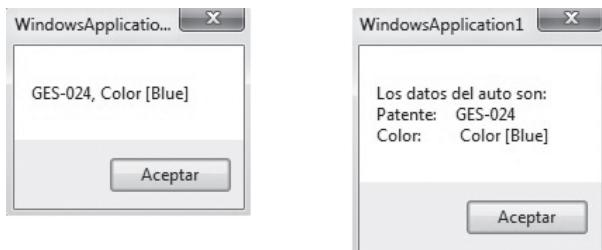


Figura 5.4.

Otro caso de sobrescritura se puede dar cuando se hereda un método virtual.

```
Public MustInherit Class Alumno
    Public MustOverride Function Promedio(Notas() As Integer)
End Class
Public Class AlumnoUniversitario
    Inherits Alumno
    Public Overrides Function Promedio(Notas() As Integer) As Object
        Dim T As Double
        Dim C As Integer
        For Each N As Integer In Notas
            T += N
            C += 1
        Next
        Return T / C
    End Function
End Class
```

Como se puede observar en el código anterior, la clase abstracta `Alumno` tiene un método virtual `Promedio`. Este método está preparado para recibir un array de `Notas`. La clase `AlumnoUniversitario` hereda de `Alumno` y sobrescribe el método virtual para calcular el promedio de las notas recibidas.

Un caso especialmente interesante dentro del contexto de la sobrescritura es cuando tenemos un método sobrecargado y sobrescribible. Al heredar de la clase que posee este método, existe la posibilidad de que no se desee aprovechar todas las sobrecargas y solo una o algunas de ellas. En este caso, debemos usar `Shadows`. En el siguiente ejemplo, la clase `Texto` posee el método `Decora` que es sobrescribible y sobrecargado. La clase `TextoEspecial` hereda de la clase `Texto` y con ello

el método. La sobrescritura se lleva adelante con Shadows (sombreado). Esto causa que se sobrecriba solo la sobrecarga que coincide con la firma mientras que la otra se oculta. En caso de que la sobrecarga posea varias firmas, supongamos cinco, y se sombrean dos con Shadows, estas serían las que se ven y las otras tres son las que se ocultan.

En el ejemplo utilizado se puede observar que en la clase `TextoEspecial` se sombra solo una de las sobrecargas y se implementa un código de decoración diferente.

```

Public Class Texto
    Public Overridable Function Decora(Origen As String)
        Return Origen.ToUpper
    End Function
    Public Overridable Function Decora(Origen As String, Separador As Char)
        'Intercala el separador entre cada carácter
        Dim Cadena() As Char = Origen.ToCharArray
        Dim Resultado As String
        For Each C As Char In Cadena
            Resultado += C & Separador
        Next
        Return Resultado.Substring(0, Resultado.Length - 1)
    End Function
End Class
Public Class TextoEspecial
    Inherits Texto
    Public Shadows Function Decora(Origen As String, Separador As Char)
        'El separador se coloca al principio y al final del texto
        Dim Cadena() As Char = Origen.ToCharArray
        Return Separador & " " & Origen & " " & Separador
    End Function
End Class

```

En la figura 5.5 se puede observar una instancia de `Texto` denominada `T`. Al intentar utilizar el método `Decora`, se observa que la sobrecarga del método es claramente 2.



```

Private Sub Form1_Load(sender As System.Object, e As System.EventArgs) Handles MyBase.Load
    Dim T As New Texto
    Dim TT As New TextoEspecial

    T.Decora()

```

Figura 5.5.

Al utilizar el mismo método pero desde `TT`, que es una instancia de `TextoEspecial`, no se observa la sobrecarga debido al efecto que causa `Shadows`, como se ve en la figura 5.6.

```

Private Sub Form1_Load(sender As System.Object, e As System.EventArgs) Handles MyBase.Load
    Dim T As New Texto
    Dim TT As New TextoEspecial

    TT.Decora(
        Decorar(Origen As String, Separador As Char) As Object
)

```

Figura 5.6.

5.4.1.4. Polimorfismo

Uno de los mecanismos más poderosos en la orientación a objetos es el polimorfismo. En realidad, lo que es polimórfico es el comportamiento de los métodos en ciertas circunstancias que se expondrán a continuación. Cuando una o más subclases heredan de una superclase uno o más métodos sobrescribibles o virtuales a ser implementados, puede ocurrir que la implementación que se realice en cada subclase sea distinta. Este hecho ocasionará que el comportamiento de ese método en cada subclase sea distinto a pesar de que el método en cada subclase posee el mismo nombre. Es por ello que en general, este concepto se define como: “la capacidad de que un método heredado posea distinto comportamiento en las subclases que lo implementan”. Esto le da al modelo mucha flexibilidad y al mismo tiempo permite abstraerse de qué subclase ejecutará el método, ya que este se denomina igual en cada una de ellas pero el resultado obtenido será especializado pues está definido de distinta manera en cada subclase que lo heredó. El ejemplo esquemático de la figura 5.7 permite comprender mejor el concepto. En una jerarquía del tipo “es-un” (herencia) entre Animal como concepto abstracto y Ave, Terrestre y Pez como especializaciones de Animal, si poseemos un método Desplazamiento heredado desde Animal hacia las tres subclases, la implementación de este método en cada una de ellas deberá ser distinto, ya que cada tipo de animal, representado por cada subclase, se desplaza de una forma distinta.

El código de la página siguiente muestra como las tres subclases Ave, Terrestre y Pez poseen una implementación diferente, cada una adaptada a las necesidades de la subclase específica. En el ejemplo se observa que la diferencia solo está dada por mostrar una leyenda distintiva, pero en realidad podrían ser procesos mucho más complejos y diferentes.

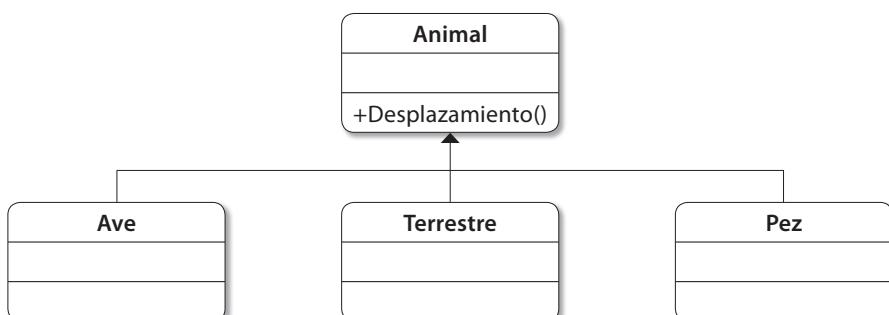


Figura 5.7.

Luego, al utilizar estas subclases, se manifiesta claramente que al iterar el `For ... Each` se independiza de cual es el elemento que obtiene en `T`. Esto permite que se envíe el mensaje `T.Desplazamiento` y el mismo funcionará acorde a la implementación que posea el objeto (`Ave`, `Terrestre` y `Pez`) que recibió `T`. Queda garantizado que el mensaje no generará una excepción ya que `T` es una variable de tipo `Animal`, lo que provocará que cualquiera sea el elemento que se reciba, será de ese tipo o cualquier otro tipo derivado de él (`Ave`, `Terrestre` o `Pez`).

```
Public Class Form1
    Private Sub Button1_Click _
        (sender As System.Object, e As System.EventArgs) Handles Button1.Click
        Dim A As New Ave
        Dim C As New Terrestre
        Dim P As New Pez
        Dim L As New List(Of Animal)
        L.AddRange({A, C, P})
        For Each T As Animal In L
            T.Desplazamiento()
        Next
    End Sub
End Class
Public Class Animal
    Public Overridable Sub Desplazamiento()
        MsgBox("Desplazamiento de un Animal")
    End Sub
End Class
Public Class Ave
    Inherits Animal
    Public Overrides Sub Desplazamiento()
        MsgBox("Desplazamiento de un Ave")
    End Sub
End Class
Public Class Terrestre
    Inherits Animal
    Public Overrides Sub Desplazamiento()
        MsgBox("Desplazamiento de un Terrestre")
    End Sub
End Class
Public Class Pez
    Inherits Animal
    Public Overrides Sub Desplazamiento()
        MsgBox("Desplazamiento de un Pez")
    End Sub
End Class
```

A continuación, se expondrá el código afectado del mismo ejemplo pero suponiendo que `Animal` es una clase abstracta y `Desplazamiento` un método virtual.

```
Public MustInherit Class Animal
    Public MustOverride Sub Desplazamiento()
End Class
```

Un aspecto importante a destacar es que cuando se utiliza correctamente el polimorfismo, se pueden desestructurar las formas de decisiones múltiples (`If ... ElseIf ... Elseif ... Else ... End If o Select Case ... Case1 ... Case2 ... End Select`).

Para analizar en detalle lo expuesto, se tomará un escenario donde se poseen dos tipos de cuentas bancarias denominadas `CuentaDeAhorro` y `CuentaCorriente`. Cada una está identificada por un número y las operaciones que pueden realizar son `Deposito`, `Extraccion` y `Transferencia`. La diferencia entre una `CuentaDeAhorro` y una `CuentaCorriente` es que las segundas admiten un giro en descubierto cuyo monto queda establecido en la propiedad `Descubierto` de la cuenta.

La clase `Cuenta` es abstracta y define las propiedades `Número` y `Saldo`, que son el número y el saldo de la cuenta, respectivamente. También posee dos procedimientos (los métodos `Deposito` y `Transferencia`) y una función (el método virtual `Extraccion`) que retorna un valor booleano verdadero si la misma se ha realizado con éxito y falso en caso contrario. Este método virtual es el que se sobrescribe en las subclases `CuentaDeAhorro` y `CuentaCorriente`, ya que en el tratamiento (contemplando el descubierto o no) que se le da a la extracción de dinero está la esencia que diferencia a un tipo de cuenta de la otra. Este método es polimórfico pues la forma en la que se sobrescribe e implementa en las subclases es distinta. En su implementación, `CajaDeAhorro` solo chequea que el saldo sea igual o superior a cero, mientras que `CuentaCorriente` observa que el saldo más el descubierto con posterioridad a la extracción de dinero que se realiza sea superior o igual a cero.

Aquí queda de manifiesto que, al momento de utilizar estas implementaciones, no importa qué tipo de cuenta sea, ya que al enviarle el mensaje `Extraccion`, que posee el mismo nombre en ambas ya que viene heredado de la superclase, cada una establecerá las validaciones que correspondan según su tipo.

```
Public MustInherit Class Cuenta
    Private Vnumero As String
    Public Property Número() As String
        Get
            Return Vnumero
        End Get
        Set(ByVal value As String)
            Vnumero = value
        End Set
    End Property
    Private Vsaldo As Decimal
    Public Property Saldo() As Decimal
        Get
            Return Vsaldo
        End Get
```

```

        Set(ByVal value As Decimal)
            Vsaldo = value
        End Set
    End Property

    Public Sub Deposito(ByVal pMonto As Decimal)
        Me.Saldo += pMonto
    End Sub
    Public MustOverride Function Extraccion(ByVal pMonto As Decimal) As Boolean
    Public Sub Transferencia _
        (ByVal pMonto As Decimal, ByVal pCuentaDestino As Cuenta)
        If Me.Extracción(pMonto) Then pCuentaDestino.Deposito(pMonto)
    End Sub
End Class

```

```

Public Class CuentaDeAhorro
    Inherits Cuenta
    Sub New()
        Me.Saldo = 0
    End Sub
    Sub New(ByVal pSaldoInicial As Decimal)
        Me.New()
        Me.Saldo = pSaldoInicial
    End Sub
    Public Overrides Function Extraccion(pMonto As Decimal) As Boolean
        Dim R As Boolean = False
        If Me.Saldo - pMonto >= 0 Then Me.Saldo -= pMonto : R = True
        Return R
    End Function
End Class

```

```

Public Class CuentaCorriente
    Inherits Cuenta
    Sub New()
        Me.Saldo = 0
    End Sub
    Sub New(ByVal pSaldoInicial As Decimal)
        Me.New()
        Me.Saldo = pSaldoInicial
    End Sub
    Private Vdescubierto As Decimal
    Public Property Descubierto() As Decimal
        Get
            Return Vdescubierto
        End Get
    End Property

```

```

    Set(ByVal value As Decimal)
        Vdescubierto = value
    End Set
End Property
Public Overrides Function Extraccion(pMonto As Decimal) As Boolean
    Dim R As Boolean = False
    If Me.Saldo + Me.Descubierto - pMonto >= 0 _
        Then Me.Saldo -= pMonto : R = True
    Return R
End Function
End Class

```

5.4.2. Agregación

Este tipo de relación permite que, conjuntamente, dos o más objetos representen un concepto donde cada uno de ellos forma parte del mismo. Esta relación es derivada del tipo de relación jerárquica “todo-parte”. En general, una clase constituye la noción del concepto que representa al todo, pero necesita de las partes para obtener el detalle estructural y de comportamiento que cada uno de estas le provee, entonces las *agrega*. Por ejemplo, conceptualmente podríamos pensar en la noción de auto y comprender que un Auto posee partes como el Motor, la Carrocería y el Chasis. Para construir esto, existirá una clase Auto que en su definición tendrá los elementos para poder tener un Motor, un Chasis y una Carrocería. No obstante, los detalles de implementación referidos a cómo un Motor se enciende o se apaga, el peso de la Carrocería o el largo y el ancho del Chasis están definidos en cada una de las clases que representan las partes del Auto.

La *agregación* como relación se genera a nivel de clases y se ve claramente en un diagrama de clases, esto nos daría una visión estática de la relación. Al utilizar esas clases e instanciarlas, obtendremos objetos que se corresponden de la misma manera que lo hacen las clases que les dieron origen.

Existen dos formas de poder llevar adelante este concepto. La primera es que las partes se agreguen haciendo referencias a ellas. Cuando esto ocurre, en general los ciclos de vida del objeto que agrega y los ciclos de vida de los objetos agregados no necesariamente son dependientes o están sincronizados. Los ciclos de vida de los objetos son dependientes o están sincronizados cuando la creación del objeto que agrega implica la creación de los objetos agregados y la eliminación del objeto que agrega implica la eliminación de los objetos agregados. La segunda forma propone que las partes sean creadas por el objeto que agrega en el momento en el que él es creado y cuando finalice su ciclo de vida (y por lo tanto sea eliminado), él sea el encargado de finalizar los ciclos de vida de los objetos que creó y agregó. Esta última forma tiene sentido solo cuando las partes existen para conformar el todo y sin la existencia de este no se justifica la existencia de las partes.

En la práctica, a la primera forma mencionada se la denomina *agregación simple* (o *agregación* solamente), mientras que a la segunda se la llama *agregación con contención física o composición*.

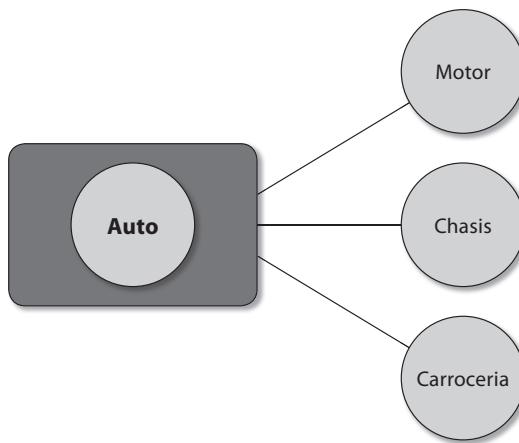


Figura 5.8. Esquema de agregación.



Figura 5.9. Esquema de composición.

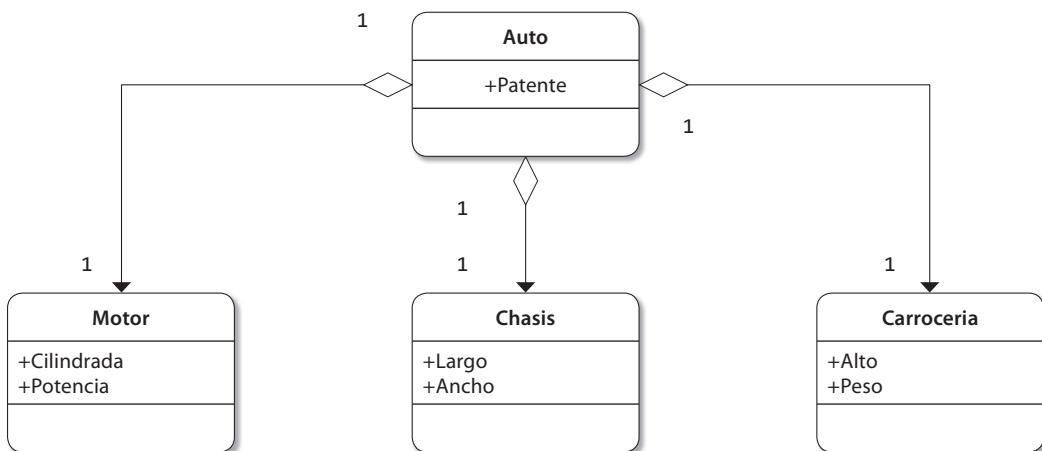


Figura 5.10. Ejemplo de agregación.

Para el siguiente ejemplo utilizaremos las clases Auto, Motor, Chasis y Carrocería. La clase Auto posee las propiedades: Patente, UnMotor, UnChasis, UnaCarrocería. Se puede observar que

UnMotor es de tipo Motor, UnChasis es de tipo Chasis, UnaCarroceria es de tipo Carroceria. Esto permitirá tener referencias a un Motor, un Chasis y una Carroceria y construir la agregación que estamos exemplificando.

```
Public Class Auto
    Sub New()
        End Sub
        Sub New(pPatente As String)
            Me.Patente = pPatente
        End Sub
        Private Vpatente As String
        Public Property Patente() As String
            Get
                Return Vpatente
            End Get
            Set(ByVal value As String)
                Vpatente = value
            End Set
        End Property
        Private Vmotor As Motor
        Public Property UnMotor() As Motor
            Get
                Return Vmotor
            End Get
            Set(ByVal value As Motor)
                Vmotor = value
            End Set
        End Property
        Private Vchasis As Chasis
        Public Property UnChasis() As Chasis
            Get
                Return Vchasis
            End Get
            Set(ByVal value As Chasis)
                Vchasis = value
            End Set
        End Property
        Private Vcarroceria As Carroceria
        Public Property UnaCarroceria() As Carroceria
            Get
                Return Vcarroceria
            End Get
            Set(ByVal value As Carroceria)
                Vcarroceria = value
            End Set
        End Property
    End Class
```

La clase Motor posee dos características que administra por medio de las propiedades Cilindrada y Potencia.

```
Public Class Motor
    Sub New()
    End Sub
    Sub New(pCilindrada As Double, pPotencia As Double)
        Me.Cilindrada = pCilindrada
        Me.Potencia = pPotencia
    End Sub
    Private Vcilindrada As Double
    Public Property Cilindrada() As Double
        Get
            Return Vcilindrada
        End Get
        Set(ByVal value As Double)
            Vcilindrada = value
        End Set
    End Property
    Private Vpotencia As Double
    Public Property Potencia() As Double
        Get
            Return Vpotencia
        End Get
        Set(ByVal value As Double)
            Vpotencia = value
        End Set
    End Property
End Class
```

```
Public Class Chasis
    Sub New()
    End Sub
    Sub New(pLargo As Double, pAncho As Double)
        Me.Largo = pLargo
        Me.Ancho = pAncho
    End Sub
    Private Vlargo As Double
    Public Property Largo() As Double
        Get
            Return Vlargo
        End Get
        Set(ByVal value As Double)
            Vlargo = value
        End Set
    End Property
End Class
```

```
        End Set
    End Property
    Private Vancho As Double
    Public Property Ancho() As Double
        Get
            Return Vancho
        End Get
        Set(ByVal value As Double)
            Vancho = value
        End Set
    End Property
End Class
```

```
Public Class Carroceria
    Sub New()
    End Sub
    Sub New(pAlto As Double, pPeso As Double)
        Me.Alto = pAlto
        Me.Peso = pPeso
    End Sub
    Private Valto As Double
    Public Property Alto() As Double
        Get
            Return Valto
        End Get
        Set(ByVal value As Double)
            Valto = value
        End Set
    End Property
    Private Vpeso As Double
    Public Property Peso() As Double
        Get
            Return Vpeso
        End Get
        Set(ByVal value As Double)
            Vpeso = value
        End Set
    End Property
End Class
```

La clase Chasis posee también dos características que administra con las propiedades Largo y Ancho, mientras que Carroceria se caracteriza por tener las propiedades Alto y Peso.

En el siguiente código se puede observar cómo se construye la agregación. Tenemos tres objetos: un Motor, un Chasis y una Carroceria, cada uno apuntados por las variables MO, CH y

CA. También un objeto Auto apuntado por la variable AA. El Auto en sus propiedades UnMotor, UnChasis y UnaCarrocería se le asignaron (están apuntando) a un Motor (M0), un Chasis (CH) y una Carrocería (CA). La agregación ha permitido que Auto esté completo con todas sus partes y generando la posibilidad de acceder (navegar) a las características que estas poseen. Es típico de la agregación navegar desde el todo a las partes. En el código del ejemplo, esto queda a la vista al observar instrucciones del tipo AA.UnMotor.Cilindrada, ya que se está navegando desde el todo AA hacia una de las partes UnMotor y allí se solicita una característica de esta, Cilindrada. Si se obtienen todas las características de las partes que componen un auto y se muestran, se visualizará algo como en el formulario de la figura 5.11.



Figura 5.11.

Los objetos **Motor**, **Chasis** y **Carrocería** existen con anterioridad al objeto **Auto** y probablemente sigan existiendo con posterioridad a la extinción de este objeto, característica esencial de la agregación simple.

Tomaremos el ejemplo anterior y lo forzaremos para generar una agregación con contención física.

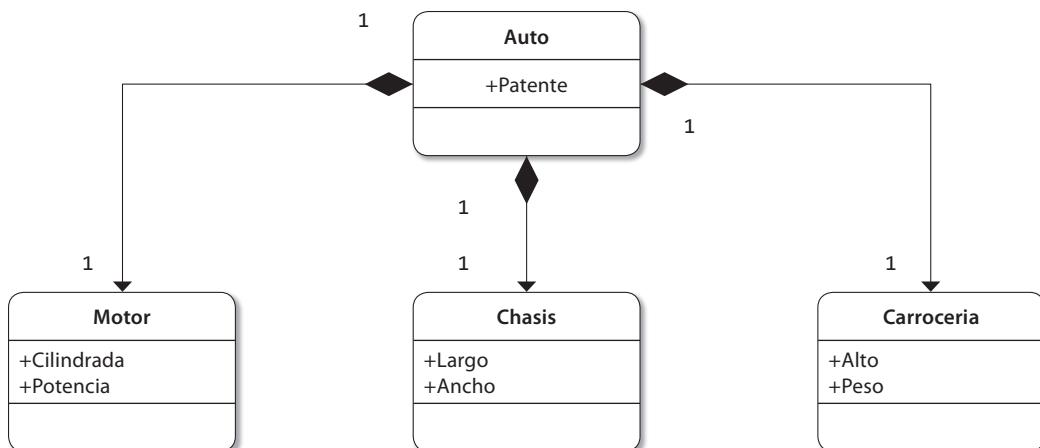


Figura 5.12. Ejemplo de Agregación con contención física o composición.

Suponiendo y tomando el ejemplo anterior, si se determina que no tiene sentido la existencia del Motor, el Chasis y la Carrocería salvo que exista un Auto, entonces deberíamos relacionar los ciclos de vida del Auto con las partes. Para ello, debemos modificar la clase Auto. La clase Auto en su constructor instancia a los objetos que conforman parte de él (Motor, Chasis y Carrocería). El procedimiento **Finalize** oficia de destructor de la clase y será lo último que se ejecute antes de que el objeto deje de existir. En él se puede observar cómo se rompe la referencia al Motor, el Chasis y la Carrocería. Cabe aclarar que lo mismo se podría haber colocado en el procedimiento **Dispose**, ya que este es ejecutado por el programador y de esta forma no se debe esperar al Garbage Collector para que se ejecuten esas líneas de código, ya que es él quien ejecuta el **Finalize**.

Otro aspecto observable al colocar el código en el **Finalize** es que si no lo hubiéramos colocado, igual las referencias a los objetos Motor, Chasis y Carrocería se perderían al finalizar la existencia del objeto Auto y estos quedarían como basura en memoria. La diferencia fundamental de colocar este código en el **Dispose** o en el **Finalize** es poder manejar el momento en el que se liberan los objetos que ofician como partes de Auto.

```
Public Class Auto
    Dim MO As Motor
    Dim CH As Chasis
    Dim CA As Carroceria
    Sub New(pMotorCilindrada As Double, pMotorPotencia As Double, _
        pChasisLargo As Double, pChasisAncho As Double, _
        pCarroceriaAlto As Double, pCarroceriaPeso As Double)
        MO = New Motor(pMotorCilindrada, pMotorPotencia)
        CH = New Chasis(pChasisLargo, pChasisAncho)
        CA = New Carroceria(pCarroceriaAlto, pCarroceriaPeso)
    End Sub
    Sub New(pPatente As String, pMotorCilindrada As Double, _
        pMotorPotencia As Double, pChasisLargo As Double, _
        pChasisAncho As Double, pCarroceriaAlto As Double, _
        pCarroceriaPeso As Double)
        Me.new(pMotorCilindrada, pMotorPotencia, pChasisLargo, _
            pChasisAncho, pCarroceriaAlto, pCarroceriaPeso)
        Me.Patente = pPatente
    End Sub
    Private Vpatente As String
    Public Property Patente() As String
        Get
            Return Vpatente
        End Get
        Set(ByVal value As String)
            Vpatente = value
        End Set
    End Property
    Private Vmotor As Motor
    Public Property UnMotor() As Motor
```

```

Get
    Return Vmotor
End Get
Set(ByVal value As Motor)
    Vmotor = value
End Set
End Property
Private Vchasis As Chasis
Public Property UnChasis() As Chasis
    Get
        Return Vchasis
    End Get
    Set(ByVal value As Chasis)
        Vchasis = value
    End Set
End Property
Private Vcarroceria As Carroceria
Public Property UnaCarroceria() As Carroceria
    Get
        Return Vcarroceria
    End Get
    Set(ByVal value As Carroceria)
        Vcarroceria = value
    End Set
End Property
Protected Overrides Sub Finalize()
    MO = Nothing
    CH = Nothing
    CA = Nothing
End Sub
End Class

```

El uso de Auto quedaría como sigue:

```

Public Class Form1
    Private Sub Form1_Load(sender As System.Object, e As System.EventArgs) _
        Handles MyBase.Load
        Dim AA As New Auto("JGK-821", 2000, 150, 3150, 1750, 1500, 1700)
        MsgBox("Patente: " & vbCrLf & AA.Patente & vbCrLf & _
            "Motor Cilindrada: " & vbCrLf & AA.UnMotor.Cilindrada & vbCrLf & _
            "Motor Potencia: " & vbCrLf & AA.UnMotor.Potencia & vbCrLf & _
            "Chasis Largo: " & vbCrLf & AA.UnChasis.Largo & vbCrLf & _
            "Chasis Ancho: " & vbCrLf & AA.UnChasis.Ancho & vbCrLf & _
            "Carrocería Alto: " & vbCrLf & AA.UnaCarroceria.Alto & vbCrLf & _
            "Carrocería Peso: " & vbCrLf & AA.UnaCarroceria.Peso & vbCrLf)
    End Sub
End Class

```

5.4.3. Asociación

La *asociación* es una relación entre clases que permite asociar las instancias (objetos) que colaboran entre sí. Esta no es una relación jerárquica, no hay dependencia de los ciclos de vida del objeto que solicita la colaboración respecto de quien la brinda.

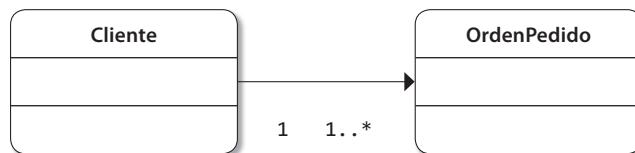


Figura 5.13.

En el ejemplo de la figura 5.13 se establece una *cardinalidad* que denota que un cliente puede tener 1 o muchas órdenes de pedido. La cardinalidad puede ser de:

- 1 a 1
- 1 a muchos
- Muchos a muchos

La flecha indica sentido de navegabilidad. En este caso, dado un cliente, puedo conoce sus órdenes de pedido.

```
Public Class Cliente
    Sub New()
        End Sub
        Sub New(pCodigo As String, pDescripcion As String)
            Me.Codigo = pCodigo
            Me.Descripcion = pDescripcion
        End Sub
        Private Vcodigo As String
        Public Property Codigo() As String
            Get
                Return Vcodigo
            End Get
            Set(ByVal value As String)
                Vcodigo = value
            End Set
        End Property
        Private Vdescripcion As String
        Public Property Descripcion() As String
            Get
                Return Vdescripcion
            End Get
            Set(ByVal value As String)
                Vdescripcion = value
            End Set
        End Property
    End Class
```

```

        End Set
    End Property
    Private VlistaOrdenPedido As New List(Of OrdenPedido)
    Public ReadOnly Property ListaOrdenPedido() As List(Of OrdenPedido)
        Get
            Return VlistaOrdenPedido
        End Get
    End Property
    Public Sub CargaOrdenPedido(pOrdenPedido As OrdenPedido)
        VlistaOrdenPedido.Add(pOrdenPedido)
    End Sub
End Class

```

En la clase `Cliente` se puede observar que posee una lista de órdenes de pedidos, la carga se maneja por medio del método `CargaOrdenPedido` y la lectura por medio de la propiedad `ListaOrdenPedido`.

Por su lado, las órdenes de pedido no conforman alguna parte de un cliente sino que simplemente interactúan con él, y por ello la relación preexistente no es una agregación. Tampoco le está prestando un servicio, por lo que no es una relación de uso (este tema se analiza en el siguiente punto). Simplemente se relacionan (*asocian*) por la semántica que existe entre ellas en el sentido de que un `Cliente` puede tener una o muchas órdenes de compras que le corresponden.

```

Public Class OrdenPedido
    Sub New()
    End Sub
    Sub New(pNumero As Integer, pDetalle As String)
        Me.Numero = pNumero
        Me.Detalle = pDetalle
    End Sub
    Private Vnumero As Integer
    Public Property Numero() As Integer
        Get
            Return Vnumero
        End Get
        Set(ByVal value As Integer)
            Vnumero = value
        End Set
    End Property
    Private Vdetalle As String
    Public Property Detalle() As String
        Get
            Return Vdetalle
        End Get
        Set(ByVal value As String)
            Vdetalle = value
        End Set
    End Property
End Class

```

El siguiente código permite ver cómo se utiliza la asociación generada entre Cliente y OrdenPedido contemplando que existe navegabilidad desde Cliente hacia OrdenPedido.

```
Public Class Form1
    Private Sub Form1_Load(sender As System.Object, e As System.EventArgs) _
        Handles MyBase.Load
        Dim C As New Cliente("C001", "Acjor S.A.")
        Dim OP1 As New OrdenPedido(1, "1 Máquina ZAD 1001")
        Dim OP2 As New OrdenPedido(2, "3 Empaquetadoras E233")
        Dim OP3 As New OrdenPedido(3, "2 Cintas Z2233")
        C.CargaOrdenPedido(OP1)
        C.CargaOrdenPedido(OP2)
        C.CargaOrdenPedido(OP3)

        Dim S As String = "Cliente: " & C.Codigo & " - " & C.Descripcion _
            & vbCrLf & vbCrLf
        S += "Ordenes de Pedido:" & vbCrLf & vbCrLf
        For Each OP As OrdenPedido In C.ListaOrdenPedido
            S += OP.Numero & " " & OP.Detalle & vbCrLf
        Next
        MsgBox(S)
    End Sub
End Class
```

5.4.4. Uso

La relación de uso se da cuando se establece una clara situación en la que una clase está haciendo uso de otra. Hacer uso de una clase se debe entender como que debe solicitar algo para resolver un comportamiento o cambio de estado propio. En una relación de uso, una clase no contiene a la otra.

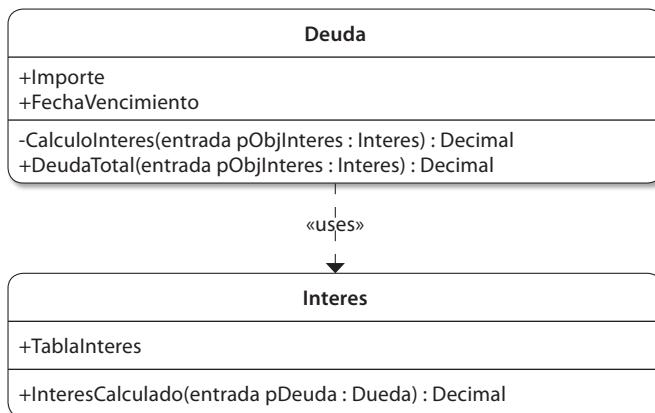


Figura 5.14.

Por ejemplo, si la clase `Deuda` necesita colaboración de la clase `Interes` para calcular los intereses de una deuda, podríamos establecer una colaboración entre ambas donde `Deuda` usa `Interes`.

En la práctica, se vería de la siguiente manera:

```
Public Class Deuda
    Private Vimpor As Decimal
    Public Property Importe() As Decimal
        Get
            Return Vimpor
        End Get
        Set(ByVal value As Decimal)
            Vimpor = value
        End Set
    End Property
    Private VfechaVencimiento As Date
    Public Property FechaVencimiento() As Date
        Get
            Return VfechaVencimiento
        End Get
        Set(ByVal value As Date)
            VfechaVencimiento = value
        End Set
    End Property
    Public Function DeudaTotal(ByRef pObjInteres As Interes) As Decimal
        Return Vimpor + pObjInteres.InteresCalculado(Me)
    End Function
End Class
```

```
Public Class Form1
    Private Sub Form1_Load(sender As System.Object, e As System.EventArgs) _
        Handles MyBase.Load
        Dim HT As New Dictionary(Of Integer, Double)
        'Carga de Días y porcentajes de interés
        HT.Add(30, 0.1)
        HT.Add(60, 0.2)
        HT.Add(90, 0.3)
        Dim I As New Interes
        I.TablaInteres = HT
        Dim D As New Deuda
        D.Importe = 10000
        D.FechaVencimiento = #1/6/2012#
        MsgBox("Deuda Original:" & vbTab & D.Importe & vbCrLf & _
            "Deuda Total:" & vbTab & D.DeudaTotal(I))
    End Sub
End Class
```

```

Public Class Interes
    Private VtablaInteres As New Dictionary(Of Integer, Double)
    Public Property TablaInteres() As Dictionary(Of Integer, Double)
        Get
            Return VtablaInteres
        End Get
        Set(ByVal value As Dictionary(Of Integer, Double))
            VtablaInteres = value
        End Set
    End Property
    Public Function InteresCalculado(pDeuda As Deuda) As Decimal
        Dim Dias As Integer
        Dim Interes As Double
        For Each X As KeyValuePair(Of Integer, Double) In Me.TablaInteres
            Dias = (Now - pDeuda.FechaVencimiento).Days
            Interes = X.Value * pDeuda.Importe
            If X.Key > Dias Then Exit For
        Next
        Return Interes
    End Function
End Class

```

La clase `Deuda` prevé poder calcular la `DeudaTotal`, que está compuesta por la deuda original más los intereses. Para el cálculo de los intereses, la clase `Deuda` necesita la colaboración de la clase `Interes` ya que es ella quien sabe calcular los intereses y cuánto hay que cobrar en función de los días de retraso. En este ejemplo particular, el método `DeudaTotal` de la clase `Deuda` posee un parámetro denominado `pObjInteres` del tipo `Interes`, que permite que se reciba una referencia y utilizar el objeto referenciado para enviarse ella misma (`ObjInteres.InteresCalculado(Me)`, `Me` representa una instancia de `Deuda`), pues su estado posee el importe (`Importe`) y la fecha de vencimiento (`FechaVencimiento`) necesarias para el cálculo.

