

UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

**XC. UN LENGUAJE ORIENTADO A COMPONENTES
PARA PROGRAMACIÓN DE SISTEMAS**

Jorge Mederos Martín

TESIS DOCTORAL

DIRECTOR: Prof. Dr. Fernando Pérez Costoya

2003

Tribunal nombrado por el Mgco. y Exmo. Sr. Rector de la Universidad Politécnica de Madrid.

PRESIDENTE D.

VOCAL D.

VOCAL D.

VOCAL D.

SECRETARIO D.

Realizado el acto de defensa y lectura de Tesis en Madrid el día de
del año 2003.

CALIFICACIÓN:

El Presidente

Los Vocales

El Secretario

*A María José
A mi familia*

Madrid, 13 de junio de 2003

Agradecimientos

Quisiera agradecer a *Fernando Pérez* el interés que desde siempre ha puesto en mí antes y después de finalizar mi licenciatura, así como su constante apoyo en este trabajo cuando todavía era tan solo un conjunto de ideas vagamente enlazadas entre sí. Quisiera agradecerle también las discusiones, comentarios, revisiones y sugerencias de mejora que han enriquecido el presente manuscrito.

En deuda estoy también con *Pedro de Miguel* y *Antonio Pérez* por sus comentarios y recomendaciones a lo largo de estos años, y que, en cierta ocasión hace ya mucho tiempo, me dieron el inestimable consejo de saber concentrar el esfuerzo en finalizar los objetivos que me hubiese impuesto.

Agradezco también el trato entrañable y amigable que siempre he recibido en el Departamento de Arquitectura y Tecnología de Sistemas Informáticos de la Facultad de Informática de la Universidad Politécnica de Madrid, desde los tiempos que era estudiante de primeros cursos.

Un recuerdo especial me quedará siempre con *José Oramas*, *Héctor Madrona*, *José Miguel Díez de la Lastra* y *Carlos Oramas* por los innumerables buenos consejos, impresiones y —a veces— acaloradas discusiones que hemos compartido y que, en el fondo, han sido imprescindibles para forjar las ideas fundamentales que se encuentran presentes en este trabajo.

Sin el apoyo de mis hermanos y mis padres, es difícil que hubiera continuado con esta labor académica.

Por último, agradezco muy especialmente a *María José Lloris* por ser tan comprensiva conmigo y haberme apoyado tanto durante el tiempo que he dedicado a este trabajo.

Resumen

El presente trabajo estudia las limitaciones en las técnicas actuales de subdivisión de sistemas informáticos y de su ensamblado. Este análisis pone en relieve deficiencias conocidas pero a menudo aceptadas en las técnicas usadas en la subdivisión del diseño y en el ensamblado de las aplicaciones finales. Carencias que aumentan la complejidad percibida de los programas construidos y limitan las posibilidades de creación de componentes. Lenguajes actuales de programación de sistemas disponibles como C, C++, Modula-3 u Objective-C no soportan directamente modelos de componentes, modelos que ofrecen únicamente una abstracción parcialmente opaca a los lenguajes de programación con los que se manipulan. Esta característica es una fuente importante de la complejidad percibida de los modelos de componentes. Los lenguajes de programación, en cambio, son capaces de dar una abstracción más consistente para la creación de componentes, ya que pueden esconder mejor los aspectos de implementación: Comprobación de tipos estricta, separación completa entre interfaz de componentes y su implementación, envío de mensajes eficiente, organización modular o una sintaxis cómoda son elementos importantes a tener en cuenta. La implementación eficiente de componentes demanda lenguajes con características de programación de sistemas y de bajo nivel. A su vez, la programación de sistemas, cada vez más compleja y flexible, se puede beneficiar notablemente del uso de componentes para mantener las barreras de abstracción apropiadas y simplificar su diseño y construcción.

Con el fin de obtener componentes con las propiedades adecuadas automáticamente, se describen y justifican mejoras en los procesos de desarrollo de programas que ayuden a acercarnos a ese objetivo. Son guías de diseño para lenguajes de programación donde se pone mayor énfasis en las propiedades de los productos resultantes —programas con componentes— que en el proceso que los genera —el lenguaje—. Este trabajo describe la definición e implementación de un prototipo de un nuevo lenguaje de programación, llamado XC, usado como ejemplo de implementación de las guías de diseño estudiadas. Se trata de un lenguaje orientado a la programación con componentes que soporta también la programación orientada a objetos y la programación de sistemas. Esta pensado para el diseño y desarrollo de sistemas informáticos donde se comparte una tensión inherente entre funcionalidad, flexibilidad, tamaño, eficiencia y programación portable de bajo nivel.

El nuevo lenguaje, que pertenece a la familia de lenguajes de C, ofrece las abstracciones necesarias dentro del lenguaje para facilitar la creación de componentes y la programación de sistemas. Se trata de un lenguaje modular que demarca una separación estricta entre interfaz e implementación. Posee herencia múltiple de interfaces y herencia simple de implementación, así como un novedoso modelo de objetos interno modular basado en prototipos, pensado para la construcción de componentes. El modelo es extensible preservando la modularidad. Es posible realizar extensiones a objetos prototipo existentes mediante categorías, y extensiones ortogonales que soportan técnicas de metaprogramación con más sencillez. El lenguaje posee comprobación estricta de tipos, tipos covariantes y tipos genéricos, así como una sintaxis expresiva. También incluye facilidades de optimización de tipos básicos, funciones en línea, funciones y objetos nativos, código seguro y no seguro, y compilación final a C.

Abstract

The present document studies limitations in the current techniques of system decomposition and assembly. This analysis uncovers several known but accepted deficiencies in application design partition and assembly that increase the perceived complexity of programs and reduce the likelihood of creating components. Several known system programming languages, like C, C++, Modula-3 or Objective-C do not have direct support for component models. Those models just offer a partial opaque abstraction in the languages being used to work with. This property is an important source of perceived complexity of component models. On the other hand, programming languages are able to offer a more consistent abstraction for components handling, as they can do a better job hiding implementation details. Aspects to take into account are: strong type checking, clear separation between component interface and implementation, efficient message sending, modular organization or flexible syntax. An efficient implementation of components requires languages with system-level programming facilities. The increased complexity of system-level programming may also obtain a fair amount of benefits with the use of components, to keep up with adequate abstraction barriers and simplified design and construction.

In order to automatically obtain components with the appropriate properties, improvements in program development processes are discussed to help us to carry them out. They are design guides for programming languages where the focus is on the products —programs with components— rather than on the process used to create them —the language—. This work describes the definition and implementation of a new programming language prototype, XC, used as an implementation example of the previous guides. It is a component-oriented programming language with facilities for object-oriented programming and system-level programming. Designed for computer systems where there is an inherent tension among functionality, flexibility, size, efficiency and portable low-level programming.

The new language is a member of the C family of languages. It provides the necessary abstractions to assist in the creation of components and system-level programming. It is a modular language with a clear separation between interfaces and implementations. It has multiple interface inheritance, single implementation inheritance, and a novel modular object model based on prototypes, designed for components construction. The model preserves modularity during extensions. Feasible extensions can be applied to existing prototypes using categories, and orthogonal extensions allow simpler meta-programming techniques. The language has strong type checking, covariant types, generic types and an expressive syntax. It also includes features like primitive type optimizations, in-line functions, native functions and objects, safe and unsafe code, and C target compilation.

Índice

1	Introducción	1
1.1	Motivación	3
1.2	Objetivos	7
1.3	Organización del documento	9
2	Componentes y técnicas de ensamblado	11
2.1	Introducción	11
2.2	Definiciones preliminares.....	11
2.3	Un sistema informático típico	14
2.4	Tendencias en sistemas de ensamblaje.....	17
2.4.1	Ingeniería de dominios	19
2.4.2	Familias de sistemas y líneas de productos	20
2.5	Principales modelos de componentes actuales	20
2.5.1	COM, DCOM y COM+.....	21
2.5.2	JavaBeans y EJB	23
2.5.3	CORBA y CCM	25
2.5.4	Servicios <i>web</i>	28
2.6	Procesos y productos	28
2.7	Interdependencia	30
2.7.1	Dependencias e interdependencia	30
2.7.2	Análisis de aserciones locales.....	40
2.8	Recapitulación.....	49
3	XC. Una propuesta	51
3.1	Introducción	51
3.2	Conceptos básicos	51
3.3	Premisas de diseño de XC.....	53
3.4	El modelo de componentes de XC.....	55
3.4.1	Implementación del modelo.....	57
3.4.2	Módulos.....	58
3.4.3	El modelo de objetos subyacente.....	58
3.4.4	Envío de mensajes	61
3.4.5	Tipos de extensiones.....	63
3.5	Programación de sistemas en XC	67
3.6	Objetivos principales de XC.....	68
3.6.1	Consistencia	68
3.6.2	Robustez.....	72
3.6.3	Escalabilidad.....	73
3.6.4	Eficiencia.....	74
3.7	Esquema de la solución propuesta	74

3.8 Breve comparación con otros lenguajes	75
3.9 Recapitulación.....	79
4 El modelo de objetos	81
4.1 Introducción	81
4.2 De prototipos a clases	81
4.3 Herencia	82
4.3.1 Delegación y concatenación	84
4.3.2 Herencia de implementación y herencia de tipos o interfaces	87
4.3.3 Polimorfismo	88
4.3.4 Monotonía	89
4.3.5 Herencia sintáctica y semántica	89
4.3.6 Herencia estática y dinámica.....	90
4.3.7 Herencia simple y múltiple	90
4.3.8 Llamadas de retorno y herencia	97
4.3.9 El problema de la clase base frágil	101
4.3.10 Discusión.....	102
4.4 Mecanismos de envío de mensajes	108
4.4.1 La resolución de métodos.....	109
4.4.2 Tablas virtuales	110
4.4.3 Selectores	111
4.4.4 Otras técnicas de resolución de métodos.....	113
4.4.5 Discusión.....	117
4.5 Reflexión y arquitecturas meta-nivel.....	118
4.5.1 Implementaciones abiertas	119
4.5.2 Metaobjetos	121
4.5.3 Envío de mensajes a metaobjetos.....	121
4.5.4 Metaclases.....	122
4.5.5 Aspectos	124
4.5.6 Discusión	125
4.6 Recapitulación.....	127
5 La estructura sintáctica	129
5.1 Introducción	129
5.2 Organización general de programas	129
5.2.1 Introducción al modelo de objetos.....	131
5.3 Extensiones	135
5.3.1 Herencia múltiple de tipos	136
5.3.2 Herencia simple de implementación.....	136
5.3.3 Extensión a tipos y objetos existentes	136
5.3.4 Extensiones ortogonales	140
5.4 Protocolos, prototipos y sus variantes	144
5.5 Módulos	147
5.5.1 Módulos y clases	148
5.5.2 Otros lenguajes modulares.....	149
5.5.3 La estructura jerárquica	150
5.5.4 Estructura de un módulo	151
5.5.5 Importación de módulos	153
5.5.6 Las reglas de ámbito	155

5.5.7 Módulos como unidad de carga	156
5.5.8 Modularidad monótona	157
5.6 Conveniencias sintácticas	157
5.6.1 Expresiones y protocolos de operador	157
5.6.2 Expresiones estructuradas	158
5.6.3 Constantes: objetos constantes y protocolos constantes.....	160
5.6.4 El nexo de unión con el soporte en tiempo de ejecución.....	162
5.7 Recapitulación	163
6 El modelo de componentes	165
6.1 Introducción	165
6.2 Características del modelo de componentes	165
6.2.1 Reglas de composición.....	166
6.2.2 Interfaces o protocolos y contratos	167
6.2.3 Nombrado	168
6.2.4 Metadatos	168
6.2.5 Interoperabilidad.....	168
6.2.6 Empaquetado y despliegue	170
6.2.7 Configuración.....	170
6.2.8 Evolución	170
6.3 Implementación de componentes.....	171
6.3.1 Estructura de un componente	171
6.3.2 Control de calidad.....	173
6.4 Recapitulación	175
7 El sistema de tipos	177
7.1 Introducción.....	177
7.2 Variantes de sistemas de tipos.....	177
7.2.1 Aspectos básicos de un sistema de tipos	178
7.3 Tipos y subtipos	181
7.3.1 Polimorfismo y comprobación estricta de tipos	183
7.3.2 Separación entre tipos e implementación	185
7.3.3 Jerarquías inversas	186
7.3.4 Comprobación semántica de tipos.....	188
7.4 Tipos avanzados	190
7.4.1 El problema de los métodos binarios.....	190
7.4.2 Covarianza y contravarianza	191
7.4.3 El tipo covariante <i>Self</i>	194
7.4.4 Variables de instancia covariantes y ruptura de invariantes.....	195
7.4.5 Encaje o <i>matching</i>	197
7.4.6 Comprobación de tipos covariantes y tipos exactos	199
7.5 Tipos genéricos	202
7.5.1 El problema de las colecciones	203
7.5.2 Tipos polimórficos	205
7.5.3 Tipos paramétricos	206
7.5.4 Tipos Virtuales	207
7.5.5 Tipos homogéneos y heterogéneos	209
7.5.6 Sustitución de tipos implícita.....	210
7.5.7 Tipos genéricos en XC: Sustitución de tipos explícita	211

7.6 Tipos opacos y modularidad.....	216
7.7 Recapitulación	218
8 El soporte en tiempo de ejecución y el compilador	219
8.1 Introducción	219
8.2 El soporte en tiempo de ejecución	219
8.3 La estructura de los objetos.....	220
8.3.1 El mapa de los objetos.....	220
8.3.2 La semántica de clases	221
8.3.3 Extensiones modulares	222
8.3.4 El algoritmo de creación de objetos	224
8.3.5 El algoritmo de linearización de categorías	225
8.3.6 La estructura de la tabla de métodos	225
8.3.7 El algoritmo de construcción de la tabla de métodos.....	227
8.4 Implementación del envío de mensajes	228
8.4.1 Envío eficiente de mensajes	229
8.4.2 Registro de activación de métodos.....	233
8.4.3 La función mensajera.....	235
8.5 Inicialización	237
8.5.1 Registro de módulos	238
8.5.2 El objeto <i>null</i>	239
8.6 Diseño general del compilador	240
8.6.1 Análisis léxico e importación de módulos.....	240
8.6.2 Evaluación sintáctica y semántica de módulos	241
8.6.3 Generación de código para módulos	242
8.7 Optimización	244
8.7.1 Modificadores de optimización de objetos.....	245
8.7.2 Optimización de tipos básicos	247
8.7.3 Optimización del envío de mensajes	248
8.7.4 Funciones y objetos nativos	250
8.7.5 Código no seguro	253
8.8 Recapitulación	255
9 Conclusiones	257
9.1 Resultados obtenidos	257
9.2 Futuras líneas de investigación	259
A Gramática de XC	261
Símbolos terminales	261
Gramática	263
B Ejemplos de código	275
Declaración de diccionarios	275
Definición de diccionarios.....	276
C Aspectos de implementación	279
La función mensajera.....	279
D Bibliografía	281

Índice de Figuras

Figura 2-1 Diferentes tipos de componentes.....	13
Figura 2-2 Sistema informático de complejidad media.	14
Figura 2-3 Estructura de un servidor de aplicaciones actual.....	16
Figura 2-4 El largo camino hacia líneas de ensamblaje de productos estándar.	18
Figura 2-5 Llamada de un cliente a un componente COM.....	21
Figura 2-6 Comunicación entre componentes COM usando DCOM.	22
Figura 2-7 Posibles beans definidos por JavaBeans.....	24
Figura 2-8 Comunicación remota entre EJB y clientes.....	24
Figura 2-9 Comunicación entre objetos CORBA.	26
Figura 2-10 Comunicación remota entre servicios web y clientes.	28
Figura 2-11 Relación entre clientes y proveedores.....	40
Figura 2-12 Llamada a función y retorno de función.....	43
Figura 2-13 Asincronía en llamadas de retorno.	47
Figura 2-14 Temporalidad.	47
Figura 2-15 Información de estado intermedio del proveedor.	47
Figura 2-16 Llamadas de retorno encadenadas.	48
Figura 3-1 Implementación de componentes.	57
Figura 3-2 Estructura modular.....	58
Figura 3-3 Objetos en un modelo de objetos.....	59
Figura 3-4 El modelo de objetos subyacente.	60
Figura 3-5 Énfasis en programación orientada a objetos y en programación estructurada.....	61
Figura 3-6 Elementos de un envío de mensajes.	62
Figura 3-7 Envío de mensajes en objetos y componentes.....	63
Figura 3-8 Extensiones no planificadas: herencia y categorías.....	65
Figura 3-9 Extensiones ortogonales.....	66
Figura 3-10 Tensión entre número de abstracciones y complejidad del lenguaje.	71
Figura 3-11 Guía general de los próximos capítulos.....	75
Figura 4-1 Conceptos básicos tradicionales de herencia.....	83
Figura 4-2 Clases en SIMULA-67 y en Smalltalk-80 (simplificado).	84
Figura 4-3 Prototipos con delegación y concatenación.....	85
Figura 4-4 Herencia en Smalltalk-80 (simplificado).	86
Figura 4-5 Herencia simple y múltiple de interfaces.....	91
Figura 4-6 Herencia simple de implementación con concatenación y delegación.	92
Figura 4-7 Herencia múltiple de implementación por concatenación.....	94
Figura 4-8 Algoritmo de linearización de herencia.....	96

Figura 4-9 Mixins como una forma disciplinada de herencia múltiple.....	97
Figura 4-10 Llamadas de retorno realizadas desde los padres.....	98
Figura 4-11 Llamadas de retorno realizadas desde los hijos.....	99
Figura 4-12 Tablas de métodos.	109
Figura 4-13 Tablas de métodos para las técnicas de tabla virtual y selectores.	112
Figura 4-14 Programas y metaprogramas en lenguajes de programación.	118
Figura 4-15 Interpretación de la metaprogramación como programas cliente/servidor.	119
Figura 4-16 Reificación y reflexión en metaprogramación.	120
Figura 4-17 Metaclases y adjetivos.	123
Figura 4-18 Metaprogramación con metaclases frente a metaprogramación en XC.....	126
Figura 5-1 Llamadas a métodos anteriores y posteriores en extensiones ortogonales.....	142
Figura 5-2 Jerarquía básica de protocolos y prototipos de XC.....	146
Figura 7-1 Jerarquías inversas.	187
Figura 7-2 Grupos, Composición y Categorías.	189
Figura 8-1 Objetos y mapas de objetos.	221
Figura 8-2 El mapa de un objeto.....	224
Figura 8-3 Métodos base, anteriores y posteriores un objeto.	226
Figura 8-4 Tablas de métodos y cadenas de métodos asociadas.	227
Figura 8-5 Arrays dispersos.....	230
Figura 8-6 Caché de selectores monomórfica en el lugar de llamada.	232
Figura 8-7 Registros de activación.	234
Figura 8-8 Analizador léxico.....	241
Figura 8-9 Análisis sintáctico, semántico y generación de código.	242
Figura 8-10 Estructura de código generado para cada módulo.	243

Índice de Tablas

Tabla 3-1 <i>Características principales de XC junto a otros lenguajes populares.</i>	76
Tabla 4-1 <i>Delegación frente a concatenación.</i>	86
Tabla 4-2 <i>Opciones de implementación de herencia.</i>	87
Tabla 5-1 <i>Variedades de protocolos.</i>	145
Tabla 5-2 <i>Variedades de objetos prototipo.</i>	145
Tabla 5-3 <i>Características modulares de XC junto a otros lenguajes con módulos conocidos.</i>	149
Tabla 5-4 <i>Protocolos y mensajes asociados a operadores.</i>	158
Tabla 6-1 <i>Elementos básicos de un modelo de componentes.</i>	166
Tabla 6-2 <i>Palabras clave de soporte para llamadas entre procesos.</i>	169
Tabla 9-1 <i>Identificadores reservados de XC.</i>	261
Tabla 9-2 <i>Símbolos terminales generados por el analizador léxico.</i>	262

1

Introducción

La sociedad actual está inmersa en la llamada revolución de la información. Esta revolución está cambiando la forma en que obtenemos, procesamos y distribuimos información, incrementando la capacidad humana para manejarla hasta límites insospechados hace unas décadas. Está haciendo con nuestra habilidad de manipular información lo que anteriormente hizo la revolución industrial con nuestra capacidad de fabricación de artefactos.

La demanda de nuevos y más complejos sistemas para satisfacer nuestros deseos de gestión de información no cesa de crecer. Los campos de aplicación de esta tecnología se extienden a multitud de actividades de la sociedad de hoy: industria, comunicación, arte, enseñanza, ocio... La posibilidad de realizar estos procedimientos de forma más automática, precisa, rápida y barata es la esencia de la revolución de la información: extender la capacidad del ser humano para procesar información en su propio beneficio. Los *sistemas de información automatizados* o *sistemas informáticos* son los encargados de automatizar esos procedimientos. Poniendo una analogía, los sistemas informáticos tienden a representar en la revolución de la información lo que las fábricas representaron en la revolución industrial.

En el eje central de esta presión de progreso se encuentra nuestra capacidad para producir y mantener programas o *software* cada vez más complejo, las unidades básicas de los sistemas informáticos. Desafortunadamente, el diseño, construcción y mantenimiento de *software* continúa siendo una actividad humana esencialmente intelectual. Los métodos y lenguajes de programación facilitan el desarrollo de *software*, proporcionando conceptos más cercanos al ser humano que a la máquina, ayudando así en la descripción de programas. Pero la construcción de sistemas informáticos no es sencilla. La experiencia ha demostrado que la complejidad de grandes sistemas informáticos va más allá de las prácticas tecnológicas actuales y aumenta notablemente con el tamaño de los proyectos [Jones, 1995; pp. 86-87]. La falta de capacidad para producir programas en la cantidad demandada, de la complejidad adecuada, de la calidad esperada, y razonablemente mantenibles, es percibida desde hace ya muchos años como la “crisis del *software*” [Randell, 1979; p. 1].

Para lidiar con la complejidad de los sistemas informáticos se han seguido dos líneas principales: mejorar los métodos y herramientas usadas para construir programas, y dividir los sistemas informáticos principales en subsistemas más especializados, comunicados mediante protocolos estándar. El primer aspecto corresponde con el estudio y mejora de los *procesos* de construcción de *software*, mediante la recolección de guías y prácticas útiles, así como el diseño de nuevos lenguajes de programación. Las guías y prácticas ayudan a las organizaciones que se dedican al desarrollo de *software* a definir áreas de procesos cuya meta sea mejorar sus procesos de construcción de *software*. Estas áreas están enfocadas más en qué hacer que en cómo hacerlo, encargándose de conseguir procesos definidos, estandarizados y repetibles, sobre los que posteriormente puedan

aplicarse mejoras. El CMM (*Capability Maturity Model*) [Paulk, 1995] y el ISO 9001 [ISO, 1991] son las descripciones más aceptadas. Las mejoras de los lenguajes de programación, dotados con el tiempo de mayor estructura y abstracción, también han ayudado a construir programas más complejos, aunque no sin cierto coste. La paulatina comprensión de las propiedades de algunas de estas mejoras básicas, como la orientación a objetos, los sistemas de tipos o la modularidad, han dado lugar durante las últimas dos décadas a numerosos progresos en la teoría e implementación de lenguajes. Desgraciadamente, los lenguajes más populares de hoy no siempre obtienen beneficio de esos avances.

El segundo aspecto trata de definir mejor el *producto* resultante de los procesos anteriores: el *software*, para que se pueda combinar mejor, y facilite la construcción y el mantenimiento de sistemas más sofisticados. La división de sistemas informáticos en subsistemas ha fomentado con el tiempo la especialización y creación de productos complejos como sistemas operativos, redes de ordenadores, bases de datos, procesadores de texto, entornos gráficos, o sistemas de mensajería electrónica. No obstante, la división de estos grandes subsistemas en otros menores o *componentes* que puedan también combinarse ha encontrado grandes dificultades, incrementándose los costes, tiempos de desarrollo y produciéndose carencias en calidad. En este nivel se disipa el concepto de producto y se continúa usando técnicas más tradicionales de desarrollo. La mayor parte del esfuerzo en la construcción de sistemas informáticos se produce en este nivel, en donde se crea *software a medida*, específico para cada organización. La doctrina tradicional en ingeniería del *software* subraya este aspecto de la construcción de *software* único, junto con el papel predominante de los procesos sobre los productos resultantes:

“Directamente relacionado con el hecho de que la ingeniería del *software* es [una actividad] intensiva en diseño, se reconoce que el desarrollo de una aplicación informática particular está caracterizada principalmente por una actividad separada de diseño y construcción. Esto es debido a que existen menos aplicaciones informáticas estándar o productos que puedan ser producidos en masa, salvo unos pocos tipos de sistemas informáticos o utilidades generales.”

“Así, para el desarrollo de *software* intensivo en diseño, los únicos elementos que pueden posiblemente ser estandarizados y reutilizados significativamente son principalmente los procesos de ingeniería del *software*, no los productos finales, como ocurre en otras disciplinas de ingeniería dedicadas a la manufactura.” [Wang & King, 2000; p. 10].

La complejidad de las aplicaciones informáticas ha sido y continúa siendo tema de intenso debate. [Brooks, 1975; pp. 18-19] describe las dificultades percibidas en el desarrollo de *software* complejo. Observó que añadir más personas a un proyecto de construcción de *software* atrasado sólo servía para atrasarlo aún más. En un artículo posterior argumenta que tales dificultades son inherentes e inevitables. Debidas a la complejidad, consistencia, maleabilidad e intangibilidad del *software*, resultado de su *esencia*, no de un *accidente*, algo que se estuviese haciendo mal en su producción [Brooks, 1987; pp. 10-12]. Esos pensamientos están en la raíz del enfoque de la ingeniería del *software* en mejorar y estandarizar los procesos de construcción de aplicaciones informáticas únicas, y no en buscar y estandarizar productos. Es más, la actitud centrada en los procesos de construcción de aplicaciones particulares, debido a la complejidad inherente de los sistemas informáticos, se retroalimenta a sí misma. Contradice la posibilidad de encontrar fórmulas tecnológicas que permitan definir productos estándar de grano más fino —componentes— que puedan reducir a su vez la complejidad de la construcción de aplicaciones especializadas. Al menos desde 1968, esta idea se encuentra sobre la mesa. En esa época, Doug MacIlroy identificó el deseo de obtener componentes producidos en masa y catálogos de componentes clasificados por características [Randell, 1979; p. 8].

[Brooks, 1987; pp. 16-18] identifica los accidentes y la esencia del *software* para concentrar la atención en la búsqueda de soluciones a los elementos esenciales, más esquivos. En su análisis es destacable el énfasis en cómo un cambio de tendencia que facilite comprar *software* en vez de crearlo expresamente, unido al desarrollo de un mercado de masas para él, ayudaría a reducir su complejidad total [Brooks, 1987; pp. 16-17]. ¿Es el *software* esencialmente complejo? ¿O es un accidente resultado de cómo se crea actualmente? [Cox, 1996; pp. 51-53] argumenta que el modo en que se crea *software*, usualmente casi desde cero, es el que da la impresión de complejidad.

“Los programadores mayoritariamente están de acuerdo en que el *software* es más complejo que la fontanería de una casa. ¿Pero es la diferencia realmente esencial en el sentido dado por Brooks [...]? ¿No es la complejidad del *software* una consecuencia de cómo nosotros lo construimos? Imagínese la complejidad de un sistema de cañerías si los fontaneros lo construyesen de la forma en que nosotros construimos *software*. Cada fontanero tendría que ser un geólogo para saber dónde realizar minería. Debería ser un experto en refinado y triturado para poder refinar mineral en sus materiales básicos. Tendría que inventar, diseñar, construir y comprobar la vasta multitud de partes que constituyen incluso el más mundano de los sistemas de cañerías. Y finalmente tendría que ser un fontanero, para hacer el trabajo que contrató su cliente en primer lugar. Cuando todo esto se ha terminado, ¿cuál es la percepción que va a tener su cliente? Probablemente que era demasiado caro, que llevó demasiado tiempo y que todavía era defectuoso, justo lo que ahora esperamos del *software* de hoy.” [Cox, 1996; p. 52].

En [Randell, 1979; p. 8] aparece una idea similar: ver a la creación de *software* como una actividad industrial, siendo esa actividad la que limita la complejidad percibida. Todas las demás disciplinas de ingeniería maduras usan componentes como medio para controlar su complejidad. Algunas veces se dice que el *software* es demasiado flexible para la construcción de componentes. Según [Szyperski, 1998; p. xiii] es, en cambio, una indicación de la inmadurez de esta disciplina.

Una propiedad de las ingenierías desarrolladas es que están sostenidas por mercados de componentes especializados en sus áreas de conocimientos. Un aspecto importante para el establecimiento de un próspero mercado de componentes que soporte a la ingeniería del *software* es conocer de dónde se obtiene el beneficio de la venta de componentes. Si bien los aspectos económicos van mucho más allá de los objetivos de este trabajo, son ilustrativas las fuerzas de fondo que gobiernan el éxito o fracaso de los mercados de componentes. Los componentes de grano más grueso como bases de datos, hojas de cálculo o procesadores de texto son rentables porque son lo suficientemente extensos, aunque en la actualidad no se puedan componer muy bien, en parte por su elevada complejidad. Si, como ocurre a menudo hoy en día, los componentes de grano más fino se venden con licencia libre de copia, es posible que tal mercado sea a lo sumo marginal. [Cox, 1996; pp. 41-42] arguye en este sentido. Al contrario de lo que ocurre con productos manufacturados, donde el coste de creación de cada copia redundante es un beneficio para el fabricante, el coste de copia de componentes *software* no lo hace. El éxito de un cliente de un producto manufacturado lleva consigo el éxito de sus proveedores, pues éstos obtienen un margen más amplio gracias al aumento de facturación de los productos vendidos al cliente. En cambio, los proveedores de componentes *software* no obtienen usualmente beneficio extra del éxito de sus clientes, porque el coste de copia es cero. La ausencia de beneficio extra obliga a subir el coste de los componentes, que a su vez limita el número de clientes potenciales. El aumento de precio hace rentable a los grandes clientes obtener la copia inicial de los componentes para luego realizar múltiples réplicas. En cambio, a pequeños clientes no les resulta rentable el alto precio. Por fortuna, no está claro que ése sea el resultado inevitable. Se han propuesto posibles soluciones para la viabilidad de los mercados de componentes como el pago por uso o la tarifa plana [Cox, 1996; pp. 153-165], [Szyperski, 1998; pp. 340-341]. [Wallnau, Hissam & Seacord, 2002; pp. 15-21] argumentan que el régimen de mercado actualmente ya modela qué funcionalidad de componentes aparece disponible, cuándo surge, cómo se distribuye entre distintos componentes, y cuánto tiempo se soporta. Determina también las interfaces usadas y qué estándares son adoptados. Nuevas técnicas y procesos se hacen entonces necesarios para acomodar la pérdida de control sobre aspectos importantes del diseño de sistemas informáticos, como su división, interfaces o coordinación —proveídos por componentes— y para adaptarse a la inestabilidad del incipiente mercado de componentes actual.

1.1 Motivación

Aires renovados en ingeniería del *software* afloran en el último lustro, observando que en el mercado sí existe cierta tendencia hacia la obtención de componentes. Estas ideas, estrechamente relacionadas con los argumentos anteriores, consideran el desarrollo de componentes similar a otros procesos industriales en los que las prácticas ingenieriles están más asentadas. Mediante una analogía con esos procesos, la complejidad debe gestionarse mediante la especialización, la especificación,

fabricación, ensamblado y control de calidad de componentes según estándares [Heineman & Councill, 2001; pp. xxi-xxiii]. Desde este nuevo punto de vista, la división de sistemas informáticos en subsistemas, junto con los nuevos modelos de componentes, en vez de ser considerados casos especiales de menor interés como hace la doctrina tradicional de ingeniería del *software*, se entienden como un paso adelante en el camino hacia una incipiente ingeniería aún por terminar de definir. Son productos que en la actualidad se distribuyen como tales, elementos embrionarios de una ingeniería centrada en la creación y suministro de productos estándar como vía para reducir la complejidad, tiempo y coste, al tiempo que aumentar la calidad de los sistemas informáticos.

Se espera que, con el tiempo, el énfasis sobre los productos resultantes sea mayor que sobre los procesos usados para construirlos. En la actualidad la situación es la inversa. Según la doctrina tradicional de ingeniería del *software*, el énfasis está más en los procesos —lenguajes y metodologías de construcción de programas— que en los productos resultantes. La tendencia hacia la combinación o ensamblaje de programas como vía para la reducción de costes en el diseño, construcción y mantenimiento, se ve socavada tanto por aspectos filosóficos establecidos acerca de la naturaleza del *software*, como por las limitaciones de los lenguajes populares actuales, problemas de los modelos de componentes, o qué condiciones hacen económicamente factible la creación de *software* mediante componentes y ensamblado.

La aceptación de que la ingeniería del *software* debe dirigirse hacia el estudio de cómo realizar mejor la combinación y ensamblado de componentes, es el primer paso para superar los aspectos filosóficos establecidos acerca de la esencia y accidentes del *software*. Esta tendencia se revisa con mayor detalle en [Cox, 1996], [Heineman & Councill, 2001] y en [Wallnau, Hissam & Seacord, 2002]. Pero tal aceptación es sólo el comienzo. Multitud de problemas técnicos separan la idea intuitiva de combinación de componentes de una realización práctica, dificultades que limitan todavía más las posibilidades de existencia de un próspero futuro mercado de componentes.

Los modelos de componentes más conocidos como CORBA [OMG, 2002b], EJB [Sun, 2001] o COM+ [Microsoft, 2003c] describen conceptos que no suelen estar soportados directamente por los lenguajes de programación: interfaces, acceso remoto, disponibilidad de servicios o modos de ejecución. Aspectos que deben ser habitualmente manipulados desde los lenguajes. Desde dentro de ellos, los componentes usualmente se exponen como objetos especiales con ciertas propiedades específicas. Generalmente, esas propiedades deben estar presentes a la hora de manipularlos, mezclándose los conceptos propios de los lenguajes con aquellos proporcionados por los componentes, y dificultando en última instancia la construcción de programas. Por ejemplo, la manipulación de objetos CORBA o COM+ en C [Kernighan & Ritchie, 1988] o C++ [Stroustrup, 1994] es bastante compleja. Se alivia mediante la construcción de objetos intermedios en el caso de C++. Aun así, los objetos intermedios que representan componentes deben tratarse con sumo cuidado, ya que parecen objetos pero en realidad no lo son y se rigen por reglas especiales.

La combinación de componentes consiste normalmente en efectuar llamadas a los objetos intermedios expuestos en los lenguajes de programación. Las interfaces de los componentes no son más que llamadas a métodos de objetos y su funcionalidad no está estandarizada, si bien existen algunos intentos serios en este sentido [OMG, 2002a]. Es difícil reusar o reemplazar unos componentes por otros, perdiéndose parte del beneficio de usar componentes. Su uso, en cambio, aumenta la complejidad de los sistemas construidos, puesto que la complejidad no está escondida detrás de abstracciones lo suficientemente sólidas. Detalles técnicos de implementación implícitos en cómo se exponen los componentes —modos de ejecución, ciclo de vida, o los mecanismos de obtención de referencias a componentes— afloran constantemente durante el desarrollo y mantenimiento, requiriendo considerables conocimientos técnicos.

El camino hacia sistemas informáticos basados en componentes será todavía arduo. La naturaleza de los problemas que se plantean no se encuentra siquiera bien definida. A la hora de plantear qué posibles mejoras se podrían realizar sobre los sistemas actuales basados en componentes, existen al menos dos alternativas interesantes. La primera consiste en identificar dificultades en los

modelos de componentes actuales y tratar de definir una variante o un nuevo modelo de componentes que alivie esas deficiencias. La segunda pretende evaluar si parte de las dificultades encontradas son más profundas y están relacionadas con los lenguajes usados para implementar los componentes, presentando en este caso una extensión o un nuevo lenguaje que solvente las carencias.

Una ventaja clara de la primera aproximación es que puede concentrarse en problemas asociados con componentes y sus resultados son aplicables a lenguajes de programación ya desarrollados que, en última instancia, pueden facilitar su difusión. Un inconveniente obvio es que los problemas que aparecen durante la especificación de la interfaz de programación del modelo de componentes se entremezclan con los problemas de los lenguajes de programación usados. Esta es una deficiencia conocida que limita la aceptación de los modelos de componentes y requiere el estudio de herramientas que asistan en el proceso.

La segunda aproximación tiene la ventaja de permitir evaluar los modelos de componentes desde una perspectiva más fundamental, teniéndose más libertad a la hora de proponer alternativas que puedan dar solución a mayor número de anomalías. Un inconveniente importante es que demanda la definición consistente de una extensión o un nuevo lenguaje. Además, debe dar soluciones específicas para los modelos de componentes. Otro inconveniente es que las ideas propuestas tienen menor probabilidad de ser aplicadas en lenguajes actuales ya maduros, por lo que su difusión puede ser más limitada.

Cuando se analiza con mayor detenimiento las deficiencias de los modelos de componentes actuales, se encuentran ciertas características y dificultades que afectan tanto a los modelos de componentes como a los lenguajes de programación. Entre ellas podemos resaltar las siguientes:

- La disponibilidad de interfaces de componentes separadas de la implementación de los mismos es una característica de todos los modelos de componentes. Curiosamente, la carencia de esa diferenciación en los lenguajes de programación es fuente de conocidos problemas recurrentes en los sistemas de tipos, tales como los que aparecen en lenguajes orientados a objetos basados en clases, donde el tipo asociado a una clase tiene propiedades distintas del tipo asociado a una interfaz, si ésta última existe [Snyder, 1986a].
- La semántica asociada a las interfaces de un componente no va más allá de la identificada por los nombres de las interfaces, operaciones y parámetros usados. La semántica asociada a llamadas a subrutinas y métodos de objetos tampoco, lo que permite una correspondencia prácticamente uno a uno entre ambas definiciones que puede incluso automatizarse. Esta particularidad sugiere que uno de los dos conceptos es redundante. Es más, la definición de interfaces de componentes y su verificación precisa la existencia de algún sistema de tipos, similar —aunque más sencillo— al de un lenguaje de programación.
- La manipulación de componentes dentro de los lenguajes de programación se realiza mediante la construcción de objetos intermedios o *proxies* soportados directamente por los lenguajes, que esconden parte de la complejidad asociada con la llamada a operaciones en los componentes. Estas llamadas requieren considerable código adicional, puesto que los modelos de componentes han de almacenar toda la información de las interfaces dadas por los componentes para poder construir esas llamadas. Los lenguajes de programación han de obtener el mismo tipo de información para ser capaces de construir las llamadas a subrutinas o a métodos de objetos, información que está disponible también en el soporte en tiempo de ejecución o *runtime* del lenguaje si éste soporta alguno. Esta peculiaridad sugiere también que una de las dos funcionalidades es redundante.
- La implementación de los modelos de componentes no suele hacer suposiciones acerca de la disponibilidad de la funcionalidad de una interfaz, puesto que el código es más genérico y es viable descubrir tal funcionalidad dinámicamente. Sin embargo, estos modelos suelen construir habitualmente objetos específicos de una interfaz dada conocida, para ofrecer objetos intermedios comprobables en tiempo de compilación por el lenguaje que los usa. Es decir, en la

práctica hacen suposiciones acerca de la disponibilidad de la funcionalidad de una interfaz. Los lenguajes con soporte en tiempo de ejecución almacenan también la misma información.

- El intercambio de datos entre componentes precisa el uso de operaciones entre varios componentes, manipulación de referencias a componentes, almacenamiento de referencias a componentes dentro de otros componentes, o paso de parámetros entre ellos. Su soporte necesita la definición e implementación de cada una de estas características, cuyo cometido es muy similar al que ha de resolver el compilador de un lenguaje de programación.
- Muchos componentes probablemente necesiten programación de más bajo nivel, como por ejemplo mecanismos de sincronización o de colaboración. Es deseable que esa funcionalidad también pueda realizarse con componentes, cobrando especial relevancia las prestaciones obtenidas por ellos, e insinuando que deben existir lenguajes que soporten a la vez la construcción de componentes y programación de bajo nivel.
- Los componentes son una abstracción adecuada para aspectos de programación de sistemas, como por ejemplo los controladores de dispositivos en sistemas operativos o mecanismos de distribución. Deben usar interfaces conocidas por el sistema operativo y, en los sistemas operativos más modernos, a veces poder activarse o desactivarse dinámicamente en tiempo de ejecución, como ocurre por ejemplo con las interfaces de acceso a redes. Sin lenguajes que soporten simultáneamente programación de sistemas y soporte de componentes, es necesario emular esas funcionalidades usando convenciones de codificación que complican notablemente la construcción de estos tipos de sistemas.
- Los modelos de componentes suelen incorporar también *servicios* adicionales, que corresponden con aspectos no funcionales ortogonales a la definición de los componentes, es decir, aspectos que no forman parte de la funcionalidad directa de los componentes. Servicios de seguridad y cifrado durante el acceso a operaciones de componentes, persistencia, o los modos de ejecución con uno o varios hilos de ejecución son algunos ejemplos. Los lenguajes de programación que incluyen protocolos de metaobjetos [Kiczales, Rivières & Bobrow 1991] o aspectos [Kiczales *et alii*, 1997] resuelven también este tipo de requisitos no funcionales.

El análisis anterior sugiere que la definición de un modelo de componentes debe resolver múltiples problemas que también son objetivo de los lenguajes de programación modernos. Además, la programación de sistemas facilita la creación de componentes más eficientes, acceso controlado a características de bajo nivel, y poder aplicar técnicas de componentes para racionalizar y simplificar el diseño e implementación de programas de sistemas, cada vez más complejos. Los lenguajes ofrecen una abstracción más fuerte para esas tareas que los modelos de componentes, puesto que los lenguajes esconden mejor los aspectos de implementación. Los modelos de componentes, para esconder esos aspectos, deben suministrar compiladores de interfaces, generadores de objetos intermedios o *proxies*, y duplicar parte de las comprobaciones realizadas por los lenguajes de programación. Aun así aportan un modelo sólo parcialmente opaco al lenguaje. Es de esperar que el uso de un lenguaje que soporte la construcción de componentes dé lugar a abstracciones más simples. Como ejemplo de este último caso, es interesante comprobar que, si bien el lenguaje Visual Basic de Microsoft [Microsoft, 2003a] posee un diseño poco consistente y falto de elegancia, buena parte de su éxito reside en que esconde de forma bastante efectiva los mecanismos de acceso a los componentes COM, facilitando el uso de componentes desde dentro del lenguaje y dando gran potencia al lenguaje. Este escenario contrasta con la dificultad de manipular componentes COM desde otro de los lenguajes más usados de la misma empresa: Visual C++ [Microsoft, 2003b]. Microsoft sigue una línea similar de razonamiento a la de Visual Basic con C# [ECMA, 2002a], prestando esta vez mucha más atención al diseño del lenguaje.

Debido a las razones anteriores, este trabajo se decanta por la segunda aproximación, es decir, mejorar el soporte de componentes mediante la definición de sus características en un lenguaje de programación con facilidades de programación de sistemas. Es en cierto modo paradójico que durante el intento de hacer mayor énfasis en los productos resultantes —en este caso los componentes— se acaben definiendo nuevas propiedades de los procesos que los construyen, es decir, de los lenguajes.

Un examen más detallado muestra que no existe tal paradoja, aunque se trata de una moneda de dos caras.

Por un lado, si la definición de un mejor proceso permite construir con mayor fiabilidad componentes con propiedades bien definidas, y si las decisiones sobre las características de un lenguaje que soporte componentes están supeditadas a la definición coherente de un mejor modelo de componentes, entonces la definición del lenguaje se convierte en un medio para conseguir un fin. El fin continúa siendo una mejor definición de productos, el medio automatiza esa definición y resuelve el fin con más eficiencia. Contrasta este enfoque de diseño de lenguajes centrado en el tipo de productos con el tradicional de los lenguajes de programación. Este último enfoque tiende a la creación bien de lenguajes minimalistas con interesantes y elegantes propiedades matemáticas y algorítmicas, bien a la definición de lenguajes con un número arbitrario de funcionalidades consideradas útiles, donde el criterio de qué debe formar parte del lenguaje depende de si esas funcionalidades son capaces de resolver problemas interesantes.

Por otro lado, sólo cuando se materialice la construcción de componentes estándar se obtendrán finalmente los beneficios del uso de sistemas basados en componentes. Esta segunda cara de la moneda está más lejana en el tiempo, esperándose que surja a partir de mejores herramientas que ayuden a la construcción de componentes y faciliten la definición de componentes estándar. Por esa razón, no debe ser objetivo del lenguaje promover un estándar particular, sino fomentar su integración con aquellos que surjan, usando las abstracciones adecuadas definidas en el lenguaje.

Optar por mejorar o definir un nuevo lenguaje simplifica la gestión de componentes dentro del lenguaje, evita definiciones redundantes, automatiza la creación final de componentes con ayuda de un compilador, y provee un modelo de componentes de más alto nivel directamente soportado por el lenguaje. Este modelo, como se verá más adelante, tiene propiedades similares a los objetos de los lenguajes orientados a objetos, factor que ayuda a eliminar redundancia en el modelo de objetos y componentes, aun cuando ambos conceptos son y deben seguir siendo diferentes. Se espera que la simplificación global obtenida permita construir en un futuro modelos de componentes más sofisticados e integrar modelos de componentes existentes sobre las abstracciones del lenguaje.

El mayor reto de esta decisión recae en la toma de conciencia de la necesidad de definir unas reglas consistentes, robustas, escalables y eficientes para el lenguaje: Diseño interno, abstracciones subyacentes, funcionalidad soportada, o sintaxis expuesta a sus usuarios son algunos aspectos a tener en cuenta. Es decir, no sólo habrá que comprender con profundidad los problemas de la construcción de *software* usando componentes, también será necesario entender y evaluar con el mismo detalle las características de un lenguaje de programación, su diseño, problemática e implementación eficiente.

1.2 Objetivos

Siguiendo las ideas sobre la posibilidad de describir mediante componentes subsistemas complejos de sistemas informáticos, que nos acerquen a las ideas de ingeniería del *software* basada en componentes, el presente trabajo estudia en primer lugar las limitaciones en las técnicas actuales de subdivisión de sistemas informáticos y de su ensamblado. Este análisis pone de relieve deficiencias —muchas veces conocidas pero aceptadas sin más— en las técnicas usadas en la subdivisión del diseño y en el ensamblado de las aplicaciones finales. Carencias que aumentan la complejidad percibida de los programas construidos, limitan las posibilidades de creación de componentes, y, en última instancia, fomentan la percepción tradicional de la complejidad del *software*.

También estudia el estado actual de los esfuerzos en la construcción de componentes. Se hace énfasis en las limitaciones de los lenguajes de programación usados para la definición de los modelos de componentes actuales —CORBA, COM+ o EJB—. Estas limitaciones distorsionan las características percibidas sobre qué es y qué debería ser un componente. Algunas de las propiedades atribuidas a los componentes —como las interfaces— deberían ser parte integrante de los lenguajes de programación,

puesto que corresponden básicamente con deficiencias de estos últimos. Estas faltas han encontrado una solución temporal en la implementación de los modelos de componentes actuales, pero no son propiedades exclusivas de los componentes. Otras características se encuentran implementadas de manera redundante en modelos de componentes y lenguajes.