5.5. Interfaces

Una interfaz es un elemento que nos brinda la posibilidad de que bajo un nombre (el de la interfaz), se puedan definir las firmas de un conjunto de miembros (procedimientos, propiedades, métodos, etcétera).

Luego, estas interfaces pueden ser implementadas en clases (también lo pueden hacer las estructuras). Se denomina *firma* al conjunto que conforma el nombre del miembro más los parámetros de entrada y sus tipos. En una misma clase no pueden existir miembros con igual firma. Un aspecto muy importante de las interfaces es que “tipan”. Cuando una clase o estructura implementa una interfaz, además de tener el tipo que crean por existir, también tendrán el tipo de la interfaz. Esto permitirá construir lo que denominamos “polimorfismo con interfaces”, que funcionalmente es igual al polimorfismo que se analizó al momento de abordar la herencia pero

con mayor flexibilidad, ya que al no estar atado a este tipo de relación (herencia), este polimorfismo puede ser transversal a varias jerarquías del tipo “es-un” generadas por herencia. Este punto en particular se analizará más adelante. Se deberá tener en cuenta que las interfaces se definen dentro de un espacio de nombres o un módulo.

Las interfaces establecen una clara separación de la definición de miembros respecto a la implementación de estos, lo que permite pensar diseños de software más flexibles. Es importante resaltar que cuando se utiliza la herencia como mecanismo para aprovechar/reutilizar el código ya definido y/o implementado en clases más abstractas, estamos restringiendo esa posibilidad a una clase, pues el entorno tecnológico en el cual se están construyendo los ejemplos soporta solo herencia simple. Las interfaces no poseen esta restricción y una clase puede implementar tantas interfaces como necesite.

Las interfaces pueden heredarse entre ellas. Una interfaz puede heredar de una interfaz o de más de una, lo que representa un aspecto muy interesante en cuanto a la flexibilidad que le otorga al programador. También las interfaces admiten anidamiento.

En los siguientes párrafos se detallará cómo crear interfaces personalizadas y se exemplificará el “polimorfismo por interfaces”.

Las interfaces se construyen con **Interface ... End Interface**. Los nombres de las interfaces, como buena práctica de programación, comienzan con “I”, por ejemplo, **IFigura**.

Ejemplo de construcción de una interfaz.

```
Interface IFigura
    Function Perimetro(pObj As List(Of Double)) As Double
    Function Superficie(pObj As List(Of Double)) As Double
End Interface
```

La interfaz **IFigura** define dos métodos que en particular, son funciones para el cálculo del **Perímetro** y la **Superficie**. Ambas poseen un parámetro (**pObj**) que es una lista de elementos **Double** para obtener los valores necesarios para el cálculo, dependiendo de la figura. El resultado lo retorna como valor **Double**.

Ejemplo de implementación de una interfaz:

```
Public Class Cuadrado
    Implements IFigura
    ''' <summary>
    ''' Determina el perímetro de un Cuadrado
    ''' </summary>
    ''' <param name="pObj">Requerido. Debe llevar 4 ítems que representan
    ''' cada uno de los lados del cuadrado</param>
    ''' <returns>El perímetro del cuadrado</returns>
    ''' <remarks></remarks>
```

```

    Public Function Perimetro(pObj As System.Collections.Generic.List(Of Double)) _
        As Double Implements IFigura.Perimetro
        Return pObj.Item(0) + pObj.Item(1) + pObj.Item(2) + pObj.Item(3)
    End Function
    ''' <summary>
    ''' Determina la superficie de un Cuadrado
    ''' </summary>
    ''' <param name="pObj">Requerido. Debe llevar 1 ítems que representa un
    ''' lado del cuadrado</param>
    ''' <returns>La superficie del cuadrado</returns>
    ''' <remarks></remarks>
    Public Function Superficie(pObj As System.Collections.Generic.List(Of Double)) _
        As Double Implements IFigura.Superficie
        Return pObj(0) * 2
    End Function
End Class

```

La implementación de una interfaz obliga a quien la implementa, en este caso una clase, a que implemente los miembros que se han definido en ella. En nuestro ejemplo, las dos funciones fueron implementadas y se puede observar la particularidad de que al final de las firmas de las mismas aparecen `Implements IFigura.Perimetro` e `Implements IFigura.Superficie` para el perímetro y la superficie. Esto permite identificar rápida y fácilmente en qué interfaz está definida la firma correspondiente a la implementación que se está realizando. Gracias a esta característica, más adelante podremos ver cómo dos miembros provenientes de distintas interfaces pueden tener el mismo nombre.

Ejemplo de más de una interfaz:

```

Interface IUnidades
    Property UnidadPerimetro As String
    Property UnidadSuperficie As String
    Function MuestraPerimetroSuperficie(pObj As System.Collections.Generic. _
        List(Of Double)) As String
End Interface

```

Para ejemplificar este punto, se agregará otra interfaz `IUnidades` que define los miembros que permitirán administrar las unidades para el perímetro (`UnidadPerimetro`) y la superficie (`UnidadSuperficie`), así como el retorno de un `String` (`MuestraPerimetroSuperficie`) con la información conjunta del perímetro y la superficie.

```

Public Class Cuadrado
    Implements IFigura

```

```

Implements IUnidades
Sub New()
End Sub
''' <summary>
'''
''' </summary>
''' <param name="pUnidadPerimetro">Requerido. Se coloca la unidad en que
''' se expresa el perímetro.</param>
''' <param name="pUnidadSuperficie">Requerido. Se coloca la unidad en que
''' se expresa la superficie.</param>
''' <remarks></remarks>
Sub New(pUnidadPerimetro As String, pUnidadSuperficie As String)
    Me.UnidadPerimetro = pUnidadPerimetro
    Me.UnidadSuperficie = pUnidadSuperficie
End Sub
''' <summary>
''' Determina el perímetro de un Cuadrado
''' </summary>
''' <param name="pObj">Requerido. Debe llevar 4 ítems que
''' representan cada uno de los lados del cuadrado</param>
''' <returns>El perímetro del cuadrado</returns>
''' <remarks></remarks>
Public Function Perimetro(pObj As System.Collections.Generic. _
                           List(Of Double)) As Double Implements IFigura.Perimetro
    Return pObj.Item(0) + pObj.Item(1) + pObj.Item(2) + pObj.Item(3)
End Function
''' <summary>
''' Determina la superficie de un Cuadrado
''' </summary>
''' <param name="pObj">Requerido. Debe llevar 1 ítem que representa un
''' lado del cuadrado</param>
''' <returns>La superficie del cuadrado</returns>
''' <remarks></remarks>
Public Function Superficie(pObj As System.Collections.Generic. _
                           List(Of Double)) As Double Implements IFigura.Superficie
    Return pObj(0) * 2
End Function
Public Function MuestraPerimetroSuperficie(pObj As _
                                             System.Collections.Generic.List(Of Double)) As String Implements _
                                             IUnidades.MuestraPerimetroSuperficie
    Return "Cuadrado" & vbCrLf & _
           "Medidas: " & pObj(0) & " x " & pObj(1) & " x " & pObj(2) & _
           " x " & pObj(3) & vbCrLf & "Perímetro: " & Me.Perimetro(pObj) & _
           " " & Me.UnidadPerimetro & vbCrLf & "Superficie: " & _
           Me.Superficie(pObj) & " " & Me.UnidadSuperficie
End Function
Dim VunidadPerimetro As String
Public Property UnidadPerimetro As String Implements IUnidades.UnidadPerimetro
    Get
        Return VunidadPerimetro
    End Get

```

```

        Set(value As String)
            VUnidadPerimetro = value
        End Set
    End Property
    Dim VUnidadSuperficie As String
    Public Property UnidadSuperficie As String Implements IUnidades.UnidadSuperficie
        Get
            Return VUnidadSuperficie
        End Get
        Set(value As String)
            VUnidadSuperficie = value
        End Set
    End Property
End Class

```

La clase Cuadrado ahora implementa dos interfaces: `IFigura` e `IUnidades`. Lo referido a la primera interface se analizó en el ejemplo anterior. La interface `IUnidades` hace que se implemente en la clase Cuadrado la posibilidad de cargar en qué unidad se mide el perímetro y la superficie del mismo, además de formatear un `String` para retornar la información calculada. De esta forma, la clase Cuadrado podría implementar muchas interfaces y colocarle a los miembros que estas definen el código que un cuadrado necesita para la ejecución de cada uno de esos miembros.

Ejemplo de herencia de interfaz:

```

Interface IOperacionesPrimerNivel
    Function Suma(N1 As Double, N2 As Double) As Double
    Function Resta(N1 As Double, N2 As Double) As Double
    Function Producto(N1 As Double, N2 As Double) As Double
    ''' <summary>
    '''
    ''' </summary>
    ''' <param name="N1">Requerido. Dividendo</param>
    ''' <param name="N2">Requerido. Divisor, debe ser distinto de 0 (cero)</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Function Cociente(N1 As Double, N2 As Double) As Double
End Interface
Interface IOperacionesSegundoNivel
    Inherits IOperacionesPrimerNivel
    ''' <summary>
    ''' Potencia de N1 elevado a la N2
    ''' </summary>
    ''' <param name="N1">Requerido. Base</param>
    ''' <param name="N2">Requerido. Exponente</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Function Potencia(N1 As Double, N2 As Double) As Double
End Interface

```

```

    <><summary>
    <></summary>
    <><param name="N1">Requerido. Radicando</param>
    <><param name="N2">Radicando. Índice</param>
    <><returns></returns>
    <><remarks></remarks>
Function Raiz(N1 As Double, N2 As Double) As Double
End Interface

```

La interfaz `IOperacionesSegundoNivel` hereda de `IOperacionesPrimerNivel` todas las definiciones que esta posee. Para realizarlo, se utiliza la palabra `Inherits`.

La clase `Operaciones` implementa la interfaz `IOperacionesSegundoNivel`, lo que provoca que tenga que implementar los miembros que esta interfaz define más lo que heredó de `IOperacionesPrimerNivel`. Para distinguir si un miembro implementado proviene de una u otra interfaz, se puede observar lo que dice luego de la palabra `Implements` en el miembro implementado. Por ejemplo, en “`Public Function Cociente(N1 As Double, N2 As Double) As Double Implements IOperacionesPrimerNivel.Cociente`” se puede observar que `Cociente` corresponde a `IOperacionesPrimerNivel` mientras que en “`Public Function Potencia(N1 As Double, N2 As Double) As Double Implements IOperacionesSegundoNivel.Potencia`” se puede ver que `Potencia` corresponde a `IOperacionesSegundoNivel`.

```

Public Class Operaciones
    Implements IOperacionesSegundoNivel
    Public Function Cociente(N1 As Double, N2 As Double) As Double _
        Implements IOperacionesPrimerNivel.Cociente
        Return IIf(N2 <> 0, N1 / N2, Nothing)
    End Function
    Public Function Producto(N1 As Double, N2 As Double) As Double _
        Implements IOperacionesPrimerNivel.Producto
        Return N1 * N2
    End Function
    Public Function Resta(N1 As Double, N2 As Double) As Double _
        Implements IOperacionesPrimerNivel.Resta
        Return N1 - N2
    End Function
    Public Function Suma(N1 As Double, N2 As Double) As Double _
        Implements IOperacionesPrimerNivel.Suma
        Return N1 + N2
    End Function
    Public Function Potencia(N1 As Double, N2 As Double) As Double _
        Implements IOperacionesSegundoNivel.Potencia
        Return N1 ^ N2
    End Function

```

```
Public Function Raiz(N1 As Double, N2 As Double) As Double _
    Implements IOperacionesSegundoNivel.Raiz
    Return N1 ^ (1 / N2)
End Function
End Class
```

Ejemplo de herencia múltiple de interfaces:

Las interfaces pueden soportar la herencia múltiple. Esto ofrece una opción flexible cuando se desean aprovechar definiciones realizadas en diversas interfaces.

En este ejemplo, la interfaz `IDescuentoCompleto` hereda de `IDescuentoPago` e `IDescuentoCantidad` además de definir una función propia denominada `DescuentoCompleto`.

Las clases que implementen la interfaz `IDescuentoCompleto` implementarán los métodos `DescuentoPago`, `DescuentoCantidad` y `DescuentoCompleto`. Cada clase le dará a la implementación las características que necesite.

Más abajo, se observa una implementación de la interfaz `IDescuentoCompleto` en la clase `Descuento`, donde el método `DescuentoCantidad` recibe un importe y una cantidad que puede ser cero, uno, dos o mayor a dos y de acuerdo a esto realiza descuentos de 10%, 20%, 30% y 40%, respectivamente. El método `DescuentoPago` hace algo similar (aplica descuentos porcentuales) pero dependiendo de la forma de pago, que puede ser 0, 1, 2 o mayor a dos. Por su lado, el método `DescuentoCompleto` le aplica al importe que recibe primero el `DescuentoCantidad` y luego el `DescuentoPago`.

```
Interface IDescuentoPago
    Function DescuentoPago(pImporte As Double, pForma As Byte) As Double
End Interface
Interface IDescuentoCantidad
    Function DescuentoCantidad(pImporte As Double, pCantidad As Byte) As Double
End Interface
Interface IDescuentoCompleto
    Inherits IDescuentoPago
    Inherits IDescuentoCantidad
    Function DescuentoCompleto(pImporte As Double, pForma As Byte, _
        pCantidad As Byte) As Double
End Interface

Public Class Descuento
    Implements IDescuentoCompleto

        Public Function DescuentoCompleto(pImporte As Double, _
            pForma As Byte, pCantidad As Byte) As Double Implements IDescuentoCompleto.DescuentoCompleto
            Return Me.DescuentoPago(Me.DescuentoCantidad(pImporte, pCantidad), pForma)
        End Function
    End Class
```

```

Public Function DescuentoCantidad(pImporte As Double, pCantidad As Byte) _
    As Double Implements IDescuentoCantidad.DescuentoCantidad
    Dim VImporteRetorno As Double
    If pCantidad = 0 Then VImporteRetorno = pImporte * 0.9
    If pCantidad = 1 Then VImporteRetorno = pImporte * 0.8
    If pCantidad = 2 Then VImporteRetorno = pImporte * 0.7
    If pCantidad > 2 Then VImporteRetorno = pImporte * 0.6
    Return VImporteRetorno
End Function

Public Function DescuentoPago(pImporte As Double, pForma As Byte) _
    As Double Implements IDescuentoPago.DescuentoPago
    Dim VImporteRetorno As Double
    If pForma = 0 Then VImporteRetorno = pImporte * 0.9
    If pForma = 1 Then VImporteRetorno = pImporte * 0.8
    If pForma = 2 Then VImporteRetorno = pImporte * 0.7
    If pForma > 2 Then VImporteRetorno = pImporte * 0.6
    Return VImporteRetorno
End Function
End Class

```

Se puede observar lo explicado ejecutando el siguiente código:

```

Public Class Form1
    Private Sub Form1_Load(sender As Object, e As System.EventArgs) Handles Me.Load
        Dim D As New Descuento
        MsgBox(D.DescuentoCompleto(10000, 1, 2))
    End Sub
End Class

```

Ejemplo sobre cómo funciona el tipado por interfaz:

Cuando una clase implementa una interfaz, las instancias que se generen a partir de ella además de implementar su propio tipo (por ejemplo, la clase `Cliente` es un cliente) y los tipos que existen como supertipos en la jerarquía de herencia que reciben (por ejemplo, si `Cliente` hereda de `Persona`, entonces las instancias de `Cliente` implementarán el/los tipos que posea `Persona`), también implementarán el tipo de la interfaz.

En el siguiente código se puede observar una clase denominada `Clase1` que implementa la interfaz `IIInterface1`. De lo dicho anteriormente se deduce que si se tiene un objeto C resultante de instanciar `Clase1`, este será del tipo `Clase1` por ser instancia de ella, del tipo `IIInterface1`, pues `Clase1` implementa esta interfaz y también `Object`, ya que en esta tecnología que utilizamos la clase más abstracta de todas es `Object` y de allí nace todo el grafo de herencia.

```

Public Class Form1
    Private Sub Form1_Load(sender As Object, e As System.EventArgs) Handles Me.Load
        Dim C As Clase1 = New Clase1
        MsgBox("El objeto C es una instancia de Clase1, " & vbCrLf &
               "y Clase1 implementa la interfaz IInterface1." & vbCrLf &
               "C es del tipo Clase1 (True / False): " & TypeOf C Is Clase1)
        MsgBox("El objeto C es una instancia de Clase1, " & vbCrLf &
               "y Clase1 implementa la interfaz IInterface1." & vbCrLf &
               "C es del tipo IInterface1 (True / False): " & TypeOf C Is IInterface1)
        MsgBox("El objeto C es una instancia de Clase1, " & vbCrLf &
               "y Clase1 implementa la interfaz IInterface1. " & vbCrLf &
               "C es del tipo Object (True / False): " & TypeOf C Is Object)
    End Sub
End Class
Interface IInterface1

End Interface
Public Class Clase1
    Implements IInterface1
End Class

```

Como puede observarse en el código, la variable `C` que apunta al objeto instanciado es del tipo `Clase1`. Eso implica que esa variable podrá apuntar a cualquier objeto del tipo `Clase1` o a cualquier objeto que pertenezca a un sub tipo de esta. Si se intentase hacer “`TypeOf C Is Auto`”, esto generaría un error en tiempo de compilación ya que, de acuerdo al tipo de la variable `C` y analizando el grafo de herencia, no existe ninguna posibilidad que la variable `C` apunte a un objeto de tipo `Auto`. En realidad, según el IDE que se utilice, el error se detectará antes de llegar a la compilación.

Si en lugar de ser de tipo `Clase1` la variable `C` fuera del tipo `Object`, entonces podría apuntar a cualquier objeto existente, ya que la variable sería del tipo más abstracto posible. Esto traería como consecuencia que en tiempo de compilación, y mucho antes de ese momento, podamos saber si la variable `C` apunta a un objeto de tipo `Auto` o no. En la práctica, no se generaría ningún tipo de error, pero si la variable `C` apunta a un objeto que no es de tipo `Auto`, al hacer “`TypeOf C Is Auto`” la respuesta será `False`. El código quedaría de la siguiente manera:

```

Public Class Form1
    Private Sub Form1_Load(sender As Object, e As System.EventArgs) Handles Me.Load
        Dim C As Object = New Clase1
        MsgBox("El objeto C es una instancia de Clase1, " & vbCrLf &
               "y Clase1 implementa la interfaz IInterface1." & vbCrLf &
               "C es del tipo Clase1 (True / False): " & TypeOf C Is Clase1)
        MsgBox("El objeto C es una instancia de Clase1, " & vbCrLf &
               "y Clase1 implementa la interfaz IInterface1." & vbCrLf &
               "C es del tipo IInterface1 (True / False): " & TypeOf C Is IInterface1)
        MsgBox("El objeto C es una instancia de Clase1, " & vbCrLf &
               "y Clase1 implementa la interfaz IInterface1. " & vbCrLf &
               "C es del tipo Object (True / False): " & TypeOf C Is Object)
    End Sub
End Class

```

```

    "C es del tipo Object (True / False): " & TypeOf C Is Object)
MsgBox("El objeto C es una instancia de Clase1, " & vbCrLf &
    "y Clase1 implementa la interfaz IInterface1." & vbCrLf &
    "C es del tipo Auto (True / False): " & TypeOf C Is Auto)

End Sub
End Class
Interface IInterface1

End Interface
Public Class Clase1
    Implements IInterface1
End Class
Public Class Auto

End Class

```

Hasta aquí se vio qué rol cumplen las interfaces respecto del tipado y podemos decir que es similar al tipado que se produce cuando se hereda, pero más flexible. Se menciona la flexibilidad pues al menos hay dos consecuencias directamente observables. Por un lado, la implementación de una interfaz puede realizarse en clases que no pertenezcan a la misma rama, considerando la jerarquía de herencia. La segunda consecuencia es que se pueden implementar tantas interfaces como se requieran, lo que permite el manejo de múltiples tipos ad hoc y no solo restringirse a los tipos de la jerarquía superior que nos da la herencia.

En función de lo mencionado, podemos pensar en generar un comportamiento polimórfico utilizando interfaces, sin perder de vista los preceptos teóricos pero con una visión pragmática, ya que si las clases que intervienen en esta construcción implementan la misma interfaz, entonces compartirán un tipo y los miembros que esa interfaz defina. El código siguiente ejemplifica cómo tres tarjetas de crédito que brindan beneficios a sus asociados los implementan de diferente manera para luego aprovecharlos polimórficamente.

```

Public Class Form1
    Function CuantoPaga(pImporte As Decimal, pObjeto As IBeneficio) As Decimal
        Return pObjeto.Beneficio(pImporte)
    End Function
    Private Sub Form1_Load(sender As Object, e As System.EventArgs) Handles Me.Load
        Dim ListaTarjetas As New List(Of IBeneficio) _
            ({New TarjetaDorada, New TarjetaPlateada, New TarjetaRoja})
        Dim Vimporte As Decimal = InputBox("Ingrese el importe:")
        For Each T In ListaTarjetas
            MsgBox("Por un importe de " & Vimporte & " la " & T.GetType.Name & _
                " debe pagar " & T.Beneficio(1000))
        Next
    End Sub
End Class

```

```

Public Interface IBeneficio
    Function Beneficio(pImporte As Decimal) As Decimal
End Interface
Public Class TarjetaDorada
    Implements IBeneficio
    Public Function Beneficio(pImporte As Decimal) As _
        Decimal Implements IBeneficio.Beneficio
        Return pImporte * 0.7
    End Function
End Class
Public Class TarjetaPlateada
    Implements IBeneficio
    Public Function Beneficio(pImporte As Decimal) As _
        Decimal Implements IBeneficio.Beneficio
        Return pImporte * 0.8
    End Function
End Class
Public Class TarjetaRoja
    Implements IBeneficio
    Public Function Beneficio(pImporte As Decimal) As _
        Decimal Implements IBeneficio.Beneficio
        Return pImporte * 0.9
    End Function
End Class

```

Las tarjetas de crédito están representadas por las clases `TarjetaDorada`, `TarjetaPlateada` y `TarjetaRoja`, la interfaz que implementarán las tres clases se denomina `IBeneficio` y define una función denominada `Beneficio`, que recibe un importe que será tratado según la tarjeta que posea el asociado. En este ejemplo, si la tarjeta es dorada, sobre sus compras recibirá un descuento del 30%, si es plateada del 20% y si es roja del 10%. Estas funcionalidades están programadas en cada una de las clases. Luego, asumiendo por la implementación que estas hicieron de la interfaz `IBeneficio`, todas implementan el tipo `IBeneficio`, y este tipo se utiliza para aprovechar el polimorfismo. Para utilizar esta ventaja, existe en la clase `Form1` una función denominada `CuantoPaga`, que retorna el importe final a pagar resultante de aplicarle el descuento que corresponda, según la tarjeta, al importe consumido. Esta función posee dos parámetros: `pImporte` para recibir el importe consumido y `pObjeto` para recibir un objeto del tipo `IBeneficio` (recordemos que tanto `TarjetaDorada`, `TarjetaPlateada` y `TarjetaRoja` se corresponden con ese tipo). Luego, cada vez que esta función reciba un objeto en `pObjeto`, retornará el resultado de “`pObjeto.Beneficio(pImporte)`” sin importar qué tarjeta es. Esto permite encapsular el conocimiento del descuento dentro de la tarjeta y hacer extensible el desarrollo ya que, en caso de surgir una nueva tarjeta, basta con crear la nueva clase y hacer que esta implemente la interfaz `IBeneficio` para que las instancias de esta clase puedan ser enviadas como parámetro a `pObjeto` y que esta función las procese.

Cuando se desarrolló el caso de polimorfismo por herencia, se analizó un ejemplo que involucraba las clases `CuentaDeAhorro` y `CuentaCorriente`, que poseían un método polimórfico `Extraccion`. Veamos cómo queda el mismo ejemplo utilizando interfaces en lugar de herencia.

Para explotar el polimorfismo, se deberá hacer desde una variable de tipo `IExtraccion`.

```
Public Interface IExtraccion
    Function Extraccion(ByVal pMonto As Decimal) As Boolean
    Sub Transferencia(ByVal pMonto As Decimal, ByVal pCuentaDestino As Cuenta)
End Interface

Public Class CuentaDeAhorro
    Inherits Cuenta
    Implements IExtraccion
    Sub New()
        Me.Saldo = 0
    End Sub
    Sub New(ByVal pSaldoInicial As Decimal)
        Me.New()
        Me.Saldo = pSaldoInicial
    End Sub
    Public Function Extraccion(pMonto As Decimal) As Boolean Implements IExtraccion.Extraccion
        Dim R As Boolean = False
        If Me.Saldo - pMonto >= 0 Then Me.Saldo -= pMonto : R = True
        Return R
    End Function
    Public Sub Transferencia1(pMonto As Decimal, pCuentaDestino As Cuenta) _
        Implements IExtraccion.Transferencia
        If Me.Extraccion(pMonto) Then pCuentaDestino.Deposito(pMonto)
    End Sub
End Class

Public Class CuentaCorriente
    Inherits Cuenta
    Implements IExtraccion
    Sub New()
        Me.Saldo = 0
    End Sub
    Sub New(ByVal pSaldoInicial As Decimal)
        Me.New()
        Me.Saldo = pSaldoInicial
    End Sub
    Private Vdescubierto As Decimal
    Public Property Descubierto() As Decimal
        Get
            Return Vdescubierto
        End Get
        Set(ByVal value As Decimal)
            Vdescubierto = value
        End Set
    End Property
End Class
```

```

End Property
Public Function Extraccion(pMonto As Decimal) As Boolean _
    Implements IExtraccion.Extraccion
    Dim R As Boolean = False
    If Me.Saldo + Me.Descubierto - pMonto >= 0 _
Then Me.Saldo -= pMonto : R = True
    Return R
End Function
Public Sub Transferencia(pMonto As Decimal, pCuentaDestino As Cuenta) _
    Implements IExtraccion.Transferencia
    If Me.Extraccion(pMonto) Then pCuentaDestino.Deposito(pMonto)
End Sub
End Class

Public MustInherit Class Cuenta
    Private Vnumero As String
    Public Property Numero() As String
        Get
            Return Vnumero
        End Get
        Set(ByVal value As String)
            Vnumero = value
        End Set
    End Property
    Private Vsaldo As Decimal
    Public Property Saldo() As Decimal
        Get
            Return Vsaldo
        End Get
        Set(ByVal value As Decimal)
            Vsaldo = value
        End Set
    End Property
    Public Sub Deposito(ByVal pMonto As Decimal)
        Me.Saldo += pMonto
    End Sub
End Class

```

Ejemplo de interfaz anidada:

Una interfaz anidada se construye colocando una interfaz dentro de la otra. Esto permite que se genere una suerte de ordenamiento. Para acceder a ella, debemos hacerlo por medio del nombre de la interfaz que anida, colocando un punto y luego el nombre de la interfaz anidada. Se pueden construir los niveles de anidamiento que se consideren necesarios. El siguiente código muestra un ejemplo donde se utiliza una clase denominada `Calculo`, que implementa una estructura de interfaces anidadas.

```
Public Class Form1
    Private Sub Form1_Load(sender As System.Object, e As System.EventArgs) _
        Handles MyBase.Load
        Dim C As New Calculo( _
            InputBox("Ingrese Número1: "), InputBox("Ingrese Número2: "))
        MsgBox("Suma: " & C.Suma())
        MsgBox("Resta: " & C.Resta())
        MsgBox("Producto: " & C.Producto())
        MsgBox("Cociente: " & C.Cociente())
        MsgBox("Potencia: " & C.Potencia())
        MsgBox("Raíz: " & C.Raiz())
    End Sub
End Class

Interface IOperaciones
    Property Numero1 As Double
    Property Numero2 As Double
    Interface IOperacionesSimples
        Function Suma() As Double
        Function Resta() As Double
        Function Producto() As Double
        Function Cociente() As Double
    End Interface
    Interface IOperacionesComplejas
        Function Potencia() As Double
        Function Raiz() As Double
    End Interface
End Interface

Public Class Calculo
    Implements IOperaciones
    Implements IOperaciones.IOperacionesSimples
    Implements IOperaciones.IOperacionesComplejas
    Sub New(N1 As Double, N2 As Double)
        Me.Numero1 = N1
        Me.Numero2 = N2
    End Sub
    Dim Vnumero1 As Double
    Public Property Numero1 As Double Implements IOperaciones.Numero1
        Get
            Return Vnumero1
        End Get
        Set(value As Double)
            Vnumero1 = value
        End Set
    End Property
    Dim Vnumero2 As Double
    Public Property Numero2 As Double Implements IOperaciones.Numero2
        Get
            Return Vnumero2
        End Get
        Set(value As Double)
```

```

        Vnumero2 = value
    End Set
End Property
Public Function Cociente() As Double _
    Implements IOperaciones.IOperacionesSimples.Cociente
    Try : Return Me.Numero1 / Me.Numero2 : Catch ex As Exception : End Try
End Function
Public Function Producto() As Double _
    Implements IOperaciones.IOperacionesSimples.Producto
    Try : Return Me.Numero1 * Me.Numero2 : Catch ex As Exception : End Try
End Function
Public Function Resta() As Double _
    Implements IOperaciones.IOperacionesSimples.Resta
    Try : Return Me.Numero1 - Me.Numero2 : Catch ex As Exception : End Try
End Function
Public Function Suma() As Double _
    Implements IOperaciones.IOperacionesSimples.Suma
    Try : Return Me.Numero1 + Me.Numero2 : Catch ex As Exception : End Try
End Function
Public Function Potencia() As Double _
    Implements IOperaciones.IoperacionesComplejas.Potencia
    Try : Return Me.Numero1 ^ Me.Numero2 : Catch ex As Exception : End Try
End Function
Public Function Raiz() As Double _
    Implements IOperaciones.IoperacionesComplejas.Raiz
    Try : Return Me.Numero1 ^ (1 / Me.Numero2) : Catch ex As Exception : End Try
End Function
End Class

```

5.5.1 La interfaz IComparable

En innumerables ocasiones es muy útil poder ordenar un *Array*. Es fácil suponer los ordenamientos de números o palabras. En ambos casos, es casi seguro que se requerirá ordenarlos de manera ascendente o descendente. En el primer caso se utilizará el valor de cada número y en el segundo, un ordenamiento alfabético, como en el diccionario. Los algoritmos que se utilizan para este tipo de ordenamientos son conocidos y es debido a esto que el framework los posee implementados.

Pero consideremos ahora que tenemos la clase *Persona* que posee en su estructura los campos *Vnombre* y *Vapellido* que la definen y las propiedades *Nombre* y *Apellido* para poder leer y grabar, respectivamente, esas características.

Si creamos un *Array* del tipo *Persona*, ¿qué significa ordenarlo? ¿cómo puede saber el framework si deseamos ordenarlo por el nombre o por el apellido? La respuesta a esta última pregunta es que, simplemente, no lo sabe. En tanto la clase *Persona* posea una estructura más compleja, más serán las opciones que se podrán considerar para ordenar un *Array* de este tipo. Parece lógico que sea la misma clase *Persona* la que defina qué significa o qué características hacen que una persona, dado un determinado criterio de ordenamiento, se ubique antes o después de otra.

Para esto, en el framework se han creado interfaces que permiten encapsular este conocimiento en la misma clase y que esta interactúe con la clase *Array*.

La primera interfaz que analizaremos es la interfaz `IComparable`. Para comprender cómo utilizarla, construiremos una clase `Persona` que implemente esta interfaz. La implementación de la interfaz tendrá al menos tres implicancias:

- La primera es que la clase `Persona` deberá implementar el método `CompareTo`. Esto se debe a que está definido en la interfaz `IComparable`. Su firma es:

```
Public Function CompareTo(other As Persona) As Integer _
    Implements System.IComparable(Of Persona).CompareTo
End Function
```

- La segunda es que la clase `Persona` ahora también tendrá el tipo `IComparable`, ya que la implementación de una interfaz causa que la clase que la implementa posea su tipo.
- La tercera está relacionada con la primera debido a la obligatoriedad que plantean las interfaces respecto a la implementación de su contenido en la clase que implementa la interfaz. Esto conlleva a que cualquier objeto que posea visibilidad sobre una instancia de `Persona` podrá enviarle un mensaje `CompareTo(...)` sin temor a que esta solicitud dé, como resultado, una excepción.

Como puede observarse, el método `CompareTo` posee un parámetro denominado `other` del tipo `Persona` para recibir una instancia de `Persona`. El método `CompareTo` es una función y como tal, retorna un valor que en este caso particular es `Integer`. El valor retornado es muy significativo para lo que deseamos hacer ya que lo que se retorna será utilizado por la clase `Array` para determinar el ordenamiento.

Consideremos que dado un `Array` de personas, una instancia de `Persona` será la que está ejecutando el método `CompareTo` y recibiendo por parámetro a otra `Persona`. Entre ambas se establecerá una comparación y de acuerdo al ordenamiento deseado se retornará el valor adecuado por el método según las siguientes reglas:

- Si el objeto `Persona` que se está ejecutando se ubica antes que la `Persona` recibida en el parámetro `other` y el criterio de ordenamiento es ascendente, entonces se deberá retornar un entero menor que cero.
- Si el objeto `Persona` que se está ejecutando se ubica en la misma posición que la `Persona` recibida en el parámetro `other` y el criterio de ordenamiento es ascendente, entonces se deberá retornar un cero.
- Si el objeto `Persona` que se está ejecutando se ubica después que la `Persona` recibida en el parámetro `other` y el criterio de ordenamiento es ascendente, entonces se deberá retornar un entero mayor que cero.

Supongamos que el criterio de ordenamiento deseado es por `Nombre` y `Apellido`, por lo que a igualdad de `Nombre` se colocará primero a la persona cuyo `Apellido` se ubica antes alfabéticamente.

Para lograrlo, escribimos el siguiente código:

```
Public Class Persona
    Implements IComparable(Of Persona)
    Private Vnombre As String
    Private Vapellido As String
    Sub New(pNombre As String, pApellido As String)
        Me.Nombre = pNombre
        Me.Apellido = pApellido
    End Sub
    Public Property Nombre() As String
        Get
            Return Vnombre
        End Get
        Set(ByVal value As String)
            Vnombre = value
        End Set
    End Property
    Public Property Apellido() As String
        Get
            Return Vapellido
        End Get
        Set(ByVal value As String)
            Vapellido = value
        End Set
    End Property
    Public Overrides Function ToString() As String
        Return Me.Nombre & " " & Me.Apellido
    End Function
    Public Function CompareTo(other As Persona) As Integer _
        Implements System.IComparable(Of Persona).CompareTo
        Dim VvalorRetorno As Integer
        If other Is Nothing Then
            VvalorRetorno = 1
        Else
            VvalorRetorno = StrComp(Me.Nombre.Trim & Me.Apellido.Trim, _
                other.Nombre.Trim & other.Apellido.Trim, _
                CompareMethod.Text)
        End If
        Return VvalorRetorno
    End Function
End Class
```

Como puede observarse en la función `CompareTo`, se opta por retornar un 1 si `other` es `Nothing`, lo que genera que los `Nothing`, si existieran, queden al principio del Array ordenado.

Si el valor recibido en `other` es distinto a `Nothing`, entonces se concatena `Nombre` y `Apellido` de la persona que se está ejecutando (`Me`) y se la compara con la misma concatenación de la persona recibida en `other`.

Para llevar adelante esta comparación se utilizó la función `StrComp`, aprovechando su funcionalidad, considerando que recibe dos `String` y retorna:

- Si el `String` que recibe como primer parámetro se ordena delante del `String` que se recibe como segundo parámetro, se retorna -1.
- Si el `String` que recibe como primer parámetro se ordena en el mismo lugar del `String` que se recibe como segundo parámetro, se retorna 0.
- Si el `String` que recibe como primer parámetro se ordena después del `String` que se recibe como segundo parámetro, se retorna 1.

Estos valores coinciden con lo expresado anteriormente en referencia a los valores que debe retornar `CompareTo`.

En la clase `Persona` también podemos observar la sobrescritura del método `ToString` con el objetivo de redefinir lo que retorna por defecto y ajustarlo para que sea `Nombre` y `Apellido` de la persona separados por un espacio. Esto último es un agregado a nuestro ejemplo que no interfiere con la interfaz `IComparable`.

```
Public Overrides Function ToString() As String
    Return Me.Nombre & " " & Me.Apellido
End Function
```

A continuación, se establece el código que nos permite utilizar la implementación que define el criterio de ordenamiento para la clase `Persona`. Para ello, en un proyecto de tipo consola escribimos:

```
Module Principal
    Public Sub Main()
        Dim Personas() As Persona = {New Persona("Juan", "Perez"), _
                                      New Persona("Pedro", "Perez"), _
                                      New Persona("Pedro", "Garcia"), _
                                      New Persona("María", "Martinez")}
        Array.Sort(Personas)
        For Each P As Persona In Personas
            Console.WriteLine(P.ToString())
        Next
        Console.ReadKey()
    End Sub
End Module
```

En el código anterior se genera un Array denominado Personas que posee cuatro instancias de personas cargadas: Juan Perez, Pedro Perez, Pedro Garcia y María Martinez.

Luego, haciendo uso del método Sort, se ordena el Array para recorrerlo con un For ... Each y observar, en la consola, cómo ha quedado ordenado. El resultado es:

```
Juan Perez  
María Martinez  
Pedro Garcia  
Pedro Perez
```

Ajustar el código para que el ordenamiento, en lugar de ser ascendente sea descendente es sencillo. Simplemente, al resultado obtenido en la variable VvalorRetorno de la función CompareTo lo invertimos multiplicándolo por -1. El código quedaría así:

```
Public Function CompareTo(other As Persona) As Integer _  
    Implements System.IComparable(Of Persona).CompareTo  
    Dim VvalorRetorno As Integer  
    If other Is Nothing Then  
        VvalorRetorno = 1  
    Else  
        VvalorRetorno = StrComp(Me.Nombre.Trim & Me.Apellido.Trim, _  
            other.Nombre.Trim & other.Apellido.Trim, _  
            CompareMethod.Text)  
    End If  
    Return VvalorRetorno * -1  
End Function
```

Luego de esta actualización, lo que se obtiene al ejecutarlo es:

```
Pedro Perez  
Pedro Garcia  
María Martinez  
Juan Perez
```

Cómo habrá podido percibir, al cambiar el ordenamiento de ascendente a descendente se alteró el código preexistente, anulando el primer ordenamiento para darle lugar al segundo. Esta limitación es propia de la interfaz que acabamos de analizar. Seguramente, Ud. estará pensando que sería deseable en la clase Persona poseer más de un criterio de ordenamiento con la capacidad de poder seleccionar uno de ellos sin tener que alterar el código programado.

Para lograrlo, utilizaremos la interfaz IComparer.

5.5.2 La interfaz IComparer

La interfaz `ICompare` permite dotar a una clase con varios criterios de ordenamiento para ser aprovechados por el método `Sort` de la clase `Array`.

Cuando se implementa la interfaz `IComparer`, se debe implementar el método `Compare` que ella define,

```
Function Compare (x As T, y As T) As Integer
End Function
```

Donde:

- X: es el primer objeto a comparar.
- Y: es el segundo objeto a comparar.
- T: el es tipo de los objetos.

Esta función recibe dos objetos del tipo T, los compara y retorna un valor entero. El significado del valor entero retornado está relacionado con el orden de los objetos X e Y. Los valores retornados pueden ser:

- Menor que cero si el objeto X se ubica antes que el objeto Y.
- Cero si el objeto X se ubica en el mismo lugar que el objeto Y.
- Mayor que cero si el objeto X se ubica después que el objeto Y.

Para observar cómo funciona esta interfaz, tomaremos la clase `Persona` y le implementaremos esta interfaz. Luego, crearemos un `Array` con objetos de este tipo para que se pueda ordenar de forma ascendente y descendente por `Nombre`, `Apellido` y `Edad`.

```
Public Class Persona
    Private Vnombre As String
    Private Vapellido As String
    Private VfechaNacimiento As Date
    Sub New(pNombre As String, pApellido As String, pFechaNacimiento As Date)
        Me.Nombre = pNombre
        Me.Apellido = pApellido
        Me.VfechaNacimiento = pFechaNacimiento
    End Sub
    Public Property Nombre() As String
        Get
            Return Vnombre
        End Get
        Set(ByVal value As String)
```

```

        Vnombre = value
    End Set
End Property
Public Property Apellido() As String
    Get
        Return Vapellido
    End Get
    Set(ByVal value As String)
        Vapellido = value
    End Set
End Property
Public Overrides Function ToString() As String
    Return Me.Nombre & " " & Me.Apellido & " " & Me.Edad
End Function
Public ReadOnly Property Edad() As Integer
    Get
        Return (Now.Year - VfechaNacimiento.Year) - _
            IIf(Now.DayOfYear - VfechaNacimiento.DayOfYear <= 0, 1, 0)
    End Get
End Property
Class OrdenNombreAscDesc
    Implements IComparer(Of Persona)
    Dim VascDesc As Boolean = True
    Sub New()
        End Sub
    Sub New(pAscDesc As Boolean)
        VascDesc = pAscDesc
    End Sub
    Public Function Compare(x As Persona, y As Persona) As Integer _
        Implements System.Collections.Generic.IComparer(Of _
        Persona).Compare
        Dim VvalorRetorno As Integer
        If x Is Nothing And y Is Nothing Then VvalorRetorno = 0
        If x Is Nothing Then VvalorRetorno = -1
        If y Is Nothing Then VvalorRetorno = 1
        If Not (x Is Nothing) And Not (y Is Nothing) Then _
            VvalorRetorno = (StrComp(x.Nombre.Trim, y.Nombre.Trim, _
                CompareMethod.Text)) * IIf(VascDesc, 1, -1)
        Return VvalorRetorno
    End Function
End Class
Class OrdenApellidoAscDesc
    Implements IComparer(Of Persona)
    Dim VascDesc As Boolean = True
    Sub New()
        End Sub
    Sub New(pAscDesc As Boolean)
        VascDesc = pAscDesc
    End Sub

```

```

Public Function Compare(x As Persona, y As Persona) As Integer _
    Implements System.Collections.Generic.IComparer(Of _
    Persona).Compare
    Dim VvalorRetorno As Integer
    If x Is Nothing And y Is Nothing Then VvalorRetorno = 0
    If x Is Nothing Then VvalorRetorno = -1
    If y Is Nothing Then VvalorRetorno = 1
    If Not (x Is Nothing) And Not (y Is Nothing) Then _
        VvalorRetorno = (StrComp(x.Apellido.Trim, y.Apellido.Trim, _
            CompareMethod.Text)) * IIf(VascDesc, 1, -1)
    Return VvalorRetorno
End Function
End Class
Class OrdenEdadAscDesc
    Implements IComparer(Of Persona)
    Dim VascDesc As Boolean = True
    Sub New()
    End Sub
    Sub New(pAscDesc As Boolean)
        VascDesc = pAscDesc
    End Sub
    Public Function Compare(x As Persona, y As Persona) As Integer _
        Implements System.Collections.Generic.IComparer(Of _
        Persona).Compare
        Dim VvalorRetorno As Integer
        If x Is Nothing And y Is Nothing Then VvalorRetorno = 0
        If x Is Nothing Then VvalorRetorno = -1
        If y Is Nothing Then VvalorRetorno = 1
        If Not x Is Nothing And Not y Is Nothing Then _
            VvalorRetorno = (IIf(x.Edad > y.Edad, 1, _
                IIf(x.Edad = y.Edad, 0, -1))) * _
                IIf(VascDesc, 1, -1)
        Return VvalorRetorno
    End Function
End Class
End Class

```

En el código anterior se pueden observar las incorporaciones que se le han realizado a la clase `Persona`. Podemos comenzar por analizar la propiedad de solo lectura `Edad`. Esta utiliza la fecha de nacimiento almacenada en el campo `VfechaNacimiento` cuando se instancian las personas.

Para determinar la edad se utiliza la fórmula:

```

Return (Now.Year - VfechaNacimiento.Year) - _
    IIf(Now.DayOfYear - VfechaNacimiento.DayOfYear < 0, 1, 0)

```

Lo que hace es restarle al año de la fecha actual, cargada en el sistema, el año de la fecha de nacimiento de la Persona. Luego, se evalúa la cantidad de días transcurridos desde el primero de enero de la fecha actual hasta la fecha actual (`Now.DayOfYear`), menos los días transcurridos desde el primero de enero del año de nacimiento de la Persona hasta la fecha de nacimiento de la Persona. Si el resultado obtenido es menor a cero, significa que la persona no ha cumplido años aún para el año corriente, lo que implica que hay que quitarle 1 año a la resta de los años; y en caso contrario no se le resta nada. En esta fórmula, si la Persona cumple años el mismo día de la fecha actual cargada en el sistema, se considera como años cumplidos.

Veamos unos casos sencillos para comprender cómo funciona esta fórmula.

Caso 1:

Fecha actual 03/11/2012
Fecha de Nacimiento 02/08/1966

Resultado esperado: 46 años.

Aplicación de la fórmula:

1. $2012 - 1966 = 46$
2. Días transcurridos desde 01/01/2012 hasta 03/11/2012 = 308
3. Días transcurridos desde 01/01/1966 hasta 02/08/1966 = 214
4. $308 - 214 = 94$
5. Como el resultado del paso 4 es mayor que cero, significa que la persona ya cumplió años en 2012 y no hay que ajustar el resultado del paso 1, lo que se logra restándole cero.
6. Resultado obtenido: 46 años.

Caso 2:

Fecha actual 03/11/2012
Fecha de Nacimiento 05/11/1966

Resultado esperado: 45 años.

Aplicación de la fórmula:

1. $2012 - 1966 = 46$
2. Días transcurridos desde 01/01/2012 hasta 03/11/2012 = 308
3. Días transcurridos desde 01/01/1966 hasta 05/11/1966 = 310
4. $308 - 310 = -2$
5. Cómo el resultado del paso 4 es menor que cero, significa que la persona aún no cumplió años en 2012 y hay que ajustar el resultado del paso 1, lo que se logra restándole uno.
6. $46 - 1 = 45$
7. Resultado obtenido: 45 años.

También la clase `Persona` posee tres clases anidadas denominadas `OrdenNombreAscDesc`, `OrdenApellidoAscDesc` y `OrdenEdadAscDesc`. Cada una de ellas implementa la interfaz `ICompare` y con ello el método `Compare`. Las clases anidadas fueron preparadas para que se le pueda pasar a través del constructor, por medio del parámetro `pAscDesc` de tipo `boolean`, el criterio de ordenamiento ascendente (`true`) o descendente (`false`) deseado. Así, tenemos seis ordenamientos posibles:

- Por nombre ascendente
- Por nombre descendente
- Por apellido ascendente
- Por apellido descendente
- Por edad ascendente
- Por edad descendente

La implementación del código que permite observar el funcionamiento de lo planteado es el siguiente:

```
Public Class Persona
    Private Vnombre As String
    Private Vapellido As String
    Private VfechaNacimiento As Date
    Sub New(pNombre As String, pApellido As String, pFechaNacimiento As Date)
        Me.Nombre = pNombre
        Me.Apellido = pApellido
        Me.VfechaNacimiento = pFechaNacimiento
    End Sub
    Public Property Nombre() As String
        Get
            Return Vnombre
        End Get
        Set(ByVal value As String)
            Vnombre = value
        End Set
    End Property
    Public Property Apellido() As String
        Get
            Return Vapellido
        End Get
        Set(ByVal value As String)
            Vapellido = value
        End Set
    End Property
    Public Overrides Function ToString() As String
        Return Me.Nombre & " " & Me.Apellido & " " & Me.Edad
    End Function
    Public ReadOnly Property Edad() As Integer
        Get
            Return (Now.Year - VfechaNacimiento.Year) - _
                IIf(Now.DayOfYear - VfechaNacimiento.DayOfYear <= 0, 1, 0)
        End Get
    End Property
End Class
```

```

        End Get
    End Property
    Class OrdenNombreAscDesc
        Implements IComparer(Of Persona)
        Dim VascDesc As Boolean = True
        Sub New()
        End Sub
        Sub New(pAscDesc As Boolean)
            VascDesc = pAscDesc
        End Sub
        Public Function Compare(x As Persona, y As Persona) As Integer _
            Implements System.Collections.Generic. _
            IComparer(Of Persona).Compare
            Dim ValorRetorno As Integer
            If x Is Nothing And y Is Nothing Then ValorRetorno = 0
            If x Is Nothing Then ValorRetorno = -1
            If y Is Nothing Then ValorRetorno = 1
            If Not (x Is Nothing) And Not (y Is Nothing) Then _
                ValorRetorno = (StrComp(x.Nombre.Trim, y.Nombre.Trim, _
                    CompareMethod.Text)) * IIf(VascDesc, 1, -1)
            Return ValorRetorno
        End Function
    End Class
    Class OrdenApellidoAscDesc
        Implements IComparer(Of Persona)
        Dim VascDesc As Boolean = True
        Sub New()
        End Sub
        Sub New(pAscDesc As Boolean)
            VascDesc = pAscDesc
        End Sub
        Public Function Compare(x As Persona, y As Persona) As Integer _
            Implements System.Collections.Generic. _
            IComparer(Of Persona).Compare
            Dim ValorRetorno As Integer
            If x Is Nothing And y Is Nothing Then ValorRetorno = 0
            If x Is Nothing Then ValorRetorno = -1
            If y Is Nothing Then ValorRetorno = 1
            If Not (x Is Nothing) And Not (y Is Nothing) Then _
                ValorRetorno = (StrComp(x.Apellido.Trim, y.Apellido.Trim, _
                    CompareMethod.Text)) * IIf(VascDesc, 1, -1)
            Return ValorRetorno
        End Function
    End Class
    Class OrdenEdadAscDesc
        Implements IComparer(Of Persona)
        Dim VascDesc As Boolean = True
        Sub New()
        End Sub

```

```

Sub New(pAscDesc As Boolean)
    VascDesc = pAscDesc
End Sub
Public Function Compare(x As Persona, y As Persona) As Integer _
    Implements System.Collections.Generic._
        IComparer(Of Persona).Compare
    Dim ValorRetorno As Integer
    If x Is Nothing And y Is Nothing Then ValorRetorno = 0
    If x Is Nothing Then ValorRetorno = -1
    If y Is Nothing Then ValorRetorno = 1
    If Not x Is Nothing And Not y Is Nothing Then _
        ValorRetorno = (IIf(x.Edad > y.Edad, 1, _ 
            IIf(x.Edad = y.Edad, 0, -1))) * IIf(VascDesc, 1, -1)
    Return ValorRetorno
End Function
End Class
End Class

```

Analicemos el código utilizado dentro de una de las clases anidadas para establecer el criterio de ordenamiento, ya que los otros son muy similares y solo se cambia el dato utilizado (nombre, apellido o edad). En este caso particular, tomaremos el ordenamiento por nombre.

```

Public Function Compare(x As Persona, y As Persona) As Integer _
    Implements System.Collections.Generic.IComparer(Of _
        Persona).Compare
    Dim VvalorRetorno As Integer
    If x Is Nothing And y Is Nothing Then VvalorRetorno = 0
    If x Is Nothing Then VvalorRetorno = -1
    If y Is Nothing Then VvalorRetorno = 1
    If Not (x Is Nothing) And Not (y Is Nothing) Then _
        VvalorRetorno = (StrComp(x.Nombre.Trim, y.Nombre.Trim, _ 
            CompareMethod.Text)) * IIf(VascDesc, 1, -1)
    Return VvalorRetorno
End Function

```

Como puede observarse, la función `Compare` posee dos parámetros: `x` e `y`, del tipo `Persona`, que representan a las personas que se utilizarán en la comparación y a partir de allí se retornará un valor `Integer` para indicar cómo se posiciona una respecto de la otra. Debemos recordar que si el valor retornado es menor que cero, significa que el objeto `X` se ubica antes que el objeto `Y`; si es cero que el objeto `X` se ubica en el mismo lugar que el objeto `Y` y si es mayor que cero, `X` se ubica después que el objeto `Y`.

Se define una variable denominada `VvalorRetorno` donde se almacenará el entero que se desea retornar. Para determinar el valor a retornar, primero se evalúa si `x` e `y` son `Nothing`, y en

caso afirmativo se retorna un 0 (cero). Si solo *x* es *Nothing* se retorna -1 y si solo *y* es *Nothing* se retorna un 1.

Con esto se logra que si en el Array de personas existen elementos *Nothing*, estos se ubiquen a la izquierda.

Más adelante se determina si *x* e *y* no son *Nothing*, se compara el *Nombre* de la persona recibida en *x* con el *Nombre* de la persona recibida en *y*. Como los nombres son de tipo *String*, se aprovecha la función *StrComp* para obtener el valor a retornar. Este valor se multiplica por 1 si la variable *VascDesc* es *true* o -1 si la variable es *false*. Esto último se realiza con el objetivo de no alterar lo almacenado en *VvalorRetorno* en el primer caso o invertirlo en el segundo. Así se logra el ordenamiento descendente. Finalmente, se almacena lo obtenido en la variable *VvalorRetorno* y se procede a retornarlo.

Cabe mencionar que en el caso del ordenamiento por edad, al determinar el valor a retornar no se utiliza la función *StrComp* debido a que la edad es un valor *Integer*. En su lugar, utilizamos el siguiente código:

```
VvalorRetorno = ( IIf(x.Edad > y.Edad, 1, _
                      IIf(x.Edad = y.Edad, 0, -1))) * _
                      IIf(VascDesc, 1, -1)
```

Para poner en práctica lo descripto y observar cómo funcionan los criterios de ordenamiento, se puede utilizar el siguiente fragmento de código.

```
Module Principal
    Dim Personas() As Persona = _
        {Nothing, _
        New Persona("Juan", "Perez", #8/2/1966#), _
        New Persona("Pedro", "Perez", #2/5/2001#), _
        New Persona("Pedro", "Garcia", #8/30/2005#), _
        New Persona("Maria", "Martinez", #10/17/1995#), _
        Nothing}

    Public Sub Main()
        Call Ordena(New Persona.OrdenNombreAscDesc, _
                   "nombre", _
                   "ascendente")
        Call Ordena(New Persona.OrdenNombreAscDesc(False), _
                   "nombre", _
                   "descendente")
        Call Ordena(New Persona.OrdenApellidoAscDesc, _
                   "apellido", _
                   "ascendente")
        Call Ordena(New Persona.OrdenApellidoAscDesc(False), _
                   "apellido", _
```

```

        "descendente")
Call Ordena(New Persona.OrdenEdadAscDesc, _
           "edad", _
           "ascendente")
Call Ordena(New Persona.OrdenEdadAscDesc(False), _
           "edad", _
           "descendente")
Console.ReadKey()
End Sub
Private Sub Ordena(pOrden As Object, _
                  pTextoConcepto As String, pTextoCriterio As String)
    Array.Sort(Personas, pOrden)
    Console.WriteLine("Ordenado por " & pTextoConcepto & " de forma " & _
                      pTextoCriterio & Constants.vbCrLf)
    For Each x As Persona In Personas
        If Not x Is Nothing Then Console.WriteLine(x.ToString())
    Next
    Console.WriteLine(Constants.vbCrLf)
End Sub
End Module

```

5.5.3 La interfaz IClonable

La clonación de objetos permite que un objeto se replique en una posición distinta de memoria. A partir de ese momento, el primer objeto utilizado como original para ser clonado y el segundo objeto, obtenido como resultado de clonar al primero, actúan de manera independiente.

Esto es muy útil ya que si todos los objetos quedaran referenciados, bastaría cambiar uno para que se alteren los otros. Un aspecto relevante para destacar es que debemos distinguir el hecho de “asignar” respecto de “clonar”. Supongamos que tenemos la clase Profesor y dos variables de ese tipo: VprofesorX y VprofesorY. Si VprofesorY apunta a un objeto Profesor, entonces no es lo mismo asignar

VprofesorX = VprofesorY

que clonar,

VprofesorX = VprofesorY.clone

En la asignación, como se explicó anteriormente, las variables `VprofesorX` y `VprofesorY` terminan apuntando al mismo objeto, mientras que en la clonación `VprofesorX` apuntará a un objeto distinto al que apunta la variable `VprofesorY`. Como resulta obvio, en el escenario de la clonación tenemos dos objetos de tipo `Persona` alojados en posiciones de memoria distintas, mientras que en la asignación solo tenemos uno.

Veremos ahora cómo hacer que un objeto sea clonable implementando la interfaz `ICloneable` en la clase que le da origen. Al implementar la interfaz, aparece la función `Clone` que podemos observar a continuación:

```
Public Function Clone() As Object Implements System.ICloneable.Clone
    Return Me.MemberwiseClone
End Function
```

Podemos percibir que esta función solo posee una instrucción en su implementación: `Me.MemberwiseClone`.

El método `MemberwiseClone` es un método protegido definido en la clase `Object` y por tal motivo, lo heredan todas las clases. Su funcionalidad es crear un nuevo objeto al que le copia los campos no estáticos del objeto original desde el cual se crea. Decimos que esta función realiza un tipo de clonación que podemos denominar simple o superficial. Esta denominación se le otorga debido a que cuando se encuentra con campos cuyo tipo es un tipo de valor, realiza una copia bit a bit del mismo, pero cuando se topa con campos cuyo tipo es de referencia, copia la referencia pero no clona al objeto referenciado. Esta forma de funcionamiento ocasiona que el objeto original desde el cual se genera la clonación y el objeto posteriormente creado como producto de la clonación posean una referencia a un tercer objeto en común, generándose una clonación parcial.

Este tema lo trataremos unos párrafos más adelante y veremos cómo evitarlo en caso de ser necesario.

El código en la clase `Profesor` que implementa la clonación simple o superficial es:

```
Public Class Profesor
    Implements ICloneable
    Private Vlegajo As Integer
    Public Property Legajo() As Integer
        Get
            Return Vlegajo
        End Get
        Set(ByVal value As Integer)
            Vlegajo = value
        End Set
    End Property
    Private Vnombre As String
```

```

Public Property Nombre() As String
    Get
        Return Vnombre
    End Get
    Set(ByVal value As String)
        Vnombre = value
    End Set
End Property
Private Vapellido As String
Public Property Apellido() As String
    Get
        Return Vapellido
    End Get
    Set(ByVal value As String)
        Vapellido = value
    End Set
End Property
Sub New()
End Sub
Sub New(pLegajo As Integer, pNombre As String, pApellido As String)
    Me.Legajo = pLegajo
    Me.Nombre = pNombre
    Me.Apellido = pApellido
End Sub
Public Function Clone() As Object Implements System.ICloneable.Clone
    Return Me.MemberwiseClone
End Function
End Class

```

Ahora veamos el código que demuestra la forma en que funciona la clonación simple o superficial:

```

Public Module Modulo1
Sub Main()
    Dim VprofesorX As Profesor
    Dim VprofesorY As New Profesor("1000", "Guillermo", "Romano")

    VprofesorX = VprofesorY.Clone

    Console.WriteLine("Estado inmediato después de la clonación.")
    Console.WriteLine("Los estados de los objetos apuntados por " & _
                      "las variables VprofesorX y VprofesorY " & _
                      "son iguales" & vbCrLf)

    Console.WriteLine("El legajo de VprofesorX es: " & _
                      VprofesorX.Legajo & vbCrLf & _
                      "El nombre de VprofesorX es: " & _
                      VprofesorX.Nombre & vbCrLf & _
                      "El apellido de VprofesorX es: " & _

```

```

        VprofesorX.Apellido & vbCrLf)
Console.WriteLine("El legajo de VprofesorY es: " & _
                  VprofesorY.Legajo & vbCrLf & _
                  "El nombre de VprofesorY es: " & _
                  VprofesorY.Nombre & vbCrLf & _
                  "El apellido de VprofesorY es: " & _
                  VprofesorY.Apellido & vbCrLf)

Console.WriteLine("* Ahora le cambiamos el nombre a VprofesorX. " & _
                  "Le colocamos Fabian.")

VprofesorX.Nombre = "Pedro"

Console.WriteLine("* Los estados de los objetos apuntados por " & _
                  "las variables VprofesorX y VprofesorY no " & _
                  "son iguales.")
Console.WriteLine("* Esto es debido al cambio realizado en " & _
                  "VprofesorX.")
Console.WriteLine("* Queda demostrado que las variables apuntan " & _
                  "a objetos distintos." & vbCrLf)
Console.WriteLine("El legajo de VprofesorX es: " & _
                  VprofesorX.Legajo & vbCrLf & _
                  "El nombre de VprofesorX es: " & _
                  VprofesorX.Nombre & vbCrLf & _
                  "El apellido de VprofesorX es: " & _
                  VprofesorX.Apellido & vbCrLf)

Console.WriteLine("El legajo de VprofesorY es: " & _
                  VprofesorY.Legajo & vbCrLf & _
                  "El nombre de VprofesorY es: " & _
                  VprofesorY.Nombre & vbCrLf & _
                  "El apellido de VprofesorY es: " & _
                  VprofesorY.Apellido & vbCrLf)

        Console.ReadKey()
End Sub
End Module

```

Es interesante analizar lo que hubiera ocurrido si en lugar de clonar se hubiera realizado una asignación. Los estados de ambos objetos quedarían iguales ya que ambas variables estarían apuntando al mismo objeto. Los cambios en uno de ellos se verían reflejados en el otro. A continuación, el código que demuestra esta situación:

```

Public Module Modulo1
Sub Main()
    Dim VprofesorX As Profesor
    Dim VprofesorY As New Profesor("1000", "Guillermo", "Romano")

    VprofesorX = VprofesorY

```

```

Console.WriteLine("*. Estado inmediato después de la clonación.")
Console.WriteLine("*. Los valores mostrados por ambas variables " & _
                  "VprofesorX y VprofesorY son iguales" & vbCrLf)

Console.WriteLine("El legajo de VprofesorX es: " & _
                  VprofesorX.Legajo & vbCrLf & _
                  "El nombre de VprofesorX es: " & _
                  VprofesorX.Nombre & vbCrLf & _
                  "El apellido de VprofesorX es: " & _
                  VprofesorX.Apellido & vbCrLf)
Console.WriteLine("El legajo de VprofesorY es: " & _
                  VprofesorY.Legajo & vbCrLf & _
                  "El nombre de VprofesorY es: " & _
                  VprofesorY.Nombre & vbCrLf & _
                  "El apellido de VprofesorY es: " & _
                  VprofesorY.Apellido & vbCrLf)
Console.WriteLine("*. Ahora le cambiamos el nombre a VprofesorX. " & _
                  "Le colocamos Fabian.")

VprofesorX.Nombre = "Pedro"

Console.WriteLine("*. Los valores mostrados por ambas variables " & _
                  "VprofesorX y VprofesorY siguen siendo iguales.")
Console.WriteLine("*. Ambos han cambiado (nombre = Pedro) debido " & _
                  "a que se realizó una asignación " & _
                  "en lugar de una clonación.")
Console.WriteLine("*. Queda demostrado que las variables apuntan " & _
                  "al mismo objeto." & vbCrLf)
Console.WriteLine("El legajo de VprofesorX es: " & _
                  VprofesorX.Legajo & vbCrLf & _
                  "El nombre de VprofesorX es: " & _
                  VprofesorX.Nombre & vbCrLf & _
                  "El apellido de VprofesorX es: " & _
                  VprofesorX.Apellido & vbCrLf)
Console.WriteLine("El legajo de VprofesorY es: " & _
                  VprofesorY.Legajo & vbCrLf & _
                  "El nombre de VprofesorY es: " & _
                  VprofesorY.Nombre & vbCrLf & _
                  "El apellido de VprofesorY es: " & _
                  VprofesorY.Apellido & vbCrLf)

Console.ReadKey()

End Sub
End Module

```

Habíamos dejado planteado el problema que enfrenta la clonación simple o superficial con respecto a los valores de tipo referencia. Para poder comprender más profundamente sus implicancias, supongamos que a la clase **Profesor** que utilizamos anteriormente la modificamos para

que también posea una referencia a otro profesor que oficia como su supervisor. La asignación o lectura del supervisor de un profesor se realizará por medio de una propiedad.

El siguiente código nos permite observar los efectos que produce la clonación simple dada la situación donde se presentan valores de tipo referencia (el supervisor).

El código de la clase Profesor queda de la siguiente forma:

```
Public Class Profesor
    Implements ICloneable
    Private Vlegajo As Integer
    Public Property Legajo() As Integer
        Get
            Return Vlegajo
        End Get
        Set(ByVal value As Integer)
            Vlegajo = value
        End Set
    End Property
    Private Vnombre As String
    Public Property Nombre() As String
        Get
            Return Vnombre
        End Get
        Set(ByVal value As String)
            Vnombre = value
        End Set
    End Property
    Private Vapellido As String
    Public Property Apellido() As String
        Get
            Return Vapellido
        End Get
        Set(ByVal value As String)
            Vapellido = value
        End Set
    End Property
    Private Vsupervisor As Profesor
    Public Property Supervisor() As Profesor
        Get
            Return Vsupervisor
        End Get
        Set(ByVal value As Profesor)
            Vsupervisor = value
        End Set
    End Property
    Sub New()
    End Sub
    Sub New(pLegajo As Integer, pNombre As String, pApellido As String)
```

```

Me.Legajo = pLegajo
Me.Nombre = pNombre
Me.Apellido = pApellido
End Sub
Public Function Clone() As Object Implements System.ICloneable.Clone
    Return Me.MemberwiseClone
End Function
End Class

```

El módulo para probar y verificar los efectos causados es:

```

Public Module Modulo1
    Sub Main()
        Dim VprofesorX As Profesor
        Dim VprofesorY As New Profesor("1000", "Guillermo", "Romano")

        VprofesorY.Supervisor = New Profesor("2000", "Pedro", "Garcia")

        VprofesorX = VprofesorY.Clone

        Console.WriteLine("Estado inmediato después de la clonación.")
        Console.WriteLine("Los estados de los objetos apuntados por " & _
                           "las variables VprofesorX y VprofesorY " & _
                           "son iguales" & vbCrLf)

        Console.WriteLine("El legajo de VprofesorX es: " & _
                           VprofesorX.Legajo & vbCrLf & _
                           "El nombre de VprofesorX es: " & _
                           VprofesorX.Nombre & vbCrLf & _
                           "El apellido de VprofesorX es: " & _
                           VprofesorX.Apellido & vbCrLf & _
                           "El legajo del supervisor de VprofesorX es: " & _
                           VprofesorX.Supervisor.Legajo & vbCrLf & _
                           "El nombre del supervisor de VprofesorX es: " & _
                           VprofesorX.Supervisor.Nombre & vbCrLf & _
                           "El apellido del supervisor de VprofesorX es: " & _
                           VprofesorX.Supervisor.Apellido & vbCrLf)

        Console.WriteLine("El legajo de VprofesorY es: " & _
                           VprofesorY.Legajo & vbCrLf & _
                           "El nombre de VprofesorY es: " & _
                           VprofesorY.Nombre & vbCrLf & _
                           "El apellido de VprofesorY es: " & _
                           VprofesorY.Apellido & vbCrLf & _
                           "El legajo del supervisor de VprofesorY es: " & _
                           VprofesorY.Supervisor.Legajo & vbCrLf & _
                           "El nombre del supervisor de VprofesorY es: " & _
                           VprofesorY.Supervisor.Nombre & vbCrLf & _
                           "El apellido del supervisor de VprofesorY es: " & _
                           VprofesorY.Supervisor.Apellido & vbCrLf)
    End Sub
End Module

```

```

Console.WriteLine("* Ahora le cambiamos el nombre al supervisor " & _
                  "de VprofesorX y le colocamos Marcelo.")

VprofesorX.Supervisor.Nombre = "Marcelo"

Console.WriteLine("* Consultamos en nombre del supervisor " & _
                  "de VprofesorX. " & vbCrLf)
Console.WriteLine(VprofesorX.Supervisor.Nombre & vbCrLf)
Console.WriteLine("* Consultamos en nombre del supervisor " & _
                  "de VprofesorY. " & vbCrLf)
Console.WriteLine(VprofesorY.Supervisor.Nombre & vbCrLf)

Console.WriteLine("* Podemos observar que los nombres de los " & _
                  "supervisores coinciden a pesar de haber " & _
                  "clonado los objetos y solo haberle cambiado " & _
                  "el nombre a VprofesorX.")
Console.WriteLine("* Esto es debido a que la clonación simple no " & _
                  "soluciona el problema de los valores de tipo " & "referencia")

Console.ReadKey()
End Sub
End Module

```

Se puede deducir que al realizar una clonación simple como la realizada de VprofesorY en VprofesorX, si VprofesorY posee valores de tipo referencia se clona la referencia (en nuestro caso la referencia al supervisor) pero no se crea un nuevo objeto para representar al supervisor. Esto causa que ambos profesores queden apuntando al mismo supervisor y al cambiarle el nombre al supervisor desde VprofesorX, el efecto se perciba desde VprofesorY ya que, como se mencionó, es el mismo supervisor para ambos.

Para solucionar este efecto, debemos reacondicionar la clase Profesor y adecuar el método Clone para que realice una clonación profunda o completa. Esto se logra reprogramando el método Clone con una rutina recursiva. A continuación, se muestra cómo quedaría la función ajustada.