Con el fin de construir productos con las propiedades adecuadas automáticamente y basándose en los estudios anteriores, el trabajo describe y justifica mejoras en los procesos de construcción de programas que nos ayuden a acercarnos a ese objetivo. Son guías de diseño para lenguajes de programación donde se pone mayor énfasis en las propiedades de los productos resultantes —programas que faciliten la construcción de componentes— que en el proceso que los genera —el lenguaje—. En vez de emplear una argumentación teórica que pueda pasar por alto problemas técnicos serios, este trabajo describe la definición e implementación de un prototipo de un nuevo lenguaje de programación, usado como ejemplo de implementación de las guías de diseño estudiadas. Para hacer el análisis más interesante y práctico, y porque es de esperar que muchos componentes relevantes se construyan con programación de sistemas, el tipo de sistemas informáticos a los que va orientado el estudio y la definición del prototipo es similar al que hoy por hoy se pueda construir en C, C++, Modula-3 [Cardelli *et alii*, 1992] u Objective-C [Cox & Novobilski, 1991]. Estos sistemas habitualmente comparten una tensión inherente entre funcionalidad, flexibilidad, tamaño, eficiencia y programación portable de bajo nivel.

Se estudian qué características deben tener los programas generados para fomentar su combinación y ensamblaje, así como técnicas para realizarlas. Estas características a veces dan lugar al diseño de una funcionalidad soportada por el lenguaje. Es decir, el razonamiento va del producto al proceso, no al revés. Este cambio de énfasis no va en detrimento de la importancia de la definición del lenguaje. Poniendo una analogía industrial, la construcción de los elementos de una fábrica va guiada por los productos que se pretenden fabricar, pero no por ello la construcción de la fábrica es menos importante. No se trata de construir la mejor fábrica que produzca cualquier producto, se trata de construir una fábrica que produzca productos con las características deseadas. La relevancia principal del cambio de énfasis radica en que es una mejor guía para el diseño de lenguajes y, por extensión, para el diseño de productos: existen ideas atractivas en el diseño de lenguajes que deben dejarse atrás por el simple hecho de ir en contra del tipo de productos generados. Sin esta guía, algunas de esas ideas podrían encontrar su camino dentro de la definición de un lenguaje y, posteriormente, generar confusión si se pretende construir productos para los que tal funcionalidad no fuese conveniente.

No existe mucha teoría que ayude a diseñar un buen lenguaje de programación, ni qué propiedades debe tener. Se trata mucho más de un arte que de una ciencia, lo cual es una manifestación más de la inmadurez de nuestra disciplina. Es posible obtener provecho de las experiencias pasadas en el diseño de lenguajes para evitar cometer errores conocidos, pero sólo la experiencia y el uso continuado podrá responder a esas preguntas. Un lenguaje madura con el paso de los años y con la invaluable aportación de conocimiento que realizan sus usuarios a través los problemas que detecten.

En resumen, los objetivos son los siguientes:

- Análisis e identificación de deficiencias en la subdivisión y el ensamblado de sistemas informáticos, causantes de un aumento de la complejidad observada en ellos.
- Análisis de características redundantes en modelos de componentes y lenguajes de programación, con el objetivo de simplificar las abstracciones disponibles para el diseñador y el programador.
- Definición precisa de guías de diseño para lenguajes de programación que fomenten la programación con componentes, su combinación y ensamblado, y que, simultáneamente, faciliten la programación de sistemas. Guías que pongan el énfasis en los productos generados, en vez de en los procesos que los generan.
- Implementación de un prototipo de un nuevo lenguaje de programación que sirva para ensayar y evaluar en la práctica las guías de diseño identificadas, incluyendo la presentación de las técnicas de implementación usadas para conseguir ese fin.

1.3 Organización del documento

El resto del documento se organiza como sigue: En el segundo capítulo se define con mayor rigor qué es un componente, y en qué lugar se encuentra la práctica y el uso de los componentes en la actualidad, las tendencias en sistemas de ensamblaje y los modelos de componentes más comunes. Luego se estudia el concepto de interdependencia, para introducir los problemas de fondo de la construcción de componentes. El tercer capítulo expone cómo se va a abordar el diseño del nuevo lenguaje. Las premisas de diseño, las características del modelo de componentes y aspectos de programación de sistemas. Al tratarse de un nuevo lenguaje, especial énfasis se da a la consistencia, robustez, escalabilidad y eficiencia. Por último, se realiza una breve comparación del lenguaje propuesto con otros lenguajes existentes. El capítulo cuarto examina con más profundidad el núcleo del lenguaje, estudiando el modelo de objetos usado para implementar el modelo de componentes. Se evalúa cuál es el mecanismo de herencia más apropiado, el proceso de envío de mensajes, y las técnicas de metaprogramación usadas. El quinto capítulo presenta el lenguaje desde el punto de vista sintáctico, incluyendo ejemplos ilustrativos de las abstracciones más relevantes. Con la información expuesta en los capítulos anteriores el capítulo sexto introduce el modelo de componentes, sus características y ejemplos de cómo construir componentes con el lenguaje. El capítulo séptimo analiza con detalle el sistema de tipos, piedra angular responsable de la flexibilidad controlada del lenguaje. El capítulo octavo se centra en el estudio del soporte en tiempo de ejecución, el diseño del prototipo del compilador, y cómo ambos se coordinan para resolver importantes aspectos de eficiencia, críticos para la viabilidad de la programación de sistemas con el lenguaje. Por último, el capítulo noveno presenta las conclusiones de este trabajo y futuras líneas de investigación.

2

Componentes y técnicas de ensamblado

2.1 Introducción

En este capítulo se estudian limitaciones en las técnicas actuales de subdivisión de sistemas informáticos y de su ensamblado. En primer lugar, y para centrar el tema, se dan algunas definiciones preliminares acerca de términos básicos que permitan definir qué es un componente y un modelo de componentes. Posteriormente se estudia la estructura de un sistema informático típico, haciendo hincapié en la efectividad de la división de estos sistemas en subsistemas estándar y cuándo empiezan a aparecer dificultades. Luego se repasan las tendencias actuales en sistemas de ensamblaje y los modelos de componentes más comunes. Por último, se presenta el concepto de interdependencia como elemento clave en la definición de un modelo de componentes. Existen deficiencias conocidas y aceptadas en el proceso de ensamblado, tanto en subrutinas como en subsistemas completos, que dificultan la construcción de aplicaciones y modelos de componentes con las propiedades adecuadas. Estas deficiencias se analizan siguiendo las ideas introducidas con la interdependencia a través de sus distintos niveles, de menor a mayor complejidad. Atención especial requieren los mecanismos de llamada entre diferentes partes de código y las condiciones de reemplazo de componentes.

2.2 Definiciones preliminares

La palabra *componente* empieza a ser usada en demasiados contextos. Es un vocablo muy genérico, que puede dar lugar fácilmente a interpretaciones inexactas o confusas. En este apartado se establece con detalle qué se considera un componente, así como otros términos relacionados. La mayoría de las definiciones se basan en las descritas en [Councill & Heineman, 2001; pp. 5-19], que representan un consenso amplio acerca de las mismas. Se introducen primero algunos términos fundamentales necesarios para las definiciones posteriores, continuando luego con ellas.

- ❖ **Def. 2-1.** Un *sistema de información* aúna el conjunto de procedimientos manuales y automáticos que permiten recopilar, procesar y distribuir información.
- ❖ **Def. 2-2.** Un *sistema informático* es un sistema de información automatizado que reúne elementos *software*, elementos *hardware* y al personal que los manipula¹.

¹ En este trabajo nos centraremos en los elementos *software* de un sistema informático.

- ❖ **Def. 2-3.** Los programas o genéricamente *software* son construidos para ejecutarse sobre una máquina de cómputo de von Neumann (de aquí en adelante una *máquina*).
- ❖ **Def. 2-4.** Un *elemento software, parte de un programa o programa* contiene secuencias ordenadas de instrucciones o sentencias abstractas que detallan las operaciones realizadas por una máquina.
- ❖ **Def. 2-5.** Un *elemento hardware* es un dispositivo físico que forma parte de una máquina de cómputo de von Neumann.
- ❖ **Def. 2-6.** Un programa es *ejecutable por una máquina* si: (1) la máquina directamente ejecuta las instrucciones descritas o (2) un intérprete ejecutable por la máquina directamente entiende las instrucciones descritas y la máquina directamente ejecuta el intérprete. Este programa se llama también *código objeto* y se dice que está en *forma binaria*.
- ❖ **Def. 2-7.** El *código fuente* de un programa es el conjunto de ficheros legibles por una máquina contenido sentencias de programa escritas en un lenguaje de programación. Estas sentencias son compiladas a código objeto, y ejecutadas directamente por una máquina o ejecutadas por un intérprete.
- ❖ **Def. 2-8.** Una *interacción* es una acción entre dos o más elementos *software* o programas.
- ❖ **Def. 2-9.** Una *interfaz o protocolo* es una abstracción del comportamiento de un componente (Def. 2-11). Consiste en un subconjunto de las interacciones de ese componente junto con un conjunto de restricciones que indican cuándo pueden éstas ocurrir. La interfaz describe el comportamiento de un componente considerando sólo las interacciones identificadas en la interfaz, manteniéndose escondidas el resto de las interacciones del componente.
- ❖ **Def. 2-10.** *Composición* es la combinación de dos o más componentes *software* (Def. 2-11) mediante interacciones que dan lugar a un nuevo comportamiento de componentes en un diferente nivel de abstracción. Las características de comportamiento del nuevo componente están determinadas por los componentes que se combinan y por el modo en que son combinados.
- ❖ **Def. 2-11.** Un *componente software* es un programa que sigue un modelo de componentes, puede ser independientemente desplegado (*deployed*) y puede componerse sin modificación de acuerdo con un estándar de composición.
- ❖ **Def. 2-12.** Una *implementación de un componente* es un programa en formato de código fuente, código binario o ambos, que contiene las operaciones que definen la funcionalidad del componente *software* para ser ejecutadas por una máquina.
- ❖ **Def. 2-13.** Un *modelo de componentes* define unas reglas estándar específicas de composición e interacción a través de interfaces.
- ❖ **Def. 2-14.** Una *implementación de un modelo de componentes* es el conjunto de elementos *software* dedicados que se requieren para soportar la ejecución de componentes que sigan un modelo dado.
- ❖ **Def. 2-15.** Una *infraestructura de componentes* es un conjunto de componentes *software* diseñados para asegurar que un sistema informático o subsistema construido usando esos componentes e interfaces satisfacen especificaciones de prestaciones bien definidas.
- ❖ **Def. 2-16.** Un *lenguaje orientado a componentes* es un lenguaje que directamente soporta en su sintaxis y semántica un modelo de componentes.

Estas definiciones dan una descripción precisa de qué es un componente y qué implica usar componentes. Un componente es un programa que tiene un conjunto de características especiales. La Figura 2-1 muestra una visión general de las mismas. Cada componente consta de tres partes diferenciadas: la interfaz del componente, la implementación del componente, y el modelo de componentes subyacente, que confiere ciertas propiedades a la interacción entre componentes. La estructura exacta de las interfaces, cómo se conecta una implementación a una interfaz, cómo se conectan varias interfaces entre sí, o cómo se construyen los componentes, son algunas de las propiedades que ha de definir el modelo de componentes. La Figura 2-1 también muestra algunas variantes importantes de la implementación de los componentes.

- ❖ **Def. 2-17.** Un componente es *simple* si tiene una implementación completamente particular, no relacionada con otros componentes.
- ❖ **Def. 2-18.** Un componente es *compuesto* si está formado por otros componentes. Puede tener dos tipos de implementación: (1) una implementación parcial particular junto con otra suministrada por algún componente interno, y (2) una implementación formada exclusivamente por componentes internos.

La importancia del caso (2) de (Def. 2-18) es que *no* precisa codificación adicional, por lo que el ahorro en desarrollo y control de calidad es mayor. La definición (Def. 2-11) de *componente software* implícitamente se adhiere a este último caso, puesto que es la organización de componentes que se desearía tener. El componente compuesto que requiere codificación adicional incurre en costes extra de desarrollo y control de calidad. El código adicional tiene dos lecturas diferentes y complementarias: puede ser necesario debido a que la manipulación del componente interno es compleja y no es dada por otros componentes, o también puede ser debido a deficiencias de abstracción en el modelo de componentes, que demanda codificación adicional para realizar la composición de los componentes internos.

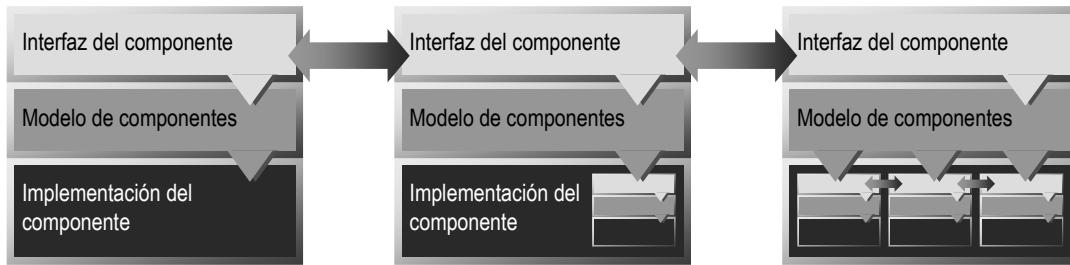


Figura 2-1 Diferentes tipos de componentes.

A la izquierda, componente simple con una implementación particular. En el centro, un componente con implementación particular parcial, el resto recae en un componente interno. A la derecha, un componente formado únicamente por otros componentes interconectados.

[Szyperski, 1998; p. 3-4] resalta la importancia de que la distribución y entrega de componentes se produzca en forma binaria, y que la composición de componentes use únicamente la forma binaria y la descripción de la interfaz del componente. Este tipo de distribución es importante porque protege la inversión realizada por los fabricantes de componentes, a la vez que enfatiza la importancia de la interfaz como único nexo de unión entre componentes.

Por último, véase que un modelo de componentes siempre tiene dos caras: una especificación y una implementación. Si la especificación es pública, entonces es posible que exista más de una implementación del modelo de componentes. Este es el caso de CORBA o EJB por ejemplo, pero no el caso de COM. La interoperabilidad entre diferentes implementaciones de un modelo de componentes es un problema serio, puesto que al tratarse de definiciones bastante complejas, no es difícil que existan lagunas en las especificaciones. Este ha sido y es un problema que ha afectado durante años a CORBA y EJB. Con la tecnología disponible, la única manera de asegurar la definición correcta de una especificación es proporcionando, además, una implementación de referencia. No es de extrañar entonces que la falta de una implementación de referencia tanto para CORBA como para EJB haya estado entre las causas de las discrepancias de interoperabilidad.

Las definiciones dadas en este apartado asientan la *ingeniería del software basada en componentes* [Heineman & Council, 2001]. Representan el deseo de construir sistemas informáticos mediante la composición de componentes que escondan la mayor parte de su complejidad en su interior. Si bien la meta es loable y es compartida en este trabajo, el estado actual de la técnica dificulta su consecución. En la práctica, la composición o ensamblado de componentes o de partes integrantes de ellos es muchas veces muy difícil. Para entender mejor la naturaleza de este problema y poner en relieve la tendencia actual hacia la

construcción de componentes, revisaremos en primer lugar un sistema informático común hoy en día. Veremos también los mecanismos de subdivisión del mismo y dónde comienzan a surgir dificultades.

2.3 Un sistema informático típico

Los sistemas informáticos de hoy presentan considerable complejidad. Para lidiar con ella se estructuran jerárquicamente, involucrando múltiples subsistemas, algunos de los cuales quizás no sean conocidos a priori. La Figura 2-2 muestra un sistema informático para Internet típico de complejidad media. Corresponde en este caso con una hipotética aplicación de banca *on-line* para Internet. En ella se pueden identificar varios subsistemas principales: aplicación bancaria, base de datos asociada con la aplicación bancaria, *host* o sistema central del banco, servidor *web* y clientes que se conectan quizás a través de Internet. La aplicación bancaria se encarga de presentar a sus clientes una interfaz amigable para ver el estado de sus cuentas y realizar operaciones. La aplicación conecta a los clientes con los mismos sistemas bancarios internos accesibles desde una sucursal o un cajero automático. La conveniencia del cliente a través de Internet para el banco se resume en que puede hacer uso de la infraestructura de Internet para dar mejor servicio, acercando su aplicación al usuario final con poco coste adicional. Este último se puede conectar desde su casa o su trabajo con mayor comodidad.

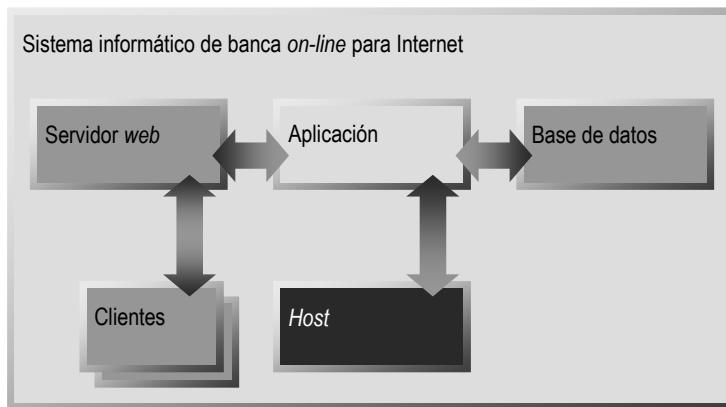


Figura 2-2 Sistema informático de complejidad media.

Involucran habitualmente diferentes subsistemas conectados por protocolos de red, que pueden residir en uno o más nodos. El mantenimiento se simplifica si cada subsistema es reemplazable por otro equivalente o una nueva versión por separado.

La disposición en subsistemas del sistema informático de la Figura 2-2 le confiere interesantes posibilidades y flexibilidad si se realiza con cuidado. Si la conexión entre los diferentes subsistemas se efectúa mediante un protocolo de red, el sistema informático puede ser configurado para ejecutarse bien en un único nodo de red —incluidos los clientes— o en múltiples nodos. Si se configura en múltiples nodos, es posible que éstos ejecuten sistemas operativos distintos. La comunicación entre los nodos es más fácil de obtener si se usan protocolos de comunicación estándar. Si el protocolo de red es estándar —por ejemplo TCP/IP— así como el formato de la información enviada a los clientes —HTML— entonces es posible que clientes desconocidos que cumplan los protocolos descritos se conecten a la aplicación bancaria, participando del sistema informático global sin esfuerzo adicional. Nótese cómo en este caso la existencia de un estándar de comunicación del formato de datos entre la aplicación y el cliente ayuda a hacer el sistema más flexible y adaptable a nuevos clientes desconocidos sin necesidad de recodificación. En cambio, el formato de datos entre el *host* y la aplicación no suele ser estándar, estando diseñado específicamente para cada *host*. La falta de estándares impide que la aplicación pueda gestionar *hosts* diferentes simultáneamente, a no ser que se programe especialmente para ello.

La división en subsistemas principales del sistema original, que corresponde con la arquitectura general del sistema informático, se ha realizado teniendo en cuenta la disponibilidad de estándares de comunicación y productos existentes con funcionalidad bastante definida. Es más, tales productos y sus formas de conexión *guían* la definición de la arquitectura. Un servidor *web*, una base de datos, y navegadores o *browsers* clientes son productos establecidos que ofrecen funcionalidad ya construida y probada que no es necesario replicar. Dan una guía sobre la arquitectura básica del sistema en su conjunto, ayudando a esconder la complejidad interna de cada subsistema. Es decir, la arquitectura del sistema refleja la disponibilidad en el mercado de productos que se interconectan de modo definido [Wallnau, Hissam & Seacord, 2002; p. 16]. Este sistema se realimenta, porque a su vez facilita la creación de nuevas aplicaciones para Internet, la difusión de la misma arquitectura, y razones de mercado para suministrar nuevos productos que sigan la arquitectura. Además, el uso de estándares proporciona cierta flexibilidad a la hora de reemplazar alguno de los subsistemas por nuevas versiones u otros subsistemas equivalentes. En ambos casos es importante que el nuevo subsistema mantenga la misma interfaz estándar que el subsistema reemplazado.

En síntesis, la estructura general de una aplicación para Internet, incluso sin llegar a detalles acerca de la aplicación en sí, puede realizarse combinando subsistemas complejos suministrados por diferentes proveedores que se adhieren a estándares conocidos. La existencia de productos asociados a esos subsistemas con funcionalidad conocida y métodos de comunicación estándar entre ellas, facilita abordar el desarrollo de una aplicación en un entorno distribuido para Internet con menor coste y mayor calidad global. La existencia de formatos estándar de comunicación —no sólo de protocolos de comunicación— hace posible que la comunicación entre los clientes y la aplicación sea automática, sin necesidad de codificación a medida. Los subsistemas —que pueden verse como componentes muy complejos— [Szyperski, 1998; p. 30] se combinan sin necesidad de modificarlos internamente y, por tanto, siguen el patrón deseado de interconexión de componentes que se analizó en el apartado anterior.

Continuando con el ejemplo en curso, la división en subsistemas principales propuesta no hace mención sobre la estructura de la aplicación en sí. Tal era el estado de construcción de aplicaciones para Internet hacia principios del año 1997. Nos centraremos ahora en un avance posterior: los servidores de aplicaciones *web*². Estos servidores se encargan de realizar automáticamente la conexión con el servidor *web*, así como la gestión más eficiente de conexiones con los clientes. Con el tiempo, los servidores de aplicaciones *web* han evolucionado hacia servidores independientes para dar mayor robustez y escalabilidad. La Figura 2-3 muestra la estructura de un servidor de aplicaciones *web* actual, como una expansión del cuadro *Aplicación* de la Figura 2-2. El servidor de aplicaciones da una estructura predefinida sobre la cual la aplicación *web* se puede construir con menor esfuerzo.

Se distinguen varios subsistemas importantes coordinados por un motor interno. El conector con servidores *web*, el conector con sistemas de base de datos, y (quizá) conectores con algún tipo de sistema bancario o *host*. Los dos primeros suelen ser gestionados automáticamente, el último raramente lo es. Un servidor de aplicaciones *web* complejo provee también dos subsistemas importantes: el modelo de páginas *web* y el modelo de componentes. El modelo de páginas *web* ofrece a la aplicación una abstracción que separa los datos de petición, los datos internos de la aplicación para cada cliente, los datos de respuesta, y un modo sencillo de crear páginas nuevas. El modelo de componentes sirve para acceder a funcionalidad externa a través del modelo, y organizar parte de la aplicación en componentes que puedan potencialmente reusarse. Los servidores de aplicaciones *web* sencillos no disponen de modelo de componentes. Modelos de componentes usados son, por ejemplo, CORBA, COM+ o EJB, que revisaremos más adelante.

² Las aplicaciones realizadas con CGI (*Common Gateway Interface*) y la evolución posterior a arquitecturas de *scripting* para aplicaciones *web* no es relevante para esta discusión.

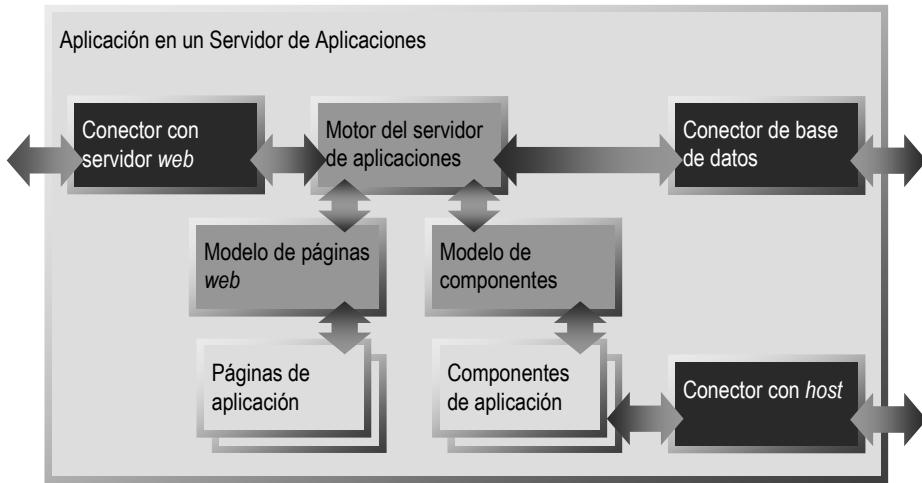


Figura 2-3 Estructura de un servidor de aplicaciones actual.