Función Clone de la clase Profesor:

```

Public Function Clone() As Object Implements System.ICloneable.Clone
    Dim Vp As Profesor = Me.MemberwiseClone()
    If Not (Vp.Supervisor Is Nothing) Then
        Vp.Supervisor = Me.Supervisor.Clone
    End If
    Return Vp
End Function

```

Es oportuno aclarar que en la función recién analizada, lo primero que se hace es una clonación simple (Dim Vp As Profesor = Me.MemberwiseClone()). Luego, al objeto clonado se le pregunta si tiene supervisor (If Not(Vp.Supervisor Is Nothing) Then) y en caso afirmativo, se clona el

supervisor. Si ese supervisor clonado a su vez posee un supervisor, la recursividad se encargará de ello, clonando todo el grafo de objetos hasta que se llegue a un objeto profesor que no posea supervisor.

Es importante destacar que esta rutina generada para supervisor se debería replicar para cada valor de tipo referencia que posea la clase Profesor. Por ejemplo, si los profesores también reportan a un profesor que es su director, en la función Clone deberíamos hacer para Director lo mismo que construimos para Supervisor.

Módulo de prueba para observar el funcionamiento de la clonación profunda o completa:

```

Public Module Modulo1
    Sub Main()
        Dim VprofesorX As Profesor
        Dim VprofesorY As New Profesor("1000", "Guillermo", "Romano")

        VprofesorY.Supervisor = New Profesor("2000", "Pedro", "Garcia")

        VprofesorX = VprofesorY.Clone

        Console.WriteLine("/* Estado inmediato después de la clonación.*/)
        Console.WriteLine("/* Los estados de los objetos apuntados por " & _
                           "las variables VprofesorX y VprofesorY " & _
                           "son iguales" & vbCrLf)

        Console.WriteLine("El legajo de VprofesorX es: " & _
                           VprofesorX.Legajo & vbCrLf & _
                           "El nombre de VprofesorX es: " & _
                           VprofesorX.Nombre & vbCrLf & _
                           "El apellido de VprofesorX es: " & _
                           VprofesorX.Apellido & vbCrLf)

        Console.WriteLine("El legajo del supervisor de VprofesorX es: " & _
                           VprofesorX.Supervisor.Legajo & vbCrLf & _
                           "El nombre del supervisor de VprofesorX es: " & _
                           VprofesorX.Supervisor.Nombre & vbCrLf & _
                           "El apellido del supervisor de VprofesorX es: " & _
                           VprofesorX.Supervisor.Apellido & vbCrLf)

        Console.WriteLine("El legajo de VprofesorY es: " & _
                           VprofesorY.Legajo & vbCrLf & _
                           "El nombre de VprofesorY es: " & _
                           VprofesorY.Nombre & vbCrLf & _
                           "El apellido de VprofesorY es: " & _
                           VprofesorY.Apellido & vbCrLf)

        Console.WriteLine("El legajo del supervisor de VprofesorY es: " & _
                           VprofesorY.Supervisor.Legajo & vbCrLf & _
                           "El nombre del supervisor de VprofesorY es: " & _
                           VprofesorY.Supervisor.Nombre & vbCrLf & _
                           "El apellido del supervisor de VprofesorY es: " & _
                           VprofesorY.Supervisor.Apellido & vbCrLf)

        Console.WriteLine("/* Ahora le cambiamos el nombre al supervisor " & _
                           "de VprofesorX. Le colocamos Marcelo.")
    End Sub

```

```

VprofesorX.Supervisor.Nombre = "Marcelo"

Console.WriteLine("* Consultamos en nombre del supervisor " & _
                  "de VprofesorX. " & vbCrLf)
Console.WriteLine(VprofesorX.Supervisor.Nombre & vbCrLf)
Console.WriteLine("* Consultamos en nombre del supervisor " & _
                  "de VprofesorY. " & vbCrLf)
Console.WriteLine(VprofesorY.Supervisor.Nombre & vbCrLf)

Console.WriteLine("* Podemos observar que los nombres de " & _
                  "los supervisores no coinciden ya que ahora " & _
                  "cada supervisor es un objeto distinto gracias " & _
                  "a utilizar la clonación profunda o completa.")

Console.ReadKey()
End Sub
End Module

```

5.5.4 Las interfaces IEnumerable e IEnumerator

Las interfaces `IEnumerable` e `IEnumerator` se utilizan en forma conjunta para aprovechar el `For ... Each` e iterar una colección.

La interfaz `IEnumerable` expone al enumerador, que admite una iteración simple en una colección no genérica. Si se desea trabajar con clases genéricas, recomendamos utilizar `IEnumerable(of T)` e `IEnumerator(of T)`.

La forma básica en que operan estas interfaces con el `For ... Each` es la siguiente: el `For ... Each` le solicita a la clase `IEnumerable` un enumerador. Para poder brindárselo, `IEnumerable` instancia un `IEnumerator` y se lo retorna al `For ... Each` de manera que este pueda interactuar con la funcionalidad del `IEnumerator`. Podemos decir que de alguna manera, el que decide qué `IEnumerator` se le da al `For ... Each` es `IEnumerable`, ya que es el responsable de exponerlo y de definir, dadas determinadas circunstancias, qué `IEnumerator` es el más adecuado.

Para ver estas interfaces en acción, construiremos un ejemplo donde se necesita iterar con un `For ... Each` las partes constitutivas de un código de barra EAN-13 y en cada iteración se espera obtener una de las partes constituyentes del código en el siguiente orden: país, empresa, producto y dígito verificador.

Para desarrollar el ejemplo necesitaremos dos clases. La primera, responsable de exponer al enumerador, se denominará `ExponeEnumeratorEAN13`. La segunda, denominada `EnumeratorEAN13`, será la responsable de contener la lógica para retornar los elementos enumerados que se espera obtener del código de barra.

Veamos como lo hace:

```

Class ExponeEnumeratorEAN13
    Implements IEnumerable

```

```

Private Vean As String
Private Vlargo() As Integer
Private Vposicion() As Integer
Private Vleyenda() As String
Sub New(pEan As String, pLargo As Integer(), _
         pPosicion As Integer(), pLeyenda As String())
    Me.Vean = pEan
    Me.Vlargo = pLargo
    Me.Vposicion = pPosicion
    Me.Vleyenda = pLeyenda
End Sub
Public Function GetEnumerator() As System.Collections.IEnumerator _
            Implements System.Collections.IEnumerable.GetEnumerator
    Return New EnumeradorEAN13(Me.Vean, Me.Vlargo, Me.Vposicion, Me.Vleyenda)
End Function
End Class

```

La clase `ExponeEnumeradorEAN13` implementa la interfaz `IEnumerable`. Posee un constructor que recibe los siguientes datos al instanciar un objeto:

<code>pEan</code>	Código EAN-13
<code>pLargo</code>	Un Array con el tamaño de cada elemento que se desea obtener del código EAN-13: 3 para país 4 para empresa 5 para producto 1 para dígito verificador
<code>pPosicion</code>	Un Array con la posición donde comienza cada elemento a obtener dentro del código EAN-13: 0 para país 3 para empresa 7 para producto 12 para dígito verificador
<code>pLeyenda</code>	Un Array con las leyendas que acompañarán a cada elemento del código EAN-13 retornado: País Empresa Producto DV

Luego, en la función `GetEnumerator`, implementada debido a la definición que posee la interfaz `IEnumerable`, se instancia un objeto `EnumeradorEAN13`, que es del tipo `IEnumerator` debido a que implementa esa interfaz y se le pasa a su constructor todos los datos recibidos en el constructor de `ExponeEnumeradorEAN13` con el objetivo de que cuente con la información necesaria para

realizar el trabajo requerido. Luego, este objeto (la instancia de EnumeradorEAN13) es retornado por la función GetEnumerator a su llamador. De esta manera, el objeto que le haya solicitado a ExponeEnumeradorEAN13 un enumerador, lo obtiene.

A continuación, veamos como opera la clase EnumeradorEAN13.

```
Class EnumeradorEAN13
    Implements IEnumerable
    Private Vean As String
    Private Vposicion() As Integer
    Private Vlargo() As Integer
    Private Vleyenda() As String

    Sub New(pEan As String, pLargo As Integer(), _
             pPosicion As Integer(), pLeyenda As String())
        Me.Vean = pEan
        Me.Vlargo = pLargo
        Me.Vposicion = pPosicion
        Me.Vleyenda = pLeyenda
    End Sub
    Dim VcadenaActual As String
    Dim Vpasada As Integer
    Public ReadOnly Property Current As Object _
        Implements System.Collections.IEnumerable.Current
        Get
            Return VcadenaActual
        End Get
    End Property

    Public Function MoveNext() As Boolean _
        Implements System.Collections.IEnumerable.MoveNext
        Dim Vtemp2 As String = ""
        'Determina si el dígito verificador que se le colocó al código es
        'correcto.
        If Vpasada = 3 Then
            Dim Vtemp As Integer = 0
            For X = 0 To 11
                Vtemp += IIf(Not (X Mod 2 = 0), Me.Vean.Substring(X, 1) * 3, _
                            Me.Vean.Substring(X, 1) * 1)
            Next
            Vtemp2 = IIf((((Vtemp \ 10) * 10) + 10) - Vtemp) = _
                      Me.Vean.Substring(12, 1), "OK", "OUT")
        End If
        If Vpasada <= 3 Then
            Me.VcadenaActual = _
                Me.Vleyenda(Vpasada) & vbTab & IIf( _
                    Me.Vleyenda(Vpasada).Length < 8, vbTab, "") & _
                    Me.Vean.Substring(Me.Vposicion(Vpasada), _
                    Me.Vlargo(Vpasada)) & Vtemp2
        End If
    End Function
```

```

    Vpasada += 1 : Return IIf(Vpasada = 5, False, True)
End Function

Public Sub Reset() Implements System.Collections.IEnumerator.Reset
End Sub
End Class

```

La clase `EnumeradorEAN13` recibe en su constructor `pEan`, `pLargo`, `pPosicion` y `pLeyenda` descriptos con anterioridad. Además, implementa la propiedad de solo lectura `Current` que retorna un `Object`, en nuestro caso más específicamente un `String` que contendrá el elemento actualmente obtenido del código EAN-13. Como hemos mencionado anteriormente, este elemento será, en la primera iteración, el país, en la segunda la empresa, en la tercera el producto y finalmente, en la cuarta, el dígito verificador.

También se encuentra implementado el procedimiento `Reset`, que en nuestro caso no posee código pues no ha sido necesario. En él se podrían reiniciar valores de variables u otros elementos que necesiten ser tratados de alguna forma peculiar.

Por último, podemos observar la función `MoveNext`, que retorna un valor `Boolean`. Esta función es la que realmente posee toda la inteligencia para poder tratar un código EAN-13 y obtener de él los elementos requeridos. El retorno de tipo `Boolean`, cuando es verdadero, estará indicando que se ha cargado un elemento a `VcadenaActual` para que se pueda consultar su valor a través de `Current`. Por el contrario, un valor `False` indicará que ya no hay más elementos por recorrer.

Esta función primero indaga qué valor posee la variable `Vpasada`. Mientras sea inferior a tres, no determina si el dígito verificador que se colocó al código EAN-13 es correcto, solo lo hará cuando `Vpasada = 3`, o sea, exactamente antes de obtener el cuarto y último elemento (`Vpasada` trabajará de 0 a 3 para cada uno de los elementos).

Luego, evalúa si `Vpasada <= 3`. Si es verdadero, recorta del código EAN-13 el elemento que corresponde y lo almacena en `VcadenaActual`, incrementa `Vpasada` en 1 y retorna `True`. Esto lo realiza hasta que `Vpasada` sea igual a 5, lo que significa que ya no hay elementos a recorrer.

El código que prueba lo expresado se encuentra en el siguiente módulo:

```

Public Module Modulo1
    Public Sub Main()
        Dim Vres As String = ""
        Dim Vcod As String = "4902778122433"
        Dim Vpos() As Integer = {0, 3, 7, 12}
        Dim Vlar() As Integer = {3, 4, 5, 1}
        Dim Vley() As String = {"País", "Empresa", "Producto", "DV"}
        Console.WriteLine("El código EAN ingresado es: " & Vcod & vbCrLf)
        For Each Velemento As String In New ExponeEnumeradorEAN13(Vcod, _
            Vlar, Vpos, Vley)
            Vres += Velemento & vbCrLf
        Next
        Console.WriteLine("Los datos obtenidos son: " & Vres)
    End Sub
End Module

```

```
    Next
    Console.WriteLine(Vres)
    Console.ReadLine()
End Sub
End Module
```

5.6. Tipos

Al abordar los conceptos referidos a los tipos de datos, se deben considerar dos aspectos fundamentales. El primero se refiere a la clasificación que dan los tipos, siendo esta la que establece “qué es” un determinado elemento. El segundo son los manejos que podemos hacer administrando adecuadamente los tipos para lograr una programación más robusta.

En la tecnología tratada existen dos grandes grupos. El primero se denomina *tipos de valor (value type)* y el segundo *tipos de referencia (reference type)*.

Los tipos de valor son: Byte, Short, Integer, Long, Byte, UShort, UInteger, ULong, Boolean, Char, Date y las **estructuras**.

Los **tipos de referencia** son las **clases**, las **matrices**, los **delegados** y el tipo **string**. Dentro de los tipos de referencia existe uno denominado “**Object**”. Este tipo es el más abstracto y donde se comienza a construir la jerarquía “es-un” dentro del grupo de los tipos de referencia.

Los tipos son utilizados, por ejemplo, cuando se desean declarar variables. Esto le otorgará a la variable sus condiciones respecto a qué pueden contener o apuntar. En la tecnología que estamos trabajando, se pueden declarar variables sin tipos. Las declaraciones y asignación de tipos pueden hacerse de manera implícita (no se indica el tipo y el compilador la construye como **Object**) o explícita (se indica el tipo).

Declaración implícita:

```
Dim Vnombre
```

Declaración explícita:

```
Dim Vnombre As String
```

Tipos genéricos

Usar tipos genéricos permite que un mismo fragmento de código adapte su funcionalidad a una multiplicidad de tipos. Por ejemplo, si se desea que una lista se adapte a trabajar con un tipo de dato en particular, podemos construirla de la siguiente manera.

```
Public Class Form1
    Private Sub Form1_Load(sender As System.Object, e As System.EventArgs) _
        Handles MyBase.Load
        Dim Vlista As New List(Of Integer)
        Vlista.AddRange({1, 3, 6, 9, 10})
        MsgBox("El promedio de los valores de la lista es: " & Vlista.Average)
    End Sub
End Class
```

En el código anterior, la lista queda preparada para trabajar con tipos `integer`. Se puede hacer que una lista reciba el tipo con el que trabajará dinámicamente, lo que hace a la rutina más flexible y reutilizable. A continuación, el código que lo ejemplifica.

```
Public Class ListaDinamica(Of T)
    Dim Vlista As New List(Of T)
    Public Sub CargarDatos(pData As T)
        Try
            If TypeOf pData Is T Then : Vlista.Add(pData)
            Else : MsgBox("Tipo Erroneo") : End If
        Catch ex As Exception
            MsgBox(ex.Message)
        End Try
    End Sub
    Public Sub BorrarDatos(pData As T)
        Try
            If TypeOf pData Is T Then : Vlista.Remove(pData)
            Else : MsgBox("Tipo Erroneo") : End If
        Catch ex As Exception
            MsgBox(ex.Message)
        End Try
    End Sub
    Public Function RetornaDatos() As List(Of T)
        Return Vlista
    End Function
End Class
```

Se puede observar como la clase `ListaDinamica` está acompañada de (`of T`), donde `T` será un tipo a recibir que será utilizado para definir el tipo de la lista: `Dim Vlista as New List(of T)`. Luego, los métodos `CargarDatos`, `BorrarDatos` y `RetornaDatos` también aprovechan el tipo `T` para tipar en los dos primeros casos al parámetro `pDatos` y en el último la lista que se retorna.

El código siguiente muestra cómo se usa la clase `ListaDinamica(Of T)`, configurándola para que funcione con datos del tipo `Integer` y `String`.

```
Public Class Form1
    Private Sub Form1_Load(sender As System.Object, e As System.EventArgs) _
        Handles MyBase.Load
        MsgBox("Ejemplo con Integer")
        Dim Ldi As New ListaDinamica(Of Integer)
        Ldi.CargarDatos(3) : Ldi.CargarDatos(6) : Ldi.CargarDatos(9)
        Ldi.BorrarDatos(6)
        Dim Vti As String = ""
        For Each Vldi As Integer In Ldi.RetornaDatos
            Vti += Vldi & vbCrLf
        Next
        MsgBox(Vti)

        MsgBox("Ejemplo con String")
        Dim Lds As New ListaDinamica(Of String)
        Lds.CargarDatos("A") : Lds.CargarDatos("C") : Lds.CargarDatos("E")
        Lds.BorrarDatos("C")
        Dim Vts As String = ""
        For Each Vlds As String In Lds.RetornaDatos
            Vts += Vlds & vbCrLf
        Next
        MsgBox(Vts)
    End Sub
End Class
```

Se presentará un ejemplo más para profundizar el concepto y la utilidad de usar tipos genéricos. En este caso, se generará una clase ordenadora genérica para ordenar una lista de objetos, donde el tipo de la lista se define dinámicamente y el ordenador podrá ordenar por cualquier tipo de orden asociado al tipo de la lista según la necesidad del programador. Para ello, se utilizan las clases `Ordenador`, que es el ordenador genérico; `Persona`, que es el tipo que tendrán los objetos que se desean ordenar; `Orden1`, que representa uno de los criterios de orden que se desea tener y `Orden2`, que representa a otro tipo de orden. Se podrían tener muchos tipos de ordenamientos definidos, tantos como sean necesarios para cada tipo. Se presentarán por orden las clases intervinientes.

La clase `Persona` posee las propiedades `Nombre` y `Edad`. También un constructor que permite ingresar estos valores cuando se instancia un objeto. De manera anidada, contiene las clases `Orden1` y `Orden2`, que son los criterios de orden deseados para una persona. En este caso

particular, Orden1 ordena de manera ascendente por edad mientras que Orden2 lo hace de forma descendente. Se utilizó para esto la interfaz Icomparer que provee el framework utilizado. A continuación, el código de lo descripto.

```
Public Class Persona
    Sub New(pNombre As String, pEdad As Byte)
        Nombre = pNombre
        Edad = pEdad
    End Sub
    Private Vnombre As String
    Public Property Nombre() As String
        Get
            Return Vnombre
        End Get
        Set(ByVal value As String)
            Vnombre = value
        End Set
    End Property
    Private Vedad As Byte
    Public Property Edad() As Byte
        Get
            Return Vedad
        End Get
        Set(ByVal value As Byte)
            Vedad = value
        End Set
    End Property
End Class

Public Class Orden1
    Implements IComparer(Of Persona)
    Public Function Compare(x As Persona, y As Persona) As Integer _
        Implements System.Collections.Generic.IComparer(Of Persona).Compare
        If x Is Nothing And y Is Nothing Then Return 0
        If x Is Nothing Then Return 1
        If y Is Nothing Then Return -1
        Return x.Edad.CompareTo(y.Edad)
    End Function
End Class

Public Class Orden2
    Implements IComparer(Of Persona)
    Public Function Compare(x As Persona, y As Persona) As Integer _
        Implements System.Collections.Generic.IComparer(Of Persona).Compare
        If x Is Nothing And y Is Nothing Then Return 0
        If x Is Nothing Then Return -1
        If y Is Nothing Then Return 1
        Return y.Edad.CompareTo(x.Edad)
    End Function
End Class
End Class
```

La clase Ordenador(Of T) permite que al instanciar un objeto de este tipo, le pasemos un tipo al parámetro T. Este tipo T será el utilizado para otorgarle el tipo a la lista Vlista, a los parámetros pElemento de los procedimientos Agregar y Eliminar y a los tipos de las listas retornadas por las funciones Ver y Ordenname. También está la propiedad Orden, que recibe un valor del tipo Type, el cual representa a la clase que posee el ordenamiento que se desea utilizar para ordenar la lista de objetos que son instancias de ella.

```
Public Class Ordenador(Of T)
    Dim Vlista As New List(Of T)
    Dim O As Object

    Private Vorden As Type
    Public Property Orden() As Type
        Get
            Return Vorden
        End Get
        Set(ByVal value As Type)
            Vorden = value
            O = Activator.CreateInstance(value)
        End Set
    End Property

    Public Sub Agregar(pElemento As T)
        Vlista.Add(pElemento)
    End Sub

    Public Sub Eliminar(pElemento As T)
        Vlista.Remove(pElemento)
    End Sub

    Public Function Ver() As List(Of T)
        Return (Vlista)
    End Function

    Public Function Ordenname() As List(Of T)
        If Not O Is Nothing Then Vlista.Sort(O)
        Return Vlista
    End Function
End Class
```

Finalmente, se encuentra el código que utiliza y aprovecha las posibilidades de la clase Ordenador. Para esto, se han instanciado cuatro personas, una de ellas sin edad (se asumirá como edad cero). Luego, se instanció O, un objeto de tipo Ordenador(Of Persona), lo que indica que servirá para ordenar una lista de personas. El siguiente paso es cargar el criterio de orden en la propiedad Orden del objeto O, lo que se logra pasándole a O.Orden el tipo de la clase que contiene el orden deseado, GetType(Persona.Orden1). Las siguientes líneas de código agregan las personas al objeto Ordenador, y se puede observar que además se agrega una

persona en Nothing (a modo de ejemplo), que será ignorada. Finalmente, se itera el retorno de `O.Ordenename` que devuelve una lista ordenada.

Primero se ve funcionar en orden ascendente y luego descendente.

```

Public Class form1
    Private Sub form1_Load(sender As Object, e As System.EventArgs) Handles Me.Load
        Dim p1 As New Persona("Juan", 22)
        Dim p2 As New Persona("Pedro", 1)
        Dim p3 As New Persona("Maria", 15)
        Dim p4 As New Persona("Ariel", Nothing)
        Dim O As New Ordenador(Of Persona)

        MsgBox("Orden Ascendente")
        O.Orden = GetType(Persona.Orden1)
        O.Agregar(p1)
        O.Agregar(p2)
        O.Agregar(p3)
        O.Agregar(p4)
        O.Agregar(Nothing)
        For Each X As Persona In O.Ordenename
            If Not X Is Nothing Then MsgBox(X.Edad & " " & X.Nombre)
        Next

        MsgBox("Orden Descendente")
        O.Orden = GetType(Persona.Orden2)
        For Each X As Persona In O.Ordenename
            If Not X Is Nothing Then MsgBox(X.Edad & " " & X.Nombre)
        Next
    End Sub
End Class

```

5.7. Comunicación entre aplicaciones distribuidas

Las aplicaciones distribuidas se denominan de esa forma debido a que se encuentran situadas en distintos puntos de una red (LAN, MAN, WAN) y existe una necesidad concreta de comunicación entre ellas con el objetivo de recibir información, enviar información, recibir y enviar información o, simplemente, sincronizarse para una tarea.

Dadas dos aplicaciones en dos computadoras diferentes, para que se puedan conectar necesitamos los siguientes elementos:

- Que las computadoras sean identificables (en una red TCP/IP, será el número de IP).
- Un protocolo de transporte (TCP).
- Los puertos o direcciones donde operan las aplicaciones.

Este conjunto de elementos se denomina *socket*. Por lo tanto, un socket contendrá las direcciones IP del equipo local y del equipo remoto, el protocolo utilizado y los puertos del equipo local y del equipo remoto.

En general, cuando una aplicación se va a conectar con otra, sucede que una está en modo de escucha y la otra intentará conectarse. Se dice que el cliente intenta conectarse al *servidor*. Esto permite afirmar que la utilización de socket nos habilita a comunicar aplicaciones en modo *cliente-servidor*.

El término servidor denota la idea de que la aplicación que así actúa dará un servicio al cliente y podría ser así, pero lo que veremos más adelante demostrará que una vez establecida la conexión, se torna indiferente el rol de cliente o servidor desde el punto de vista técnico, quedando esta clasificación sujeta a la función que cumpla la aplicación y no a la tarea desarrollada al momento de establecer la conexión.

Antes de que las aplicaciones estén conectadas, una adoptará el rol de servidor y la otra de cliente, con el objetivo de que el servidor se predisponga a aceptar una conexión por parte del cliente. Ocurrido esto, y establecida la conexión deseada entre las aplicaciones, como mencionamos en el párrafo anterior, dependerá de las tareas que efectivamente realicen para ser consideradas cliente, servidor o cliente-servidor.

Para estas conexiones se podrá utilizar el protocolo TCP (*Transmission Control Protocol*) o el UDP (*User Datagram Protocol*), ambos protocolos de transporte. Una de las diferencias más importantes entre ambos protocolos es que TCP es orientado a la conexión, mientras que UDP no lo es. Antes de iniciar la comunicación entre las partes, un protocolo orientado a la conexión verifica algunos datos como disponibilidad, alcance y credenciales para lograr una conexión segura y eficiente. En los protocolos no orientados a la conexión no existe un acuerdo previo entre las partes antes del envío de la información. Evidentemente, recargan menos a la red pero dan menos garantías.

Si nos conectamos utilizando TCP como protocolo de transporte, tendremos ciertas garantías. En primer lugar, que la transmisión de todos los paquetes se produzca sin errores ni omisiones. La segunda es que todo paquete llegará a su destino en el mismo orden en el que se ha transmitido.

El protocolo TCP usa un número de puerto para identificar a las aplicaciones emisoras y receptoras. El número de puerto oscila entre 0 y 65535 (valor de 16 bit sin signo), y se debe tener la precaución de utilizar puertos libres para comunicar las aplicaciones pues en caso de no hacerlo, se generan errores. Los puertos se encuentran divididos en tres grandes grupos por IANA (*Internet Assigned Numbers Authority*). Los bien conocidos van del 0 al 1023, por ejemplo 80=HTTP, 21=FTP y 25=SMTP. Los registrados van del 1024 al 49151 y los usan aplicaciones de usuarios (software desarrollado por terceras partes para que lo utilicen los usuarios) en forma temporal. Los dinámicos van del 49152 al 65535 y también son utilizados por usuarios pero dentro de una conexión TCP definida, y son menos usados que los del grupo anterior.

Con el protocolo de transporte definido (TCP) y el concepto de puerto para identificar las aplicaciones emisora y receptora, solo resta la identificación de las computadoras. Esto se realizará por medio del protocolo IP. El protocolo IP es una dirección numérica que identifica de

manera lógica y jerárquica a una computadora (en el mundo de las redes se dice que identifican una interfaz) dentro de una red.

Las direcciones IP versión 4, denominado normalmente IPv4, consisten en un número binario de 32 bits. Esto permite direccionar 2^{32} direcciones, equivalente a 4.294.967.296. Estos 32 bits se dividen en octetos, más precisamente 4 octetos de bits. Un octeto de bits permite identificar 256 elementos entre 0 y 255. Ejemplos de direcciones IP podrían ser: 192.100.10.22, 10.1.0.110 y 198.122.100.1.

La versión 6 de IP (denominada IPv6) amplía el rango de direccionamiento llevando esta capacidad a 2^{64} direcciones para una sub red. La dirección completa es de 128 bits pero 64 quedan reservados para el enrutamiento.

Hecha esta breve presentación de los elementos necesarios para comunicar dos aplicaciones, nos adentraremos en los conceptos más cercanos al software que debemos desarrollar para lograrlo.

En colaboración con otras, la clase `Socket` permite comunicar dos aplicaciones de manera sincrónica y asincrónica en una red para hacer la transferencia de datos.

Ejemplo 1. Veamos un ejemplo sencillo que clarifique lo expuesto hasta aquí. Se crearán dos aplicaciones: una que funcione como servidor y otra como cliente. Como ya se ha mencionado oportunamente, estos roles tendrán sentido cuando se establezca la conexión, luego ambas podrán enviar y recibir información.

Aplicación servidor:

```

Imports System
Imports System.Threading
Imports System.Net.Sockets
Imports System.IO
Imports System.Text
Public Class Form1
    'Declaración de una variable de tipo TcpListener
    Dim Vescucha As TcpListener
    'Declaración de una variable para apuntar a un subproceso
    Dim VsubProceso1 As Thread
    'Declaración de una variable para apuntar a un subproceso
    Dim VsubProceso2 As Thread
    'Declaración de una variable de tipo Socket
    Dim Vsocket As Socket
    Private Sub Form1_Load(sender As Object, e As System.EventArgs) Handles Me.Load
        'Se crea un objeto de tipo TcpListener para escuchar conexiones de clientes
        Vescucha = New TcpListener(New System.Net.IPAddress({127, 0, 0, 1}), 8050)
        'Se inicia la escucha
        Vescucha.Start()
        'Se crea un subproceso donde se ejecuta el procedimiento EsperarCliente
        'para que se quede escuchando la solicitud de conexión de un cliente
        VsubProceso1 = New Thread(AddressOf EsperarCliente)
        'Se pone a correr el procedimiento EsperarCliente
    End Sub
End Class

```

```

    VsubProceso1.Start()
End Sub
Public Sub EnviarDatos(ByVal Datos As String)
    'Se envia un mensaje al Cliente
    Vsocket.Send(Encoding.ASCII.GetBytes(Datos))
End Sub
Private Sub EsperarCliente()
    While True
        'Cuando se recibe la petición de conexión se acepta.
        'Se guarda el Socket que utilizo para mantener la conexión con el cliente
        Vsocket = Vescucha.AcceptSocket() 'Se queda esperando la conexión de un cliente
        'Se crea un Thread para que se encargue de escuchar los mensajes del cliente
        VsubProceso2 = New Thread(AddressOf LeerSocket)
        'Se inicia el thread encargado de escuchar los mensajes del cliente
        VsubProceso2.Start()
    End While
End Sub
Private Sub LeerSocket()
    'Se declara un array que se utiliza para recibir los datos que llegan
    Dim Vrecibir() As Byte
    Dim Vret As Integer = 0
    While True
        If Vsocket.Connected Then
            Vrecibir = New Byte(100) {}
            Try
                'Se espera a que llegue un mensaje desde el cliente
                Vret = Vsocket.Receive(Vrecibir, Vrecibir.Length, SocketFlags.None)
                If Vret > 0 Then
                    'Se muestra el mensaje recibido
                    MsgBox(Encoding.ASCII.GetString(Vrecibir), , "Servidor")
                Else
                    Exit While
                End If
            Catch e As Exception
                If Not Vsocket.Connected Then
                    Exit While
                End If
            End Try
        End If
    End While
End Sub
Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) _
Handles Button1.Click
    Me.EnviarDatos("Desde el Servidor -> " & Me.TextBox1.Text)
End Sub
End Class

```

Para construir la aplicación servidor, se deben implementar los siguientes namespaces: `System`, `System.Threading`, `System.Net.Sockets`, `System.IO` y `System.Text`. La estrategia es la siguiente: por medio de un objeto de tipo `TcpListener` (`Vescucha`) se dotará a esta aplicación de las capacidades para poder quedar a la escucha y a la espera de que un cliente solicite conectarse. Para poder lograrlo, se configura pasándole al constructor parámetros que lo ponen a funcionar en la IP 127.0.0.1 y el puerto 8050, y por defecto trabajará con el protocolo de transporte TCP. Luego, se lo pone a funcionar por medio del método `Start`. El siguiente paso es que el procedimiento `EsperarCliente` funcione en un subproceso propio (`VsubProceso1`). Cuando `EsperarCliente` comienza a funcionar (`VsubProceso1.Start`), queda a la espera del pedido de comunicación de un cliente. Cuando esto ocurre, se pone a correr en otro subproceso (`VsubProceso2`) el procedimiento `LeerSocket` para receptionar los mensajes que provienen del cliente. En el procedimiento `LeerSocket` se declara un array de tipo `Byte` denominado `Recibir()`, que tendrá un tamaño de 100 bytes con la intención de utilizarlo para contener lo receptionado en el buffer de recepción (`Vsocket.Receive(Recibir, Recibir.Length, SocketFlags.None)`). Finalmente, si se recibe algo, el método `Receive` del `Socket` además retorna el número de bytes recibidos, que será mayor que cero. En caso de ocurrir esto, se muestra transformando el array de bytes en un `String` (`MsgBox(Encoding.ASCII.GetString(Recibir), , "Servidor")`).

Para enviarle datos al cliente se utiliza el procedimiento `EnviarDatos`. Este utiliza el método `Send` del `Socket` establecido en la conexión, codificando el `String` que se desea enviar como un array de bytes (`Vsocket.Send(Encoding.ASCII.GetBytes(Datos))`).

La interfaz gráfica de la aplicación se verá como la de la figura 5.15.



Figura 5.15.

Para que este ejemplo funcione correctamente, es muy importante que se ejecute la aplicación servidor antes que la del cliente, pues toda la lógica que coloca al servidor a escuchar se encuentra programada en la carga del formulario expuesto anteriormente.

Aplicación cliente:

```

Imports System
Imports System.Threading
Imports System.Net.Sockets
Imports System.IO
Imports System.Text
Public Class Form1
    'Se declara una variable de tipo TcpClient
    Dim VtcpCliente As TcpClient

```

```

'Se declara una variable de tipo Thread
Dim VsubProceso1 As Thread
'Se declara una variable de tipo Stream
Private Vstream As Stream
Private Sub Form1_Load(sender As Object, e As System.EventArgs) Handles Me.Load
    'Se crea una instancia de TcpClient para conectarse con el servidor
    VtcpCliente = New TcpClient()
    'Solicitud de conexión al objeto Servidor determinado por las propiedades
    'IPDelHost y PuertoDelHost
    VtcpCliente.Connect("127.0.0.1", 8050)
    'Se devuelve la NetworkStream usada para enviar y recibir datos.
    Vstream = VtcpCliente.GetStream()
    'Se creo un thread para que escuche los mensajes enviados por el Servidor
    VsubProceso1 = New Thread(AddressOf LeerSocket)
    'Se inicio el subproceso
    VsubProceso1.Start()
End Sub
Private Sub LeerSocket()
    Dim VbufferDeLectura() As Byte
    While True
        Try
            VbufferDeLectura = New Byte(100) {}
            'Se lee la llegada de datos desde el servidor
            Vstream.Read(VbufferDeLectura, 0, VbufferDeLectura.Length)
            'Se muestran los datos recibidos desde el Servidor
            MsgBox(Encoding.ASCII.GetString(VbufferDeLectura), , "Cliente")
        Catch e As Exception
            Exit While
        End Try
    End While
End Sub
Public Sub EnviarDatos(ByVal Datos As String)
    'Se envían los datos al Servidor
    Vstream.Write(Encoding.ASCII.GetBytes(Datos), 0, _
    Encoding.ASCII.GetBytes(Datos).Length)
End Sub
Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) _ 
    Handles Button1.Click
    Me.EnviarDatos("Desde el Cliente -> " & Me.TextBox1.Text)
End Sub
End Class

```

La aplicación cliente es otra aplicación separada de la anterior y también implementa los siguientes namespaces: `System`, `System.Threading`, `System.Net.Sockets`, `System.IO` y `System.Text`.