Define una estructura general para una aplicación *web* y simplifica su desarrollo. En color gris claro está la aplicación *web*, en color gris medio y gris oscuro se ven los subsistemas principales del servidor de aplicaciones *web*.

Los mecanismos de reemplazo de subsistemas se encuentran más limitados en este nivel. Un servidor de aplicaciones *web* típico admite el reemplazo de los conectores y, por supuesto, incorporar nuevas páginas o componentes de la aplicación. Los conectores de *host* no suelen estar soportados, por lo que la responsabilidad recae completamente en el lado de la aplicación. Los servidores de aplicaciones, en cambio, no permiten modificar el motor o los modelos de página y componentes, que en la práctica quiere decir que es necesario cambiar el servidor de aplicaciones por completo si se desea reemplazarlo por otra versión o por otro proveedor. No hay guías adicionales para el diseño de páginas o para la construcción de componentes más allá de la suministrada por el modelo de componentes. Una aplicación *web* compleja debe organizarse a partir de entonces usando técnicas más tradicionales y el apoyo del modelo de componentes suministrado, si éste existe. La disponibilidad de estándares que puedan ayudar a construir subsistemas más independientes también se empieza a ver limitada en este nivel. La aplicación *web* debe generar directamente HTML para presentación de datos, usualmente con poca o ninguna ayuda adicional por parte del servidor de aplicaciones. El conector de base de datos suele estar especificado mediante una interfaz. No existe habitualmente ayuda adicional salvo el lenguaje SQL estándar de base de datos. Si el servidor de aplicaciones provee un modelo de componentes, tal modelo indica únicamente el protocolo de comunicación, cómo describir interfaces de componentes y, a veces, cómo conectarse a componentes externos o remotos. No especifica formatos de datos enviados ni suele proporcionar muchos componentes preconstruidos.

En definitiva, el servidor de aplicaciones aporta una infraestructura para adaptar manualmente una aplicación a medida tradicional o cliente-servidor a Internet, pero no facilita la construcción de la aplicación en sí. La ventaja del uso del servidor de aplicaciones es que un importante conjunto de problemas serios relacionados con las aplicaciones *web* es resuelto directamente por el servidor de aplicaciones. Los servidores de aplicaciones y la estructura general de aplicaciones *web* sirven como ejemplo de las nuevas tendencias dedicadas a dividir las arquitecturas en componentes para hacerlas más asequibles, facilitando su adopción y puesta en funcionamiento. La demanda de mercado por tener aplicaciones *web* ha hecho posible que se creen estándares y productos que más o menos los cumplen, que exista una arquitectura general definida, e incluso que los productos sean razonablemente reemplazables. Contrastá esta situación con la encontrada dentro del ámbito de cada aplicación particular. Aquí las posibilidades son más limitadas, quizás porque aún no existe un mercado de componentes que lo haga posible. Paradójicamente, gracias a la existencia de productos especializados semireemplazables, resulta más sencillo construir un diseño distribuido y escalable para una aplicación *web* que escribir la propia aplicación, quizás mucho más simple técnicamente.

Pero no conviene confundir una tendencia con una realidad. Los servidores de aplicaciones, las bases de datos, los modelos de componentes, o los sistemas CRM (*Customer Relationship Management*), por poner sólo unos pocos ejemplos, distan de ser perfectos. Son a veces llamados productos o componentes COTS

(*Commercial off-the-Shelf*) [Tracz, 2001; p. 100]. Si bien la trayectoria actual se dirige a la integración de estos elementos, como vehículo para la construcción de sistemas informáticos más complejos con menor esfuerzo, lo cierto es que la calidad y las expectativas creadas acerca de los COTS no deben llevar a un entusiasmo prematuro. Los COTS son a menudo grandes y complejos, a veces incluso difíciles de poner en funcionamiento, y suelen requerir un conocimiento extensivo de los mismos para obtener buenos resultados [Gorton & Liu, 2002; pp. 557-558]. [Tracz, 2001; pp. 102-107] identifica algunos mitos y realidades acerca de ellos. Algunos extractos a resaltar son:

- Existe a menudo una diferencia apreciable entre lo que un COTS anuncia y lo que realmente hace, que obliga a realizar más pruebas sobre los COTS de lo que se podría esperar.
- La selección de COTS a veces se basa en demos, búsquedas en la red o lecturas en revistas especializadas, debido a que las infraestructuras de componentes son un campo relativamente nuevo.
- La regla del 80%/20% se aplica a muchos COTS. El cliente puede satisfacer el 80% de sus necesidades con el 20% del coste y tiempo de un sistema a medida. Las dificultades surgen de suponer que el 20% restante se puede obtener con la tarifa habitual de desarrollo *software*. En la práctica este coste es mucho mayor, debido al poco control sobre el producto COTS.
- Los procesos de los COTS no suelen reflejar las mejores prácticas de la industria. Éstos reflejan muchas veces la experiencia del proveedor en el dominio y sus planificaciones de salida a mercado.
- Es difícil influenciar a un proveedor de COTS, aunque el cliente sea lo suficientemente grande. El mercado los influencia. Tampoco suele ser posible contratar al proveedor para modificar los COTS y cumplir ciertos requisitos. Sólo suele ser viable contratar a una empresa externa para que haga las modificaciones.
- Los proveedores a menudo no resuelven los problemas de un producto en la versión actual. Quizá lo hagan en la siguiente versión. El nivel de servicio recibido del proveedor es negociable. A menos que el contrato diga lo contrario, las correcciones recibidas estarán regidas por el mercado.
- No se pueden ignorar las actualizaciones de los COTS. Es común perder el soporte de sistemas anteriores si no se realizan actualizaciones. Además, las actualizaciones del proveedor suelen ser requeridas para reparar defectos.

Si los productos o componentes COTS no tienen las prestaciones o calidad adecuadas, es necesario dejarlos de lado y resolver el problema con desarrollos a medida especializados [Illback, 1999; p. 12]. Las causas de estas deficiencias son tres. (1) Los proveedores suelen asumir erróneamente cómo van a ser integrados sus componentes. (2) Los proveedores quieren retener a sus clientes, usando técnicas de integración particulares o propietarias. (3) Los proveedores intentan innovar, creando nuevas funcionalidades con el mero propósito de introducir dificultades de integración [Wallnau, Hissam & Seacord, 2002; pp. 18-19]. Algunas guías para valorar el uso de COTS actuales en proyectos se dan en [Tracz, 2001; pp. 107-109] y en [Wallnau, Hissam & Seacord, 2002], consejos que ayudan a lidiar con COTS no tan idílicos.

El mercado actual de componentes y productos COTS se encuentra aún en su infancia. En [Traas & Hillegersberg, 2000; pp. 115-117] se identifican condiciones para su crecimiento. Las más relevantes son: atención del mercado hacia los componentes de grano más fino, disponibilidad de información adecuada acerca de los mismos, listas de precios de productos similares, especificación siguiendo estándares, tendencia al reuso de caja negra, protección de copia, buscadores de componentes, y la existencia de intermediarios que den valor añadido, soporte técnico y tengan criterios comunes para comparar componentes.

2.4 Tendencias en sistemas de ensamblaje

La tendencia de los métodos de ensamblaje que se está produciendo actualmente se resume en la Figura 2-4. Puede observarse que se dirige a sistemas más especializados, en los que se intenta sacar mayor partido de los conocimientos de campos específicos, y técnicas que permitan reducir los costes de organización, desarrollo y mantenimiento de *software*.

Tradicionalmente se han construido productos únicos, creados a medida para resolver problemas particulares, y habitualmente usados como subsistemas de un sistema informático mayor. En los años 80 han surgido las primeras infraestructuras de componentes, cuya naturaleza sigue siendo única, puesto que dos infraestructuras con distinta implementación que sirvan para una misma función usualmente no son intercambiables, incluso dentro de una misma organización. Un ejemplo de este tipo de infraestructuras de componentes son los entornos gráficos y los componentes especializados en generación de informes, gráficas o tablas. El reemplazo de estas infraestructuras sólo es posible con una nueva versión de la misma. Dado que desarrollar *software* único es muy costoso, la tendencia tradicional para reducir ese coste ha sido intentar mejorar los procesos que ayudan a desarrollar *software* único. Ejemplos de estas tendencias son el CMM y el RUP (*Rational Unified Process*) [Jacobson, Booch & Rumbaugh, 1999].

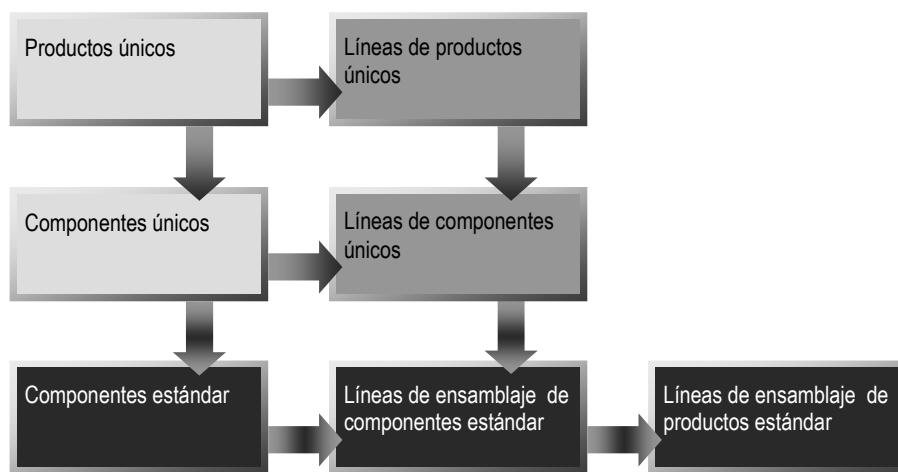


Figura 2-4 El largo camino hacia líneas de ensamblaje de productos estándar.

En color claro, los tipos de productos desarrollados tradicionalmente. En color medio, nuevas tendencias de tipos de productos. En color oscuro, tipos de productos estandarizados, no disponibles hoy día.

Nuevas tendencias en la construcción de productos únicos surgen a mitad de los 80 con el análisis de dominios y FODA (*Feature-Oriented Domain Analysis*) [Kaen et alii, 1990]. A mitad de los 90 las tendencias se refinan, poniendo de manifiesto la necesidad de especializarse más y construir productos parecidos o *líneas de productos*, incluyendo guías técnicas y organizativas que ayuden en estos procesos [Clements & Northrop, 2002]. Así, dentro de un mismo dominio se puede sacar provecho del conocimiento especializado, los procesos de construcción y la estructura de gestión de la organización, para poder crear paulatinamente productos más competitivos a menor coste. En grandes organizaciones una tendencia es construir líneas de componentes reutilizables dentro de la organización [Clements & Northrop, 2002; p. 10]. En ambos casos se continúa hablando de productos o infraestructuras de componentes únicos, puesto que los productos o componentes desarrollados por diferentes organizaciones son incompatibles, aun cuando resuelvan problemas similares.

El siguiente paso lógico es la construcción de componentes estándar. Componentes que son intercambiables entre distintas organizaciones. Este paso sólo se está empezando a producir, porque requiere la definición de estándares que puedan ser verificables. Las especificaciones de Java sobre multitud de subsistemas, como Servlets, JSP, JMS, o JDBC [Sun, 2002], entre otros, van en esta última dirección, aunque son usuales los problemas de compatibilidad entre distintos proveedores. El modo de ensamblado del *software* dentro de un sistema informático introduce dependencias no deseadas que dificulta que esos estándares maduren. Genera también obstáculos para describir y construir herramientas que ayuden a verificar que el ensamblado se ha realizado correctamente. La incapacidad para definir y producir componentes estándar afecta a la evolución posterior de esos procesos de producción hacia otras formas más eficientes: Líneas de componentes estándar y líneas de productos estándar, cuya posible existencia tan solo se puede intuir hoy en día.

2.4.1 Ingeniería de dominios

Dejar de pensar en productos únicos y centrarse en cambio en conjuntos de sistemas con características similares —de las que se pueda sacar provecho para reducir costes y tiempos mejorando la calidad— constituye la base de la ingeniería de dominios. Un *dominio* corresponde con un área de conocimiento y su vocabulario especializado. Un *dominio vertical* es la especialidad de un sistema informático: puede ser un sistema bancario, un sistema de reservas aéreas o un sistema de inventario, por ejemplo. Un *dominio horizontal*, en cambio, es un área de conocimiento aplicable a muchos dominios verticales como gestión de base de datos, interfaces gráficas o sistemas de flujos de trabajo. La ingeniería de dominios trata de capturar el conocimiento de un dominio —sea éste horizontal o vertical— explícitamente para hacerlo reusable y no reinventarlo cada vez. El objetivo es aplicar los elementos reutilizables para construir sistemas informáticos en menos tiempo, con menor coste y con mayor calidad.

“La ingeniería de dominios es la actividad de obtener, organizar y guardar experiencia pasada en la construcción de sistemas o partes de sistemas en un dominio particular en forma de bienes reusables (por ejemplo, productos de trabajo reusables), así como dar guías adecuadas sobre cómo reusar esos bienes (por ejemplo, recolección, calificación, diseminación, adaptación, ensamblado, ...) cuando se construyen nuevos sistemas.” [Czarnecki & Eisenecker, 2000; p. 20].

A los bienes reusables se los denomina a menudo *activos reusables* o *reusable assets*. La ingeniería de dominios se diferencia de la ingeniería del *software* tradicional en que la primera trata de construir familias de sistemas reusables dentro de un dominio, mientras que la segunda intenta construir sistemas informáticos únicos. La ingeniería de dominios se divide en análisis de dominios, diseño de dominios e implementación de dominios.

El *análisis de dominios* intenta definir un dominio identificando su ámbito, obteniendo la información relevante e integrándola en un *modelo del dominio*. El modelo identifica aquellos aspectos comunes al dominio, y propiedades variables de sistemas informáticos que formen parte del dominio, junto con dependencias entre ellas. El modelo de dominio incluye el ámbito del dominio, su vocabulario, conceptos asociados al dominio —descritos mediante términos informáticos: diagramas de estado, objetos de interacción, flujos de datos, documentación— y modelos de capacidades o *feature models*. Estos últimos incluyen información de elementos configurables dentro del dominio, combinaciones que tienen sentido y cuáles son las más apropiadas en condiciones conocidas. Es decir, recogen los puntos de variabilidad configurables por los sistemas informáticos del dominio. Este aspecto es uno de los rasgos más característicos de la ingeniería de dominios.

El *diseño de dominios* trata de desarrollar una arquitectura para una familia de sistemas informáticos dentro del dominio, e imaginar vías de llevar a la práctica cada uno de esos sistemas mediante el *plan de producción*. El plan describe cómo se construye cada sistema final. Según [Cohen, 1999] se puede hacer mediante:

- *Ensamblado manual*. La arquitectura y componentes están acompañados de una guía de desarrollo, y los sistemas son ensamblados a partir de los elementos reusables manualmente.
- *Soporte para el ensamblado automático*. Se usan herramientas que asisten en el ensamblado de elementos reusables, incluyendo buscadores de componentes y generadores que automatizan determinados aspectos del desarrollo de aplicaciones.
- *Ensamblado automático*. Un conjunto de herramientas soporta el proceso de construir una aplicación a partir de una orden o petición de construcción. Toda la aplicación es generada acorde a ese orden, con la excepción de aquellas partes que requieran desarrollo a medida.

La *implementación de dominios* es el resultado de aplicar las guías dadas en el diseño de dominios. Requiere implementar la arquitectura, los componentes y otros elementos *software* reusables, quizás con la ayuda de herramientas de soporte de ensamblado o de ensamblado automático. En el mismo contexto, la *ingeniería de aplicaciones* es el proceso de construir sistemas informáticos basados en los resultados de la ingeniería de dominios [Czarnecki & Eisenecker, 2000; p. 30].

2.4.2 Familias de sistemas y líneas de productos

Muy relacionado con la ingeniería de dominios están las familias de sistemas y líneas de productos. Ambas usan las técnicas de la ingeniería de dominios para identificar aspectos comunes que puedan ser reutilizados. Las familias de sistemas comparten un número importante de activos comunes, reusables en cada sistema. Las líneas de productos pueden verse como familias de sistemas centradas en servir un mercado particular. Las prácticas de construcción de líneas de productos reconocen la importancia de la gestión que ha de llevar a cabo una organización que pretenda crear líneas de productos. La organización es responsable de facilitar un clima y unos objetivos donde se fomente la creación de líneas de productos y no la construcción de productos independientes que reusan algunas partes de otros productos. Las prácticas de líneas de productos se centran en tres procesos interrelacionados. (1) La ingeniería de dominios para el desarrollo de los *activos básicos* o *core assets*. (2) La ingeniería de aplicaciones para el desarrollo de productos usando los activos básicos. Y (3) la gestión de la organización para fomentar los dos primeros procesos [Clements & Northrop, 2002; pp. 29-30]. Los tres procesos son iterativos y se retroalimentan. Así, mejores conocimientos en la construcción de productos pueden dar lugar a mejoras en los activos básicos o viceversa. Lo mismo ocurre con los aspectos organizativos, mejoras en los procesos de creación de activos básicos o productos pueden ser asumidos por la organización para fomentarlos y apoyarlos.

Los aspectos tecnológicos relacionados con el análisis y el diseño de líneas de productos son cubiertos fundamentalmente por la ingeniería de dominios, que brevemente se revisó en el apartado 2.4.1. El ámbito de los productos que pueden crearse, incluyendo características como formas de variación, elementos comunes, funcionalidad, o información sobre prestaciones, debe ser definido con mucho cuidado. Si es demasiado amplio, entonces se corre el riesgo de que los activos básicos requieran demasiadas modificaciones y no sean útiles. Si es demasiado estrecho, los activos básicos pueden no ser lo suficientemente flexibles y perder también su utilidad. Los activos básicos incluyen la arquitectura compartida por la línea de productos y sus puntos de variabilidad, componentes compartidos, especificaciones de requisitos comunes, casos de prueba y documentación. Un aspecto importante de cada activo básico es que debe incorporar un *proceso anexo* o *attached process* que especifica cómo debe usarse el activo en el desarrollo de productos, incluyendo las herramientas apropiadas para su manipulación. Su conjunto forma el *plan de producción* de la línea de productos, que incorpora, además, la secuenciación de actividades y métricas de uso. Las guías de implementación pueden obtener beneficios de las incipientes técnicas de la ingeniería basada en componentes, pero es posible usar otras técnicas más tradicionales. Una técnica interesante es la *programación generativa*, que intenta obtener procesos de ensamblado completamente automáticos a través del uso de *generadores* [Czarnecki & Eisenecker, 2000; pp. 5-13]. Los generadores poseen cierta variabilidad, permitiendo parametrizar algunos aspectos de los elementos *software* generados para adaptarlos a circunstancias particulares. Los elementos *software* ensamblados mediante generadores no tienen porqué ser componentes en el sentido de la definición (Def. 2-11), sino simplemente *software* adaptable automáticamente a ciertos requisitos. Por ejemplo, es posible construir generadores usando plantillas o *templates* de C++, *scripts*, o traductores. El conocimiento adquirido durante el proceso de construcción de los productos finales a su vez suele modificar el ámbito de parte de los activos básicos, que deben ser actualizados. Igualmente, el plan de producción puede verse alterado para acomodar los cambios. Es responsabilidad de la organización que fomenta la aplicación de las ideas de las líneas de productos asegurar que esas operaciones se realizan con el apoyo adecuado.

2.5 Principales modelos de componentes actuales

El paso siguiente a las líneas de productos únicos, que se está empezando a producir actualmente, son las líneas de componentes únicos. Resulta posible con la aplicación simultánea de las prácticas de líneas de productos y las técnicas que permiten construir componentes únicos. Los modelos de componentes son el elemento básico que facilita la construcción de componentes, y son una ayuda para organizar la arquitectura final de las aplicaciones. Con el tiempo ha ido formándose la idea de que los componentes se encuentran implícitamente en la base de muchos desarrollos pasados. Bibliotecas de rutinas o sistemas de bases de datos

pueden verse como componentes de grano grueso [Szyperski, 1998; p. 30]. También es posible considerar a los sistemas operativos como una implementación de un modelo de componentes primitivo donde los componentes son aplicaciones completas [Szyperski, 1998; p. 11]. Las aplicaciones siguen el estándar definido por el sistema operativo y éste se encarga de abstraer y resolver multitud de problemas de bajo nivel, simplificando el desarrollo de aplicaciones. Ahora bien, las aplicaciones vistas como componentes tienen ciertas deficiencias importantes: granularidad demasiado elevada, falta de soporte para la composición de aplicaciones, y falta de estándares específicos para los dominios de aplicación, puesto que un sistema operativo es una herramienta demasiado general [Wienreich & Sametinger, 2001; p. 35].

Los modelos de componentes actuales están bastante evolucionados. Van mucho más allá de la descripción del modelo de componentes básico, incluyendo distribución, configuración, empaquetado y despliegue, modos de ejecución, servicios adicionales de transaccionalidad, seguridad, persistencia, o bibliotecas de ayuda para la programación con componentes. La mayor parte de estos añadidos corresponden con implementaciones particulares de funcionalidad útil. Por ejemplo, un modelo de componentes puede o no proporcionar persistencia, seguridad o distribución transparente. De hecho, a veces esa funcionalidad ha sido añadida con el tiempo.

Los modelos de componentes que se repasan a continuación son similares en espíritu, aunque tienen diferencias importantes de implementación. La similitud entre ellos es debida en parte a que han servido de inspiración unos a otros durante varias iteraciones de los modelos. Para realzar los aspectos comunes, y ponerlos en contraste con la estructura de componentes básica, las figuras seguirán la misma codificación de color que la Figura 2-1, y el esquema mostrado será similar en todos ellos, incluyendo distribución, que es inherente a CORBA, EJB y a los servicios *web*.

2.5.1 COM, DCOM y COM+

El modelo de componentes básico de Microsoft es COM (*Component Object Model*) [Microsoft, 2003c]. Es independiente del lenguaje y de plataforma, siendo usado muy mayoritariamente en los sistemas operativos de Microsoft. En COM, únicamente se describe la estructura binaria de las interfaces de los componentes. Nada se dice acerca de qué implementación se usa para dar vida a los componentes. Un componente COM puede estar implementado por procedimientos o por objetos. Cada interfaz en COM lista las operaciones que contiene y mantiene una referencia al objeto o datos asociados a esa interfaz, que es pasada en cada llamada a las operaciones, simulando un modelo orientado a objetos. Un componente COM puede contener múltiples interfaces. Las interfaces en COM se escriben con MIDL (*Microsoft Interface Definition Language*).

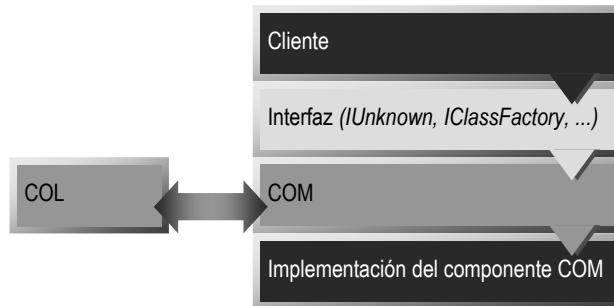


Figura 2-5 Llamada de un cliente a un componente COM.

El modelo básico de un componente COM es el mostrado en la Figura 2-5. Todos los componentes COM incluyen una interfaz básica llamada `IUnknown`, que se encarga de buscar el conjunto de interfaces particulares que contiene un componente COM, y controla la gestión básica de memoria mediante cuenta de referencias. Para crear componentes COM se usa un servidor COM. La interfaz de creación es

`IClassFactory`, que accede a la biblioteca de componentes o COL (*Component Object Library*) para crear la imagen en tiempo de ejecución de los componentes.

La variante DCOM (*Distributed COM*) [Microsoft, 2003d] es usada para implementar llamadas distribuidas. La llamada distribuida incluye comprobaciones de un proveedor de seguridad o *security provider* externo configurable que autoriza la llamada remota. La Figura 2-6 muestra cómo se realiza la llamada. Es responsabilidad del modelo de componentes —en este caso DCOM— dar una implementación adecuada de las llamadas remotas para evitar que el cliente dependa de esa información.

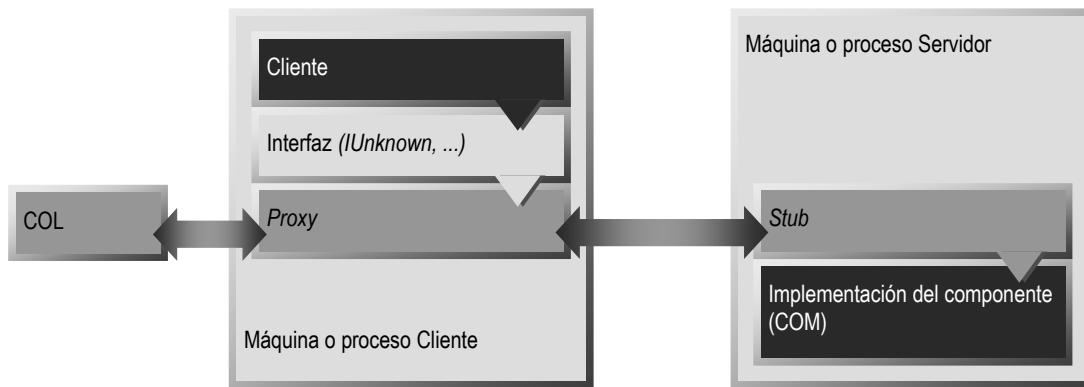


Figura 2-6 Comunicación entre componentes COM usando DCOM.

La invocación remota de un componente se realiza con dos segmentos de código intermedios: el *proxy* o delegado, que envía los parámetros por la red, y el *stub* que los interpreta y los transforma en una invocación al objeto remoto. Para devolver los valores de retorno se invierte el proceso.

El cliente obtiene una referencia al componente COM de igual modo que antes, usando la interfaz `IClassFactory`, accediendo a la biblioteca de objetos COM, esta vez remotamente. A través de la interfaz `IUnknown` el cliente obtiene una referencia a la interfaz concreta del componente que desee usar. La llamada del cliente es interceptada por un código intermedio o *proxy* que delega la llamada en la implementación remota, reenviándola a otro código intermedio o *stub* situado en el servidor. El *stub* es el encargado de reconstruir la llamada final al componente remoto. Para soportar comportamientos de componentes más complejos, COM define nuevas interfaces sobre la misma tecnología de componentes básicas. Por ejemplo, la interfaz `IPersistentObject` es usada para guardar en un dispositivo de almacenamiento secundario el estado asociado a una imagen de un componente. La interfaz `IDataObject` permite a varios componentes comunicarse a través del portapapeles y mediante arrastre de unos componentes sobre otros. La interfaz `IDispatch` sirve para invocar operaciones compuestas dinámicamente, siendo un intento de dar un limitado soporte en tiempo de ejecución. Las interfaces de salida u *outgoing interfaces* conectan varios componentes entre sí. Otras interfaces COM más complejas definen los componentes OLE (*Object Linking and Embedding*) para construir documentos compuestos o *compound documents*, y los controles ActiveX, pensados para ser distribuibles por Internet.

Un producto separado, MTS (*Microsoft Transaction Server*) [Microsoft, 2003f], aporta escalabilidad, transaccionalidad, seguridad y control del entorno de ejecución a los componentes COM, usando DCOM como base. MTS gestiona la lista de conexiones abiertas con bases de datos, el uso de múltiples hilos de ejecución, y el ciclo de vida de los componentes. Usando MTS y COM, el cliente llama a MTS para obtener un contexto de ejecución del componente COM y ejecutar el componente dentro de él. El contexto indica propiedades de seguridad, de transaccionalidad y modo de ejecución. La seguridad suele ser gestionada por el sistema operativo, pero otras opciones son también posibles. La transaccionalidad es gestionada automáticamente por MTS. Se puede también elegir ejecutar los componentes en un solo hilo de ejecución o en múltiples.

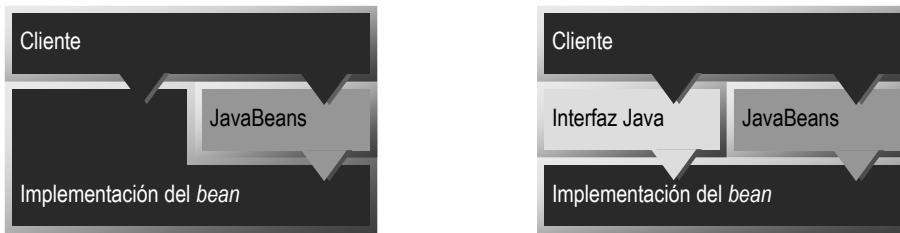
Con el tiempo COM ha evolucionado a COM+ [Microsoft, 2003e], una versión extendida del modelo básico COM que también integra MTS para dar una visión más unificada, e incluye otras bibliotecas de utilidad. Los componentes de COM+ son más adaptables a diferentes entornos, como lenguajes de *scripting* o Java. Incluyen más información o metadatos no presentes originalmente en COM para facilitar la integración con lenguajes y herramientas de bajo nivel como depuradores. Los objetos en COM+ usan un recolector de basura, en vez del sistema de cuenta de referencias de COM. Los objetos COM+ son compatibles con los objetos COM creando adaptadores COM+ para ellos. Las llamadas remotas se siguen gestionando con DCOM. COM+ incorpora, además, múltiples bibliotecas de utilidad para simplificar el desarrollo con componentes COM y COM+. Recientemente, Microsoft ha unificado aún más la representación interna de los objetos COM+ en el CLR (*Common Language Runtime*) que define un modelo de objetos básico compatible con múltiples lenguajes: C++, Java, C# y VisualBasic entre otros, especificando un único soporte en tiempo de ejecución para todos ellos. También es usado para implementar componentes COM y es accesible desde COM+. La versión en proceso de estandarización se llama CLI (*Common Language Infrastructure*) [ECMA, 2002b].

2.5.2 JavaBeans y EJB

JavaBeans [Sun, 1997] es una especificación de Sun Microsystems que describe un modelo de componentes para Java. Un componente se denomina un *bean*, si bien no tienen nada que ver con los *beans* de los EJB que se verán posteriormente. Los *beans* están pensados para ser manipulables por un entorno gráfico de desarrollo. Soportan dos modos de ejecución, un modo interactivo usado durante el desarrollo y el modo habitual de ejecución. Se espera que sirvan para crear controles de tamaño pequeño o medio [Sun, 1997; p. 7]. Los componentes de JavaBeans tienen las siguientes características:

- *Introspección*. Los componentes describen sus propiedades, eventos y métodos para que puedan ser inspeccionados por una herramienta externa de ensamblaje.
- *Configuración*. La herramienta externa de ensamblaje es capaz de configurar un componente dando ciertos valores a sus propiedades.
- *Eventos*. Indican fuentes de notificación de eventos para ser consumidos por otros componentes. Cada evento enviado es un objeto. Los *beans* incluyen emisores y receptores de eventos.
- *Propiedades*. Identifican el estado que almacenará el componente e incluye siempre métodos de acceso y escritura. Los cambios de propiedades pueden lanzar eventos, y es posible controlar y limitar el intento de cambio de una propiedad.
- *Persistencia*. Es usada para guardar instancias de componentes configuradas.
- *Métodos*. Representan otras operaciones realizadas por los componentes. Son métodos Java exportados que pueden ser llamados desde un entorno de *scripting* externo.

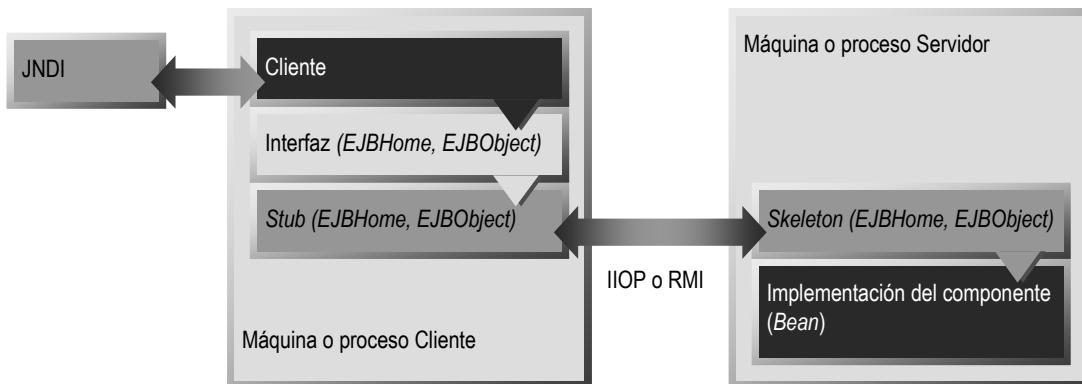
El modelo de JavaBeans se fundamenta en la definición de propiedades, métodos y eventos. Los componentes se ejecutan en el mismo espacio de direcciones que su aplicación contenedora. Es posible construir *beans* distribuidos usando bibliotecas RMI o CORBA, si bien el modelo de componentes no da facilidades específicas para ello. El tipo asociado a un *bean* es identificado por una clase o una interfaz Java, por lo que no siempre existe el concepto de interfaz independiente de implementación. Aun cuando un *bean* esté implementado únicamente con una clase, el modelo JavaBeans espera que se obtengan referencias a los objetos e información acerca de su tipo usando métodos estándar definidos en el modelo. Fomenta así la separación conceptual entre objetos y componentes y facilita extensiones futuras del modelo, a costa de añadir un modo alternativo preferente de creación de objetos sobre el estándar existente en el lenguaje de base. Por tanto, un cliente puede acceder a un componente usando directamente las facilidades del lenguaje sin pasar por el modelo de componentes, usando el modelo de componentes, o, más habitualmente, usando ambas opciones.

**Figura 2-7** Posibles beans definidos por JavaBeans.

Los *beans* pueden implementarse directamente con clases Java o con clases e interfaces. El modelo de JavaBeans es complementario a las definiciones del lenguaje incluyendo convenciones para acceder a la funcionalidad de componentes.

Existe una herramienta de ensamblaje que conecta unos *beans* con otros. La herramienta usa archivos JAR (*Java ARchive*) para empaquetar las clases que forman un JavaBean, instancias del JavaBean configuradas, ficheros de ayuda, localización, iconos y otros ficheros extra anexos.

Los EJB (*Enterprise Java Beans*) [Sun, 2001] son otra especificación de componentes, más robusta, y orientada a aplicaciones empresariales. Comparten únicamente el nombre *bean* con la especificación de JavaBeans. Un EJB es un componente de servidor reusable que sus clientes pueden invocar remotamente. Los EJB sí demarcán claramente la separación entre interfaz e implementación, división que es más difusa en los JavaBeans. La imagen en tiempo de ejecución de los componentes se implementa con objetos Java. El procedimiento exacto de invocación remota usualmente es IIOP (*Internet Inter-ORB Protocol*) de CORBA, o RMI de Java. Un componente EJB puede tener múltiples interfaces, al igual que un objeto Java. La Figura 2-8 muestra el proceso y los elementos participantes.

**Figura 2-8** Comunicación remota entre EJB y clientes.

La invocación remota de un componente se realiza a través de dos fragmentos de código intermedios: el *stub*, que envía los parámetros por la red, y el *skeleton* que los interpreta y los transforma en una invocación al objeto remoto. Para devolver los valores de retorno se invierte el proceso.

El cliente localiza al componente remoto usando el servicio de directorio JNDI (*Java Naming and Directory Interface*). Luego crea el objeto asociado al componente usando la interfaz EJBHome. A partir de entonces ya se pueden enviar mensajes remotos, que pasan a través del *stub* local al esqueleto remoto y finalmente a la implementación del componente. Cada EJB está formado por un conjunto de elementos:

- **Interfaz EJBHome.** Es la interfaz percibida por el cliente que cuenta con los mensajes básicos del ciclo de vida de los objetos del componente: búsqueda, creación y eliminación de objetos.
- **Interfaz EJBObject.** Es la interfaz percibida por el cliente de los métodos que implementan la funcionalidad del componente.

- *Interfaz EJBLocalHome*. Es una interfaz local opcional de ciclo de vida, válida para llamadas entre beans dentro de una misma máquina virtual de Java, sin el coste añadido de una llamada remota.
- *Interfaz EJBLocalObject*. Es otra interfaz local opcional de la funcionalidad del componente. Es válida para llamadas entre beans dentro de una misma máquina virtual de Java, sin el coste añadido de una llamada remota.
- *Enterprise bean*. Es el objeto que implementa la funcionalidad del componente.
- *Stub y Skeleton de las interfaces EJBHome y EJBObject*. Adaptadores que se encargan del envío, serialización y recepción de peticiones remotas.
- *Descriptor de despliegue o deployment descriptor*. Contiene información adicional en XML (*eXtended Markup Language*) [W3C, 2000] para configurar el bean en el servidor.

Un EJB debe ser desplegado en un servidor que contenga un *contenedor* de EJB. El contenedor controla la activación y comportamiento en tiempo de ejecución del componente. El contenedor es responsable de la transaccionalidad, seguridad, gestión de conexiones, gestión de estado y multitarea. La idea del contenedor de EJB está inspirada en el uso de MTS con componentes COM. El procedimiento de despliegue incluye información adicional para realizar el proceso y la configuración final de cada bean. Para asegurar el funcionamiento correcto, el contenedor bloquea ciertas posibles llamadas del componente: acceso al sistema de ficheros, a sockets, al entorno gráfico, a la gestión de hilos de ejecución, al cargador de clases y a bibliotecas nativas. El contenedor maneja dos tipos de componentes: beans de sesión y beans de entidad.

- Los beans de sesión pueden almacenar estado en base de datos y participar en transacciones. Si el estado se mantiene entre peticiones, el bean de sesión es exclusivo de un único cliente (*stateful session bean*). Si el estado no se mantiene, puede ser usado por múltiples clientes (*stateless session bean*). Ambos tipos de beans de sesión son capaces de usar transacciones, marcando durante su despliegue atributos de transaccionalidad. Por ejemplo, si soporta o no transacciones, o si un método ha de ejecutarse siempre en una nueva transacción dejando la transacción en curso en suspenso mientras tanto. El contenedor puede gestionar las transacciones haciendo uso de los atributos. O para obtener mayor flexibilidad, es posible gestionarlas directamente desde el bean haciendo llamadas a la biblioteca de transacciones.
- Los beans de entidad representan datos en un almacenamiento secundario persistente, como una base de datos. Siempre son compartidos por múltiples clientes. Puesto que los datos del bean pueden cambiar, se hace necesario un procedimiento de sincronización con el almacenamiento persistente. El sistema de transacciones del contenedor se encarga de invocar los procedimientos de sincronización, que cargan o almacenan la información del bean desde o hacia el almacenamiento persistente. Es posible que el contenedor gestione la persistencia del bean, si bien los mecanismos usados para hacer persistente cada bean no son parte de la especificación. La persistencia también puede ser gestionada directamente por el bean, programándose la lógica requerida en ese caso dentro del bean.

El contenedor aplica la transaccionalidad, la seguridad y la persistencia usando los mecanismos de reflexión de la máquina virtual Java, y apoyándose en la generación apropiada del código intermedio de transmisión y recepción remota. La especificación de EJB es larga y compleja. Con el tiempo distintos fabricantes han ido implementándola con mayor extensión. La diferencia fundamental entre JavaBeans y EJB es que los últimos están pensados para ser usados en un entorno más exigente y más complejo. La implementación y propósito de ambos son completamente distintos. Los JavaBeans, por el contrario, son más sencillos de usar incluso en un entorno distribuido, a costa de no contar con las funcionalidades más robustas de los EJB.

2.5.3 CORBA y CCM

Desde sus orígenes el modelo de objetos CORBA (*Common Object Request Broker Architecture*) es inherentemente un modelo de objetos distribuido, multilenguaje y multiplataforma. Cuando un cliente accede a un objeto CORBA, no es consciente de en qué máquina o proceso se encuentra ni en qué lenguaje ha sido implementado. Su última versión CORBA 3.0 [OMG, 2002b] incluye también el modelo de componentes CCM (*CORBA Component Model*) [OMG, 2002a], que se construye sobre el modelo de objetos. La imagen de

un componente CCM en tiempo de ejecución está compuesta por una referencia a un componente y un objeto CORBA subyacente. Debido a los objetivos de independencia de máquina, lenguaje y plataforma de CORBA, los objetos CORBA muchas veces se han usado directamente como componentes, puesto que fomentan la separación crítica entre interfaz e implementación. La interfaz se describe independientemente de la plataforma con IDL (*Interface Definition Language*) y la implementación puede realizarse en múltiples lenguajes. Para cada uno de ellos existe una especificación para traducir el IDL a cada lenguaje. Un objeto CORBA puede tener múltiples interfaces. La organización básica de CORBA se presenta en la Figura 2-9.

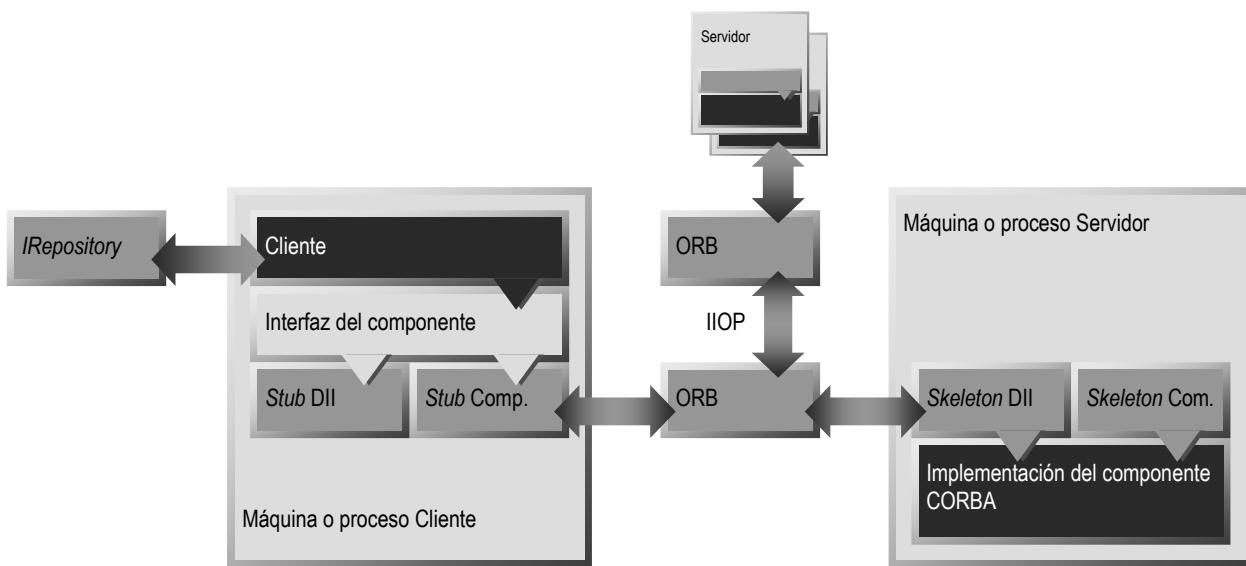


Figura 2-9 Comunicación entre objetos CORBA.

La invocación remota de un componente usa dos fragmentos de código intermedios: el *stub*, que envía los parámetros por la red a través del ORB, y el *skeleton* que los interpreta y los transforma en una invocación al objeto remoto. Para devolver los valores de retorno se usa el camino inverso.

El cliente localiza el objeto remoto que representa un componente usando el repositorio de interfaces o IR (*Interface Repository*). El cliente realiza una llamada a un componente CORBA localmente. La interfaz del componente incluye una interfaz de invocación dinámica DII (*Dynamic Invocation Interface*) —que permite enviar cualquier mensaje al componente— y un *stub* usado para enviar los mensajes especificados en la interfaz del componente. La primera interfaz es compartida por todos los componentes, la segunda es particular para cada uno de ellos. La invocación es pasada al ORB (*Object Request Broker*) que la canaliza hasta el componente de destino o servidor. El esqueleto o *skeleton* del servidor se encarga de recuperar la llamada —mediante DII o mediante el esqueleto particular del componente— y efectúa la invocación final en el componente servidor, que en tiempo de ejecución es siempre un objeto. Puesto que CORBA es una especificación, pueden haber diferentes ORB de distintos proveedores. La Figura 2-9 muestra también un segundo ORB —quizá de otro proveedor— en el que están registrados otros componentes. Si el componente servidor se encuentra gestionado por un ORB diferente del que recibe originalmente la petición, entonces ambos ORB negocian —usualmente mediante IIOP (*Internet Inter-ORB Protocol*)— el traslado de la petición entre ambos. Los valores de retorno de la invocación siguen en todo caso el camino inverso hasta llegar otra vez al cliente original.

El añadido más significativo de la última versión de CORBA es el CCM, que es un superconjunto de la especificación de EJB. El CCM facilita la construcción de aplicaciones CORBA más complejas. Resuelve un número importante de anomalías y limitaciones de las versiones anteriores, descritas por ejemplo en [Wang, Schmidt & O’Ryan, 2001; pp. 559-560]. Cada componente puede tener múltiples interfaces, no necesariamente relacionadas por herencia. Los componentes son manejados mediante referencias, que esconden la localización real del componente, el objeto CORBA subyacente y las interfaces que cumplen.

Junto a ellos se encuentran los conectores que guardan referencias a las interfaces de otros componentes. Existen dos tipos de componentes, los *básicos* y los *extendidos*. Los primeros contienen únicamente atributos, y están pensados para ser similares a los componentes EJB 1.1. Los componentes extendidos disponen además de cuatro funcionalidades o *surface features* adicionales, llamadas *puertos* o *ports*, que son accesibles a través de la interfaz de navegación o *navigation interface*.

- *Atributos.* Determinan aspectos de configuración de un componente. Un atributo puede ser parte del estado temporal o persistente de un componente.
- *Facetas.* Son interfaces proporcionadas por el componente, que dan diferentes vistas a sus clientes.
- *Receptáculos.* Identifican las interfaces requeridas por componentes externos.
- *Fuentes y sumideros de eventos.* Facilitan monitorizar eventos asíncronos. Los eventos fuente sirven para notificar a otros componentes. Los eventos sumidero se usan para recibir notificaciones externas.

Para crear y eliminar instancias de los componentes, cada componente tiene una interfaz *home*, cuya referencia puede encontrarse a través de la interfaz *HomeFinder*. Existen otras interfaces estándar que ha de cumplir un componente, para acceder a otros servicios y permitir la instalación y registro inicial de los componentes en el repositorio. Es posible supervisar la construcción de un componente con un componente especial o *configurador*, que ayuda a crear las conexiones con otros componentes de la manera más adecuada. Los componentes pueden tener también estado persistente, para almacenarlos en algún dispositivo externo persistente, como una base de datos.

Los componentes en el CCM se empaquetan, instalan y registran en *servidores de componentes*. El servidor de componentes instancia *contenedores* para encapsular el comportamiento en tiempo de ejecución de los componentes registrados, permitir activarlos y desactivarlos, o enviar y recibir peticiones de servicios usuales: transacción, seguridad, persistencia y notificación. Los servicios usuales —junto con otros menos comunes adicionales— son prestados en última instancia por el ORB, pues es el que tiene la capacidad de interceptar las llamadas en el momento adecuado para poder aplicarlos. El contenedor se comunica con el ORB para emplear los servicios apropiados. Al igual que con EJB, el contenedor puede gestionar las transacciones o delegar la responsabilidad en el componente. Del mismo modo, es posible gestionar la persistencia con el contenedor o ser el componente quien la lleve a cabo. El contenedor también maneja el tiempo de vida de los componentes, existiendo cuatro políticas que indican la duración de las imágenes de los componentes en memoria: método, sesión, componente y contenedor. Hay cuatro categorías de componentes:

- *Servicio.* No tiene estado, ni identidad, ni persistencia. La transaccionalidad puede ser gestionada por el contenedor o por el componente. El tiempo de vida es siempre de método. Es equivalente a un *bean* sin estado de EJB.
- *Sesión.* Tiene estado temporal e identidad, pero no persistencia. La transaccionalidad puede ser gestionada por el contenedor o por el componente. El tiempo de vida puede ser cualquiera. Es equivalente a un *bean* con estado de EJB. Si soporta transacciones, es equivalente a un componente MTS.
- *Proceso.* Tiene estado e identidad persistente, pero no es visible a los clientes, tan sólo opcionalmente a través de métodos. La transaccionalidad puede ser gestionada por el contenedor o por el componente. La persistencia la gestiona el componente. El tiempo de vida puede ser cualquiera.
- *Entidad.* Tiene estado e identidad persistente, visible a los clientes. La transaccionalidad y la persistencia son gestionables por el contenedor o por el componente. El tiempo de vida también puede ser cualquiera. Es equivalente a un *bean* de entidad de EJB.