Esta aplicación le solicita una petición de conexión al servidor por medio de un objeto de tipo `TcpClient` denominado `VtcpCliente`, que es un cliente que utilizará el protocolo TCP. Este cliente, a través del método `Connect`, solicita la conexión al servidor indicando en los parámetros solicitados la IP (127.0.0.1 dirección de *loopback*) y el puerto (8050). Luego, mediante el

método `GetStream()`, se obtiene un `NetworkStream` que es almacenado en una variable de tipo `Stream` denominada `Vstream`. Este `NetworkStream` representa al servidor en los términos de poder enviarle y recibir datos. El próximo paso es ejecutar el procedimiento `LeerSocket` en un procedimiento separado identificado a través de la variable `Vsubproceso1`, que se pone a funcionar por medio de `VsubProceso1.Start()`.

El procedimiento `LeerSocket` establece un array de tipo `Byte` con un tamaño de 100 bytes, que se utilizará para almacenar lo recepcionado enviado desde el servidor. `Vstream` posee un método `Read` que permite leer el buffer de recepción y lo realiza con la siguiente línea de código: `Vstream.Read(BufferDeLectura, 0, BufferDeLectura.Length)`.

Luego, lo recepcionado se codifica como `String` y se muestra con `MsgBox(Encoding.ASCII.GetString(BufferDeLectura), , "Cliente")`.

Para enviar datos se utiliza el procedimiento `EnviarDatos`. Este hace uso del método `Write` de `Vstream`, pasándole como parámetro el `String` a enviar codificado como un array de bytes.

El aspecto de la interfaz gráfica de esta aplicación es la que vemos en la figura 5.16.



Figura 5.16.

Ejemplo 2. Se verá un segundo ejemplo basado en el anterior de manera que una aplicación cliente le pueda solicitar a una aplicación servidor que le envíe el dígito verificador y el valor hash de un código EAN-13. Para ver el ejemplo en funcionamiento, es importante ejecutar primero la aplicación servidor y luego la aplicación cliente. A continuación, el código del servidor:

```

Imports System
Imports System.Threading
Imports System.Net.Sockets
Imports System.IO
Imports System.Text
Imports System.Security.Cryptography
Public Class Form1
    'Declaración de una variable de tipo TcpListener
    Dim Vescucha As TcpListener
    'Declaración de una variable para apuntar a un subproceso
    Dim VsubProceso1 As Thread
    'Declaración de una variable para apuntar a un subproceso
    Dim VsubProceso2 As Thread
    'Declaración de una variable de tipo Socket
    Dim Vsocket As Socket
    Private Sub Form1_Load(sender As Object, e As System.EventArgs) Handles Me.Load
        'Se crea un objeto de tipo TcpListener para escuchar conexiones de clientes
        Vescucha = New TcpListener(New System.Net.IPAddress({127, 0, 0, 1}), 8051)
    End Sub
End Class

```

```

    'Se inicia la escucha
    Vescucha.Start()
    'Se crea un subproceso donde se ejecuta el procedimiento EsperarCliente
    'para que se quede escuchando la solicitud de conexión de un cliente
    VsubProceso1 = New Thread(AddressOf EsperarCliente)
    'Se pone a correr el procedimiento EsperarCliente
    VsubProceso1.Start()
End Sub
Public Sub EnviarDatos(ByVal Datos As String)
    'Se envia un mensaje al Cliente
    Vsocket.Send(Encoding.ASCII.GetBytes(Datos))
End Sub
Private Sub EsperarCliente()
    While True
        'Cuando se recibe la petición de conexión se acepta.
        'Se guarda el Socket que utilizo para mantener la conexión con el cliente
        Vsocket = Vescucha.AcceptSocket() 'Se queda esperando la conexión de un cliente
        'Se crea un Thread para que se encargue de escuchar los mensajes del cliente
        VsubProceso2 = New Thread(AddressOf LeerSocket)
        'Se inicia el thread encargado de escuchar los mensajes del cliente
        VsubProceso2.Start()
    End While
End Sub
Private Sub LeerSocket()
    'Se declara un array que se utiliza para recibir los datos que llegan
    Dim Vrecibir() As Byte
    Dim Vret As Integer = 0
    Dim Vcabecera As String = ""
    Dim Vdatos As String = ""
    Dim ByteConverter As New UnicodeEncoding()

    While True
        If Vsocket.Connected Then
            Vrecibir = New Byte(100) {}
            Try
                'Se espera a que llegue un mensaje desde el cliente
                Vret = Vsocket.Receive(Vrecibir, Vrecibir.Length, SocketFlags.None)
                If Vret > 0 Then
                    'Se evalúa si la cabecera es igual a "COD"
                    If Encoding.ASCII.GetString(Vrecibir).Substring(0, 3) = "COD" Then
                        Vdatos = Encoding.ASCII.GetString(Vrecibir).Substring(3, Vret)
                        Me.EnviarDatos("VER" & Me.DígitoVerificador(Vdatos))
                    ElseIf Encoding.ASCII.GetString(Vrecibir).Substring(0, 3) = "ENC" Then
                        Vdatos = Encoding.ASCII.GetString(Vrecibir).Substring(3, Vret)
                        Dim sha As New SHA1CryptoServiceProvider()
                        Dim result() As Byte =
                            sha.ComputeHash(ByteConverter.GetBytes(Vdatos))
                        Me.EnviarDatos("XNC" & Encoding.ASCII.GetString(result))
                    End If
                Else
            End If
        End While
    End Sub

```

```

        Exit While
    End If
    Catch e As Exception
        If Not Vsocket.Connected Then
            Exit While
        End If
    End Try
End If
End While
End Sub
Private Function DígitoVerificador(pDatos As String) As String
    Dim Vchar() As Char
    Dim Vtot As Integer
    Dim Vn As Byte = 1
    Dim Vd As Integer
    Vchar = pDatos.ToCharArray : ReDim Preserve Vchar(11)
    For Each Vc As Char In Vchar
        Vtot += Char.GetNumericValue(Vc) * Vn
        If Vn = 1 Then : Vn = 3 : Else : Vn = 1 : End If
    Next
    While True
        Vd += 10 : If Vd > Vtot Then Exit While
    End While
    Return Vd - Vtot
End Function
End Class

```

Analicemos el código precedente. Existen los procedimientos `Form1_Load`, `EnviarDatos`, `EsperarCliente` y `LeerSocket` más la función `DígitoVerificador`.

Para que el servidor pueda realizar el trabajo previsto, fue necesario utilizar un objeto `TcpListener` denominado `Vescucha`, dos objetos `Thread` denominados `VsubProceso1` y `VsubProceso2`, y un objeto `Socket` denominado `Vsocket`.

En el procedimiento `Form1_Load` se configura el objeto `TcpListener`, indicándole que deberá ponerse a escuchar en el puerto 8051 de la IP 127.0.0.1 (dirección de *loopback*) para que la aplicación cliente y la aplicación servidor funcionen en el mismo equipo. Luego, por medio del método `Start`, se da inicio a la escucha. Para finalizar esta etapa, se delega en un subproceso (`VsubProceso1`) la ejecución del procedimiento `EsperarCliente` y con el método `Start` del subproceso se lo pone a funcionar, finalizando la configuración y la puesta en funcionamiento del servidor.

El procedimiento `EsperarCliente` pone en funcionamiento un bucle infinito (`While True ... End While`). Este bucle deja escuchando al objeto `TCPListener` en el puerto configurado (`Vescucha.AcceptSocket()`) y al receptionar una petición de conexión la acepta y obtiene un `Socket`, que pasa a ser apuntado por la variable `Vsocket` (`Vsocket=Vescucha.AcceptSocket()`). Al ocurrir esto, se delega la ejecución del procedimiento `LeerSocket` en otro subproceso (`VsubProceso2`) y se lo pone a funcionar por medio del método `Start` del subproceso.

Dentro del procedimiento LeerSocket hay un bucle infinito (`While True ... End While`) que recepcionará lo enviado al servidor, le dará un tratamiento y retornará información al cliente en caso de ser necesario. Para lograr que el servidor pueda gestionar efectivamente el flujo de datos entre sí mismo y el cliente, se estableció una cabecera de tres caracteres en los mensajes recibidos que significan lo siguiente:

“COD”	Solicitud del dígito verificador
“ENC”	Solicitud del valor hash
“VER”	Envío del dígito verificador
“XNC”	Envío del valor hash

Analicemos más en detalle el código de LeerSocket. Si se receptiona una cadena cuyos tres primeros caracteres son “COD”, entonces se le envía al cliente una cadena que comienza con “VER” más el dígito verificador. El dígito verificador es calculado por la función `DigitoVerificador`, que recibe un `String` de 12 caracteres numéricos. El cálculo implica descomponer los 12 caracteres en dígitos individuales y multiplicarlos alternativamente por 1 y por 3, comenzando por la izquierda. Luego, se suman todos los resultados obtenidos y, en el ejemplo, se almacenan en `Vtot`. A continuación, se busca el múltiplo de 10 mayor y más próximo al resultado obtenido, y finalmente se le resta al múltiplo de 10 obtenido el valor de `Vtot`. El resultado obtenido es el dígito verificador.

Si se receptiona una cadena cuyos tres primeros caracteres son “ENC”, utilizando un objeto `SHA1CryptoServiceProvider` se genera un valor hash para el `String` recibido. Luego, se lo envía al cliente anteponiéndole “XNC” a la cadena enviada. Por último, el procedimiento `EnviarDatos` se encarga de enviar los datos haciendo uso del método `Send` del objeto `Socket`.

La interfaz del servidor es la que muestra la figura 5.17.

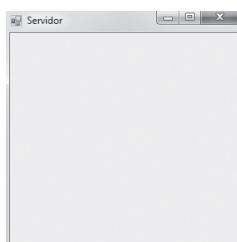


Figura 5.17.

Veamos el código del cliente:

```
Imports System
Imports System.Threading
Imports System.Net.Sockets
```

```
Imports System.IO
Imports System.Text
Public Class Form1
    'Se declara una variable de tipo TcpClient
    Dim VtcpCliente As TcpClient
    'Se declara una variable de tipo Thread
    Dim VsubProceso1 As Thread
    'Se declara una variable de tipo Stream
    Private Vstream As Stream
Private Sub Form1_Load(sender As Object, e As System.EventArgs) Handles Me.Load
    CheckForIllegalCrossThreadCalls = False
    'Se crea una instancia de TcpClient para conectarse con el servidor
    VtcpCliente = New TcpClient()
    'Solicitud de conexión al objeto Servidor determinado por las propiedades
    'IPDelHost y PuertoDelHost
    VtcpCliente.Connect("127.0.0.1", 8051)
    'Se devuelve la NetworkStream usada para enviar y recibir datos.
    Vstream = VtcpCliente.GetStream()
    'Se crea un thread para que escuche los mensajes enviados por el Servidor
    VsubProceso1 = New Thread(AddressOf LeerSocket)
    'Se inicia el subproceso
    VsubProceso1.Start()
End Sub
Private Sub LeerSocket()
    Dim VbufferDeLectura() As Byte
    While True
        Try
            VbufferDeLectura = New Byte(100) {}
            'Se lee la llegada de datos desde el servidor
            Vstream.Read(VbufferDeLectura, 0, VbufferDeLectura.Length)
            'Se transforman los datos recibidos a String
            Dim Vrecibido As String = Encoding.ASCII.GetString(VbufferDeLectura)
            If Vrecibido.Substring(0, 3) = "VER" Then
                Me.TextBox2.Text = Vrecibido.Substring(3, Vrecibido.Length - 3)
                MeEnviarDatos("ENC" & Me.TextBox1.Text.Trim & Me.TextBox2.Text.Trim)
            ElseIf Vrecibido.Substring(0, 3) = "XNC" Then
                Me.TextBox3.Text = Vrecibido.Substring(3, Vrecibido.Length - 3)
            End If
        Catch e As Exception
            Exit While
        End Try
    End While
End Sub
Public Sub EnviarDatos(ByVal Datos As String)
    'Se envían los datos al Servidor
    Vstream.Write(Encoding.ASCII.GetBytes(Datos), 0, _
                  Encoding.ASCII.GetBytes(Datos).Length)
End Sub
Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) _
Handles Button1.Click
    MeEnviarDatos("COD" & Me.TextBox1.Text)

```

```
End Sub  
End Class
```

Analicemos el código de la aplicación cliente. Esta posee los procedimientos `Form1_Load`, `EnviarDatos`, `Button1_Click` y `LeerSocket`.

Para que el cliente pueda cumplir con su objetivo, fue necesario utilizar un objeto `TcpClient` denominado `VtcpCliente`, un objeto `Thread` denominado `VsubProceso1` y un objeto `Stream` denominado `Vstream`.

En el procedimiento `Form1_Load` se configura al cliente y se gestiona su conexión con el servidor. Para ello, se utiliza el método `Connect` del objeto `TcpClient` y se le pasa como parámetros la IP del servidor 127.0.0.1 y el puerto 8051 donde el servidor está escuchando. Cuando el servidor acepta la conexión, se utiliza el método `GetStream` del objeto `TcpClient` para obtener la `NetworkStream` que permitirá estar en contacto con el servidor. El siguiente paso es delegar la ejecución del procedimiento `LeerSocket` en un subproceso (`VsubProceso1`) y ponerlo a funcionar con el método `Start` del subproceso.

El procedimiento `LeerSocket` utiliza la `NetworkStream` (`Vstream`) para leer el buffer de lectura donde se encuentra lo enviado por el servidor. Si analizamos en detalle el código de `LeerSocket`, nos encontramos con las siguientes posibilidades:

Si la cadena recibida comienza con “VER”, lo que ha arribado es el dígito verificador. Se muestra y se procede a solicitarle al servidor el valor Hash, enviándole una cadena que contiene el código EAN-13 más el dígito verificador, todo esto precedido por “ENC”.

Si la cadena recibida comienza con “XNC”, significa que se ha recibido el código Hash y se visualiza.

El procedimiento `EnviarDatos` se encarga de enviarle los datos al servidor, utilizando el objeto `Stream` (`Vstream`). El procedimiento `Button1_Click` es el encargado de iniciar la secuencia de solicitudes enviándole al servidor un `String` que comienza con “COD” más el código EAN-13 del cual se desea obtener el dígito verificador.

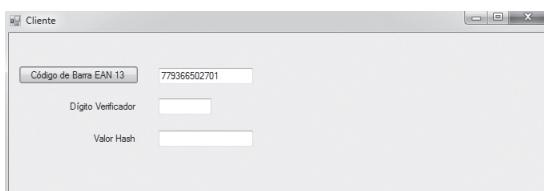


Figura 5.18. La interfaz del cliente.

Para finalizar la serie de ejemplos sobre comunicaciones de aplicaciones con socket, veremos un servidor de chat elemental pero que cumple con las funciones básicas de este tipo de aplicaciones.

Como todo lo analizado hasta aquí, contaremos con dos tipos de aplicaciones: una que funcione como servidor y otra como cliente. En este ejemplo particular, se podrá tener más de una aplicación cliente funcionando simultáneamente, ya que si nos referimos a un chat, seguramente varios usuarios querrán comunicarse al mismo tiempo. Las funcionalidades previstas cubren las siguientes posibilidades:

- Que uno o más clientes se puedan comunicar con el servidor.
- Que el servidor mantenga una lista de los clientes conectados.
- Que cada vez que un nuevo cliente se conecte, además de actualizarse la lista del servidor se informe a los clientes conectados sobre quienes están activos.
- Que un cliente les pueda enviar un mensaje a todos los clientes.
- Que un cliente le pueda enviar un mensaje a otro cliente en particular.
- Que el servidor pueda enviarle un mensaje a todos los clientes.
- Que el servidor pueda enviarle un mensaje a otro cliente en particular.

```

Public Class Servidor
#Region "ESTRUCTURAS"
    Public Structure InfoDeUnCliente
        'Esta estructura permite guardar la información sobre un cliente.
        Public Socket As Socket 'Socket utilizado para mantener la conexión con el cliente.
        Public Thread As Thread 'Thread utilizado para escuchar al cliente.
        Public UltimosDatosRecibidos As String 'Últimos datos enviados por el cliente.
        Public Quien As String 'Nick del cliente.
        Public Para As String 'Nick del cliente destinatario al que se le envía un mensaje.
    End Structure
#End Region
#Region "VARIABLES"
    Private VtcpLsn As TcpListener 'Listener de escucha.
    Private Vclientes As New Hashtable() 'Aquí se guarda la información de todos los Vclientes
                                         'conectados.
    Private VtcpThd As Thread 'Proceso donde se coloca a escuchar los clientes que llaman.
    Private ViDClienteActual As Net.IPEndPoint 'Último cliente conectado.
    Private VpuertoDeEscucha As String 'Puerto de escucha.
#End Region
#Region "EVENTOS"
    'Sucede cada vez que un cliente requiere una nueva conexión.
    Public Event NuevaConexion(ByVal pInfoClienteActual As InfoDeUnCliente)
    'Sucede cada vez que un nuevo cliente se conecta o desconecta.
    Public Event TodosLosClientes(ByVal pClientes As Hashtable)
    'Sucede cada vez que se reciben datos.
    Public Event DatosRecibidos(ByVal pIDTerminal As Net.IPEndPoint)
    'Sucede cada vez que un cliente se desconecta.
    Public Event ConexionTerminada(ByVal pIDTerminal As Net.IPEndPoint)
#End Region
#Region "PROPIEDADES"
    Property PuertoDeEscucha() As String

```

```

Get
    PuertoDeEscucha = VpuertoDeEscucha
End Get
Set(ByVal Value As String)
    VpuertoDeEscucha = Value
End Set
End Property
#End Region
#Region "METODOS"
    'Coloca al servidor a escuchar por el puerto configurado.
    Public Sub Escuchar()
        VtcpLsn = New TcpListener(New System.Net.IPEndPoint({127, 0, 0, 1}), _
            Convert.ToInt16(Me.PuertoDeEscucha))
        'Se inicia la escucha.
        VtcpLsn.Start()
        'Se crea un subproceso para que se quede escuchando la llegada de un nuevo cliente.
        VtcpThd = New Thread(AddressOf EsperarCliente)
        'Se inicia el subproceso.
        VtcpThd.Start()
    End Sub
    'Obtiene los datos enviados por un cliente en particular.
    Public Function ObtenerDatos(ByVal pIDCliente As Net.IPEndPoint) As String
        Dim InfoClienteSolicitado As InfoDeUnCliente
        'Se obtiene la información del cliente destino, origen y el mensaje.
        InfoClienteSolicitado = Vclientes(pIDCliente)
        Return Now & "--- " & InfoClienteSolicitado.Quien & " To " &
            InfoClienteSolicitado.Para & _
            InfoClienteSolicitado.UltimosDatosRecibidos & Constants.vbCrLf
    End Function
    'Cierra la conexión con un cliente particular.
    Public Sub Cerrar(ByVal pIDCliente As Net.IPEndPoint)
        Dim VinfoClienteActual As InfoDeUnCliente
        'Se obtiene la información del cliente desconectado.
        VinfoClienteActual = Vclientes(pIDCliente)
        'Se cierra la conexión con el cliente
        VinfoClienteActual.Socket.Close()
    End Sub
    'Cierra la conexión con todos los clientes.
    Public Sub Cerrar()
        Dim VinfoClienteActual As InfoDeUnCliente
        'Se recorren todos los clientes y se cierran las conexiones.
        For Each VinfoClienteActual In Vclientes.Values
            Call Cerrar(VinfoClienteActual.Socket.RemoteEndPoint)
        Next
    End Sub
    'Envía datos a un cliente particular.
    Public Sub EnviarDatos(ByVal pIDCliente As Net.IPEndPoint, ByVal pDatos As String)
        Dim Cliente As InfoDeUnCliente
        'Se obtiene la información del cliente al que se le quiere enviar el mensaje
        Cliente = Vclientes(pIDCliente)

```

```

    'Se envia el mensaje
    Cliente.Socket.Send(Encoding.ASCII.GetBytes(pDatos))
End Sub
'Envía datos a todos los clientes.
Public Sub EnviarDatos(ByVal pDatos As String)
    Dim Vcliente As InfoDeUnCliente
    'Se recorren todos los clientes conectados y se les envia el mensaje recibido
    'en el parametro pDatos
    For Each Vcliente In Vclientes.Values
        EnviarDatos(Vcliente.Socket.RemoteEndPoint, pDatos)
    Next
End Sub
#End Region
#Region "FUNCIONES PRIVADAS"
    'Procedimiento que espera que un cliente se conecte.
Private Sub EsperarCliente()
    Dim VinfoClienteActual As InfoDeUnCliente
    With VinfoClienteActual
        While True
            'Cuando se recibe la conexion, se guarda la información del cliente.
            'Se queda esperando la conexion de un cliente y se guarda el Socket
            'que utilizo para mantener la conexion con el cliente.
            .Socket = VtcpLsn.AcceptSocket()
            'Se guarda el RemoteEndPoint, que utilizo para identificar al cliente.
            ViDClienteActual = .Socket.RemoteEndPoint
            'Se crea un Thread para que se encargue de escuchar los mensaje del 'cliente.
            .Thread = New Thread(AddressOf LeerSocket)
            'Se agrega la informacion del cliente al HashTable Vclientes, donde se
            'encuentra la informacion de todos ellos.
            SyncLock Me
                Vclientes.Add(ViDClienteActual, VinfoClienteActual)
            End SyncLock
            'Se dispara el evento NuevaConexion.
            RaiseEvent NuevaConexion(VinfoClienteActual)
            'Se pone a funcionar el thread encargado de escuchar los mensajes del cliente.
            .Thread.Start()
            'Se indaga quien es es el cliente.
            Me.EnviarDatos(ViDClienteActual, "QUIEN")
        End While
    End With
End Sub

Private Sub LeerSocket()
    Dim VidReal As Net.IPEndPoint 'Para almacenar el IPEndPoint del cliente que se va a
                                    'escuchar.
    Dim Vrecibir() As Byte 'Para recibir los datos que arriban.
    Dim VinfoClienteActual As InfoDeUnCliente 'Información del cliente que se va a
                                                'escuchar.

    Dim Vret As Integer = 0
    VidReal = ViDClienteActual
    VinfoClienteActual = Vclientes(VidReal)

```

```

With VinfoClienteActual
    While True
        If .Socket.Connected Then
            Vrecibir = New Byte(100) {}
            Try
                'Se espera a que llegue un mensaje desde el cliente.
                Vret = .Socket.Receive(Vrecibir, Vrecibir.Length, SocketFlags.None)
                If Vret > 0 Then
                    If Encoding.ASCII.GetString(Vrecibir).Substring(0, 5) = "NICK-" Then
                        .Quien = Encoding.ASCII.GetString(Vrecibir).Substring(5, Vret).Trim
                        Vclientes(VidReal) = VinfoClienteActual
                        'Se ejecuta el evento que envía lo arribado a todos los Clientes.
                        RaiseEvent TodosLosClientes(Vclientes)
                    ElseIf Encoding.ASCII.GetString(Vrecibir). _
                        Substring(0, 5) = "PARA-" Then
                        .Quien = Encoding.ASCII.GetString(Vrecibir).Substring(6, _
                            Encoding.ASCII.GetString(Vrecibir).IndexOf("$") - 6)
                        'Si el mensaje es para un cliente en particular se obtiene para quien es.
                        .Para = Encoding.ASCII.GetString(Vrecibir). _
                            Substring(Encoding.ASCII.GetString(Vrecibir). _
                                IndexOf("$") + 1, (Encoding.ASCII.GetString(Vrecibir).IndexOf("%") - _
                                (Encoding.ASCII.GetString(Vrecibir).IndexOf("$") + 1)))
                        'Se guarda el mensaje recibido
                        .UltimosDatosRecibidos = Encoding.ASCII.GetString(Vrecibir). _
                            Substring(Encoding.ASCII.GetString(Vrecibir).IndexOf("%") + 1)
                        Vclientes(VidReal) = VinfoClienteActual
                        'Se dispara el evento que alerta sobre la recepción de datos.
                        RaiseEvent DatosRecibidos(VidReal)
                        'Se controla que quien genera el mensaje de datos sea distinto al
                        'destinatario.
                    If .Quien <> .Para Then
                        For Each x As DictionaryEntry In Vclientes
                            'Se ubica el destinatario y se le envía el mensaje.
                            Dim Vtemp As InfoDeUnCliente = DirectCast(x.Value, _
                                InfoDeUnCliente)
                            If Vtemp.Quien = .Para Then Me.EnviarDatos(x.Key, .Quien & _
                                " --> " & .UltimosDatosRecibidos)
                        Next
                    End If
                ElseIf Encoding.ASCII.GetString(Vrecibir).Substring(0, 5) = "TODOS" Then
                    .Quien = Encoding.ASCII.GetString(Vrecibir).Substring(6, _
                        Encoding.ASCII.GetString(Vrecibir).IndexOf("$") - 6)
                    'Se guarda el mensaje recibido.
                    .UltimosDatosRecibidos = Encoding.ASCII.GetString(Vrecibir). _
                        Substring(Encoding.ASCII.GetString(Vrecibir).IndexOf("$") + 1)
                    Vclientes(VidReal) = VinfoClienteActual
                    'Se dispara el evento que alerta sobre la recepción de datos.
                    RaiseEvent DatosRecibidos(VidReal)
                    'Se le envía el mensaje a cada cliente.
                    For Each x As DictionaryEntry In Vclientes
                        Me.EnviarDatos(x.Key, .Quien & " --> " & .UltimosDatosRecibidos)
                    Next
                End If
            End Try
        End If
    End While
End With

```

```

    Else
        'Se genera el evento de la finalización de la conexión.
        RaiseEvent ConexionTerminada(VidReal)
        Exit While
    End If
    Catch e As Exception
        If Not .Socket.Connected Then
            'Se genera el evento que alerta sobre la finalización de la conexión.
            RaiseEvent ConexionTerminada(VidReal)
            Exit While
        End If
    End Try
End If
End While
'Se elimina el cliente del hashtable que guarda la información de los clientes.
Vclientes.Remove(VidReal)
'Se ejecuta el evento que alerta sobre la actualización de la lista de clientes.
RaiseEvent TodosLosClientes(Vclientes)
Call CerrarThread(VidReal)
End With
End Sub
Private Sub CerrarThread(ByVal IDCiente As Net.IPEndPoint)
    Dim VinfoClienteActual As InfoDeUnCliente
    VinfoClienteActual = Vclientes(IDCiente)
    'Se cierra el thread que se encargaba de escuchar al cliente especificado.
    Try
        VinfoClienteActual.Thread.Abort()
    Catch e As Exception
        SyncLock Me
            'Se elimina el cliente del hashtable.
            Vclientes.Remove(IDCiente)
        End SyncLock
    End Try
End Sub
#End Region
End Class

```

Comenzaremos explicando los elementos de la clase **Servidor**. En ella se utiliza una estructura denominada **InfoDeUnCliente**, que tiene por objetivo mantener toda la información referida a un cliente conectado. Para ello posee:

- Un **Socket** denominado **Socket**, que se utiliza para mantener la conexión de un cliente.
- Un **Thread** denominado **Thread**, por donde se escuchará al cliente.
- Un **String** denominado **UltimosDatosRecibidos**, que mantendrá los últimos datos enviados por el cliente.
- Un **String** para mantener el nick del cliente actual que se conecta y/o envía un mensaje.
- Un **String** para mantener el nick del cliente al cual se le envía el mensaje.

También se definen las variables:

- **VtcpLsn** del tipo `TcpListener`. Se utiliza para escuchar el puerto por donde se recibirán las solicitudes de conexión de todos los clientes.
- **Vclientes** del tipo `HashTable`. Se utiliza para tener una tabla con la información (`InfoDeUnCliente`) de todos los clientes conectados al servidor. Como el `HashTable` lo permite, se almacenará un Id único del cliente como identificador y sus datos (almacenados en la estructura) como valor de ese identificador.
- **VtcpThd** del tipo `Thread`. Se utiliza para ejecutar el código que espera la conexión de un cliente en un proceso diferente.
- **VidClienteActual** del tipo `IPEndPoint`. Se utiliza para almacenar el `IPEndPoint` de un cliente y usarlo como identificador único.
- **VpuertoDeEscucha** del tipo `String`. Se utiliza para almacenar el puerto por donde se pondrá a escuchar a la aplicación `Servidor`.

Los eventos utilizados son:

- `NuevaConexion` se desencadena cuando un nuevo cliente se conecta.
- `TodosLosClientes` se desencadena cada vez que un nuevo cliente se conecta con el objetivo de informarle a cada cliente cual es la lista de todos los clientes conectados.
- `DatosRecibidos` se desencadena cuando se reciben datos de un cliente.
- `ConexionTerminada` se desencadena cuando se aborta o finaliza una conexión con algún cliente.

La propiedad `PuertoDeEscucha` se utiliza para configurar el puerto por donde el servidor se quedará escuchando las solicitudes de conexión de los clientes.

El procedimiento `Escuchar` configura el objeto `TcpListener` en el puerto configurado y lo pone a escuchar con el método `Start`. Luego, se delega la ejecución del método `EsperarCliente` en un nuevo proceso (`VtcpThd`) y se lo pone a funcionar con el método `Start`.

La función `ObtenerDatos` recibe un `IPEndPoint` que identifica al cliente del cual se desean obtener los datos que envió. Con el identificador ubica al cliente en el `HashTable` de clientes y formatea un `String` que retorna con la siguiente información:

Fecha y Hora del sistema + El cliente que envía el mensaje + To + El cliente destinatario del mensaje + ---> + el mensaje recibido + un salto y retorno de carro.

Esta información se utilizará para visualizar en la interfaz del servidor.

El procedimiento `Cerrar` está sobrecargado y permite cerrar la conexión con un cliente si se pasa por parámetro el `IPEndPoint` o todos los clientes si no se pasa nada. Para concretar la acción, cierra el socket `Socket.Close`.

El procedimiento `EnviarDatos` también es un procedimiento sobrecargado y se utiliza para hacerle llegar un mensaje a un cliente en particular o a todos. Para que un cliente particular reciba un mensaje, se debe pasar por parámetro el `IPEndPoint` del cliente y el mensaje. Si solo se pasa el mensaje, este le llegará a todos los clientes.

El procedimiento privado denominado `EsperarCliente` es el que permite que el servidor quede a la espera de la conexión de algún cliente. Al detectar esta petición por medio del `RemoteEndPoint` del socket que se conecta con el cliente, se obtiene el identificador del cliente e inmediatamente se delega la ejecución del procedimiento `LeerSocket` en un proceso nuevo. Luego, se desencadena el evento `NuevaConexion` informando el identificador (`IPEndPoint`) del cliente que se ha conectado. Finalmente, se pone a funcionar el subprocesso donde se ejecutará `LeerSocket`. A continuación, el servidor indaga quién es el cliente que se ha conectado con el objetivo de recibir su nickname. La indagación se realiza enviándole el mensaje “QUIEN” al cliente, el cual lo interpretará como una indagación.

El procedimiento `LeerSocket` es el responsable de recibir los datos enviados por los clientes. Para lograrlo, se estableció un protocolo elemental que trabajará en la cabecera de los mensajes, ocupando los primeros cinco caracteres. Básicamente, este procedimiento se queda esperando que algún cliente envíe algo e interpreta por la cabecera del mensaje lo que envió el cliente. A continuación, se expondrá lo que se puede recibir en la cabecera de un mensaje que llega al servidor.

- “NICK-” Indica que lo que se recibió en el mensaje es el nickname del cliente. Este tipo de mensaje se recibe luego de que el servidor haya indagado a un cliente con un mensaje del tipo “QUIEN”.
- “PARA-” Significa que el mensaje va dirigido a un cliente en particular. Para extraer qué cliente en particular, se busca su nickname dentro del mensaje ya que estará delimitado por los caracteres “\$” al comienzo y “%” al final.
- “TODOS” Indica que el mensaje va dirigido a todos los clientes.

Veamos ahora más en detalle qué hace este procedimiento. Se declaran las variables `VidReal` del tipo `IPEndPoint` para almacenar el `IPEndPoint` del cliente que se va a escuchar. `VinfoClienteActual` para mantener la información del cliente que se va a escuchar y el vector `Vrecibir` para almacenar los bytes que arriben en los mensajes, que en nuestro ejemplo tendrá una capacidad de 100 bytes. Además, se utiliza la variable `Vret` para medir el largo en caracteres del mensaje.

El procedimiento aguarda a que la variable `Vret` sea mayor a cero, esto indica que ha llegado un mensaje al buffer. El siguiente paso es evaluar qué llegó en la cabecera del mensaje (los primeros cinco caracteres del mismo).

Si es “NICK-”, se extrae del mensaje el nickname de quien lo envió. Esto estará alojado a partir del sexto carácter hasta el final del mensaje, y por este motivo en el código observamos (...`Substring(5, Vret)`). Este dato es cargado en la estructura de `VinfoClienteActual` en la variable `Quien` y a continuación, esa estructura se actualiza en el `HashTable`, precisamente en la posición cuyo identificador coincide con `VidReal` (el `IPEndPoint` del cliente, su identificador). Por último, se desencadena el evento `TodosLosClientes` y se envía el `HashTable` (recordemos que este evento avisa a todos los clientes que un cliente nuevo se ha agregado a los conectados).

Si es “PARA-”, se extrae del mensaje quién lo envió y para quién es, y se almacena en la estructura `VinfoClienteActual` en las variables `Quien` y `Para`, respectivamente. `Quien` está desde el sexto carácter hasta el carácter “\$” en el mensaje. `Para` se encuentra entre los caracteres

“\$” y “%” del mensaje. Lo enviado como dato propio del mensaje se almacena en la variable `UltimosDatosRecibidos` de la estructura y se encuentra a partir del carácter “%”. Luego, cuando la estructura esté cargada, se actualiza en el `HashTable` de clientes. A continuación, se desencadena el evento `DatosRecibidos` que alerta sobre la recepción de datos por parte de un cliente. Para finalizar, se verifica si `Quien` y `Para` son iguales y se evita que el mensaje sea enviado y receptionado por el mismo cliente. En caso de que no exista coincidencia, se recorre el `HashTable` de clientes y se ubica por el nickname al cliente (`Para`) al que se le debe enviar el mensaje. Una vez ubicado el destinatario, quien remite utiliza el método `EnviarDatos`.

Si es “TODOS”, significa que el mensaje recibido desde un cliente debe llegarle a todos los demás clientes. Para ello, se ubica quien lo envió y se carga en la variable `Quien` de la estructura `VinfoClienteActual` desde el sexto carácter del mensaje hasta el carácter “\$”. A partir de ahí hasta el final están los datos puros del mensaje, que son almacenados en la variable `UltimosDatosRecibidos` de la estructura. Luego se recorre el `HashTable` de clientes y, por medio del método `EnviarDatos`, se le envía el mensaje a todos ellos.

Lo que se detalló hasta aquí para el procedimiento `LeerSocket` ocurre si el socket está conectado, y si se desconecta se desencadena el evento `ConexionTerminada`. Luego, se elimina al cliente del `HashTable` de clientes, se desencadena el evento `TodosLosClientes` y, finalmente, se cierra el proceso de ese cliente que se desconectó (`CerrarThread`).

El aspecto de la interfaz de la aplicación servidora es la que se ve en la figura 5.19.

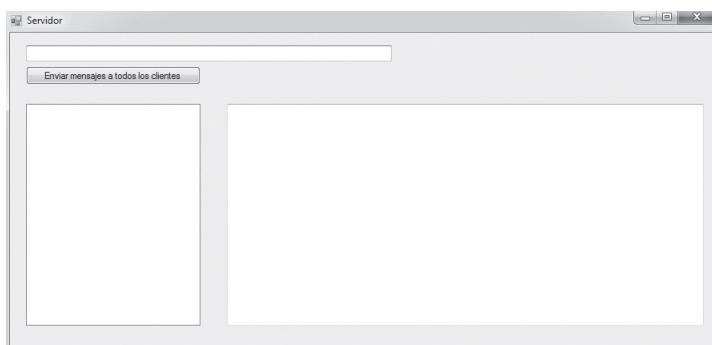


Figura 5.19.

A continuación, se puede observar el código que le da soporte a la interfaz gráfica y de esta forma conocer los valores que adoptan las propiedades de los controles y los procedimientos que articulan el código que se ejecutará.

```
Imports System
Imports System.Threading
Imports System.Net.Sockets
```

```
Imports System.IO
Imports System.Text
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim WithEvents WinSockServer As New Servidor()
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles MyBase.Load
        Form1.CheckForIllegalCrossThreadCalls = False
        Control.CheckForIllegalCrossThreadCalls = False
        TextBox.CheckForIllegalCrossThreadCalls = False
        With WinSockServer
            'Se establece el puerto donde escuchar
            .PuertoDeEscucha = 8050
            'Comienza la escucha
            .Escuchar()
        End With
    End Sub
    Private Sub WinSockServer_ConexionTerminada(ByVal pIDTerminal As System.Net.IPEndPoint) _
        Handles WinSockServer.ConexionTerminada
        'Se muestra con quien se termino la conexion
        MsgBox("Se ha desconectado el cliente desde la IP= " & pIDTerminal.Address.ToString & _
            ", Puerto = " & pIDTerminal.Port)
    End Sub
    Private Sub WinSockServer_DatosRecibidos(ByVal pIDTerminal As System.Net.IPEndPoint) _
        Handles WinSockServer.DatosRecibidos
        'Se obtienen los datos recibidos.
        Dim Vdatos As String = WinSockServer.ObtenerDatos(pIDTerminal).Replace(WinSockServer. _
            ObtenerDatos(pIDTerminal).Chars(WinSockServer. _
            ObtenerDatos(pIDTerminal).Length - 1), " ").Trim
        'Se muestran los datos recibidos.
        Me.TextBox1.Text = Vdatos.Replace(Vdatos.Chars(Vdatos.Length - 1), " "). _
            Trim & Constants.vbCrLf & Me.TextBox1.Text
    End Sub
    Private Sub btnEnviarMensaje_Click(ByVal sender As System.Object, ByVal e As _
        System.EventArgs) Handles btnEnviarMensaje.Click
        'Se envía el texto escrito en el textbox txtMensaje a todos los clientes.
        WinSockServer.EnviarDatos(txtMensaje.Text)
    End Sub
    Private Sub WinSockServer_NuevaConexion(pInfoClienteActual As Servidor.InfoDeUnCliente) _
        Handles WinSockServer.NuevaConexion
        Dim VtempIpEndPoint As System.Net.IPEndPoint = _
            DirectCast(pInfoClienteActual.Socket.RemoteEndPoint, System.Net.IPEndPoint)
        'Se muestra que cliente se conecto
        MsgBox("Se ha conectado un cliente desde la IP= " & _
            VtempIpEndPoint.Address.ToString & ", Puerto = " & VtempIpEndPoint.Port)
    End Sub
    Private Sub WinSockServer_TodosLosClientes(pClientes As System.Collections.Hashtable) _
        Handles WinSockServer.TodosLosClientes
        'Se muestran todos los clientes en el ListBox
        Me.ListBox1.Items.Clear()
        For Each X As DictionaryEntry In pClientes
```

```

        Me.ListBox1.Items.Add(DirectCast(X.Value, Servidor.InfoDeUnCliente).Quien & " ")
    Next
    For Each X As DictionaryEntry In pClientes
        WinSockServerEnviarDatos("CLIEB")
        Thread.Sleep(300)
        For Each L In Me.ListBox1.Items
            WinSockServerEnviarDatos("CLIE-" & L.ToString.Trim)
        Next
    Next
End Sub
End Class

```

El siguiente código puede colocarse en el archivo Form1.Designer.vb:

```

<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
Partial Class Form1
    Inherits System.Windows.Forms.Form

    'Form reemplaza a Dispose para limpiar la lista de componentes.
    <System.Diagnostics.DebuggerNonUserCode()> _
    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        Try
            If disposing AndAlso components IsNot Nothing Then
                components.Dispose()
            End If
        Finally
            MyBase.Dispose(disposing)
        End Try
    End Sub

    'Requerido por el Diseñador de Windows Forms
    Private components As System.ComponentModel.IContainer

    'NOTA: el Diseñador de Windows Forms necesita el siguiente procedimiento
    'Se puede modificar usando el Diseñador de Windows Forms.
    'No lo modifique con el editor de código.
    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()
        Me.txtMensaje = New System.Windows.Forms.TextBox()
        Me.btnEnviarMensaje = New System.Windows.Forms.Button()
        Me.ListBox1 = New System.Windows.Forms.ListBox()
        Me.TextBox1 = New System.Windows.Forms.TextBox()
        Me.SuspendLayout()
        '
        'txtMensaje
        '
        Me.txtMensaje.Location = New System.Drawing.Point(1, 12)
        Me.txtMensaje.Name = "txtMensaje"

```

```
Me.txtMensaje.Size = New System.Drawing.Size(457, 20)
Me.txtMensaje.TabIndex = 0
'
'btnEnviarMensaje
'
Me.btnAddMensaje.Location = New System.Drawing.Point(1, 38)
Me.btnAddMensaje.Name = "btnEnviarMensaje"
Me.btnAddMensaje.Size = New System.Drawing.Size(218, 23)
Me.btnAddMensaje.TabIndex = 1
Me.btnAddMensaje.Text = "Enviar mensajes a todos los clientes"
Me.btnAddMensaje.UseVisualStyleBackColor = True
'
'
'ListBox1
'
Me.ListBox1.FormattingEnabled = True
Me.ListBox1.Location = New System.Drawing.Point(1, 85)
Me.ListBox1.Name = "ListBox1"
Me.ListBox1.Size = New System.Drawing.Size(218, 277)
Me.ListBox1.TabIndex = 2
'
'TextBox1
'
Me.TextBox1.Location = New System.Drawing.Point(252, 85)
Me.TextBox1.Multiline = True
Me.TextBox1.Name = "TextBox1"
Me.TextBox1.Size = New System.Drawing.Size(596, 277)
Me.TextBox1.TabIndex = 3
'
'Form1
'
Me.AutoScaleDimensions = New System.Drawing.SizeF(6.0!, 13.0!)
Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font
Me.ClientSize = New System.Drawing.Size(881, 436)
Me.Controls.Add(Me.TextBox1)
Me.Controls.Add(Me.ListBox1)
Me.Controls.Add(Me.btnAddMensaje)
Me.Controls.Add(Me.txtMensaje)
Me.Name = "Form1"
Me.Text = "Servidor"
Me.ResumeLayout(False)
Me.PerformLayout()

End Sub
Friend WithEvents txtMensaje As System.Windows.Forms.TextBox
Friend WithEvents btnEnviarMensaje As System.Windows.Forms.Button
Friend WithEvents ListBox1 As System.Windows.Forms.ListBox
Friend WithEvents TextBox1 As System.Windows.Forms.TextBox
End Class
```

La aplicación cliente

La aplicación cliente tiene por objetivo conectarse con el servidor y, de esta forma, enviarle mensajes para que sean vistos por todos los demás clientes conectados o por uno en particular.

El código de la aplicación cliente es el siguiente y se desarrollará para comprender en detalle su funcionamiento.

```
Public Class Cliente
#Region "VARIABLES"
    Private Vstm As Stream 'Se utiliza para enviar datos al Servidor y recibir datos del mismo.
    Private ViPDelHost As String 'Se coloca la dirección IP donde está el objeto de la clase _
        'Servidor.
    Private VpuertoDelHost As String 'Se coloca el puerto donde escucha el objeto de la clase _
        'Servidor.
#End Region
#Region "EVENTOS"
    Public Event ConexionTerminada()
    Public Event DatosRecibidos(ByVal pDatos As String)
    Public Event ClienteRecibido(ByVal pCliente As String)
#End Region
#Region "PROPIEDADES"
    Public Property IPDelHost() As String
        Get
            IPDelHost = ViPDelHost
        End Get
        Set(ByVal Value As String)
            ViPDelHost = Value
        End Set
    End Property
    Public Property PuertoDelHost() As String
        Get
            PuertoDelHost = VpuertoDelHost
        End Get
        Set(ByVal Value As String)
            VpuertoDelHost = Value
        End Set
    End Property
#End Region
#Region "METODOS"
    Public Sub Conectar()
        Dim VtcpCln As TcpClient 'Cliente TCP utilizado para conectarse con el servidor.
        Dim VtcpThd As Thread 'Subproceso que se encargará de escuchar los mensajes _
            'enviados por el servidor.
        VtcpCln = New TcpClient()
        'Se conecta al objeto de la clase Servidor,
        'determinado por las propiedades IPDelHost y PuertoDelHost.
        VtcpCln.Connect(IPDelHost, PuertoDelHost)
        'Se apunta por medio de Stm al NetworkStream retornado por el cliente TCP.
    End Sub
#End Region
```

```

Vstm = VtcpClnt.GetStream()
‘Se crea e inicia un thread para que escuche los mensajes enviados por el servidor.
VtcpThd = New Thread(AddressOf LeerSocket)
‘Se arranca el subproceso.
VtcpThd.Start()
End Sub
Public Sub EnviarDatos(ByVal pDatos As String)
    Dim VbufferDeEscritura() As Byte
    VbufferDeEscritura = Encoding.ASCII.GetBytes(pDatos)
    If Not (Vstm Is Nothing) Then
        ‘Se envían los datos al Servidor
        Vstm.Write(VbufferDeEscritura, 0, VbufferDeEscritura.Length)
    End If
End Sub
#End Region

#Region “FUNCIONES PRIVADAS”
Private Sub LeerSocket()
    Dim VbufferDeLectura() As Byte
    While True
        Try
            VbufferDeLectura = New Byte(100) {}
            ‘Se queda esperando a que llegue algún mensaje.
            Vstm.Read(VbufferDeLectura, 0, VbufferDeLectura.Length)
            ‘Se evalúa que llega en la cabecera del mensaje.
            ‘QUIEN: Solicita el NICK del cliente.
            If Encoding.ASCII.GetString(VbufferDeLectura).Substring(0, 5) = “QUIEN” Then
                ‘Se envía el NICK del cliente.
                Me.EnviarDatos(“NICK-” & Form1.txtNombre.Text)
                ‘CLIEB: Solicita que se borre la lista de clientes conectados.
            ElseIf Encoding.ASCII.GetString(VbufferDeLectura).Substring(0, 5) = “CLIEB” Then
                ‘Se desencadena el evento que borra la lista de clientes conectados.
                RaiseEvent ClienteRecibido(“CLIEB”)
                ‘CLIE-: Solicita que se agregue un cliente conectado a la lista de clientes.
            ElseIf Encoding.ASCII.GetString(VbufferDeLectura).Substring(0, 5) = “CLIE-” Then
                ‘Se desencadena el evento que agrega un cliente a la lista de clientes.
                RaiseEvent ClienteRecibido(Encoding.ASCII.GetString(VbufferDeLectura). _
                    Substring(5).Trim)
            Else
                ‘Se desencadena el evento DatosRecibidos, pues se han recibido datos desde el
                ‘Servidor.
                RaiseEvent DatosRecibidos(Now & “---” & _
                    Encoding.ASCII.GetString(VbufferDeLectura))
            End If
            Catch e As Exception
                Exit While
            End Try
        End While
        ‘Se finaliza la conexión, por lo tanto desencadena el evento correspondiente.
        RaiseEvent ConexionTerminada()
    End Sub

```

```
#End Region  
End Class
```

La clase cliente declara las siguientes variables:

- `Vstm` del tipo `stream` para poder enviar datos al servidor y recibir datos de este.
- `VipDelHost` del tipo `String` para almacenar la IP de la máquina donde se encuentra funcionando la aplicación servidora.
- `VpuertoDelHost` del tipo `String` para almacenar el puerto por donde la aplicación servidora escucha las peticiones de los clientes para conectarse.

También posee eventos para alertar sobre aspectos que pueden estar sucediendo. Los eventos utilizados son:

- `ConexionTerminada`. Se desencadena cuando la conexión con el servidor se ha interrumpido o cerrado.
- `DatosRecibidos`. Se desencadena cuando arriban datos al cliente. Posee un `String` como parámetro que contiene el mensaje/datos recibidos.
- `ClienteRecibido`. Se desencadena cuando los datos recibidos poseen una cabecera de tipo “CLIEB” o “CLIE-” con el objetivo de borrar la lista de nicknames de clientes conectados o bien agregar un nickname a esa lista, dependiendo de la cabecera recibida. Este aspecto se desarrollará con mayor amplitud al analizar la forma de lectura de los datos recibidos.

Las propiedades utilizadas para configurar el Host son:

- `IPDelHost`. Permite configurar la IP del host (servidor).
- `PuertoDelHost`. Permite configurar el número de puerto donde el cliente peticionará la solicitud de conexión y el servidor se encuentra escuchando.

Los métodos son dos más un procedimiento privado. Se utilizan para que el cliente se conecte con el servidor, que pueda enviar datos y leer los datos recibidos. Se implementan por los siguientes procedimientos:

- `Conectar`. Este método realiza la petición de conexión al servidor. Para ello, se utiliza un objeto `TcpClient` denominado `VtcpC1nt`. Este objeto, por medio del método `Connect`, y pasando como parámetros la `IPDelHost` y el `PuertoDelHost`, realiza la petición de conexión. Al ser aceptada, se apunta al objeto `NetworkStream` obtenido, por medio del cual se gestionará el flujo de datos con el servidor. Una vez logrado el `NetworkStream`, se predispone todo para que el procedimiento `LeerSocket` funcione en un proceso independiente (`VtcpThd`). A través de él se administrarán los datos que arriben al cliente. Finalmente, este proceso se inicia con el método `Start`.

- **EnviarDatos.** Este método recibe por parámetro (*pDatos*) los datos a enviar. Para hacerlo efectivo, codifica (*Encoding.ASCII.GetBytes(pDatos)*) los datos como un Array de bytes. Luego, utilizando el método *Write* del objeto *NetworkStream*, se envían los datos.
- **LeerSocket.** Es el responsable de recibir los datos enviados por el servidor. Para lograrlo, se estableció un protocolo elemental que trabajará en la cabecera de los mensajes ocupando los primeros cinco caracteres. Básicamente, este procedimiento se queda esperando que el servidor envíe información y al hacerlo, se interpreta la cabecera del mensaje para determinar qué es lo que se debe hacer con él. A continuación, se expondrá lo que se puede recibir en la cabecera de un mensaje recibido en el cliente que fue enviado por el servidor.

- “QUIEN” Se le solicita al cliente que se identifique enviando su nickname. El cliente le envía al servidor el nickname con la cabecera “NICK-”.
- “CLIEB” Indica que el cliente debe borrar la lista de clientes conectados que posee.
- “CLIE-” Indica que el cliente debe agregar a un nuevo cliente conectado a la lista de clientes.

Si el mensaje arribado no posee como cabecera alguna de las mencionadas, entonces el mensaje se interpreta como simples datos recibidos y debido a esto se desencadena el evento *Datosrecibidos*. La interfaz de la aplicación cliente es la que se ve en la figura 5.20.



Figura 5.20.

A continuación, se puede observar el código que le da soporte a la interfaz gráfica y los valores que adoptan las propiedades de los controles y los procedimientos que articulan el código que se ejecutará:

```

Imports System
Imports System.Threading
Imports System.Net.Sockets
Imports System.IO
Imports System.Text

Public Class Form1
    Dim WithEvents WinSockCliente As New Cliente
    Private Sub btnConectar_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnConectar.Click

```

```

With WinSockCliente
    'Se determina a donde se quiere conectar el usuario
    .IPDelHost = txtIP.Text
    .PuertoDelHost = txtPuerto.Text
    'Se realiza la conexión
    .Conectar()
    'Se desabilitan los elementos relacionados con la posibilidad de conexión
    txtIP.Enabled = False
    txtPuerto.Enabled = False
    btnConectar.Enabled = False
    txtNombre.Enabled = False
    'Habilito la posibilidad de enviar mensajes
    btnEnviarMensaje.Enabled = True
    btnEnviarA.Enabled = True
    txtTexto.Enabled = True
End With
End Sub
Private Sub WinSockCliente_ClienteRecibido(pCliente As String) Handles _
    WinSockCliente.ClienteRecibido
    'Se actualiza la lista de clientes conectados.
    If pCliente = "CLIEB" Then
        ListBox1.Items.Clear()
    Else
        Me.ListBox1.Items.Add(pCliente)
    End If
End Sub
Private Sub WinSockCliente_DatosRecibidos(ByVal pDatos As String) Handles _
    WinSockCliente.DatosRecibidos
    'Se muestran los datos recibidos.
    Me.TextBox1.Text = pDatos.Replace(pDatos.Chars(pDatos.Length - 1), " ").Trim & _
        Constants.vbCrLf & Me.TextBox1.Text
End Sub
Private Sub WinSockCliente_ConexionTerminada() Handles WinSockCliente.ConexionTerminada
    MsgBox("Finalizó la conexión")
    'Se habilita la posibilidad de una reconexión.
    txtIP.Enabled = True
    txtPuerto.Enabled = True
    btnConectar.Enabled = True
    txtNombre.Enabled = True
    'Se deshabilita la posibilidad de enviar mensajes.
    btnEnviarMensaje.Enabled = False
    btnEnviarA.Enabled = False
    txtTexto.Enabled = False
End Sub
Private Sub Form1_Closing(ByVal sender As Object, ByVal e As _
    System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
    End
End Sub
Private Sub btnEnviarMensaje_Click_1(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnEnviarMensaje.Click
    'Se envía lo que está escrito en la caja de texto del mensaje.

```

```

        WinSockClienteEnviarDatos("TODOS*" & txtNombre.Text & "$" & txtTexto.Text)
End Sub
Private Sub Form1_Load(sender As Object, e As System.EventArgs) Handles Me.Load
    Form1.CheckForIllegalCrossThreadCalls = False
    Control.CheckForIllegalCrossThreadCalls = False
End Sub
Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) Handles _
    BtnEnviarA.Click
    Try
        'Se envia lo que está escrito en la caja de texto para mensajes a un cliente
        'seleccionado en la lista.
        WinSockClienteEnviarDatos(String.Concat({{"PARA-*", txtNombre.Text, "$", _
            me.ListBox1.SelectedItem.ToString.Replace(Me.ListBox1.SelectedItem. _
            ToString.Chars(Me.ListBox1.SelectedItem.ToString.Length - 1), " ").Trim, _
            "%", txtTexto.Text}}))
    Catch ex As Exception
    End Try
End Sub
End Class

```

El siguiente código puede colocarse en el archivo Form1.Designer.vb:

```

<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
Partial Class Form1
    Inherits System.Windows.Forms.Form

    'Form reemplaza a Dispose para limpiar la lista de componentes.
    <System.Diagnostics.DebuggerNonUserCode()> _
    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        Try
            If disposing AndAlso components IsNot Nothing Then
                components.Dispose()
            End If
        Finally
            MyBase.Dispose(disposing)
        End Try
    End Sub

    'Requerido por el Diseñador de Windows Forms
    Private components As System.ComponentModel.IContainer

    'NOTA: el Diseñador de Windows Forms necesita el siguiente procedimiento
    'Se puede modificar usando el Diseñador de Windows Forms.
    'No lo modifique con el editor de código.
    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()
        Me.btnAddConectar = New System.Windows.Forms.Button()
        Me.txtIP = New System.Windows.Forms.TextBox()
        Me.txtPuerto = New System.Windows.Forms.TextBox()
    End Sub

```

```

Me.Label1 = New System.Windows.Forms.Label()
Me.Label2 = New System.Windows.Forms.Label()
Me.txtTexto = New System.Windows.Forms.TextBox()
Me.btnEnviarMensaje = New System.Windows.Forms.Button()
Me.Nombre = New System.Windows.Forms.Label()
Me.txtNombre = New System.Windows.Forms.TextBox()
Me.ListBox1 = New System.Windows.Forms.ListBox()
Me.TextBox1 = New System.Windows.Forms.TextBox()
Me.BtnEnviarA = New System.Windows.Forms.Button()
Me.SuspendLayout()
'
'btnConectar
'
Me.btnConectar.Location = New System.Drawing.Point(226, 91)
Me.btnConectar.Name = "btnConectar"
Me.btnConectar.Size = New System.Drawing.Size(66, 23)
Me.btnConectar.TabIndex = 3
Me.btnConectar.Text = "Conectar"
Me.btnConectar.UseVisualStyleBackColor = True
'
'txtIP
'
Me.txtIP.Location = New System.Drawing.Point(55, 12)
Me.txtIP.Name = "txtIP"
Me.txtIP.Size = New System.Drawing.Size(165, 20)
Me.txtIP.TabIndex = 2
Me.txtIP.Text = "127.0.0.1"
'
'txtPuerto
'
Me.txtPuerto.Location = New System.Drawing.Point(55, 38)
Me.txtPuerto.Name = "txtPuerto"
Me.txtPuerto.Size = New System.Drawing.Size(165, 20)
Me.txtPuerto.TabIndex = 4
Me.txtPuerto.Text = "8050"
'
'Label1
'
Me.Label1.AutoSize = True
Me.Label1.Location = New System.Drawing.Point(12, 15)
Me.Label1.Name = "Label1"
Me.Label1.Size = New System.Drawing.Size(17, 13)
Me.Label1.TabIndex = 5
Me.Label1.Text = "IP"
'
'Label2
'
Me.Label2.AutoSize = True
Me.Label2.Location = New System.Drawing.Point(12, 41)
Me.Label2.Name = "Label2"
Me.Label2.Size = New System.Drawing.Size(38, 13)
Me.Label2.TabIndex = 6

```

```
Me.Label2.Text = "Puerto"
'
'txtTexto
'
Me.txtTexto.Enabled = False
Me.txtTexto.Location = New System.Drawing.Point(298, 12)
Me.txtTexto.Name = "txtTexto"
Me.txtTexto.Size = New System.Drawing.Size(544, 20)
Me.txtTexto.TabIndex = 7
'
'btnEnviarMensaje
'
Me.btnEnviarMensaje.Enabled = False
Me.btnEnviarMensaje.Location = New System.Drawing.Point(226, 117)
Me.btnEnviarMensaje.Name = "btnEnviarMensaje"
Me.btnEnviarMensaje.Size = New System.Drawing.Size(66, 23)
Me.btnEnviarMensaje.TabIndex = 8
Me.btnEnviarMensaje.Text = "Enviar"
Me.btnEnviarMensaje.UseVisualStyleBackColor = True
'
'Nombre
'
Me.Nombre.AutoSize = True
Me.Nombre.Location = New System.Drawing.Point(12, 67)
Me.Nombre.Name = "Nombre"
Me.Nombre.Size = New System.Drawing.Size(44, 13)
Me.Nombre.TabIndex = 10
Me.Nombre.Text = "Nombre"
'
'txtNombre
'
Me.txtNombre.Location = New System.Drawing.Point(55, 64)
Me.txtNombre.Name = "txtNombre"
Me.txtNombre.Size = New System.Drawing.Size(165, 20)
Me.txtNombre.TabIndex = 9
Me.txtNombre.Text = "Pedro"
'
'ListBox1
'
Me.ListBox1.FormattingEnabled = True
Me.ListBox1.Location = New System.Drawing.Point(55, 90)
Me.ListBox1.Name = "ListBox1"
Me.ListBox1.Size = New System.Drawing.Size(165, 134)
Me.ListBox1.TabIndex = 11
'
'TextBox1
'
Me.TextBox1.Location = New System.Drawing.Point(298, 45)
Me.TextBox1.Multiline = True
Me.TextBox1.Name = "TextBox1"
Me.TextBox1.Size = New System.Drawing.Size(544, 179)
Me.TextBox1.TabIndex = 12
```

```

'
'BtnEnviarA
'

Me.BtnEnviarA.Enabled = False
Me.BtnEnviarA.Location = New System.Drawing.Point(226, 146)
Me.BtnEnviarA.Name = "BtnEnviarA"
Me.BtnEnviarA.Size = New System.Drawing.Size(66, 23)
Me.BtnEnviarA.TabIndex = 13
Me.BtnEnviarA.Text = "Enviar a"
Me.BtnEnviarA.UseVisualStyleBackColor = True
'

'Form1
'

Me.AutoScaleDimensions = New System.Drawing.SizeF(6.0!, 13.0!)
Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font
Me.ClientSize = New System.Drawing.Size(857, 247)
Me.Controls.Add(Me.BtnEnviarA)
Me.Controls.Add(Me.TextBox1)
Me.Controls.Add(Me.ListBox1)
Me.Controls.Add(Me.Nombre)
Me.Controls.Add(Me.txtNombre)
Me.Controls.Add(Me.btnEnviarMensaje)
Me.Controls.Add(Me.txtTexto)
Me.Controls.Add(Me.Label2)
Me.Controls.Add(Me.Label1)
Me.Controls.Add(Me.txtPuerto)
Me.Controls.Add(Me.btnConectar)
Me.Controls.Add(Me.txtIP)
Me.Name = "Form1"
Me.Text = "Cliente"
Me.ResumeLayout(False)
Me.PerformLayout()

End Sub

Friend WithEvents btnConectar As System.Windows.Forms.Button
Friend WithEvents txtIP As System.Windows.Forms.TextBox
Friend WithEvents txtPuerto As System.Windows.Forms.TextBox
Friend WithEvents Label1 As System.Windows.Forms.Label
Friend WithEvents Label2 As System.Windows.Forms.Label
Friend WithEvents txtTexto As System.Windows.Forms.TextBox
Friend WithEvents btnEnviarMensaje As System.Windows.Forms.Button
Friend WithEvents Nombre As System.Windows.Forms.Label
Friend WithEvents txtNombre As System.Windows.Forms.TextBox
Friend WithEvents ListBox1 As System.Windows.Forms.ListBox
Friend WithEvents TextBox1 As System.Windows.Forms.TextBox
Friend WithEvents BtnEnviarA As System.Windows.Forms.Button

End Class

```

Introducción a la arquitectura de software

Desde que se desarrolla software, la complejidad asociada a su construcción ha ido creciendo exponencialmente. Originalmente, la actividad se visualizaba casi como una construcción artística, donde cada profesional aproximaba sus prácticas a las formas que conocía o le habían dado mejores resultados. En los tiempos que corren, sería inconcebible pensar un producto de software de esa manera, ya que se utilizarían recursos de manera inadecuada y sin sentido para llevar adelante actividades que resultan poco productivas cuando en realidad, ya existen estudios que avalan las maneras más efectivas de realizar lo mismo de forma eficiente y eficaz.

La ingeniería de software aporta formas, guías y técnicas para desarrollar software aprovechando el conocimiento adquirido para que los resultados obtenidos sean más previsibles. Estas formas permiten considerar buenas prácticas y, de esta manera, los desarrolladores pueden abstraerse a formas previamente utilizadas que han dados buenos resultados.

Existen actividades profesionales relacionadas con el desarrollo de productos de software, como la arquitectura de software, donde estos conceptos están muy afianzados y llevan un largo camino recorrido. Si pensamos en cómo un arquitecto concibe una casa, podremos observar que gran parte del esfuerzo se concentra en el diseño de la misma. Al diseñarla puede establecer las estructuras básicas que compondrán esa vivienda, y es en ese momento en el que quizás pueda reutilizar parte de diseños anteriores que definieron buenas formas de resolver problemas funcionales de otras viviendas. Siendo más específico, podría mencionar aspectos como la orientación, la ubicación de la cocina, las habitaciones y el escritorio, la distribución de la circulación entre los ambientes, las dimensiones según los requerimientos, etcétera. Se puede observar que en esta etapa del diseño se está lejos de considerar aspectos muy específicos, ya que ellos serán dependientes del diseño que se utilice.

Volviendo al mundo del desarrollo de software, podemos pensar en la arquitectura del software como el diseño previo a la programación. En esta etapa, pondremos especial énfasis en los aspectos estructurales que le dan al producto flexibilidad, extensibilidad, capacidad de reutilización y, además, que permiten reducir los costos de mantenimiento y mejorar la calidad del producto. Se podrá observar que no he mencionado ningún aspecto estrictamente relacionado

con la tecnología ni con los aspectos algorítmicos del producto ya que, seguramente, ambos serán en gran medida dependientes de la arquitectura que diseñemos. En definitiva, la arquitectura de nuestro software será el diseño estructural global con el que dotemos a nuestro sistema para lograr las cualidades antes mencionadas.

El diseño de software es una actividad que abarca muchas tareas, que generan responsabilidades concretas en los actores intervenientes en el desarrollo del producto. Estas tareas constituyen un factor clave para el éxito en lo referido a la concreción satisfactoria del objetivo planteado. Estas tareas podrían clasificarse jerárquicamente, y si bien no lo haremos aquí debido a que el texto no intenta atender esos aspectos de la ingeniería de software, visualizando el concepto de jerarquía podemos afirmar que la arquitectura de software estaría en la cima del diseño. En la actualidad, esta actividad ha cobrado mucha notoriedad y resulta muy común escuchar hablar del rol de arquitecto de software en las organizaciones.

Lo que expondremos de aquí en adelante no intenta ser una recopilación histórica de las arquitecturas más difundidas ni un tratado para el desarrollo formal de las arquitecturas de software. Las ideas sobre las que reflexionaremos a continuación intentan que quienes no poseen conocimientos sobre esta área puedan reconocer ciertos problemas bastante comunes en el desarrollo de software e intentar, de manera práctica, visualizar una posible solución. Esta solución no pretende ser la mejor pero sí lo suficientemente explícita y fácil de comprender para que el lector pueda apropiarse del aporte que propone la arquitectura planteada.

Pensemos en un sistema sencillo que intenta proporcionar la posibilidad de guardar datos suministrados por un usuario, modificarlos, eliminarlos y recuperarlos. A esto se lo conoce también por la sigla **CRUD** (*Create, Read, Update and Delete*). Básicamente, podemos detectar tres elementos muy característicos que deberían proporcionar funcionalidades muy distintas para satisfacer las necesidades planteadas. Estas funcionalidades se detallan a continuación:

- **Vista:** las relacionadas con la interacción con el usuario, la captura de los datos ingresados por él y la visualización de los datos obtenidos cuando se consulte para recuperar datos previamente almacenados.
- **Lógica:** las reglas lógicas que afecten a los datos ingresados generando nueva información a partir de ellos, o bien al requerir una consulta y, por medio de cálculos sobre los datos almacenados, obtener nuevos para ser visualizados.
- **Datos:** los aspectos referidos al tratamiento de los estados de los objetos para ser almacenados físicamente en las tablas de la base de datos.

De acuerdo a lo planteado, el esquema básico sería el que aparece en la figura 6.1.

Este esquema plantea tres divisiones lógicas que podemos denominar **capas**, entonces podemos hablar de la capa de vista, la capa de lógica y la capa de datos.

Estas capas tendrán una manifestación física al momento de crear la aplicación. Por ejemplo, cada capa podría construirse en un proyecto o un componente de software individual, lo que permitiría que se ejecuten en servidores independientes. Esto le otorga a un sistema niveles de seguridad adicional a los tradicionales e independencia respecto a los servicios que proporciona,

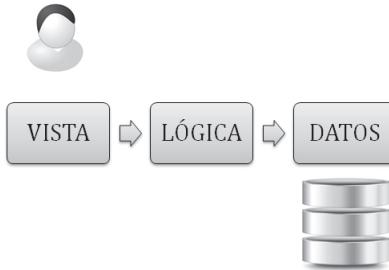


Figura 6.1.

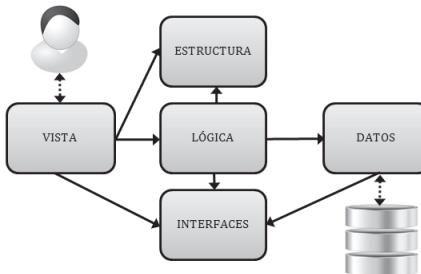
ya que cada componente de software prestará los servicios para los que fue desarrollado y estos servicios serán consumidos por los elementos que lo necesiten, constituyendo de esta manera una forma más optimizada respecto a la distribución de las responsabilidades y un mejor balanceo de las cargas de procesamiento en caso de ser necesario. Por otro lado, debemos mencionar que la orientación a objetos propone la distribución de las responsabilidades y la generación de colaboración entre las partes. Debido a esto, cuando utilizamos orientación a objetos, estamos frente a una de las formas metodológicas que mejor aprovecha las posibilidades de diseñar un software considerando las ventajas de realizar el diseño arquitectónico del mismo. En contraposición con esta postura, podríamos pensar en desarrollar un sistema con características monolíticas sin realizar ninguna división, con las consecuencias que ya se han experimentado hace algunas décadas atrás, cuando estas prácticas eran comunes. Este enfoque antiguo posee consecuencias no deseadas entre las que podemos mencionar: poca reutilización del código, poca flexibilidad, alto impacto de los errores puntuales sobre la totalidad del desarrollo, etcétera.

A continuación, comenzaremos a elaborar una arquitectura muy elemental con el objetivo de que el lector pueda introducirse en el tema. Para poder ver más de cerca lo planteado, pensemos en el siguiente ejemplo. Tenemos que desarrollar una parte de un sistema que permita almacenar, modificar, consultar y borrar los vales de dinero que una organización entrega a sus empleados. Los datos que manejan son:

Número de vale	Numérico
Importe	Numérico
Concepto	Texto
Fecha	Fecha
Legajo del empleado	Numérico
Hash de control	Texto

La idea será mantener lo suficientemente independientes los aspectos relacionados con la vista de la aplicación, la lógica y la manipulación de los datos.

Por **vista** entendemos todos los aspectos que estén relacionados con la interfaz gráfica del usuario, así como lo necesario para interpretar las acciones que este solicite y las formas de mostrarle la información.

**Figura 6.2.**

La **lógica** definirá las clases que modelan la solución que implementaremos y la lógica funcional, en nuestro caso, los vales.

El **almacenamiento** contendrá lo necesario para administrar aspectos referidos a la conexión con la base de datos y las transformaciones en sentido bidireccional de los estados de los objetos a los requerimientos de la base de datos relacional y viceversa.

En la figura 6.2 planteamos gráficamente la forma que adoptará nuestra arquitectura y vemos más precisamente el rol que cumple cada capa.

Antes de comenzar con la explicación pormenorizada, es oportuno aclarar que para el acceso a datos se ha utilizado ADO.NET, la aplicación fue construida como una solución de formularios de Windows, el lenguaje de programación es VB.NET y la base de datos SQL Server 2012.

Cada una de las capas fue desarrollada como un proyecto independiente, luego agrupados en una solución común. Esto último se realizó así pues facilita la explicación, aunque no constituye un requerimiento. Cada capa podría haberse desarrollado en soluciones diferentes luego de haber establecido las referencias a los componentes que necesita para funcionar.