La configuración de componentes de CCM tiene un cierto parecido con la especificación de EJB que se repasa en el apartado 2.5.2. La especificación de EJB es considerada un subconjunto de la especificación de CCM [Baker, 2000; p. 614] para facilitar la interoperabilidad y evitar la confrontación entre ambos modelos de componentes en el mercado. Un EJB puede verse desde CCM como un componente CCM y a la inversa. La especificación de CORBA 3.0 es reciente. También es larga y compleja, existiendo todavía pocas

implementaciones que son generalmente parciales. El requisito mínimo para obtener la certificación de la OMG es cumplir con el núcleo de la especificación CORBA [OMG, 2002b] y una traducción a algún lenguaje.

2.5.4 Servicios web

Otra tendencia actual son los servicios *web* o *web services*, cuya organización básica es similar en espíritu a los modelos de componentes distribuidos ya estudiados. Los servicios *web* son un intento de recuperar simplicidad en los modelos de componentes distribuidos, aprovechando varios estándares que han ganado aceptación con la difusión de Internet y están públicamente disponibles en el *World Wide Web Consortium* (W3C) [W3C, 2003a]. Un servicio *web* es un componente que publica un conjunto de operaciones que definen un servicio. Los servicios *web* son independientes de plataforma y de lenguaje. El acceso a las operaciones de un servicio *web* usa el protocolo SOAP (*Simple Object Access Protocol*) [W3C, 2001b], [W3C, 2003b], que básicamente implementa una llamada a un procedimiento remoto. SOAP especifica en un documento XML la información que se transmite por la red para invocar la operación remota, los parámetros y los valores de retorno. El protocolo de transporte es habitualmente HTTP o HTTPS seguros, aunque es posible usar también otros protocolos como FTP o SMTP.

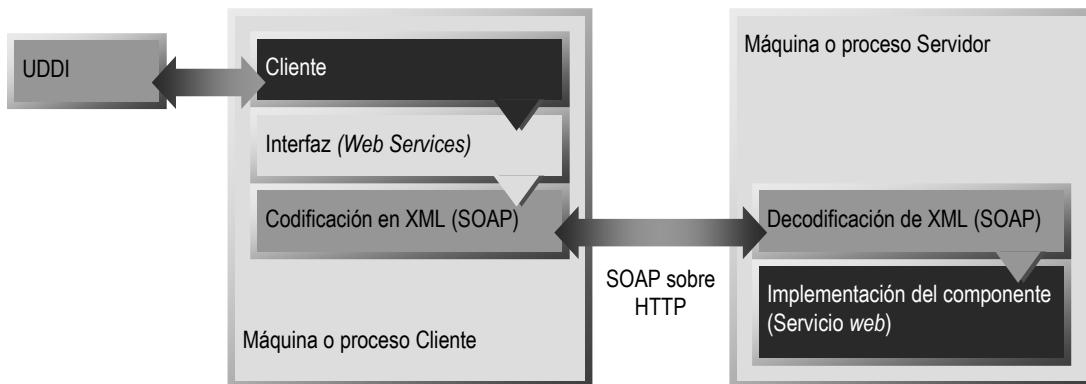


Figura 2-10 Comunicación remota entre servicios web y clientes.

La invocación remota de un componente codifica los el mensaje de envío con la operación a ejecutar y sus parámetros en un documento XML usando SOAP. El servidor lee la información del documento XML y los transforma en una invocación al componente remoto. Para devolver los valores de retorno se invierte el proceso, usándose también otro documento XML.

Un cliente localiza servicios *web* con un servicio de directorio, que usualmente es UDDI (*Universal Description Discovery and Integration*) [OASIS, 2003]. El resultado es un documento descrito en el lenguaje WSDL (*Web Services Description Language*) [W3C, 2001a], que almacena la información del protocolo SOAP necesaria para la ejecución de las operaciones del servicio buscado. Un código intermedio traduce la llamada del cliente al servicio *web* a un documento XML, que es enviado al servidor y decodificado para efectuar la llamada remota. Aunque no estrictamente necesario, es común que el acceso a un servicio *web* se haga a través de un servidor *web*. Esta configuración aprovecha directamente la infraestructura de Internet existente para soportar la construcción de componentes remotos y reducir la complejidad total.

2.6 Procesos y productos

A lo largo del apartado 2.5 *Principales modelos de componentes actuales* se observa una clara tendencia a incorporar nueva funcionalidad a los modelos de componentes bien en forma de servicios, bien como elementos implícitos a alguna especificación. Por ejemplo, CORBA y EJB incorporan distribución desde sus orígenes. En cambio COM y JavaBeans no, fue la extensión a DCOM quien trajo consigo la asimilación de esa funcionalidad para COM y la biblioteca RMI para JavaBeans. Otro ejemplo interesante es

el soporte de transaccionalidad y un entorno de ejecución de componentes. MTS es originalmente un producto externo, que sigue a grandes rasgos la arquitectura del producto CICS TS (*Customer Information Control System Transaction Server*) de IBM [IBM, 1999]. Luego la funcionalidad de MTS es incorporada dentro de COM+. Posteriormente, la arquitectura de EJB se inspira en el modelo de componentes COM con MTS. Y recientemente, el CCM incorpora el modelo de EJB como un subconjunto de su especificación. El objetivo perseguido es realizar la configuración de esas funcionalidades adicionales —tipo de seguridad, modo de ejecución o características de transaccionalidad— y simplificar el diseño y construcción de los componentes en sí. Estos requisitos a su vez demandan diseños más flexibles, que faciliten añadir o eliminar las funcionalidades adicionales sin tener que modificar el código fuente de los componentes. Es decir, parte de la funcionalidad de los componentes se factoriza, combinándose o ensamblándose luego mediante configuración y/o generación de código automática o manual. El camino recorrido en el apartado anterior empieza a dibujar una división entre los productos —los componentes y la funcionalidad a combinar— y los procesos de combinación —abstraídos en los configuradores y los generadores de código— y ejecutados en los contenedores. Pero, otra vez, la tendencia aún no se ha convertido en una entera realidad. La complejidad de las herramientas usadas, de los modelos de componentes, y la inmadurez de los productos COTS y su mercado, vista en el apartado 2.3, limitan la evolución de la tendencia. ¿Por qué pervive la sensación de dificultad? Quizá una posible respuesta esté en la identificación más precisa de qué debe ser un proceso y un producto. En conocer cuáles son sus propiedades deseables.

Si un producto lleva a cabo su función, entonces debiera perder relevancia qué proceso se usa para construirlo. Esconder los procesos utilizados permite ocultar complejidad que de otro modo estaría presente. Abre la posibilidad de poder reemplazar un producto por otro equivalente con menor esfuerzo y coste. Es una situación tan natural en otras disciplinas que generalmente no se repara en ello. Cuando, por ejemplo, se compra cualquier producto en unos grandes almacenes, muy raramente se presta atención al proceso que se ha usado para obtener el producto comprado, aun cuando el proceso haya sido de considerable importancia para ofrecer ese producto más competitivamente.

La posibilidad de reemplazar un producto *software* por otro es una de las claves de la reusabilidad y la reducción de costes de diseño, construcción y mantenimiento. Implícitamente se encuentra en la definición de componente (Def. 2-11). Existen ciertas características que han de cumplir dos productos genéricos supuestamente equivalentes para que el cambio de un producto por otro sea posible. Estas características son sus especificaciones funcionales y su comportamiento dinámico en tiempo de ejecución.

Un lenguaje de programación puede visualizarse también como un proceso³ que describe un conjunto de reglas para construir programas. Los programas, sean éstos completos o parciales, son los productos resultantes del proceso que se ha usado para generarlos. Estos productos han sido creados aplicando las reglas definidas en el proceso. Las reglas confieren ciertas propiedades a los productos resultantes, como por ejemplo los mecanismos de llamada a subrutina, su estructura modular —si es que tienen— o si están orientados a objetos. Esas propiedades son independientes de la función que ha de realizar el producto.

Sin embargo, el tipo de proceso —el lenguaje usado— continúa siendo muy relevante para decidir si un producto es válido o si puede combinarse con otros para constituir un producto más complejo. Es más, dado que resulta extremadamente esquivo definir las reglas que han de cumplir bien programas completos bien partes de ellos para poder reemplazarse, es necesario recurrir a la información sobre el proceso de construcción de los mismos, es decir, al código fuente. La barrera del lenguaje es usada por COM y CORBA precisamente para limitar de alguna manera esa dependencia, pero se trata únicamente de un primer paso.

Se pueden distinguir varios niveles de abstracción sobre los que podría interesar tener elementos reemplazables, que corresponden con diferentes capas o niveles en un sistema informático. Si una capa da una abstracción adecuada, entonces es posible obviar los procesos que han hecho posible la construcción de ese nivel de abstracción, escondiendo su complejidad inherente. Por ejemplo, los registros y puertos de

³ Se usa el término *proceso* como sinónimo de lenguaje de programación para simplificar la discusión. El proceso de construcción de programas no sólo incluye al lenguaje, sino también técnicas de recompilación periódicas, gestores de ficheros fuente, o pruebas de regresión, por poner unos ejemplos. Pero es el lenguaje la parte que permanece visible cuando se intentan esconder los procesos usados para construir programas.

entrada/salida de un ordenador revelan un nivel de abstracción tal que permite obviar los procesos usados en la construcción de los microprocesadores o los dispositivos de entrada/salida: Es viable cambiar un procesador por otro de la misma familia compatible, como un Intel Pentium III por un Intel Pentium 4; o incluso de diferentes fabricantes, como un Intel Pentium 4 por un AMD Athlon. También es posible cambiar discos por otros que tengan la misma interfaz de comunicación como SCSI o ATA. Las subrutinas y bibliotecas poseen abstracciones adecuadas en algunos aspectos: los modos de llamada a subrutina y el enlace de bibliotecas están bien definidos. Pero no lo está su semántica o la gestión de los parámetros, por lo que el reemplazo se torna más difícil. Los modelos de componentes actuales facilitan la conexión de componentes, pero, al igual que con subrutinas y bibliotecas, la semántica de cada componente tampoco está bien definida. La pregunta ahora es ¿por qué resulta esquivo definir las reglas que han de cumplir las partes reemplazables?

2.7 Interdependencia

Una preocupación muy común en la práctica de la programación es aquella que pone en relieve que los fragmentos de código de un programa son a menudo poco reemplazables. Por reemplazo se entiende desde la simple realización de modificaciones puntuales en un fragmento de código a la sustitución de un fragmento o un subsistema completo por otro totalmente distinto. Puede observarse que el reemplazo es difícil porque no se puede asegurar que los cambios funcionen siempre correctamente. Es decir, existe una falta de adaptabilidad ante los cambios, que es precisamente lo que cada reemplazo en última instancia produce. La inflexibilidad es debida a que existe un vacío semántico acerca de cuál es la funcionalidad real del fragmento de código en proceso de reemplazo. Esta reflexión es aplicable enteramente al desarrollo con componentes.

El problema de fondo que complica la obtención de componentes reemplazables se puede expresar de forma más rigurosa como un problema de *interdependencia*. En este apartado estudiaremos más detenidamente este concepto, proponiendo una descripción formal del mismo. Un nuevo análisis que expone con mayor claridad y rigor las dependencias, su razón de ser, cómo afectan a la obtención de componentes reemplazables, cómo se relacionan con el principio de encapsulación, con la ruptura de encapsulación, y con sus formas de propagación. Tiene cierta relación con la *interferencia*, tradicional de la programación concurrente, paralela y distribuida. En esos ámbitos, se define *aserción* e *interferencia* como sigue [Andrews, 2000; pp. 63-66]:

- ❖ **Def. 2-19.** Una *aserción* caracteriza aquello que un hilo de ejecución asume que es verdad antes y después de cada sentencia.
- ❖ **Def. 2-20.** Un hilo de ejecución *interfiere* con otro si el primero ejecuta una asignación que invalida una aserción hecha en el segundo hilo de ejecución.

La interdependencia, en cambio, se centra en un único hilo de ejecución o *thread*. El concepto está relacionado con las *dependencias* de [Biddle & Temporo, 1996] introducidas informalmente para estudiar características cualitativas de lenguajes de programación que fomenten la reusabilidad. [Szyperski, 1998; pp. 68-69] identifica el mismo problema de interdependencia en el contexto de llamadas de retorno en presencia de la herencia. Este trabajo realiza un nuevo análisis con ayuda de la teoría de conjuntos.

2.7.1 Dependencias e interdependencia

Las siguientes definiciones caracterizan el concepto de interdependencia. La descripción se realiza de manera más formal para centrar ya en detalle la discusión sobre las dificultades de obtención de componentes reemplazables.

- ❖ **Def. 2-21.** Un elemento *software ES* está formado por un conjunto ordenado de sentencias σ_i escritas en algún lenguaje de programación, que representan la ejecución secuencial de instrucciones de máquina.

$$(2-1) \quad ES = \bigcup_{i=1}^n \sigma_i$$

- ❖ **Def. 2-22.** Un *bloque de sentencias* es una secuencia ordenada de sentencias consecutivas:

$$(2-2) \quad \sigma_{\text{bloque}} = \sigma_1 \ \sigma_2 \ \sigma_3$$

Se admite que una sentencia pueda ser vacía:

$$(2-3) \quad \sigma_1 \ \sigma_2 \ \sigma_3 \Rightarrow \sigma_1 \ \sigma_3 \text{ si } \sigma_2 = \emptyset$$

Un elemento *software* compuesto ES_C puede estar formado por dos o más elementos *software*. El elemento compuesto es equivalente a la concatenación de las sentencias de los elementos *software* que lo forman:

$$(2-4) \quad ES_C = ES_1 \cup ES_2 = \bigcup_{i=1}^n \sigma_i \cup \bigcup_{j=1}^m \sigma_j$$

- ❖ **Def. 2-23.** Una *aserción simple* α es una afirmación que se realiza acerca de una sentencia de un elemento *software*. Por ejemplo, se puede afirmar que el valor de una variable se encuentra en una posición de memoria dada.
- ❖ **Def. 2-24.** Una *aserción local* a caracteriza aquello que un elemento *software* ES asume que es verdad en cada sentencia σ , dentro de un mismo hilo de ejecución. En general corresponde con un conjunto de aserciones simples. Por ejemplo, cuando se accede a una variable se asumen tres cosas: (1) que la variable tiene un nombre dado, (2) que a ese nombre hay asociada determinada dirección de memoria, y (3) que el contenido de la variable se encuentra dentro de ciertos valores y límites esperados. Se representa como un modificador o filtro sobre el elemento *software* que, a partir de una aserción local y una sentencia, devuelve el conjunto de aserciones simples asociadas a ellas.

$$(2-5) \quad ES(a_i, \sigma) = \bigcup_{j=1}^m \alpha_j \text{ si } a_i = \bigcup_{j=1}^m \alpha_j \wedge \sigma \in ES$$

Las aserciones locales forman un conjunto paralelo de afirmaciones α asociadas a cada sentencia σ . En particular, si σ es vacía, se está realizando una aserción acerca de un estado de ES en algún momento de su ejecución. Si tenemos un bloque de dos sentencias se cumple que las aserciones son *acumulables*:

$$(2-6) \quad \begin{aligned} & ES(a_1, \sigma_1); \quad ES(a_2, \sigma_2) \Rightarrow \\ & ES(a_{\text{acum}}, \sigma_{\text{acum}}) \text{ si } a_{\text{acum}} = a_1 \cup a_2 \wedge \sigma_{\text{acum}} = \sigma_1 \sigma_2 \wedge \sigma_{\text{acum}} \in ES \end{aligned}$$

La notación compacta $ES(a_i)$ obvia a qué bloque de sentencias particular hace referencia la aserción local acumulada a_i .

- ❖ **Def. 2-25.** Una *dependencia* de un elemento *software* ES_1 respecto de otro ES_2 es una aserción local a_i que hace el primer elemento acerca del segundo. Se representa con el símbolo \rightarrow .

$$(2-7) \quad ES_1(a_i) \rightarrow ES_2$$

La dependencia a_i tiene asociado siempre algún bloque de sentencias, aunque no se especifique cuál por brevedad. Una consecuencia importante de la definición de dependencia es que la aserción hecha en ES_1 debe ser verdad también en ES_2 :

$$(2-8) \quad ES_1(a_i) \rightarrow ES_2 \Rightarrow \exists ES_2(a_j) / a_i = a_j$$

Si no fuese así, entonces ES_1 estaría asumiendo algo acerca de ES_2 que no es verdad en ES_2 y, por tanto, sería incorrecto asumirlo en ES_1 . Además, puesto que la aserción a_i debe ser válida en ES_2 , debe existir en ES_2 una aserción a_j que exprese exactamente lo mismo que a_i . Véase también que a_j está asociado a un bloque de sentencias distinto que a_i , esta vez pertenecientes a ES_2 , y que por ser $a_i = a_j$ ambos bloques hacen referencia a las mismas aserciones simples:

$$(2-9) \quad ES_1(a_i, \sigma_i) = ES_2(a_j, \sigma_j) \quad \text{si} \quad a_i = a_j = \bigcup_{k=1}^m \alpha_k$$

- ❖ **Def. 2-26.** Una *dependencia externa* de un elemento *software* ES_1 respecto de otro ES_2 es una aserción local a_j que hace el segundo elemento acerca del primero, es decir, es una *dependencia* del segundo elemento respecto del primero. Se representa con el símbolo \leftarrow .

$$(2-10) \quad ES_1 \leftarrow ES_2(a_j)$$

Y según (2-8) también se cumple que:

$$(2-11) \quad ES_1 \leftarrow ES_2(a_j) \Rightarrow \exists ES_1(a_i) / a_i = a_j$$

- ❖ **Def. 2-27.** Dos elementos *software* ES_1 y ES_2 que forman parte de un mismo hilo de ejecución tienen *interdependencia* si existe dependencia o dependencia externa entre ellos. Se representa con el símbolo \leftrightarrow .

$$(2-12) \quad ES_1 \leftrightarrow ES_2 \Leftrightarrow \exists (ES_1(a_i) \rightarrow ES_2 \vee ES_1 \leftarrow ES_2(a_j))$$

Intuitivamente, la interdependencia representa aquel conocimiento asumido como verdad entre dos elementos *software* dentro de un mismo hilo de ejecución. Nótese que la definición de interdependencia es una relación \vee entre elementos y no una relación \wedge . La relación \wedge es un caso particular de interdependencia.

2.7.1.1 Tipos de funcionalidad

Conocer qué se asume como verdad es crítico a la hora de poder reemplazar un elemento *software* por una versión modificada o por otro elemento completamente distinto, es decir, para reemplazar su funcionalidad.

- ❖ **Def. 2-28.** Una *funcionalidad* F es el conjunto de aserciones locales (conocidas o no) que son asumidas por el elemento *software* que la implementa. Se representa como una función sobre un elemento *software* cuyo resultado es en última instancia un conjunto de aserciones simples.

$$(2-13) \quad F(ES) = \bigcup_{i=1}^n ES(a_i) = \bigcup_{i=1}^n \bigcup_{j=1}^m \alpha_{ij}$$

La funcionalidad incluye todas las aserciones asumidas como verdad por la implementación de un elemento *software*, sean éstas conocidas o no. Si bien no todas las aserciones de una funcionalidad son conocidas, en cambio sí pueden ir descubriendose bajo demanda —por ejemplo mediante depuración— si se dispone del código fuente del elemento *software*.

- ❖ **Def. 2-29.** Una *funcionalidad pública* FP es el conjunto de dependencias externas (conocidas o no) de un elemento *software*.

$$(2-14) \quad FP(ES; ES_1, \dots, ES_m) = \bigcup_{j=1}^m \bigcup_{i=1}^n \begin{cases} ES_j(a_i) & \text{si } ES \leftarrow ES_j(a_i) \\ \emptyset & \text{en otro caso} \end{cases}$$

Nótese que la definición se hace a partir de las dependencias externas, es decir, de lo que *otros* elementos *software* asumen como verdad sobre el elemento *software* en estudio. La notación compacta incluye únicamente el elemento en estudio:

$$(2-15) \quad FP(ES)$$

Se puede demostrar que la funcionalidad pública de un elemento *software* es un subconjunto de la funcionalidad de ese elemento. Según (2-11) y (2-14) para cada dependencia externa de $FP(ES)$ debe existir una aserción en ES equivalente:

$$(2-16) \quad \forall ES_j(a_i) \in FP(ES) \Rightarrow \exists ES(a_k)/a_k = a_i$$

entonces, el conjunto de aserciones de $FP(ES)$ sustituidas según (2-16) pertenece a ES , y siguiendo (2-13) es un subconjunto de $F(ES)$:

$$(2-17) \quad FP(ES) \subseteq F(ES)$$

- ❖ **Def. 2-30.** Una *funcionalidad interna FI* es aquella funcionalidad que no es pública.

$$(2-18) \quad FI(ES) \subseteq F(ES)$$

Teniendo en cuenta (2-17) entonces se cumple:

$$(2-19) \quad F(ES) = FP(ES) \cup FI(ES)$$

La funcionalidad pública muchas veces no se conoce con exactitud, representando todas las dependencias externas entre elementos *software*.

- ❖ **Def. 2-31.** Una *funcionalidad pública explícita FPe* o para abreviar *funcionalidad explícita* es el subconjunto de dependencias de la funcionalidad pública de un elemento *software* que son expresamente conocidas de alguna forma, como por ejemplo una interfaz o documentación.

$$(2-20) \quad FPe(ES) \subseteq FP(ES)$$

Normalmente, la funcionalidad explícita describe un conjunto pequeño de la funcionalidad pública de un elemento *software*. Una funcionalidad explícita no es en general igual a la funcionalidad pública, puesto que pueden existir dependencias externas que no estén identificadas en la funcionalidad explícita. Una información transmitida verbalmente sobre una característica de un elemento *software* puede dar lugar a una dependencia de este último tipo.

- ❖ **Def. 2-32.** Una *funcionalidad pública implícita FPi* o *funcionalidad implícita* para abreviar, es el subconjunto de dependencias de la funcionalidad pública de un elemento *software* que *no* son expresamente conocidas.

$$(2-21) \quad FPi(ES) \subseteq FP(ES)$$

A partir de las definiciones (Def. 2-31) y (Def. 2-32) de funcionalidad explícita e implícita se obtiene:

$$(2-22) \quad FP(ES) = FPe(ES) \cup FPi(ES)$$

La funcionalidad implícita mide la deficiencia de especificación de la funcionalidad pública. Si se pudiesen eliminar todas las dependencias externas de un elemento *software* que no formen parte de una funcionalidad explícita, es decir, si la funcionalidad implícita fuese vacía, entonces ésta coincidiría con la funcionalidad pública. En ese caso, la funcionalidad pública estaría completamente especificada.

$$(2-23) \quad FPe(ES) = FP(ES) \quad si \quad FPi(ES) = \emptyset$$

2.7.1.1 Asociatividad de elementos compuestos

La funcionalidad de un elemento *software* compuesto es la unión de la funcionalidad de los elementos que forman parte de él. Si ES_C está compuesto por ES_1 y ES_2 :

$$(2-24) \quad F(ES_C) = F(ES_1 \cup ES_2) = F(ES_1) \cup F(ES_2) = \bigcup_{i=1}^n ES(a_i) \cup \bigcup_{j=1}^m ES(a_j)$$

Aplicando (2-19) se obtiene:

$$(2-25) \quad \begin{aligned} F(ES_C) &= F(ES_1) \cup F(ES_2) = FP(ES_1) \cup FI(ES_1) \cup FP(ES_2) \cup FI(ES_2) = \\ &= FP(ES_1 \cup ES_2) \cup FI(ES_1 \cup ES_2) \end{aligned}$$

y con (2-22) se tiene para la funcionalidad pública:

$$(2-26) \quad FP(ES_C) = FP(ES_1 \cup ES_2) = FPe(ES_1 \cup ES_2) \cup FPi(ES_1 \cup ES_2)$$

2.7.1.2 Funcionalidad observable

Si tenemos dos elementos *software* que cooperan, usualmente uno de ellos hará de proveedor y otro de cliente. El cliente tiene dependencias con el proveedor a través de un subconjunto de la funcionalidad pública del proveedor, pero potencialmente puede tener dependencias con cualquier parte de la funcionalidad pública de su proveedor, es decir, puede llegar a *observar* esas dependencias.

- ❖ **Def. 2-33.** La *funcionalidad observable* FO por un elemento *software* cliente ES_{Cli} respecto de un elemento *software* proveedor ES_{Pro} es la funcionalidad pública del proveedor:

$$(2-27) \quad FO(ES_{Pro})|_{ES_{Cli}} = FP(ES_{Pro})$$

Por ejemplo, si un proveedor tiene una interfaz de dos funciones, y el cliente sólo es dependiente de la primera función, la funcionalidad observable por el cliente sigue siendo dos funciones. Para el caso de un elemento *software* compuesto se cumple la asociatividad.

$$(2-28) \quad FO(ES_{Pro})|_{ES_{Cli}} = FP(ES_1 \cup ES_2) = FP(ES_1) \cup FP(ES_2) \quad \text{si } ES_{Pro} = ES_1 \cup ES_2$$

2.7.1.3 El principio de encapsulación

El resultado (2-28) muestra que la funcionalidad observable por un cliente de un elemento *software* proveedor compuesto es la unión de las funcionalidades públicas de los elementos que lo forman. Se puede reducir la interdependencia entre el cliente y el proveedor si el cliente no tiene dependencias o dependencias externas con alguno de los elementos que constituyen al proveedor, mediante aplicación directa de (2-14).

$$(2-29) \quad FO(ES_{Pro})|_{ES_{Cli}} = FP(ES_1 \cup ES_2) = FP(ES_1) \quad \text{si } ES_2 \not\leftrightarrow ES_{Cli}$$

Desde el punto de vista de ES_{Cli} , ES_2 es un elemento interno de ES_{Pro} , porque no existe interdependencia entre ellos. Al resultado (2-29) se le conoce como *principio de encapsulación*. Simplemente se ha reformulado a partir de las definiciones dadas en este apartado. Intuitivamente significa que si un elemento *software* compuesto ES_{Pro} posee un elemento principal ES_1 y otro interno ES_2 , y éste último sólo interacciona con el principal y no con otros elementos externos, entonces la funcionalidad observable por un elemento externo ES_{Cli} es únicamente la del elemento principal. El resultado es la reducción de la funcionalidad observable del proveedor y, por tanto, de la interdependencia entre un proveedor y cualquiera de sus clientes.

- ❖ **Def. 2-34.** El *principio de encapsulación* expone el deseo de esconder los detalles de implementación de un elemento *software* ES a otros elementos *software*, como medio para controlar la complejidad percibida de ES desde otros elementos *software*.
- ❖ **Def. 2-35.** Un elemento *software* ES_1 está *encapsulado* con respecto a otro elemento *software* ES_2 si no existe interdependencia entre ellos. Es decir, la existencia de ES_1 no es percibida directa o indirectamente por ES_2 y viceversa.

$$(2-30) \quad ES_1 \not\leftrightarrow ES_2$$

La encapsulación, es decir, la falta de interdependencia, es fundamental para garantizar que las modificaciones realizadas en ES_1 no afectan a ES_2 o viceversa. Una definición relacionada, pero más problemática es la siguiente:

- ❖ **Def. 2-36.** Un elemento *software* ES_1 está *parcialmente encapsulado* con respecto a otro elemento *software* ES_2 si el primero depende del segundo y el segundo no tiene dependencias externas con el primero. Es decir, la existencia de ES_1 no es percibida directa o indirectamente por ES_2 .

$$(2-31) \quad ES_1 \rightarrow ES_2 \wedge ES_1 \leftarrow ES_2$$

La encapsulación parcial ocurre a veces entre clientes y proveedores. Un cliente ES_1 está parcialmente encapsulado con respecto a su proveedor ES_2 , si el proveedor no tiene dependencias con el cliente, pero éste último sí las tiene —como es lógico— con su proveedor. En este caso, el cliente puede cambiar sin afectar al proveedor, aunque no a la inversa. Nótese que las dependencias del cliente respecto del proveedor son exactamente las dependencias externas del proveedor respecto del cliente. En la medida que las dependencias del cliente correspondan con la funcionalidad pública explícita del proveedor, las dependencias estarán controladas. La encapsulación parcial es menos habitual de lo que frecuentemente se piensa, como se verá en el apartado 2.7.2.5 *Aserciones de llamadas de retorno*.

El principio de encapsulación se define tradicionalmente a partir del concepto de *abstracción*. A continuación se presentan dos definiciones:

Def. 2-37. “Una *abstracción* surge del reconocimiento de similitudes entre ciertos objetos, situaciones o procesos en el mundo real, y en la decisión de concentrarse en esas similaridades e ignorar a partir de entonces las diferencias”. [Dahl, Dijkstra & Hoare, 1972; p. 83].

Def. 2-38. “Una *abstracción* denota las características esenciales de un objeto que lo distingue de todos los otros tipos de objetos, dando unas fronteras conceptuales bien definidas, relativas a la perspectiva de un observador”. [Booch, 1994; p. 41].

La encapsulación según Booch se define como:

Def. 2-39. “*Encapsulación* es el proceso de separar los elementos de una abstracción que constituyen su estructura y comportamiento; la encapsulación sirve para dividir la interfaz contractual entre una abstracción y su implementación”. [Booch, 1994; p. 50].

Ambos conceptos están muy relacionados, tal y como lo describe Booch en el siguiente texto:

“Abstracción y encapsulación son conceptos complementarios: abstracción se centra en el comportamiento observable de un objeto, mientras que encapsulación se centra en la implementación que da lugar a ese comportamiento. La encapsulación es a menudo obtenida mediante la *ocultación de información*, que es el proceso de esconder todos los secretos de un objeto que no contribuyen a sus características esenciales [...].” [Booch, 1994; p. 49].

Un elemento *software*, tal y como se ha definido en (Def. 2-4) no asume que implementa una abstracción, sino que corresponde sólo con un conjunto de sentencias. La razón por la que en este trabajo se aporta una nueva definición de encapsulación (Def. 2-35) se debe a que es más precisa que la tradicional, al identificar si un elemento *software* está encapsulado mediante el estudio de sus dependencias con otros elementos *software*. Es en cierto sentido más general, al no relacionar la definición de encapsulación con una abstracción, que presupone una forma determinada de organización del código. Como resultado, se puede aplicar el concepto de encapsulación con mayor rigor y a más bajo nivel.

Existe un paralelo entre la definición de abstracción y la definición de funcionalidad pública (Def. 2-29), pero este paralelo es sólo superficial. Una abstracción es un concepto mental que se expresa en programación mediante la construcción de una interfaz para la abstracción. Esta interfaz corresponde con la funcionalidad pública explícita, pero no necesariamente hace referencia a toda la funcionalidad pública. Ésta última incluye, además, aspectos no definidos explícitamente que son, aún así, asumidos como verdad por otros elementos *software*. La funcionalidad implícita es habitualmente la gran olvidada en las definiciones tradicionales.

Existe también un paralelo entre la funcionalidad interna (Def. 2-30), la implementación de una abstracción, y la encapsulación, que está relacionado con la ocultación de información. La funcionalidad interna de un elemento *software* es igual a la implementación encapsulada de una abstracción *sólo si* esta implementación no tiene interdependencia con otros elementos *software*. Si una fracción de la implementación de una abstracción tiene interdependencia, entonces esa fracción no forma parte de la funcionalidad interna del elemento *software*, sino de su funcionalidad pública. El concepto de encapsulación tradicional es más indeterminado y, por tanto, más propenso a diferentes interpretaciones. Una descripción más precisa como

la dada en (Def. 2-35) caracteriza mejor la situación anterior, donde existe una fracción de la implementación de una abstracción que tiene interdependencias con otros elementos *software*, y que veremos a continuación.

2.7.1.4 Ruptura de la encapsulación

Un proveedor puede ser a su vez cliente de otro elemento *software*, como por ejemplo una biblioteca. En alguna parte de la funcionalidad del proveedor deben encontrarse las aserciones relacionadas con la biblioteca. Y se ha de cumplir que la funcionalidad observable por el proveedor acerca de la biblioteca es:

$$(2-32) \quad FO(ES_{Bib})|_{ES_{Pro}} = FP(ES_{Bib})$$

Si el proveedor tiene un cliente como en (2-27), el proveedor esconderá el acceso a la biblioteca a su cliente si depende de la biblioteca únicamente con su funcionalidad interna $FI(ES_{Pro})$. Si, por el contrario, depende con parte de la funcionalidad pública del proveedor, entonces su cliente también puede observar esa funcionalidad.

Veamos con más detalle este último punto. Supongamos una aserción a_i en ES_{Pro} que depende de una funcionalidad pública de ES_{Bib} . Entonces, según (2-7) y (2-8) se cumple que:

$$(2-33) \quad ES_{Pro}(a_i) \rightarrow ES_{Bib} \quad \wedge \quad \exists ES_{Bib}(a_j) / a_i = a_j$$

donde a_j corresponde con la aserción en ES_{Bib} de la que es dependiente ES_{Pro} . Por (2-9) también se cumple que:

$$(2-34) \quad ES_{Pro}(a_i, \sigma_i) = ES_{Bib}(a_j, \sigma_j)$$

Supongamos ahora que a_i y a_j son aserciones públicas, así que podemos expresarlas como tales:

$$(2-35) \quad ES_{Pro}(a_i, \sigma_i) = FP'(ES_{Pro}) \quad \text{con} \quad FP'(ES_{Pro}) \in FP(ES_{Pro})$$

$$(2-36) \quad ES_{Bib}(a_j, \sigma_j) = FP'(ES_{Bib}) \quad \text{con} \quad FP'(ES_{Bib}) \in FP(ES_{Bib})$$

Sustituyendo (2-35) y (2-36) en (2-34) se tiene que:

$$(2-37) \quad FP'(ES_{Pro}) = FP'(ES_{Bib})$$

Por tanto, la funcionalidad pública de ES_{Bib} forma parte también de la funcionalidad pública de ES_{Pro} , ya que:

$$(2-38) \quad FP'(ES_{Bib}) \in FP(ES_{Pro}) \quad \text{si} \quad FP'(ES_{Bib}) = FP'(ES_{Pro})$$

Apliquemos este último resultado a la relación entre el proveedor y cualquiera de sus clientes. Si bien la funcionalidad observable por el cliente acerca del proveedor es siempre:

$$(2-39) \quad FO(ES_{Pro})|_{ES_{Cli}} = FP(ES_{Pro})$$

El conjunto de aserciones locales $FP(ES_{Pro})$ no tiene porqué ser siempre el mismo. Revisemos porqué. La ecuación (2-35) no es aplicable si $ES_{Pro}(a_i)$ forma parte de la funcionalidad interna de ES_{Pro} y, como resultado, la funcionalidad $FP'(ES_{Bib})$ no es visible. Llamemos $FP_{Orig}(ES_{Pro})$ a esa funcionalidad pública.

$$(2-40) \quad FO(ES_{Pro})|_{ES_{Cli}} = FP_{Orig}(ES_{Pro})$$

En cambio, si $ES_{Pro}(a_i)$ es parte de la funcionalidad pública de ES_{Pro} se cumple (2-35) y (2-38). En ese caso, la funcionalidad $FP'(ES_{Bib})$ forma parte de la funcionalidad observable del cliente.

$$(2-41) \quad FO(ES_{Pro})|_{ES_{Cli}} = FP(ES_{Pro}) \quad \text{con} \quad FP(ES_{Pro}) = FP_{Orig}(ES_{Pro}) \cup FP'(ES_{Bib})$$

A este efecto se le conoce como *ruptura de la encapsulación*, cuyo resultado es un aumento de la funcionalidad pública observable por elementos externos. La ruptura de la encapsulación es problemática por dos razones principales: (1) porque puede ocurrir de forma sutil y desapercibida si la funcionalidad de la

que depende no es conocida, y (2) —la más importante— porque la nueva dependencia puede *propagarse de forma combinatoria*. Es capaz de hacerlo porque $PF'(ES_{Bib})$ es observable por *cualquier* cliente del proveedor y, por tanto, todos los clientes pueden adquirir potencialmente dependencias con la biblioteca. A este último efecto lo llamaremos *permeabilidad*.

- ❖ **Def. 2-40.** Un elemento *software* es *permeable* si inadvertidamente hace público un elemento *software* que se pretendía encapsulado, sea éste interno o con el que coopera.

Sorprendentemente, la ruptura de la encapsulación es extremadamente común en los mecanismos habituales de ensamblado de programas y en técnicas tan de moda como la orientación a objetos. La permeabilidad subyacente de los elementos *software* creados con estas técnicas distribuye exponencialmente las dependencias expuestas hacia otros elementos *software*. El resultado es que la funcionalidad pública crece mucho más que la funcionalidad explícita, cuantas más interacciones existan entre los elementos *software*. De este modo, llega un momento en que el aumento de la deficiencia de especificación de la funcionalidad pública no se puede controlar, y desemboca en última instancia en una mayor complejidad final percibida durante la construcción y mantenimiento de *software*.

Para disminuir la diferencia entre la funcionalidad explícita y la funcionalidad pública, la estrategia general que se seguirá en este trabajo consiste en revisar las técnicas de ensamblado y construcción de programas, buscar puntos donde se rompa la encapsulación y modificar las técnicas adecuadamente para que ésta no se rompa. Obviamente, por motivos de eficiencia, existirán situaciones donde se desee mantener varios elementos *software* no encapsulados, pero tal situación deberá ser más la excepción que la norma. Es decir, los comportamientos por defecto deberán mantener la encapsulación y no al revés.

2.7.1.5 Elementos *software* reemplazables

Dos elementos *software* que cumplan una misma funcionalidad pública pero tengan diferente implementación generalmente tendrán un conjunto distinto de aserciones locales, puesto que cada implementación asumirá cosas diferentes en sus bloques de sentencias:

- ❖ **Def. 2-41.** Dos elementos *software* son *equivalentes* si y sólo si tienen la misma funcionalidad pública. Se representa con el símbolo \equiv .

$$(2-42) \quad \begin{aligned} F(ES_1) &= FP(ES_1) \cup FI(ES_1), \quad F(ES_2) = FP(ES_2) \cup FI(ES_2) \\ F(ES_1) \equiv F(ES_2) &\Leftrightarrow FP(ES_1) = FP(ES_2) \end{aligned}$$

La definición (Def. 2-41) de equivalencia permite simplificar las relaciones entre elementos *software* reduciendo las dependencias al subconjunto público de ellas.

- ❖ **Def. 2-42.** Un elemento *software* es *reemplazable* por otro si y sólo si ambos son equivalentes. Se representa con el símbolo \succ .

$$(2-43) \quad ES_1 \succ ES_2 \Leftrightarrow F(ES_1) \equiv F(ES_2)$$

y combinando (2-42) (2-43) se obtiene:

$$(2-44) \quad ES_1 \succ ES_2 \Leftrightarrow FP(ES_1) = FP(ES_2)$$

Es decir, se puede reemplazar un elemento *software* por otro si este último asume las mismas aserciones públicas que asumía el elemento original. En la práctica no se puede asegurar que dos elementos *software* sean reemplazables, porque es muy difícil afirmar que ambos son equivalentes. No se puede en general asegurar tal cosa porque la única información conocida acerca de la funcionalidad pública de un elemento *software* es su funcionalidad explícita, y ésta es sólo un subconjunto de la funcionalidad pública⁴.

⁴ Para simplificar la discusión, se ha supuesto en la ecuación (2-44) que un elemento *software* implementa un único protocolo, donde F corresponde a la funcionalidad del protocolo y FP a su funcionalidad pública. Si un elemento *software* ES_1 implementa más de un protocolo, el elemento ES_2 que lo reemplace ha de reemplazar todos sus protocolos, pero no tiene porqué cumplirse a la inversa. Es decir, basta con que ES_2 cubra la funcionalidad pública de ES_1 .

2.7.1.6 Reemplazo de elementos software en la práctica

Llegados a este punto nos encontramos con un dilema de difícil solución. Si no se puede asegurar el reemplazo de elementos *software*, ¿cómo entonces se pueden construir elementos *software* reemplazables, que es el objetivo fundamental de la ingeniería del *software* basada en componentes? Habitualmente, se supone con más frecuencia de la debida que el reemplazo es posible sin más, sin reparar en las dificultades del proceso. ¿Cómo se reemplazan dos elementos *software* en la actualidad?

La solución, que es en realidad una aproximación a ella —aunque no por ello menos práctica— consiste en disponer del código fuente del elemento *software* original. Teniéndolo accesible es posible llegar a descubrir *bajo demanda* aserciones no conocidas e incorporarlas en el elemento *software* que lo reemplaza. El precio que hay que pagar es el coste de la aproximación. Tal coste es posiblemente proporcional a la diferencia entre la funcionalidad explícita del elemento de reemplazo y la funcionalidad pública del original, puesto que a mayor diferencia, mayor probabilidad de descubrir aserciones no conocidas del elemento original que sean necesarias para el funcionamiento correcto del elemento de reemplazo. Pero la solución es incompleta, ya que se trata únicamente de una aproximación a la funcionalidad implícita del elemento original. Podemos proponer entonces la siguiente definición cualitativa:

- ❖ **Def. 2-43.** El *coste* de reemplazo de un elemento *software* ES_1 por otro ES_2 es la diferencia de aserciones locales entre la funcionalidad pública del primero y la funcionalidad explícita del segundo. Dicho con otros términos, es la diferencia entre lo que se debe reemplazar de ES_1 y aquello que ya conocemos del elemento ES_2 usado para hacer el reemplazo.

$$(2-45) \quad CR(ES_1 \succ ES_2) = Card(FP(ES_1) - FPe(ES_2))$$

Supongamos primero el caso “ideal”, aquél en el que ES_1 y ES_2 son verdaderamente reemplazables. Según (2-44) quiere decir que ambos tienen igual funcionalidad pública. Entonces, aplicando (2-44) en (2-45), también se puede escribir el caso “ideal” como la diferencia entre la funcionalidad pública y explícita del elemento reemplazado ES_2 .

$$(2-46) \quad CR(ES_1 \succ ES_2) = Card(FP(ES_2) - FPe(ES_2))$$

Y según (2-22) ese resultado es la funcionalidad implícita de ES_2 :

$$(2-47) \quad Card(FP(ES_2) - FPe(ES_2)) = Card(FPi(ES_2))$$

Ahora se puede apreciar mejor la dificultad del reemplazo de elementos *software*. El coste de reemplazo requiere encontrar $FPi(ES_2)$ que, por definición (Def. 2-32), no se conoce. Usualmente se descubre por aproximación *bajo demanda* mediante depuración de ES_2 . Además, $FPi(ES_2)$ quizá incluya funcionalidad adicional a la compartida con $FPi(ES_1)$.

En la práctica es difícil asegurar el caso ideal. Es decir, que $FP(ES_1) = FP(ES_2)$, tal y como requiere (2-44). Siendo más realistas asumamos que, al menos, aquella funcionalidad que sí es conocida —la funcionalidad explícita— sí es igual entre ambos elementos, es decir, que $FPe(ES_1) = FPe(ES_2)$. Entonces, sustituyendo en (2-45), falta por conocer:

$$(2-48) \quad Card(FP(ES_1) - FPe(ES_2)) = Card(FP(ES_1) - FPe(ES_1))$$

que según (2-22) es:

$$(2-49) \quad Card(FP(ES_1) - FPe(ES_1)) = Card(FPi(ES_1))$$

$$ES_1 \succ ES_2 \Leftrightarrow FP(ES_1) \subset FP(ES_2)$$

Esta propiedad puede constatarse intuitivamente con ejemplo sencillo, si suponemos que ES_1 implementa el protocolo `Array` y ES_2 los protocolos `Lista` y `Array`. En lo sucesivo, continuaremos suponiendo que se reemplaza un único protocolo, pues los protocolos adicionales de ES_2 no son relevantes para el proceso de reemplazo si éstos últimos no afectan a la funcionalidad pública del protocolo siendo reemplazado.

Este último resultado se convierte en un problema serio para alguien que intente realizar el reemplazo si no dispone al menos del código fuente de ES_1 . $FPi(ES_1)$ es funcionalidad implícita que por definición no se conoce y que, por tanto, deberá adivinarse de algún modo.

Si se desea reemplazar ES_1 por ES_2 sin añadir funcionalidad pública implícita adicional, entonces el coste es aún mayor. No sólo hay que identificar $FPi(ES_1)$, también hay que restringir $FPi(ES_2)$ para que no publique implícitamente nada que no publicara $FPi(ES_1)$ originalmente. El coste de reemplazo será ahora proporcional a la suma de (2-47) y (2-49), que intuitivamente es el coste de descubrir la funcionalidad implícita de ES_1 y eliminar aquella implícita que provea ES_2 por su cuenta:

$$(2-50) \quad CR(ES_1 \succ ES_2) = Card(FPi(ES_1) \cup FPi(ES_2))$$

Un fabricante de ES_2 que no haya construido el elemento ES_1 se encuentra en considerable desventaja respecto a un reemplazo realizado por el fabricante original. El nuevo fabricante no conoce el ámbito de $FPi(ES_1)$ ya que éste no está explícitamente especificado en ningún sitio. El resultado es que ni siquiera es capaz de evaluar el posible coste de reemplazo. Por esa razón esta última posibilidad es muy rara hoy en día. El modelo presentado en estos últimos apartados explica con sencillez porqué funciona razonablemente bien el reemplazo con los elementos *software* producidos por una misma organización, que usualmente tiene acceso al código fuente de los elementos *software* en juego. También da una razón verosímil a la existencia de *software* abierto u *open source*, particularmente en sistemas complejos que requieren frecuentes revisiones como los sistemas operativos. Igualmente, da una explicación a la tendencia de poner en el dominio público algunos elementos *software* considerados antes estrictamente propietarios. En todos los casos, el objetivo puede resumirse en minimizar el coste de reemplazo.

2.7.1.7 Condiciones de reemplazo

El coste de reemplazo o de simple evolución de partes de programas está directamente relacionado con la capacidad de reemplazarlas fácilmente. El coste de reemplazo da una primera medida cualitativa de la dificultad de reemplazo, calculando la diferencia entre la funcionalidad explícita y la funcionalidad pública de dos elementos *software*. Sin embargo, esta medida es de difícil cálculo. Principalmente porque la funcionalidad pública está poco definida y no se conoce con exactitud. Como ya hemos visto, en todo programa que se ha estructurado en varias partes que cooperan, existen dos extremos importantes: el *proveedor* y el *cliente*.

- ❖ **Def. 2-44.** Un *proveedor* realiza cierta funcionalidad. La descripción de esa funcionalidad es su funcionalidad explícita, siendo esta última usable por varios clientes. Un ejemplo sencillo de proveedor es una subrutina. Un ejemplo más complejo es una biblioteca.
- ❖ **Def. 2-45.** Un *cliente* es capaz de combinar la funcionalidad de varios proveedores para proporcionar la suya propia. Una subrutina o una biblioteca son también ejemplos de clientes. Si un cliente es accedido por otros clientes, entonces actúa también como un proveedor.