Las flechas con líneas continuas representan referencias. Una referencia entre una capa y otra establece que la capa que posee la referencia posee visibilidad sobre los componentes públicos de la referenciada. Esto se hace imperioso cuando necesitamos tener visibilidad de tipos definidos en una capa desde otra.

Podemos observar que ahora pasamos de tres capas a cinco capas. Las cinco capas son: **vista**, **estructura**, **lógica**, **interfaces** y **datos**. Para nuestro ejemplo de vales, cada capa se denominará como lo indicamos con la palabra **Vales** antepuesta, por ejemolo, “**ValesVista**”. Pasamos de tres a cinco capas porque la capa lógica se dividió en dos. Dada una entidad, tendremos en la capa lógica “**ValesLogica**” su comportamiento (lo que se conoce normalmente como “las reglas de negocio”), y su estructura en la capa denominada “**ValesEstructura**”. Esto nos proporcionará más flexibilidad en el futuro, como explicaremos más adelante.

Por otro lado, aparece la capa “**ValesInterfaces**”, que contendrá las interfaces que utiliza el sistema. Las capas que necesiten hacer uso de ellas la referenciarán. El uso de las interfaces permitirá que clases de distintas capas compartan parte de su protocolo, además de poseer un tipo común entre ellas.

Por último, y antes de adentrarnos en el código, recuerde que la arquitectura propuesta no intenta mostrar los aspectos más sofisticados que se pueden lograr sino ser lo suficientemente

sencilla para que pueda introducirse en el tema y comprender de qué se trata este apasionante mundo de la arquitectura de software. Para reforzar la idea anterior, pensemos que un sistema medianamente complejo puede tener una docena o más de capas especializadas (seguridad, manejo de transacciones de datos, encriptación, administración de múltiples idiomas, etcétera).

Ha llegado el momento de desarrollar conceptualmente el contenido de nuestro proyecto y ver qué funciones cumple cada capa de la arquitectura propuesta, comenzando por la de datos.

Capa de datos

La capa de datos contendrá todas las clases que manejen o transformen datos en colecciones de objetos desde y hacia la base de datos. Encontraremos en ella tres tipos de clases:

- a) Clases muy relacionadas con la base de datos que estemos utilizando (en nuestro caso, Microsoft SQL 2012). Básicamente, se encargan de realizar tres actividades:
 1. Administrar la conexión a la base de datos.
 2. Construir los comandos (recuerde que estamos trabajando con ADO.NET), personalizándolos para cada acción requerida con respecto a la base de datos.
 3. Oficiar de **ORM** (*object-relational mapping* o mapeo objeto-relacional), en tanto tomará objetos con sus estados y se encargará de utilizar los datos que transportan en su estado para adecuarlos y construir la lista de parámetros que necesita el objeto comando, con el objetivo de ejecutar el procedimiento almacenado en la base de datos que corresponde. También, cuando se realiza una consulta, ADO.NET retornará un **DataTable** con la información obtenida. En esta capa se transforman los datos que contiene el **DataTable** en colecciones de objetos que se correspondan con entidades de la capa “**ValesEstructura**”.

Analicemos en detalle cómo ocurre todo esto. La solución se denomina **AdministracionVales** y, como puede observarse, posee los cinco proyectos: **ValesDatos**, **ValesEstructura**, **ValesInterfaces**, **ValesLogica** y **ValesVista**. Veremos qué contiene el proyecto.

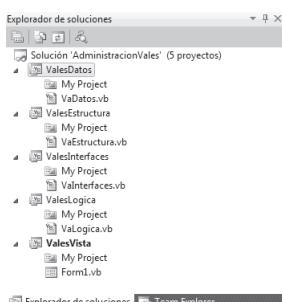


Figura 6.3.

El proyecto **ValesDatos** posee las siguientes clases: **VaDatos**, **VaConexion**, **VaComando** y **VaDatosParametros**.

VaDatos: es la clase que oficia realmente de ORM para los vales. Implementa la interfaz **ValesInterfaces.Iabmc**, lo que implica que implementa los siguientes métodos definidos en la interfaz:

```

Sub Alta(ByRef pObjeto As Object)
Sub Baja(ByRef pObjeto As Object)
Sub Modificacion(ByRef pObjeto As Object)
Sub Consulta(ByRef pObjeto As Object)
Function ConsultaRango(ByRef pObjeto1 As Object, _
    ByRef pObjeto2 As Object) As List(Of Object)
Function ConsultaIncremental(ByRef pObjeto As Object) _ 
    As List(Of Object)

```

VaConexion: retorna un objeto **SqlConnection** configurado con la cadena de conexión necesaria para acceder a la base de datos requerida.

VaComando: construye objetos **SqlCommand** con la configuración adecuada, dependiendo del requerimiento (alta, baja, modificación o consultas). También ejecuta ese comando logrando resultados reales sobre la base de datos.

VaDatosParametros: permite crear objetos que representan a un parámetro que se corresponde con el requerimiento de un procedimiento almacenado. La representación la realiza a través de: el nombre del parámetro, el tipo del parámetro (que será coincidente con algún tipo **SqlDbType**) y el valor que ese parámetro deberá llevar al momento de ejecutar alguna acción en la base de datos.

Antes de interiorizarnos en cada una de las clases mencionadas con el objetivo de analizar en detalle cómo realiza lo enunciado, haremos un apartado para reflejar cómo está construida la base de datos **GESTION** y su tabla **Vale**. La base de datos **GESTION** es una base de datos Microsoft SQL 2012 y la estructura de la tabla **Vale** es la siguiente:

Nombre de columna	Tipo de datos	Permitir valores NULL
Va_Número	numeric(10, 0)	<input type="checkbox"/>
Va.Importe	numeric(6, 2)	<input checked="" type="checkbox"/>
Va_Concepto	nvarchar(50)	<input checked="" type="checkbox"/>
Va_Fecha	datetime	<input checked="" type="checkbox"/>
Va_Legajo	numeric(6, 0)	<input checked="" type="checkbox"/>
Va_Hash	nvarchar(50)	<input checked="" type="checkbox"/>

Figura 6.4.

Como se puede observar, todos los campos menos el número de vale `Va_Número` aceptan valores nulos. En particular, `Va_Número` no los acepta porque la idea es que cumpla el rol de identificador del vale, entonces no puede aceptar valores nulos ni repetidos. Esto último lo logramos marcándolo como clave principal, hecho que se puede constatar al ver el ícono con forma de llave que lo acompaña a la izquierda (figura 6.4).

También la base de datos posee procedimientos almacenados para cada una de las acciones que se desea realizar:

- Alta
- Baja
- Modificación
- Consulta por código
- Consulta por rango de códigos
- Consulta incremental por concepto

El nombre de los procedimientos almacenados para cada una de estas acciones es:

- `ProcedimientoAlta`
- `ProcedimientoBaja`
- `ProcedimientoModificar`
- `ProcedimientoConsulta`
- `ProcedimientoConsultaDesdeHasta`
- `ProcedimientoConsultaIncremental`

Para crear estos procedimientos almacenados, se ha utilizado un archivo que contiene el código que los genera. Su nombre es `SQLQueryCrearProcedimientos.sql` y a continuación está su contenido.

```
-- =====
-- Author:      Dario Cardacci
-- Create date: 01/12/2012
-- Description: Crear Procedimientos
-- =====
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
USE GESTION;
GO
IF OBJECT_ID('ProcedimientoAlta') is not null
DROP PROCEDURE ProcedimientoAlta
GO
CREATE PROCEDURE ProcedimientoAlta
    @Va_Número numeric(10,0),
    @Va_Importe numeric(6,2),
```

```
@Va_Concepto nvarchar(50),
@Va_Fecha datetime,
@Va_Legajo numeric(6,0),
@Va_Hash nvarchar(50)
AS
    SET NOCOUNT ON;
    INSERT INTO Vale
        (Va_Numer, Va_Importe, Va_Concepto, Va_Fecha, Va_Legajo, Va_Hash)
    VALUES
        (@Va_Numer, @Va_Importe, @Va_Concepto, @Va_Fecha, @Va_Legajo, @Va_Hash)
GO
IF OBJECT_ID('ProcedimientoBaja') is not null
DROP PROCEDURE ProcedimientoBaja
GO
CREATE PROCEDURE ProcedimientoBaja
    @Va_Numer numeric(10,0)
AS
    SET NOCOUNT ON;
    DELETE Vale
        WHERE Va_Numer = @Va_Numer
GO
IF OBJECT_ID('ProcedimientoModificar') is not null
DROP PROCEDURE ProcedimientoModificar
GO
CREATE PROCEDURE ProcedimientoModificar
    @Va_Numer numeric(10,0),
    @Va_Importe numeric(6,2),
    @Va_Concepto nvarchar(50),
    @Va_Fecha datetime,
    @Va_Legajo numeric(6,0),
    @Va_Hash nvarchar(50)
AS
    SET NOCOUNT ON;
    UPDATE Vale
        SET
            Va_Importe = @Va_Importe,
            Va_Concepto = @Va_Concepto,
            Va_Fecha = @Va_Fecha,
            Va_Legajo = @Va_Legajo,
            Va_Hash = @Va_Hash
        WHERE
            Va_Numer = @Va_Numer
GO
IF OBJECT_ID('ProcedimientoConsulta') is not null
DROP PROCEDURE ProcedimientoConsulta
GO
CREATE PROCEDURE ProcedimientoConsulta
```

```

@Va_Numer0 numeric(10,0)
AS
    SET NOCOUNT ON;
    Select * From Vale
    WHERE
        Va_Numer0 = @Va_Numer0
GO

IF OBJECT_ID('ProcedimientoConsultaDesdeHasta') is not null
DROP PROCEDURE ProcedimientoConsultaDesdeHasta
GO
CREATE PROCEDURE ProcedimientoConsultaDesdeHasta
    @Va_Numer01 numeric(10,0),
    @Va_Numer02 numeric(10,0)
AS
    SET NOCOUNT ON;
    Select * From Vale
    WHERE
        Va_Numer0 Between @Va_Numer01 and @Va_Numer02
GO

IF OBJECT_ID('ProcedimientoConsultaIncremental') is not null
DROP PROCEDURE ProcedimientoConsultaIncremental
GO
CREATE PROCEDURE ProcedimientoConsultaIncremental
    @Va_Concepto nvarchar(50)
AS
    SET NOCOUNT ON;
    Select * From Vale
    WHERE
        Va_Concepto like @Va_Concepto + '%'

```

Si lo abrimos en la base de datos y lo ejecutamos, se crearán los procedimientos almacenados como se ve en la figura 6.5.

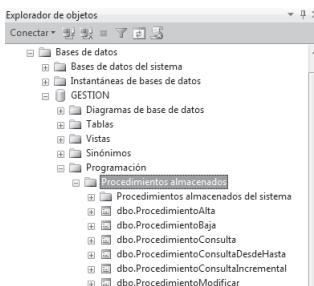


Figura 6.5.

El código de cada procedimiento almacenado es:

■ ProcedimientoAlta

```
USE [GESTION]
GO
***** Object: StoredProcedure [dbo].[ProcedimientoAlta]      Script Date: 04/12/2012
14:14:58 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[ProcedimientoAlta]
    @Va_Numer0 numeric(10,0),
    @Va_Importe numeric(6,2),
    @Va_Concepto nvarchar(50),
    @Va_Fecha datetime,
    @Va_Legajo numeric(6,0),
    @Va_Hash nvarchar(50)
AS
    SET NOCOUNT ON;
    INSERT INTO Vale
    (Va_Numer0, Va_Importe, Va_Concepto, Va_Fecha, Va_Legajo, Va_Hash)
    VALUES
    (@Va_Numer0, @Va_Importe, @Va_Concepto, @Va_Fecha, @Va_Legajo, @Va_Hash)
```

■ ProcedimientoBaja

```
USE [GESTION]
GO
***** Object: StoredProcedure [dbo].[ProcedimientoBaja]      Script Date: 04/12/2012
14:16:07 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[ProcedimientoBaja]
    @Va_Numer0 numeric(10,0)
AS
    SET NOCOUNT ON;
    DELETE Vale
    WHERE Va_Numer0 = @Va_Numer0
```

■ ProcedimientoModificar

```
USE [GESTION]
GO
/******** Object: StoredProcedure [dbo].[ProcedimientoModificar]      Script Date:
04/12/2012 14:17:08 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[ProcedimientoModificar]
    @Va_Numer0 numeric(10,0),
    @Va_Importe numeric(6,2),
    @Va_Concepto  nvarchar(50),
    @Va_Fecha datetime,
    @Va_Legajo numeric(6,0),
    @Va_Hash  nvarchar(50)
AS
    SET NOCOUNT ON;
    UPDATE Vale
        SET
            Va_Importe = @Va_Importe,
            Va_Concepto = @Va_Concepto,
            Va_Fecha = @Va_Fecha,
            Va_Legajo = @Va_Legajo,
            Va_Hash = @Va_Hash
        WHERE
            Va_Numer0 = @Va_Numer0
```

■ ProcedimientoConsulta

```
USE [GESTION]
GO
/******** Object: StoredProcedure [dbo].[ProcedimientoConsulta]      Script Date:
04/12/2012 14:17:51 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[ProcedimientoConsulta]
    @Va_Numer0 numeric(10,0)
AS
    SET NOCOUNT ON;
    Select * From Vale
    WHERE
        Va_Numer0 = @Va_Numer0
```

■ ProcedimientoConsultaDesdeHasta

```
USE [GESTION]
GO
***** Object: StoredProcedure [dbo].[ProcedimientoConsultaDesdeHasta] Script Date:
04/12/2012 14:18:36 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[ProcedimientoConsultaDesdeHasta]
    @Va_Numero1 numeric(10,0),
    @Va_Numero2 numeric(10,0)
AS
    SET NOCOUNT ON;
    Select * From Vale
    WHERE
        Va_Numero Between @Va_Numero1 and @Va_Numero2
```

■ ProcedimientoConsultaIncremental

```
USE [GESTION]
GO
***** Object: StoredProcedure [dbo].[ProcedimientoConsultaIncremental] Script
Date: 04/12/2012 14:20:03 *****/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
ALTER PROCEDURE [dbo].[ProcedimientoConsultaIncremental]
    @Va_Concepto nvarchar(50)
AS
    SET NOCOUNT ON;
    Select * From Vale
    WHERE
        Va_Concepto like @Va_Concepto + '%'
```

Una vez visto el contenido de nuestra base de datos GESTION, retornemos a ver en detalle el contenido de las clases de la capa “**ValesDatos**” y cómo funcionan internamente. Esta capa es apuntada por la capa “**ValesLogica**” y apunta a “**ValesEstructura**” y “**ValesInterfaces**”.

La clase VaDatos posee la siguiente implementación:

```
Public Class VaDatos
    Implements ValesInterfaces.Iabmc

    Public Sub Alta(ByRef pObjeto As Object) Implements _
        ValesInterfaces.Iabmc.Altar

        Dim Vobjeto As ValesEstructura.VaEstructura = _
            Activator.CreateInstance(pObjeto.GetType)
        Vobjeto = pObjeto
        Dim VlistaParametros As New List(Of VaDatosParametros)
        VlistaParametros.AddRange({ _
            New VaDatosParametros("@Va_Numer0", SqlDbType.Decimal, Vobjeto.Numer0), _
            New VaDatosParametros("@Va_Importe", SqlDbType.Decimal, Vobjeto.Importe), _
            New VaDatosParametros("@Va_Concepto", SqlDbType.NVarChar, Vobjeto.Concepto), _
            New VaDatosParametros("@Va_Fecha", SqlDbType.DateTime, Vobjeto.Fecha), _
            New VaDatosParametros("@Va_Legajo", SqlDbType.Decimal, Vobjeto.Legajo), _
            New VaDatosParametros("@Va_Hash", SqlDbType.NVarChar, Vobjeto.Hash)})
        VaComando.ObjEjec("ProcedimientoAlta", VlistaParametros)
    End Sub

    Public Sub Baja(ByRef pObjeto As Object) Implements ValesInterfaces.Iabmc.Baja
        Dim Vobjeto As ValesEstructura.VaEstructura = _
            Activator.CreateInstance(pObjeto.GetType)
        Vobjeto = pObjeto
        Dim VlistaParametros As New List(Of VaDatosParametros)
        VlistaParametros.Add(New VaDatosParametros("@Va_Numer0", SqlDbType.Decimal, _
            Vobjeto.Numer0))
        VaComando.ObjEjec("ProcedimientoBaja", VlistaParametros)
    End Sub

    Public Sub Consulta(ByRef pObjeto As Object) Implements _
        ValesInterfaces.Iabmc.Consulta
        Dim Vobjeto As ValesEstructura.VaEstructura = _
            Activator.CreateInstance(pObjeto.GetType)
        Vobjeto = pObjeto
        Dim VlistaParametros As New List(Of VaDatosParametros)
        VlistaParametros.Add(New VaDatosParametros("@Va_Numer0", SqlDbType.Decimal, _
            Vobjeto.Numer0))
        Dim Vdt As DataTable = VaComando.ObjEjecRetDataTable("ProcedimientoConsulta", _
            VlistaParametros)
        Vobjeto.Importe = Vdt.Rows(0).Item("Va_Importe")
        Vobjeto.Concepto = Vdt.Rows(0).Item("Va_Concepto")
        Vobjeto.Fecha = Vdt.Rows(0).Item("Va_Fecha")
        Vobjeto.Legajo = Vdt.Rows(0).Item("Va_Legajo")
        Vobjeto.Hash = Vdt.Rows(0).Item("Va_Hash")
    End Sub
```

```
Public Function ConsultaIncremental(ByRef pObjeto As Object) As _
    System.Collections.Generic.List(Of Object) Implements _
    ValesInterfaces.Iabmc.ConsultaIncremental
    Dim Vobjeto As ValesEstructura.VaEstructura = _
        Activator.CreateInstance(pObjeto.GetType)
    Vobjeto = pObjeto
    Dim VlistaParametros As New List(Of VaDatosParametros)
    VlistaParametros.Add(New VaDatosParametros("@Va_Concepto", SqlDbType.NVarChar, _
        Vobjeto.Concepto))
    Dim Vdt As DataTable = _
        VaComando.ObjEjecRetDataTable("ProcedimientoConsultaIncremental", _
        VlistaParametros)
    Dim ClienteLista As New List(Of Object)
    If Vdt.Rows.Count > 0 Then
        For Each Vdr As DataRow In Vdt.Rows
            ClienteLista.Add(New ValesEstructura.VaEstructura(Vdr.Item(0), _
                Vdr.Item(1), Vdr.Item(2), Vdr.Item(3), Vdr.Item(4), Vdr.Item(5)))
        Next
    End If
    Return ClienteLista
End Function

Public Function ConsultaRango(ByRef pObjeto1 As Object, ByRef pObjeto2 As Object) _ 
    As System.Collections.Generic.List(Of Object) Implements _
    ValesInterfaces.Iabmc.ConsultaRango
    Dim Vobjeto1 As ValesEstructura.VaEstructura = _
        Activator.CreateInstance(pObjeto1.GetType)
    Dim Vobjeto2 As ValesEstructura.VaEstructura = _
        Activator.CreateInstance(pObjeto2.GetType)
    Vobjeto1 = pObjeto1
    Vobjeto2 = pObjeto2
    Dim VlistaParametros As New List(Of VaDatosParametros)
    VlistaParametros.AddRange({ _
        New VaDatosParametros("@Va_Numer01", SqlDbType.Decimal, Vobjeto1.Numero), _
        New VaDatosParametros("@Va_Numer02", SqlDbType.Decimal, Vobjeto2.Numero)})
    Dim Vdt As DataTable = VaComando.ObjEjecRetDataTable( _
        "ProcedimientoConsultaDesdeHasta", VlistaParametros)
    Dim ClienteLista As New List(Of Object)
    If Vdt.Rows.Count > 0 Then
        For Each Vdr As DataRow In Vdt.Rows
            ClienteLista.Add(New ValesEstructura.VaEstructura(Vdr.Item(0), _
                Vdr.Item(1), Vdr.Item(2), Vdr.Item(3), Vdr.Item(4), Vdr.Item(5)))
        Next
    End If
    Return ClienteLista
End Function

Public Sub Modificacion(ByRef pObjeto As Object) Implements _
    ValesInterfaces.Iabmc.Modificacion
    Dim Vobjeto As ValesEstructura.VaEstructura = _
        Activator.CreateInstance(pObjeto.GetType)
```

```

Vobjeto = pObjeto
Dim VlistaParametros As New List(Of VaDatosParametros)
VlistaParametros.AddRange({ _
    New VaDatosParametros("@Va_Numer0", SqlDbType.Decimal, Vobjeto.Numer0), _
    New VaDatosParametros("@Va_Importe", SqlDbType.Decimal, Vobjeto.Importe), _
    New VaDatosParametros("@Va_Concepto", SqlDbType.NVarChar, Vobjeto.Concepto), _
    New VaDatosParametros("@Va_Fecha", SqlDbType.DateTime, Vobjeto.Fecha), _
    New VaDatosParametros("@Va_Legajo", SqlDbType.Decimal, Vobjeto.Legajo), _
    New VaDatosParametros("@Va_Hash", SqlDbType.NVarChar, Vobjeto.Hash)})}
VaComando.ObjEjec("ProcedimientoModificar", VlistaParametros)
End Sub

End Class

```

Analicemos detenidamente este código, por ejemplo el procedimiento Alta. Podemos observar que recibe un objeto en el parámetro `pObjeto`, de hecho será un objeto de tipo `Vale`. A pesar de esto, se puede observar que el parámetro `pObjeto` es de tipo `Object` y esto es debido a la definición que se le ha dado en la interfaz `ValesInterfaces.Iabmc`. El motivo de que sea de tipo `Object` radica en que en un sistema real, además de vales tendríamos otras entidades. Al ser de tipo `Object`, esta interfaz servirá para cualquiera de las entidades definidas, y si el parámetro lo personalizamos al tipo `ValesEstructura.VaEstructura`, solo servirá para los vales. En este caso se ha optado por adaptar el objeto recibido dentro del procedimiento, asignándolo a una variable de tipo `ValesEstructura.VaEstructura`. Para ello, se crea una instancia del tipo del objeto recibido y se le asigna lo que arribó en el parámetro `pObjeto` para poder tener acceso a la interfaz del tipo `VaEstructura` y así a los datos que arribaron en su estado.

```

Dim Vobjeto As ValesEstructura.VaEstructura = _
Activator.CreateInstance(pObjeto.GetType)
Vobjeto = pObjeto

```

Luego, se crea una lista del tipo `VaDatosParametros` con el objetivo de preparar los parámetros necesarios para ejecutar el procedimiento almacenado y que, de esta forma, suceda el Alta del vale. Cada parámetro se define con tres elementos que son: el nombre del parámetro en el procedimiento almacenado, el tipo de dato (que deberá coincidir con alguno definido en los `SqlDbType`) y el valor que deseamos grabar al momento de ejecutar el alta.

```

Dim VlistaParametros As New List(Of VaDatosParametros)
VlistaParametros.AddRange({ _
    New VaDatosParametros("@Va_Numer0", SqlDbType.Decimal, Vobjeto.Numer0), _
    New VaDatosParametros("@Va_Importe", SqlDbType.Decimal, Vobjeto.Importe), _
    New VaDatosParametros("@Va_Concepto", SqlDbType.NVarChar, Vobjeto.Concepto), _

```

```
New VaDatosParametros("@Va_Fecha", SqlDbType.DateTime, Vobjeto.Fecha), _
New VaDatosParametros("@Va_Legajo", SqlDbType.Decimal, Vobjeto.Legajo), _
New VaDatosParametros("@Va_Hash", SqlDbType.NVarChar, Vobjeto.Hash)})}
```

Finalmente, se utiliza el método `ObjEjec` que define la clase `VaComando`. Este método se encarga de ejecutar el procedimiento almacenado que se le pasa en el primer parámetro y utiliza la lista de parámetros que se le envía en el segundo parámetro del método.

```
VaComando.ObjEjec("ProcedimientoAlta", VlistaParametros)
```

El resto de los métodos que implementa esta clase funciona de manera similar a lo explicado para el `Alta`, pero invocando al procedimiento almacenado correspondiente.

Un detalle a considerar es lo que implementa el método `ConsultaRango` debido a que en lugar de recibir un parámetro recibe dos: `pObjeto1` y `pObjeto2`. Esto es debido a que la consulta por rango necesita dos valores; el primero oficia como el valor inferior del rango a filtrar y el segundo como el valor superior, el resto se corresponde con la misma lógica expuesta anteriormente.

La clase `VaConexion` provee la conexión a la base de datos y posee la siguiente implementación:

```
Public Class VaConexion
    Private Shared VobjConexion As SqlConnection
    Shared Function ObjConexion(Optional ByVal pStringDeConexion As String = _
        "Data Source=WIN-BI2EL3C0HF8\SQL2912" & _
        ";Initial Catalog=GESTION;Integrated Security=True") _
        As SqlConnection
        If VobjConexion Is Nothing Then VobjConexion = _
            New SqlConnection(pStringDeConexion)
        If VobjConexion.State = ConnectionState.Closed Then VobjConexion.Open()
        Return VobjConexion
    End Function
End Class
```

Esta clase posee un método privado denominado `VobjConexion` que retorna una conexión del tipo `SqlConnection`. Se puede observar que el método `Shared` permite utilizar el método sin necesidad de instanciar la clase (por ejemplo, `VaConexion.VobjConexion`). El parámetro que posee es opcional y en caso de que no se pase una cadena de conexión personalizada, se utilizará la colocada por defecto.

```
"Data Source=WIN-BI2EL3C0HF8\SQL2912;Initial Catalog=GESTION;Integrated Security=True"
```

El lector debe considerar que esta cadena es la utilizada en el ejemplo y tendrá que ajustar lo necesario para que funcione con su gestor de base de datos en el Data Source y que la misma responda de manera adecuada en su máquina.

El código implementado mantiene solo una instancia de esta clase. Lo logra indagando al momento de retornar la conexión si ya existe, en caso afirmativo retorna la conexión que ya instanció, en caso contrario instancia una y la retorna. También verifica el estado de la misma y si está cerrada la abre.

La clase VaComando

La clase `VaComando` posee la responsabilidad de administrar los `SqlCommand` que se necesiten para la efectiva ejecución de las acciones en la base de datos.

```
Public Class VaComando
    Private Shared VobjComando As SqlCommand
    Private Shared Function ObjComando(ByVal pNombreProcedimiento As String, _
                                         ByVal pConexion As SqlConnection, ByVal pParametros As _
                                         List(Of VaDatosParametros)) As SqlCommand
        VobjComando = New SqlCommand
        VobjComando.CommandText = pNombreProcedimiento
        VobjComando.CommandType = CommandType.StoredProcedure
        VobjComando.Connection = pConexion
        For Each Vparam As VaDatosParametros In pParametros
            Dim Vp1 As New SqlParameter
            Vp1.ParameterName = Vparam.NombreParametro
            Vp1.SqlDbType = Vparam.TipoParametro
            Vp1.Value = Vparam.Valor
            VobjComando.Parameters.Add(Vp1)
        Next
        Return VobjComando
    End Function
    Shared Function ObjEjecRetDataTable(ByVal pNombreProcedimiento As String, _
                                         ByVal pParametros As List(Of VaDatosParametros)) As DataTable
        Dim Vda As New SqlDataAdapter(ObjComando(pNombreProcedimiento, _
                                                 VaConexion.ObjConexion, pParametros))
        Dim Vdt As New DataTable
        Vda.Fill(Vdt)
        Return Vdt
    End Function
    Shared Sub ObjEjec(ByVal pNombreProcedimiento As String, ByVal pParametros As _
                      List(Of VaDatosParametros))
        Try
            ObjComando(pNombreProcedimiento, VaConexion.ObjConexion, _
                       pParametros).ExecuteNonQuery()
        Catch ex As Exception
            End Try
    End Sub
End Class
```

Esta clase posee la responsabilidad de retornar objetos `SqlCommand` configurados para ejecutar un procedimiento almacenado. Tiene tres métodos; el primero es un método privado y compartido denominado `ObjComando`, que recibe el nombre del procedimiento almacenado a ejecutar, una conexión válida de tipo `SqlConnection` a la base de datos y una lista de parámetros cargados con sus valores respectivos para ser utilizados por el procedimiento almacenado.

```
Private Shared Function ObjComando(ByVal pNombreProcedimiento As String, _
    ByVal pConexion As SqlConnection, ByVal pParametros As _
    List(Of VaDatosParametros)) As SqlCommand
```

Básicamente, esta función logra configurar el objeto `SqlCommand` que retornará, adecuando las propiedades `CommandText` con el nombre del procedimiento almacenado recibido por parámetro `CommandType`, indicando que lo que se ejecutará es un procedimiento almacenado (`CommandType.StoredProcedure`) `Connection` con la conexión recibida por parámetro y, finalmente, recorre la colección de parámetros recibidos y por cada uno crea un objeto `SqlParameter`, lo configura con la información que trae la lista de parámetros recibida y los agrega a la colección `Parameters` del objeto `SqlCommand`. A continuación, se puede observar esto último.

```
For Each Vparam As VaDatosParametros In pParametros
    Dim Vp1 As New SqlParameter
    Vp1.ParameterName = Vparam.NombreParametro
    Vp1.SqlDbType = Vparam.TipoParametro
    Vp1.Value = Vparam.Valor
    VobjComando.Parameters.Add(Vp1)
Next
```

El segundo método es compartido y se denomina `ObjEjecRetDataTable`. Su objetivo es retornar un `DataTable` con el resultado de ejecutar un procedimiento almacenado. Recibe como parámetros el nombre del procedimiento almacenado a ejecutar y la lista de parámetros que utilizará el objeto comando para configurarse y ejecutar el procedimiento almacenado requerido. Como puede observarse, se crea un objeto `SqlDataAdapter` y al momento de instanciarlo se le pasa, por medio de su constructor, un objeto `SqlCommand` configurado que se recibe como resultado de invocar al método `ObjComando`. Ese objeto `SqlDataAdapter` es el que se encarga de llenar el `DataTable` que se retornará por medio de su método `Fill`.

```
Dim Vda As New SqlDataAdapter(ObjComando(pNombreProcedimiento, _
    VaConexion.ObjConexion, pParametros))
Dim Vdt As New DataTable
Vda.Fill(Vdt)
Return Vdt
```

El tercer método que posee esta clase es ObjEjec. Su implementación se realiza a partir de un procedimiento y su objetivo es, simplemente, ejecutar un comando y no retornar nada. Es utilizado, por ejemplo, en las altas, bajas y modificaciones. Es oportuno mencionar que algunos desarrolladores podrían preferir que estas acciones retornasen algún valor que indique el éxito o no de la ejecución de la operación, y esta sería una postura muy válida. No obstante, optamos por no hacerlo y de esta forma simplificar el ejemplo.

La clase VaDatosParametros

Esta clase define la estructura necesaria para poder trabajar con los datos que un parámetro de un procedimiento almacenado requiere. Estos datos son: el nombre del parámetro, el tipo del parámetro y el valor que contiene. Su implementación es la siguiente:

```
Public Class VaDatosParametros
    Sub New()
        End Sub
        Sub New(ByVal pNombre As String, ByVal pTipo As SqlDbType, pValor As String)
            Me.NombreParametro = pNombre
            Me.TipoParametro = pTipo
            Me.Valor = pValor
        End Sub
        Private VnombreParametro As String
        Public Property NombreParametro() As String
            Get
                Return VnombreParametro
            End Get
            Set(ByVal value As String)
                VnombreParametro = value
            End Set
        End Property
        Private VtipoParametro As SqlDbType
        Public Property TipoParametro() As SqlDbType
            Get
                Return VtipoParametro
            End Get
            Set(ByVal value As SqlDbType)
                VtipoParametro = value
            End Set
        End Property
        Private Vvalor As String
        Public Property Valor() As String
            Get
                Return Vvalor
            End Get
            Set(ByVal value As String)
                Vvalor = value
            End Set
        End Property
    End Class
```

Capa de interfaces

La capa “**ValesInterfaces**” fue creada para alojar las interfaces del sistema. En nuestro ejemplo, solo alojará una interfaz para lograr homogeneizar la mensajería entre las capas “**ValesVista**”, “**ValesLogica**” y “**ValesDatos**”, y por lo tanto será apuntada por ellas.

La interfaz que posee esta capa de denominación **Iabmc** y las definiciones que contiene son las siguientes:

```
Public Interface Iabmc
    Sub Alta(ByRef pObjeto As Object)
    Sub Baja(ByRef pObjeto As Object)
    Sub Modificacion(ByRef pObjeto As Object)
    Sub Consulta(ByRef pObjeto As Object)
    Function ConsultaRango(ByRef pObjeto1 As Object, ByRef pObjeto2 As Object) As _
        List(Of Object)
    Function ConsultaIncremental(ByRef pObjeto As Object) As _
        List(Of Object)
End Interface
```

Como mencionamos anteriormente, los parámetros son de tipo **Object**, lo que facilita su reutilización en otras entidades además de Vales si fuera necesario.

Capa de estructura

La capa “**ValesEstructura**” es la encargada de contener la parte estructural de las clases que representan las entidades necesarias para el sistema de vales. Esta capa es apuntada por “**ValesVista**” y “**ValesLogica**”. En nuestro ejemplo se simplificó al máximo el alcance estructural de la aplicación de manera que con una sola entidad se puede representar significativamente lo que se necesita. Está claro que es un caso extremo y en los sistemas reales la modelización del dominio del problema involucrará más de una entidad, lo que se verá reflejado en un conjunto de clases interrelacionadas.

Esta capa contiene la clase **VaEstructura**. Esta clase posee solo la parte estructural de un vale. Las instancias de ella contendrán los datos de los vales. El concepto de vale se ha representado por dos clases: **VaEstructura** y **VaLogica**, esta última se analizará cuando abordemos la capa “**ValesLogica**”. El motivo de la división responde a poder exemplificar que en casos extremos, podemos pensar en combinar una estructura con distintos comportamientos y viceversa a través de la *composición*. Esto no siempre es necesario pero si lo fuera, otorga una manera versátil de lograrlo. La implementación de la clase **VaEstructura** es:

```
Public Class VaEstructura
    Sub New()
        End Sub
        Sub New(ByVal pNumero As Integer, ByVal pImporte As Integer, ByVal pConcepto As String, ByVal pFecha As Date, ByVal pLegajo As Integer, ByVal pHash As String)
            Me.Numero = pNumero
            Me.Importe = pImporte
            Me.Concepto = pConcepto
            Me.Fecha = pFecha
            Me.Legajo = pLegajo
            Me.Hash = pHash
        End Sub
        Private Vnumero As Integer
        Public Property Numero() As Integer
            Get
                Return Vnumero
            End Get
            Set(ByVal value As Integer)
                Vnumero = value
            End Set
        End Property
        Private Vimporte As Integer
        Public Property Importe() As Integer
            Get
                Return Vimporte
            End Get
            Set(ByVal value As Integer)
                Vimporte = value
            End Set
        End Property
        Private Vconcepto As String
        Public Property Concepto() As String
            Get
                Return Vconcepto
            End Get
            Set(ByVal value As String)
                Vconcepto = value
            End Set
        End Property
        Private Vfecha As Date
        Public Property Fecha() As Date
            Get
                Return Vfecha
            End Get
            Set(ByVal value As Date)
                Vfecha = value
            End Set
        End Property
        Private Vlegajo As Integer
        Public Property Legajo() As Integer
```

```

    Get
        Return Vlegajo
    End Get
    Set(ByVal value As Integer)
        Vlegajo = value
    End Set
End Property
Private Vhash As String
Public Property Hash() As String
    Get
        Return Vhash
    End Get
    Set(ByVal value As String)
        Vhash = value
    End Set
End Property
End Class

```

Capa de lógica

La capa “**ValesLogica**” posee la clase **Valogica**. Ella contiene la parte dinámica de un vale, es decir, las acciones que puede realizar. Esta capa apunta a las capas “**ValesInterfaces**” y “**ValesEstructura**”, y es apuntada por la capa “**ValesVista**”. También en esta capa y en esta clase en particular se deben definir las denominadas “reglas del negocio” para los vales: la lógica implícita al comportamiento de un vale y los cálculos que este debe realizar. En nuestro caso particular, y dentro del escenario del ejemplo propuesto, esto está representado por la posibilidad de calcular un valor **Hash** sobre el vale. También es oportuno mencionar que las otras funcionalidades están dadas por las acciones definidas en la interfaz **Iabm**. En la implementación de los métodos: Alta, Baja, Modificacion, Consulta, ConsultaIncremental y ConsultaRango que a continuación podremos observar, notamos que, básicamente, se recibe en el parámetro **pObjeto** un objeto y se lo pasa a la clase **VaDatos** de la capa “**ValeDatos**”. Esto, que parecería ser una pascuala por la que transitan los objetos sin ningún valor agregado, es así porque no hemos querido agregarle más “lógica de negocio” al ejemplo, pero en caso de existir, este sería el lugar donde se debe colocar la programación asociada a las mismas. La implementación de la clase **VaLogica** es:

```

Public Class VaLogica
    Implements ValesInterfaces.Iabmc

    Dim VvaleData As New ValesDatos.VaDatos

    Public Sub Alta(ByRef pObjeto As Object) Implements ValesInterfaces.Iabmc.Alta
        DirectCast(pObjeto, ValesEstructura.VaEstructura).Hash = _
            Me.CalcularHash(pObjeto)
        VvaleData.Alta(pObjeto)
    End Sub

```

```

End Sub

Public Sub Baja(ByRef pObjeto As Object) Implements ValesInterfaces.Iabmc.Baja
    VvaleData.Baja(pObjeto)
End Sub

Public Sub Consulta(ByRef pObjeto As Object) Implements _
    ValesInterfaces.Iabmc.Consulta
    VvaleData.Consulta(pObjeto)
End Sub

Public Function ConsultaIncremental(ByRef pObjeto As Object) As _
    System.Collections.Generic.List(Of Object) Implements
    ValesInterfaces.Iabmc.ConsultaIncremental
    Return VvaleData.ConsultaIncremental(pObjeto)
End Function

Public Function ConsultaRango(ByRef pObjeto1 As Object, ByRef pObjeto2 As Object) _
    As System.Collections.Generic.List(Of Object) Implements
    ValesInterfaces.Iabmc.ConsultaRango
    Return VvaleData.ConsultaRango(pObjeto1, pObjeto2)
End Function

Public Sub Modificacion(ByRef pObjeto As Object) Implements
    ValesInterfaces.Iabmc.Modificacion
    DirectCast(pObjeto, ValesEstructura.VaEstructura).Hash = _
        Me.CalcularHash(pObjeto)
    VvaleData.Modificacion(pObjeto)
End Sub

Public Function CalcularHash(ById pObjeto As ValesEstructura.VaEstructura) As _
    Integer
    Return (pObjeto.Numero.ToString & pObjeto.Importe.ToString &
        pObjeto.Concepto & pObjeto.Fecha.ToString &
        pObjeto.Legajo.ToString).GetHashCode
End Function
End Class

```

Capa de vista

La capa “**ValesVista**” es la encargada de contener las clases referidas a la *GUI (Graphic User Interface)* del programa y a la interpretación de las solicitudes y la presentación de la información para los usuarios. Esta capa apunta a “**ValesEstructura**”, “**ValesLogica**” y “**ValesInterfaces**”. Esta capa posee dos clases: **Form1** y **VaVista**. Sobre la primera se construye la GUI. En la segunda se realizan dos cosas, la primera es agregar una instancia de **VaEstructura** y otra de **VaLogica** para tener una representación completa del vale que consta de una estructura y un comportamiento. La segunda es contener el conocimiento para recibir los datos de la GUI y colocarlos en el estado de **VaEstructura**, así como retornar los valores obtenidos cuando se realizan consultas.

Veamos la implementación de la clase **VaVista**.

```
Public Class VaVista
    Implements ValesInterfaces.Iabmc
    Sub New()
        VvaleEstructura = New ValesEstructura.VaEstructura
        VvaleLogica = New ValesLogica.VaLogica
    End Sub
    Private VvaleEstructura As ValesEstructura.VaEstructura
    Public ReadOnly Property ValeEstructura() As ValesEstructura.VaEstructura
        Get
            Return VvaleEstructura
        End Get
    End Property
    Private VvaleLogica As ValesLogica.VaLogica
    Public ReadOnly Property ValeLogica() As ValesLogica.VaLogica
        Get
            Return VvaleLogica
        End Get
    End Property
    Public Sub Alta(ByRef pObjeto As Object) Implements ValesInterfaces.Iabmc.Alta
        Try
            Me.LimpiaEstadoValeEstructura()
            Me.ValeEstructura.Numero = pObjeto(0)
            Me.ValeEstructura.Importe = pObjeto(1)
            Me.ValeEstructura.Concepto = pObjeto(2)
            Me.ValeEstructura.Fecha = pObjeto(3)
            Me.ValeEstructura.Legajo = pObjeto(4)
            Me.ValeLogica.Alta(Me.ValeEstructura)
        Catch ex As Exception
        End Try
    End Sub

    Public Sub Baja(ByRef pObjeto As Object) Implements ValesInterfaces.Iabmc.Baja
        Try
            Me.LimpiaEstadoValeEstructura()
            Me.ValeEstructura.Numero = pObjeto(0)
            Me.ValeLogica.Baja(Me.ValeEstructura)
        Catch ex As Exception
        End Try
    End Sub

    Public Sub Consulta(ByRef pObjeto As Object) Implements _
                        ValesInterfaces.Iabmc.Consulta
        Try
            Me.LimpiaEstadoValeEstructura()
            Me.ValeEstructura.Numero = pObjeto(0)
            Me.ValeLogica.Consulta(Me.ValeEstructura)
            pObjeto(1) = Me.ValeEstructura.Importe.ToString
            pObjeto(2) = Me.ValeEstructura.Concepto.ToString
            pObjeto(3) = Me.ValeEstructura.Fecha.ToShortDateString
            pObjeto(4) = Me.ValeEstructura.Legajo.ToString
            pObjeto(5) = Me.ValeEstructura.Hash.ToString
        End Try
    End Sub
```

```

        Catch ex As Exception

    End Try

End Sub

Public Function ConsultaIncremental(ByRef pObjeto As Object) As _
                                    System.Collections.Generic.List(Of Object) Implements _
                                    ValesInterfaces.Iabmc.ConsultaIncremental
    Try
        Me.ValeEstructura.Concepto = pObjeto(0)
        Return Me.ValeLogica.ConsultaIncremental(Me.ValeEstructura)
    Catch ex As Exception

    End Try
End Function

Public Function ConsultaRango(ByRef pObjeto1 As Object, ByRef pObjeto2 As Object) _
                            As System.Collections.Generic.List(Of Object) Implements _
                            ValesInterfaces.Iabmc.ConsultaRango
    Try
        Dim Vhasta As New ValesEstructura.VaEstructura
        Vhasta.Numero = pObjeto2(0)
        Me.LimpiaEstadoValeEstructura()
        Me.ValeEstructura.Numero = Convert.ToDecimal(pObjeto1(0))
        Return Me.ValeLogica.ConsultaRango(ValeEstructura, Vhasta)
    Catch ex As Exception

    End Try
End Function

Public Sub Modificacion(ByRef pObjeto As Object) Implements _
                        ValesInterfaces.Iabmc.Modificacion
    Try
        Me.LimpiaEstadoValeEstructura()
        Me.ValeEstructura.Numero = pObjeto(0)
        Me.ValeEstructura.Importe = pObjeto(1)
        Me.ValeEstructura.Concepto = pObjeto(2)
        Me.ValeEstructura.Fecha = pObjeto(3)
        Me.ValeEstructura.Legajo = pObjeto(4)
        Me.ValeLogica.Modificacion(Me.ValeEstructura)
    Catch ex As Exception

    End Try
End Sub

Private Sub LimpiaEstadoValeEstructura()
    Me.ValeEstructura.Numero = 0
    Me.ValeEstructura.Importe = 0
    Me.ValeEstructura.Concepto = ""
    Me.ValeEstructura.Fecha = Nothing
    Me.ValeEstructura.Legajo = 0
    Me.ValeEstructura.Hash = ""
End Sub
End Class

```

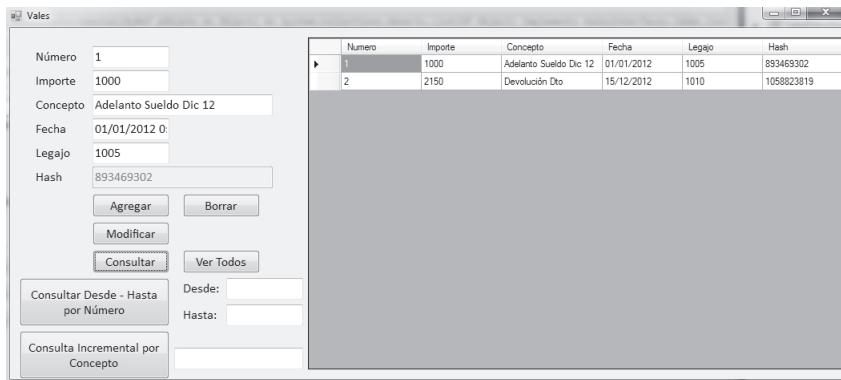


Figura 6.6.

Como se observa, la agregación de una estructura y un comportamiento se realiza a partir de la implementación de dos propiedades en la clase, denominadas `VaEstructura` y `VaLogica`. Para finalizar con este ejemplo, analizaremos la clase `Form1` y el formulario Windows asociado a ella, que oficia como GUI.

La imagen de la GUI es la que se ve en la figura 6.6.

El código asociado como parte de la `Partial Class Form1` que corresponde al diseño de esta interfaz es:

```

<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
Partial Class Form1
    Inherits System.Windows.Forms.Form

    'Form overrides dispose to clean up the component list.
    <System.Diagnostics.DebuggerNonUserCode()> _
    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        Try
            If disposing AndAlso components IsNot Nothing Then
                components.Dispose()
            End If
        Finally
            MyBase.Dispose(disposing)
        End Try
    End Sub

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.IContainer

    'NOTE: The following procedure is required by the Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()

```

```

Me.TextBox1 = New System.Windows.Forms.TextBox()
Me.TextBox2 = New System.Windows.Forms.TextBox()
Me.TextBox3 = New System.Windows.Forms.TextBox()
Me.TextBox4 = New System.Windows.Forms.TextBox()
Me.TextBox5 = New System.Windows.Forms.TextBox()
Me.TextBox6 = New System.Windows.Forms.TextBox()
Me.Label1 = New System.Windows.Forms.Label()
Me.Label2 = New System.Windows.Forms.Label()
Me.Label3 = New System.Windows.Forms.Label()
Me.Label4 = New System.Windows.Forms.Label()
Me.Label5 = New System.Windows.Forms.Label()
Me.Label6 = New System.Windows.Forms.Label()
Me.DataGridView1 = New System.Windows.Forms.DataGridView()
Me.Button1 = New System.Windows.Forms.Button()
Me.Button2 = New System.Windows.Forms.Button()
Me.Button3 = New System.Windows.Forms.Button()
Me.Button4 = New System.Windows.Forms.Button()
Me.Button5 = New System.Windows.Forms.Button()
Me.TextBox7 = New System.Windows.Forms.TextBox()
Me.TextBox8 = New System.Windows.Forms.TextBox()
Me.Label7 = New System.Windows.Forms.Label()
Me.Label8 = New System.Windows.Forms.Label()
Me.Button6 = New System.Windows.Forms.Button()
Me.TextBox9 = New System.Windows.Forms.TextBox()
Me.Button7 = New System.Windows.Forms.Button()
 CType(Me.DataGridView1, System.ComponentModel.ISupportInitialize).BeginInit()
Me.SuspendLayout()
‘
‘TextBox1
‘
Me.TextBox1.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.TextBox1.Location = New System.Drawing.Point(103, 23)
Me.TextBox1.Name = "TextBox1"
Me.TextBox1.Size = New System.Drawing.Size(96, 27)
Me.TextBox1.TabIndex = 0
‘
‘TextBox2
‘
Me.TextBox2.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.TextBox2.Location = New System.Drawing.Point(103, 53)
Me.TextBox2.Name = "TextBox2"
Me.TextBox2.Size = New System.Drawing.Size(96, 27)
Me.TextBox2.TabIndex = 1
‘
‘TextBox3
‘
Me.TextBox3.Font = New System.Drawing.Font("Calibri", 12.0!, _

```

```
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
 CType(0, Byte))
Me.TextBox3.Location = New System.Drawing.Point(103, 83)
Me.TextBox3.Name = "TextBox3"
Me.TextBox3.Size = New System.Drawing.Size(225, 27)
Me.TextBox3.TabIndex = 2
'
'TextBox4
'
Me.TextBox4.Font = New System.Drawing.Font("Calibri", 12.0!, _
 System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
 CType(0, Byte))
Me.TextBox4.Location = New System.Drawing.Point(103, 113)
Me.TextBox4.Name = "TextBox4"
Me.TextBox4.Size = New System.Drawing.Size(96, 27)
Me.TextBox4.TabIndex = 3
'
'TextBox5
'
Me.TextBox5.Font = New System.Drawing.Font("Calibri", 12.0!, _
 System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
 CType(0, Byte))
Me.TextBox5.Location = New System.Drawing.Point(103, 143)
Me.TextBox5.Name = "TextBox5"
Me.TextBox5.Size = New System.Drawing.Size(96, 27)
Me.TextBox5.TabIndex = 4
'
'TextBox6
'
Me.TextBox6.Enabled = False
Me.TextBox6.Font = New System.Drawing.Font("Calibri", 12.0!, _
 System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
 CType(0, Byte))
Me.TextBox6.Location = New System.Drawing.Point(103, 173)
Me.TextBox6.Name = "TextBox6"
Me.TextBox6.Size = New System.Drawing.Size(225, 27)
Me.TextBox6.TabIndex = 5
'
'Label1
'
Me.Label1.AutoSize = True
Me.Label1.Font = New System.Drawing.Font("Calibri", 12.0!, _
 System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
 CType(0, Byte))
Me.Label1.Location = New System.Drawing.Point(28, 26)
Me.Label1.Name = "Label1"
Me.Label1.Size = New System.Drawing.Size(60, 19)
Me.Label1.TabIndex = 6
Me.Label1.Text = "Número"
'
'Label2
```

```

'
Me.Label2.AutoSize = True
Me.Label2.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Label2.Location = New System.Drawing.Point(28, 56)
Me.Label2.Name = "Label2"
Me.Label2.Size = New System.Drawing.Size(59, 19)
Me.Label2.TabIndex = 7
Me.Label2.Text = "Importe"
'
'Label3
'
Me.Label3.AutoSize = True
Me.Label3.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Label3.Location = New System.Drawing.Point(28, 86)
Me.Label3.Name = "Label3"
Me.Label3.Size = New System.Drawing.Size(70, 19)
Me.Label3.TabIndex = 8
Me.Label3.Text = "Concepto"
'
'Label4
'
Me.Label4.AutoSize = True
Me.Label4.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Label4.Location = New System.Drawing.Point(28, 116)
Me.Label4.Name = "Label4"
Me.Label4.Size = New System.Drawing.Size(47, 19)
Me.Label4.TabIndex = 9
Me.Label4.Text = "Fecha"
'
'Label5
'
Me.Label5.AutoSize = True
Me.Label5.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Label5.Location = New System.Drawing.Point(28, 146)
Me.Label5.Name = "Label5"
Me.Label5.Size = New System.Drawing.Size(52, 19)
Me.Label5.TabIndex = 10
Me.Label5.Text = "Legajo"
'
'Label6
'
Me.Label6.AutoSize = True
Me.Label6.Font = New System.Drawing.Font("Calibri", 12.0!, _

```

```
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
 CType(0, Byte))
Me.Label6.Location = New System.Drawing.Point(28, 176)
Me.Label6.Name = "Label6"
Me.Label6.Size = New System.Drawing.Size(42, 19)
Me.Label6.TabIndex = 11
Me.Label6.Text = "Hash"
'
'DataGridView1
'

Me.DataGridView1.AllowUserToAddRows = False
Me.DataGridView1.AllowUserToDeleteRows = False
Me.DataGridView1.ColumnHeadersHeightSizeMode = _
    System.Windows.Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize
Me.DataGridView1.Location = New System.Drawing.Point(372, 12)
Me.DataGridView1.Name = "DataGridView1"
Me.DataGridView1.ReadOnly = True
Me.DataGridView1.Size = New System.Drawing.Size(715, 414)
Me.DataGridView1.TabIndex = 12
'

'Button1
'

Me.Button1.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Button1.Location = New System.Drawing.Point(103, 207)
Me.Button1.Name = "Button1"
Me.Button1.Size = New System.Drawing.Size(96, 29)
Me.Button1.TabIndex = 13
Me.Button1.Text = "Aregar"
Me.Button1.UseVisualStyleBackColor = True
'

'Button2
'

Me.Button2.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Button2.Location = New System.Drawing.Point(216, 206)
Me.Button2.Name = "Button2"
Me.Button2.Size = New System.Drawing.Size(96, 29)
Me.Button2.TabIndex = 14
Me.Button2.Text = "Borrar"
Me.Button2.UseVisualStyleBackColor = True
'

'Button3
'

Me.Button3.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Button3.Location = New System.Drawing.Point(103, 242)
Me.Button3.Name = "Button3"
```

```

Me.Button3.Size = New System.Drawing.Size(96, 29)
Me.Button3.TabIndex = 15
Me.Button3.Text = "Modificar"
Me.Button3.UseVisualStyleBackColor = True
'
'Button4
'
Me.Button4.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Button4.Location = New System.Drawing.Point(103, 277)
Me.Button4.Name = "Button4"
Me.Button4.Size = New System.Drawing.Size(96, 29)
Me.Button4.TabIndex = 16
Me.Button4.Text = "Consultar"
Me.Button4.UseVisualStyleBackColor = True
'
'Button5
'
Me.Button5.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Button5.Location = New System.Drawing.Point(12, 312)
Me.Button5.Name = "Button5"
Me.Button5.Size = New System.Drawing.Size(187, 60)
Me.Button5.TabIndex = 17
Me.Button5.Text = "Consultar Desde - Hasta por Número"
Me.Button5.UseVisualStyleBackColor = True
'
'TextBox7
'
Me.TextBox7.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.TextBox7.Location = New System.Drawing.Point(270, 312)
Me.TextBox7.Name = "TextBox7"
Me.TextBox7.Size = New System.Drawing.Size(96, 27)
Me.TextBox7.TabIndex = 18
'
'TextBox8
'
Me.TextBox8.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.TextBox8.Location = New System.Drawing.Point(270, 345)
Me.TextBox8.Name = "TextBox8"
Me.TextBox8.Size = New System.Drawing.Size(96, 27)
Me.TextBox8.TabIndex = 19
'
'Label7
'

```

```
Me.Label7.AutoSize = True
Me.Label7.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Label7.Location = New System.Drawing.Point(212, 315)
Me.Label7.Name = "Label7"
Me.Label7.Size = New System.Drawing.Size(54, 19)
Me.Label7.TabIndex = 20
Me.Label7.Text = "Desde:"
'

'Label8
'

Me.Label8.AutoSize = True
Me.Label8.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Label8.Location = New System.Drawing.Point(212, 348)
Me.Label8.Name = "Label8"
Me.Label8.Size = New System.Drawing.Size(51, 19)
Me.Label8.TabIndex = 21
Me.Label8.Text = "Hasta:"
'

'Button6
'

Me.Button6.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Button6.Location = New System.Drawing.Point(12, 378)
Me.Button6.Name = "Button6"
Me.Button6.Size = New System.Drawing.Size(187, 60)
Me.Button6.TabIndex = 22
Me.Button6.Text = "Consulta Incremental por Concepto"
Me.Button6.UseVisualStyleBackColor = True
'

'TextBox9
'

Me.TextBox9.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.TextBox9.Location = New System.Drawing.Point(205, 399)
Me.TextBox9.Name = "TextBox9"
Me.TextBox9.Size = New System.Drawing.Size(161, 27)
Me.TextBox9.TabIndex = 23
'

'Button7
'

Me.Button7.Font = New System.Drawing.Font("Calibri", 12.0!, _
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
    CType(0, Byte))
Me.Button7.Location = New System.Drawing.Point(216, 277)
Me.Button7.Name = "Button7"
```

```

Me.Button7.Size = New System.Drawing.Size(96, 29)
Me.Button7.TabIndex = 24
Me.Button7.Text = "Ver Todos"
Me.Button7.UseVisualStyleBackColor = True
'
'Form1
'
Me.AutoScaleDimensions = New System.Drawing.SizeF(6.0!, 13.0!)
Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font
Me.ClientSize = New System.Drawing.Size(1093, 438)
Me.Controls.Add(Me.Button7)
Me.Controls.Add(Me.TextBox9)
Me.Controls.Add(Me.Button6)
Me.Controls.Add(Me.Label18)
Me.Controls.Add(Me.Label17)
Me.Controls.Add(Me.TextBox8)
Me.Controls.Add(Me.TextBox7)
Me.Controls.Add(Me.Button5)
Me.Controls.Add(Me.Button4)
Me.Controls.Add(Me.Button3)
Me.Controls.Add(Me.Button2)
Me.Controls.Add(Me.Button1)
Me.Controls.Add(Me.DataGridView1)
Me.Controls.Add(Me.Label6)
Me.Controls.Add(Me.Label5)
Me.Controls.Add(Me.Label4)
Me.Controls.Add(Me.Label3)
Me.Controls.Add(Me.Label2)
Me.Controls.Add(Me.Label1)
Me.Controls.Add(Me.TextBox6)
Me.Controls.Add(Me.TextBox5)
Me.Controls.Add(Me.TextBox4)
Me.Controls.Add(Me.TextBox3)
Me.Controls.Add(Me.TextBox2)
Me.Controls.Add(Me.TextBox1)
Me.Name = "Form1"
Me.Text = "Vales"
 CType(Me.DataGridView1, System.ComponentModel.ISupportInitialize).EndInit()
Me.ResumeLayout(False)
Me.PerformLayout()

End Sub
Friend WithEvents TextBox1 As System.Windows.Forms.TextBox
Friend WithEvents TextBox2 As System.Windows.Forms.TextBox
Friend WithEvents TextBox3 As System.Windows.Forms.TextBox
Friend WithEvents TextBox4 As System.Windows.Forms.TextBox
Friend WithEvents TextBox5 As System.Windows.Forms.TextBox
Friend WithEvents TextBox6 As System.Windows.Forms.TextBox
Friend WithEvents Label1 As System.Windows.Forms.Label
Friend WithEvents Label2 As System.Windows.Forms.Label
Friend WithEvents Label3 As System.Windows.Forms.Label
Friend WithEvents Label4 As System.Windows.Forms.Label

```

```

Friend WithEvents Label5 As System.Windows.Forms.Label
Friend WithEvents Label6 As System.Windows.Forms.Label
Friend WithEvents DataGridView1 As System.Windows.Forms.DataGridView
Friend WithEvents Button1 As System.Windows.Forms.Button
Friend WithEvents Button2 As System.Windows.Forms.Button
Friend WithEvents Button3 As System.Windows.Forms.Button
Friend WithEvents Button4 As System.Windows.Forms.Button
Friend WithEvents Button5 As System.Windows.Forms.Button
Friend WithEvents TextBox7 As System.Windows.Forms.TextBox
Friend WithEvents TextBox8 As System.Windows.Forms.TextBox
Friend WithEvents Label7 As System.Windows.Forms.Label
Friend WithEvents Label8 As System.Windows.Forms.Label
Friend WithEvents Button6 As System.Windows.Forms.Button
Friend WithEvents TextBox9 As System.Windows.Forms.TextBox
Friend WithEvents Button7 As System.Windows.Forms.Button

End Class

```

La implementación de la clase Form1 es:

```

Public Class Form1

    Dim Vvale As New VaVista
    Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) Handles _
        Button1.Click
        Vvale.Alta({Convert.ToDecimal(TextBox1.Text), _
            Convert.ToDecimal(TextBox2.Text), _
            TextBox3.Text, _
            Convert.ToDateTime(TextBox4.Text), _
            Convert.ToDecimal(TextBox5.Text)})
        Me.MuestraValesTodos()
        Me.LimpiaCajaTexto()
    End Sub

    Private Sub Form1_Load(sender As System.Object, e As System.EventArgs) Handles _
        MyBase.Load
        Me.Button7_Click(Nothing, Nothing)
    End Sub
    Private Sub LimpiaCajaTexto()
        For Each Vctrl As Control In Me.Controls
            If TypeOf Vctrl Is TextBox Then Vctrl.Text = ""
        Next
    End Sub
    Private Sub MuestraValesTodos()
        Vvale.ValeEstructura.Concepto = ""
        Me.DataGridView1.DataSource = _
            Vvale.ValeLogica.ConsultaIncremental(Vvale.ValeEstructura)
    End Sub

    Private Sub Button2_Click(sender As System.Object, e As System.EventArgs) Handles _

```

```

        Button2.Click
    Vvale.Baja({Convert.ToDecimal(TextBox1.Text)})
    Me.MuestraValesTodos()
    Me.LimpiaCajaTexto()
End Sub

Private Sub Button3_Click(sender As System.Object, e As System.EventArgs) Handles _
    Button3.Click
    Vvale.Modificacion({Convert.ToDecimal(TextBox1.Text)}, _
        Convert.ToDecimal(TextBox2.Text), _
        TextBox3.Text, _
        Convert.ToDateTime(TextBox4.Text), _
        Convert.ToDecimal(TextBox5.Text)})
    Me.MuestraValesTodos()
    Me.LimpiaCajaTexto()
End Sub

Private Sub Button4_Click(sender As System.Object, e As System.EventArgs) Handles _
    Button4.Click
    Dim Varr() = {Convert.ToDecimal(TextBox1.Text), New Object, New Object, _
        New Object, New Object, New Object}
    Vvale.Consulta(Varr)
    Me.TextBox2.Text = Varr(1)
    Me.TextBox3.Text = Varr(2)
    Me.TextBox4.Text = Varr(3)
    Me.TextBox5.Text = Varr(4)
    Me.TextBox6.Text = Varr(5)
    Me.MuestraValesTodos()
End Sub

Private Sub Button5_Click(sender As System.Object, e As System.EventArgs) Handles _
    Button5.Click
    Dim VvaleHasta As New ValesEstructura.VaEstructura
    Me.DataGridView1.DataSource = _
        Vvale.ConsultaRango({Convert.ToDecimal(TextBox7.Text)}, _
        {Convert.ToDecimal(TextBox8.Text)})
    Me.LimpiaCajaTexto()
End Sub

Private Sub Button7_Click(sender As System.Object, e As System.EventArgs) Handles _
    Button7.Click
    Try
        Me.LimpiaCajaTexto()
        Me.MuestraValesTodos()
    Catch ex As Exception
    End Try
End Sub

Private Sub Button6_Click(sender As System.Object, e As System.EventArgs) Handles _
    Button6.Click
    Me.DataGridView1.DataSource = Vvale.ConsultaIncremental({TextBox9.Text})
End Sub

```

```
Private Sub TextBox9_TextChanged(sender As System.Object, e As System.EventArgs) _
    Handles TextBox9.TextChanged
    Me.DataGridView1.DataSource = Vvale.ConsultaIncremental({TextBox9.Text})
End Sub
End Class
```

Con lo visto hasta aquí, el lector posee una noción introductoria sobre arquitectura de software y debería haber modificado su visión sobre cómo encarar el diseño de un sistema.

También podría seguir especializando esta arquitectura de manera de obtener más capas especializadas y así lograr mayor independencia entre ellas.

Como ejemplo, podemos mencionar dos aspectos para que el lector que esté interesado pueda implementarlos.

Si pensamos en la capa “**ValesDatos**”, se puede observar en ella que hay clases que están muy relacionadas con la base de datos seleccionada y otras que no. Dentro del primer grupo podemos enumerar a las clases que utilizan los servicios de las clases `SqlConnection`, `SqlCommand` y `SqlDataAdapter`. En el segundo grupo están las que utilizan objetos no relacionados con una tecnología específica de base de datos y ofician como ORM. Si separáramos estas clases en dos capas, lograríamos mayor flexibilidad e independencia si en algún momento tenemos que cambiar la base de datos. Con estos ajustes realizados, solo deberíamos adaptar la capa que posee las clases que se relacionan con la tecnología de base de datos.

Otro aspecto a considerar sería la capa “**ValesVista**”. En ella tenemos clases que constituyen la GUI y otras que interpretan las acciones solicitadas y predisponen cómo se mostrarán los datos que aparecerán en la GUI. Si las separamos en capas distintas, podríamos pensar en utilizar distintas GUI para un mismo set de datos, por ejemplo, un formulario Windows en algunos casos y en otros un formulario web.

Finalmente, alentamos al lector a que acceda a otras arquitecturas como MVC (Modelo, Vista, Controlador), SOA (Arquitectura Orientada a Servicios), ESB (Bus de Servicios de Empresa) e inclusive a otros modelos arquitectónicos, ya que le ayudarán a ampliar su visión sobre cómo diseñar sistemas.

Conclusiones

Los conceptos abordados en la segunda parte del libro ejemplifican algunas de las cosas que se pueden realizar y cómo hacerlas en la práctica. Si bien se ha revisado un puñado de conceptos, estos no agotan todas las posibilidades de la orientación a objetos. Esperamos que el texto le haya servido al lector como una introducción y punto de partida para llevar adelante la programación orientada con una visión arquitectónica de los sistemas y así poder profundizar los conceptos y adaptarlos a sus necesidades.

Glosario

Por Grady Booch

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, utilizado 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 solo 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 solo por las subclases, implantación y amigos de la clase en cuestión; los elementos privados son accesibles solo por la implantación y amigos de la clase que contiene el elemento; los elementos de implantación son accesibles solo 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 dise-

ño 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 reconoci-

miento 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 la 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 Algún 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 solo 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) solo 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 , *variable de instancia*, *objeto miembro* y *ranura* (slot) son intercambiables.

objeto pasivo Un objeto que no contiene su propio hilo de control.

objeto secuencial Objeto pasivo cuya semántica se garantiza solo 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 *mensaje*, *método* y *operación* 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 de la interfaz de una clase, objeto o módulo; lo que se declara como privado (*private*) 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 de la 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 en 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 de la 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 , , y *ranura* son intercambiables.

responsabilidad Algún 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 solo 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 (solo conlleva un hilo de control), síncrona (cita entre dos procesos), de abandono inmediato (un proceso puede citarse con otro solo si el segundo proceso ya está esperando), de tiempo limitado (un proceso puede citarse con otro, pero esperará por el segundo solo 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 solo 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 *campo*, *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 solo donde sus ámbitos se solapan. El control de exportación puede restringir además el acceso a las abstracciones visibles.

Vocabulario técnico bilingüe¹

Por Grady Booch

Término original en inglés	Término usado en el libro	Vocablos alternativos de uso común
INF	IFN	—
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

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.*).

Término original en inglés	Término usado en el libro	Vocablos alternativos de uso común
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	—
client/supplier	cliente/proveedor	cliente/servidor
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	parallelismo
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

Término original en inglés	Término usado en el libro	Vocablos alternativos de uso común
debugging	depuración	puesta a punto
deep copy	copia profunda	copia honda
delegation	delegación	—
deque	cola doble	<i>deque</i>
dereferencing	derreferenciar	desreferenciar, indireccionar
decomposable	descomponible	desdoblable
destructor	destructor	—
dispatch	seleccionar	distribuir, lanzar
domain analysis	análisis de dominio	—
drive	unidad	—
dynamic binding	ligadura dinámica	ligadura 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
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 gráfica de usuario, IGU	GUI
hard drive	disco duro	—
header file	archivo de cabecera	archivo de encabezamiento
heap	montículo, <i>heap</i>	montón, cúmulo

Término original en inglés	Término usado en el libro	Vocablos alternativos de uso común
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	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	—
client	cliente	—
link	enlace	—
linkages	enlaces	enlazados
lookup	búsqueda	consulta
macro	macro	—
maintenance	mantenimiento	—
many-to-many	muchas-a-muchas	—

Término original en inglés	Término usado en el libro	Vocablos alternativos de uso común
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
modificator	modificador	—
module	module	—
multiple inheritance	herencia múltiple	—
nesting	anidamiento	encajamiento
network	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	empaquetamiento	—
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	—

Término original en inglés	Término usado en el libro	Vocablos alternativos de uso común
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
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
shallow copy	copia superficial	—
scaling	escalado	escalamiento
scaling down	escalado descendente	—
scaling up	escalado ascendente	—
scenario	escenario	—
scenario planning	planificación del escenario	—
script	guión	escritura, manuscrito

Término original en inglés	Término usado en el libro	Vocablos alternativos de uso común
schedule	planificar	—
selector	selector	—
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	<i>stack</i>
state transition	diagrama de transición de estados	—
static binding	ligadura estática	enlace, ligado estático
store	almacenar	guardar
storyboarding	narración de sucesos	—
stream	flujo	corriente
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
synergistic	sinérgico	sinergístico
task	tarea	—
template	plantilla	patrón, modelo

Término original en inglés	Término usado en el libro	Vocables alternativos de uso común
thread	hilo	hebra
thread of control	hilo 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>

La orientación a objetos le ha aportado al mundo del diseño y desarrollo de software avances significativos que permitieron manejar la complejidad de manera controlable y efectiva.

Si bien nos resulta familiar escuchar hablar de OOP (programación orientada a objetos), preferimos referirnos en la bibliografía a OO (orientación a objetos) para reforzar la idea de que, cuando nos referimos a ella, queremos abarcar los conceptos involucrados en una forma metodológica y sistemática que nos lleva al análisis, diseño y desarrollo de software.

Este abordaje respeta un paradigma que incluye la concepción filosófica de cómo encarar el problema, priorizando este aspecto muy por encima de las herramientas con las que se implementa la solución obtenida.

Uno de los aspectos más relevantes aportados por la OO es la posibilidad de simular y modelizar problemas reales con gran facilidad. El problema en sí no es ni más sencillo ni más difícil: la diferencia radica en la forma de tratarlo que nos propone la OO.

Esperamos que el lector disfrute el fascinante mundo de la OO para el desarrollo de software tanto como lo hemos hecho los autores algún tiempo atrás. Creemos que este libro puede hacer más ameno lograr el objetivo.