La Figura 2-11 muestra los tipos de reemplazo posibles. La variante más común es mantener a un proveedor y cambiar el cliente, tal y como ocurre cuando distintos programas usan una misma biblioteca. La otra variante de reemplazo consiste en mantener el cliente y cambiar al proveedor. Este caso se da, por ejemplo, cuando un proveedor cambia de versión y los clientes deben usar la nueva versión. Puesto que el proveedor no sabe de qué parte de su funcionalidad implícita son dependientes sus clientes, el proveedor debe mantener toda la funcionalidad implícita en la siguiente versión, si es que puede.

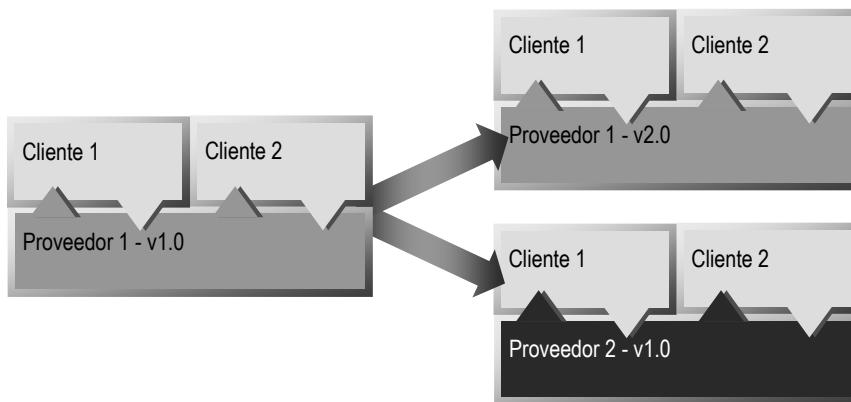


Figura 2-11 Relación entre clientes y proveedores.

Un mismo proveedor puede tener varios clientes y varios clientes pueden usar múltiples proveedores. El primer caso es común, pero el segundo no lo es.

Se debe tener en cuenta que el proveedor también puede llamar al cliente. Ocurre, por ejemplo, cuando una biblioteca gráfica realiza una llamada de retorno o *callback* hacia el cliente, cuando una petición asíncrona de entrada/salida notifica que se ha terminado la petición y ya se puede volver a procesar, o cuando una clase base definida por un proveedor llama a un método de una clase derivada definida en un cliente. Nótese que ambas interfaces, la *interfaz de llamada del cliente* y la *interfaz de llamada de retorno* deben estar especificadas por el proveedor y, por tanto, deben formar parte de la funcionalidad explícita del proveedor. Las condiciones de reemplazo entre clientes y proveedores son fáciles de enunciar:

- ❖ **Def. 2-46.** Las *condiciones de reemplazo para un cliente* son que el nuevo cliente se adapte a la funcionalidad explícita del proveedor, que debe incluir la interfaz de llamada de cliente y la interfaz de llamada de retorno. Si la funcionalidad explícita del proveedor es deficiente, el cliente se deberá adaptar también a la funcionalidad implícita.
- ❖ **Def. 2-47.** Las *condiciones de reemplazo para un proveedor* son que éste ofrezca la misma funcionalidad explícita e implícita a los clientes.

Estas condiciones han demostrado ser difíciles de cumplir debido a los problemas que surgen de la existencia de funcionalidad implícita en el proveedor. El peso de sustitución de ésta recae en el proveedor, y es la razón por la cual es más sencillo escribir nuevos clientes sobre un mismo proveedor que cambiar un proveedor. A medida que los programas se hacen más complejos el número de capas proveedor/cliente aumenta. Los clientes se vuelven a su vez proveedores de otros clientes, y en ese momento adquieren las mismas responsabilidades que sus antiguos proveedores tenían con ellos. Esos nuevos compromisos aumentan las dificultades de realizar cambios en cualquiera de las capas.

La funcionalidad implícita es responsable de buena parte de la complejidad percibida durante el reemplazo de elementos *software*. Por ello, una parte importante de este trabajo está centrada en disminuir la diferencia entre la funcionalidad explícita y la funcionalidad pública de elementos *software*, con el objetivo de que se acerque a cero. Se espera que así también se reduzca el coste de reemplazo y en consecuencia el coste de desarrollo y mantenimiento. En las próximas secciones se presentan algunos ejemplos de aserciones simples, locales y funcionalidad pública implícita.

2.7.2 Análisis de aserciones locales

Comprender las fuerzas que gobiernan la complejidad del *software* de sistemas informáticos es el primer paso para identificar por qué se pierde el control de los mismos. El concepto de aserción local (Def. 2-23), que se encuentra en la raíz de la definición de interdependencia (Def. 2-27), puede ayudar a entender mejor cuáles son esas fuerzas. Comenzaremos con el estudio de las aserciones locales más básicas, aquellas asociadas a las operaciones más sencillas que realiza un sistema informático. Luego continuaremos con

elementos *software* gradualmente más complejos que ayudarán a identificar cuáles son las aserciones más problemáticas, identificando posibles soluciones. El estudio de aserciones de objetos y herencia se dejará para el capítulo 4 *El modelo de objetos*.

Para estudiar qué se asume como verdad al ejecutar instrucciones de máquina, elegiremos un procesador relativamente sencillo. Supondremos que el procesador es de 32 bits y dispone de un contador de instrucciones PC, un registro de pila SP, y un conjunto de registros de propósito general R1, ..., Rn que aceptan un índice opcional de desplazamiento. El procesador ejecuta diferentes clases de instrucciones: lectura y escritura en memoria, operaciones aritméticas y lógicas, comparaciones y saltos. Las operaciones de entrada/salida se considerarán proyectadas en memoria para evitar incluir otro nuevo tipo de instrucciones.

- ❖ **Def. 2-48.** Una aserción es *correcta por construcción* si una implementación asegura que la aserción siempre se cumple.

Las aserciones que se asumen correctas por construcción no requieren más atención, puesto que esas aserciones son siempre verdad y no van a cambiar. En cambio, las aserciones que no son correctas por construcción pueden ser potencialmente erróneas. A veces, también es posible que una aserción sea incorrecta por construcción. Usualmente ocurre si la implementación asume algo que no siempre es verdad, en cuyo caso se trata de un error o problema asociado a la implementación que habrá que subsanar. Si nuestra meta es construir programas más complejos, un criterio para decidir si una implementación ayuda más que otra a acercarnos a esa meta, consiste en evaluar el número de aserciones que pueden ser potencialmente erróneas en ambas implementaciones.

A continuación se estudiarán varios tipos de aserciones que se producen en la construcción rutinaria de programas. Se verán primero aserciones relacionadas con bloques de sentencias, en particular: variables globales, expresiones, secuencias de control y ensamblaje de bloques de sentencias. Luego se verán aquellas relacionadas con las subrutinas, las estructuras, los registros, los arrays, el uso de memoria y, finalmente, las llamadas de retorno. De estas últimas sacaremos conclusiones importantes aplicables a la herencia.

2.7.2.1 Aserciones de bloques de sentencias

Un bloque de sentencias es una secuencia ordenada de operaciones elementales. En un bloque de sentencias se asume como verdad no sólo que cada operación elemental es correcta, sino que la ejecución de la secuencia de instrucciones es también correcta respecto a algún algoritmo. El procesador garantiza que el valor del contador de programa y el registro de pila son correctos por construcción al procesar las distintas instrucciones.

- ❖ **Def. 2-49.** *Las operaciones elementales son correctas* si son correctas las direcciones de memoria referenciadas, los valores almacenados en memoria y los valores almacenados en registros.
- ❖ **Def. 2-50.** *Un bloque de sentencias es correcto* si son correctas las operaciones elementales y el bloque de sentencias cumple un algoritmo conocido.

Comprobar el algoritmo conocido es un elemento crítico. Normalmente se verifica revisando el código fuente y mediante la ejecución de pruebas. Aquellas construcciones que complican los algoritmos son potencialmente peligrosas, puesto que la verificación de un algoritmo sigue siendo una actividad intelectual no fácilmente automatizable. Existen algunos bloques de sentencias especiales que se repiten a menudo: variables globales y locales, expresiones y secuencias de control.

2.7.2.1.1 Aserciones de variables globales y locales

Una variable es una abstracción de un conjunto de bloques de sentencias que acceden a una posición de memoria nombrada. Los bloques de sentencias se encargan de definir el almacenamiento de la variable si es necesario. También leer y escribir el valor de la variable. Si un compilador o intérprete genera el código de definición, acceso y escritura correctamente por construcción, entonces muchas de las aserciones de las operaciones elementales son verdaderas por construcción y podemos obviarlas.

Surgen también aserciones nuevas: una variable tiene asociado un nombre que la identifica. Una variable global tiene asociada una dirección de memoria fija, y una variable local tiene asociada una dirección relativa fija. Las dos últimas aserciones son correctas por construcción en todos los compiladores e intérpretes, únicamente hay que preocuparse por el nombre de la variable y su contenido.

- ❖ **Def. 2-51.** *Una variable es correcta si su nombre lo es y el contenido que almacena es válido.*

Las variables globales son fuente de problemas conocidos, debido a que dos bloques de sentencias distintos son interdependientes si usan una misma variable global. Ambos asumen (1) el nombre de la variable, (2) la dirección de la variable global, y (3) que el contenido de la variable es válido si es manipulado por ambos bloques. Es decir, *se asume que los algoritmos de los dos bloques son compatibles con los valores de la variable global no importando el orden ni el número de veces que se ejecuten*. Nótese que la aserción segunda es correcta por construcción, ya que es garantizada por el compilador. El uso de variables locales evita la necesidad de las dos aserciones restantes anteriores, manteniendo más separados los algoritmos de los dos bloques y simplificando su comprobación. También se reduce la interdependencia entre ambos algoritmos si uno de ellos realiza sólo lecturas de la variable global.

Dando una lectura más formal, véase cómo cada una de las tres aserciones anteriores corresponden con una aserción simple α (Def. 2-23), y que su conjunto corresponde con una aserción local a (Def. 2-24), que se asume como verdad en el bloque de sentencias (Def. 2-22) que accede o modifica una variable global.

2.7.2.1.2 Aserciones de expresiones

Una expresión es un bloque de sentencias del que se obtiene un resultado, e involucra operaciones aritméticas, de desplazamiento, lógicas, o llamadas a funciones. Por ejemplo, la expresión siguiente define un algoritmo cuyo resultado debe ser dos veces x sumado con dos veces y :

$$x + y * 2 + x$$

Su representación en ensamblador precisa el encadenado de registros y gestión de valores temporales en pila o registros, existiendo varias implementaciones posibles: se pueden usar operadores de desplazamiento u operadores aritméticos para realizar la multiplicación binaria, o aplicar distintas estrategias de almacenamiento en pila y registros. Las expresiones son un buen ejemplo de la capacidad de simplificación que consigue un compilador o intérprete, puesto que el encadenado de registros y valores temporales es realizado automáticamente y es correcto por construcción. Hay que hacer notar que, si se programase directamente en ensamblador, tales aserciones sí jugarian un papel significativo en la escritura del código de una expresión y aumentaría la complejidad percibida de su algoritmo.

- ❖ **Def. 2-52.** *Una expresión es correcta si los nombres de las variables son correctos, los valores de las variables son válidos y el algoritmo de la expresión es el que se desea.*

Se puede facilitar la comprobación de la validez de los valores de las variables mediante un sistema de tipos de datos comprobable automáticamente. El sistema de tipos asigna una etiqueta a cada variable, admite expresiones sólo con ciertas etiquetas, y evita manipular erróneamente variables de tipos de datos no compatibles en una expresión.

2.7.2.1.3 Aserciones de secuencias de control

Las secuencias de control como bucles y la sentencia `if` son también bloques de sentencias correctos por construcción. Incluyen expresiones, operaciones de comparación y otros bloques de sentencias internos, que describen algoritmos más complejos. Los bucles, además, pueden incluir asignaciones.

- ❖ **Def. 2-53.** *Una secuencia de control es correcta si la expresión a evaluar es correcta, si la asignación —si existe— es correcta, y si los bloques de sentencias internos siguen el algoritmo que se desea.*

2.7.2.1.4 Aserciones de ensamblaje de bloques de sentencias

El compilador y el enlazador o *linker* aseguran la corrección de la traducción de variables y estructuras de control de programa a instrucciones de máquina. Con ellas resuelve las direcciones de memoria de símbolos y los asigna a segmentos de código. Los cambios de arquitectura en los procesadores son resueltos por los fabricantes de sistemas operativos, compiladores y enlazadores. Pueden dar lugar a aserciones implícitas si las nuevas versiones de tales herramientas no están disponibles. Como ocurre con cualquier otro programa, los enlazadores no siempre resuelven todas las aserciones. Por ejemplo, el enlazador usual disponible en un sistema operativo no distingue tipos de datos asociados a variables globales, siendo el programador responsable de mantener su coherencia. Así, para ANSI C, enlazará erróneamente un programa si en un lugar del mismo se declara una referencia externa a una variable global como:

```
extern int x;
```

mientras que la variable es definida como:

```
float x;
```

2.7.2.2 Aserciones de subrutinas

Una subrutina es un bloque de sentencias parametrizado que implementa un algoritmo. Una subrutina es proveedora del algoritmo. La ejecución se traslada temporalmente al código de la subrutina y luego vuelve otra vez al código cliente, como se puede ver en la Figura 2-12.

La definición y procedimiento de ensamblado de subrutinas suelen estar definidas por los lenguajes de programación. Cada compilador puede implementar una o varias de estas técnicas permitiendo así el uso de subrutinas creadas con diferentes lenguajes. Para facilitar esta labor, es habitual que el fabricante del procesador central dé guías sobre la interfaz binaria de aplicación de bajo nivel (ABI: *Application Binary Interface*). Esta interfaz indica cuáles son los registros reservados para la pila, el registro de activación, el valor de retorno de subrutinas, o los posibles órdenes de los valores en la pila. Con estas reglas es posible intercalar código escrito en diferentes lenguajes de modo razonablemente eficiente.

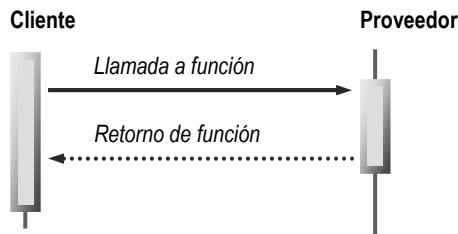


Figura 2-12 Llamada a función y retorno de función.

Una subrutina divide el algoritmo de un bloque de código en algoritmos parciales menores que pueden ser verificados separadamente, haciendo más fácil comprobar si son correctos. También ayuda a reducir la complejidad de un nuevo algoritmo al reaprovechar subrutinas existentes. La llamada a subrutina requiere un número importante de aserciones locales. Para implementar una llamada, en primer lugar es necesario definir el marco de pila, es decir, cómo se van a organizar los parámetros, su orden, dirección de retorno y variables locales en la pila. Típicamente se usa una pila para evitar el uso de variables globales y admitir llamadas recursivas. Los parámetros se han de introducir en un orden preestablecido, asignando unos desplazamientos fijos a cada argumento. El valor de retorno se suele devolver en un registro. La forma en que se produce la llamada a subrutina permite obviar todo el proceso de llamada y centrar la atención sólo en los parámetros de entrada y los valores de retorno. Esto es posible porque el compilador o intérprete es el que gestiona los registros y la pila. Todas las aserciones relacionadas con el proceso de llamada son correctas por construcción.

- ❖ **Def. 2-54.** Una llamada de cliente a subrutina es correcta si el nombre de función es correcto, los valores de los parámetros, el orden en la pila y el valor de retorno son válidos, y el algoritmo de la subrutina es el que se desea.

El nombre de función, los parámetros y los límites de validez de los mismos, así como el valor de retorno y sus límites de validez corresponden con la interfaz de la subrutina. La interfaz esconde la implementación de la subrutina, que es su algoritmo. A menudo la comprobación de los límites de validez de parámetros y valor de retorno precisa comprobaciones en tiempo de ejecución. En cambio, el nombre de función y el tipo y orden de los parámetros pueden ser comprobados en tiempo de compilación por un sistema de tipos. Si están especificados, corresponden con la funcionalidad explícita de la subrutina.

Si los parámetros se pasan por valor a una subrutina y la subrutina no asume nada acerca del código cliente que la llama, entonces la interdependencia entre el cliente y la subrutina proveedora es mínima, correspondiendo con la interfaz de la subrutina. Los parámetros por referencia se ven en el siguiente subapartado, e incurren en mayor interdependencia.

El ensamblaje de subrutinas de una biblioteca estática es muy similar al ensamblaje de una subrutina dentro del propio programa, resolviendo las direcciones finales de cada rutina dentro del programa, y no incurriendo en ninguna aserción adicional importante. El ensamblaje de una subrutina de una biblioteca dinámica requiere esfuerzo extra por parte del enlazador dinámico, que básicamente ha de realizar la labor del enlazador estático en el momento de cargar la biblioteca en tiempo de ejecución. La semántica de la carga dinámica de una biblioteca es ligeramente diferente al enlazado estático, siendo responsable de la aparición de nuevas aserciones implícitas. La carga dinámica incurre en aserciones de *temporalidad*. En una carga dinámica es importante poder ejecutar el código de arranque de la biblioteca dinámica, cuyo orden de ejecución está definido únicamente por el orden de carga. Este orden habitualmente depende del orden ejecución del programa, puesto que una biblioteca dinámica se suele cargar en demanda cuando se intenta ejecutar una de sus funciones.

2.7.2.3 Aserciones de estructuras, registros y arrays

Un estructura o registro es un conjunto de variables que se tratan como una unidad y tienen direcciones de memoria consecutivas. Un *array* es una estructura cuyas variables están identificadas por un índice numérico. Una variable en una estructura está definida por una dirección base de la estructura y un desplazamiento dentro de la estructura. Un bloque de sentencias asume la geometría de una estructura si accede directamente a las variables de la estructura. Es decir, el bloque conoce todos los desplazamientos de las variables dentro de la estructura.

- ❖ **Def. 2-55.** Un acceso a una variable de estructura es correcta si la dirección de la estructura es correcta y si el desplazamiento de la variable es también correcto.

Un compilador, a través de declaraciones para el sistema de tipos de datos, habitualmente resuelve automáticamente los desplazamientos de variables de una estructura asegurando también la corrección de acceso y la eficiencia. Por ejemplo, el código siguiente en C:

```
struct point { int x; int y; } p = { 0, 0 };
// ...
p.x = 1;
p.y = 2;
```

declara una estructura en el sistema de tipos y así el compilador puede generar el código correcto para acceder a las variables de la estructura. Se puede traducir por:

```
p: DW 0      ; Reserva 32 bits inicializados a cero, y asigna la etiqueta 'p' a esa dir.
DW 0      ; Reserva otros 32 bits inicializados a cero
// ...
MOV R1, p
MOV [R1], 1
```

```
MOV [R1+4], 2
```

Sin embargo, el deterioro de la modularidad es significativo. Al replicar el código de acceso a cada variable, el compilador introduce implícitamente una nueva aserción en cada uno de ellos: Asumir que los desplazamientos de los campos de la estructura son siempre los mismos. Es trivial evitar este conocimiento si el acceso a una estructura se hace a través de una subrutina, como en el siguiente ejemplo:

```
struct point { int x; int y; } *p;
int px (struct point *p)
{ return p->x; }
int py (struct point *p)
{ return p->y; }
void setpx (struct point *p, int x)
{ p->x = x; }
void setpy (struct point *p, int y)
{ p->y = y; }
// ...
setpx (p, 1);
setpy (p, 2);
```

La subrutina asegura la corrección del acceso a una variable de la estructura y no se codifican desplazamientos en el código cliente. Hacen falta dos subrutinas por variable de estructura: una para leer el valor y otra para escribir un nuevo valor.

Ahora bien, si como en el ejemplo anterior una estructura se pasa como referencia en un parámetro de una subrutina, entonces de forma efectiva se está usando una variable compartida entre el código del cliente y el código del proveedor. Situación similar temporalmente a una variable global que aumenta la interdependencia entre ambos. En el ejemplo es precisamente lo que se desea, pero no siempre es así. Muchas veces no se pretende modificar una variable pasada por referencia, pero si el modo de envío de parámetros no prohíbe la modificación, no es difícil que una subrutina inadvertidamente modifique un parámetro. Además, el mero hecho de leer un valor de una estructura por referencia puede revelar los desplazamientos internos de esa estructura, aumentando la interdependencia.

2.7.2.4 Aserciones de uso de memoria

El almacenamiento de variables y estructuras se realiza en memoria. Dependiendo del lenguaje de programación el uso de memoria es distinto. Siguiendo a [Pratt & Zelkowitz, 1999; pp. 217-219] existen las siguientes categorías de uso de memoria gestionadas automáticamente por compiladores, enlazadores o bibliotecas de sistema. Todas ellas generan aserciones válidas por construcción.

- Asignación de segmentos de código a programas de usuario.
- Programas de sistema en tiempo de ejecución —incluyendo bibliotecas estáticas y dinámicas— y soporte en tiempo de ejecución o *runtime* de lenguajes.
- Estructuras estáticas y constantes definidas por el usuario.
- Entornos de llamada o *referencing environments*.
- Variables temporales en evaluación de expresiones.
- *Buffers* de entrada/salida.
- Tablas, estructuras y estados internos de sistema.
- Marcos de llamada y retorno de subrutinas.

Las operaciones de creación y destrucción de estructuras de datos son un caso distinto. Muchos lenguajes tienen gestión dinámica de estructuras, creadas en puntos arbitrarios de la ejecución, y no en puntos bien definidos como una llamada a subrutina. La gestión manual de la memoria dinámica trae consigo la

responsabilidad de conocer quién es el encargado de crear una nueva estructura en memoria y quién es el encargado de liberarla finalmente. Cuando una estructura es pasada como parámetro a múltiples subrutinas —y por tanto es compartida— ¿cuál de ellas debe liberar su memoria? Ciertamente, esta pregunta es muy dependiente del algoritmo usado por *todas* las subrutinas que manipulan la estructura, puesto que esta última actúa como una variable compartida temporalmente. Cada subrutina ha de suponer —y realiza una aserción acerca de— si otra subrutina va a liberar o no la memoria de la estructura pasada como parámetro, o si la memoria de la misma es aún válida. La gestión manual también incurre en aserciones de *temporalidad* sobre la validez de estructuras dinámicas, que muchas veces son implícitas y complican la modificación de los algoritmos.

Buena parte de los errores habituales y a veces oscuros de C y C++ se originan en la gestión manual de memoria, resultado de la interdependencia causada por las dependencias de gestión de cada dato dinámico manejado por el programa. La corrección de la gestión de memoria se garantiza sólo si la petición y liberación de memoria es automática, en cuyo caso la responsabilidad de liberar la memoria recae en un proceso automático llamado recolector de basura o *garbage collector*. Los recolectores traen consigo sus propios problemas como por ejemplo saber cuándo deben realizar la recolección de la memoria no utilizada, pero aun así, una clase completa de posibles errores se elimina del código principal del programa.

- ❖ **Def. 2-56.** *La gestión de una variable o estructura en memoria dinámica* es correcta si la dirección de la variable o estructura es válida, sus datos son válidos mientras se usa y, finalmente, la variable o estructura es liberada cuando ya no se usa.

Naturalmente, en caso de realizar programación de sistemas de bajo nivel, ciertas aserciones tomadas como correctas por construcción dejarán de serlo y deberán de tenerse en cuenta igualmente. Es decir, si bien existen técnicas que permiten obviar la mayor parte de los problemas de gestión de memoria, tales técnicas no son aplicables a todos los casos y se hace necesario reducir el nivel de abstracción de programación. La complejidad percibida a la hora de trabajar a bajo nivel aumenta porque es necesario tener en cuenta multitud de detalles de implementación que usualmente son resueltos por herramientas automáticamente.

2.7.2.5 Aserciones de llamadas de retorno

Un importante tipo de llamadas a subrutinas son las llamadas de retorno. Dan lugar a comportamientos más complejos que los obtenidos con el encadenamiento usual de llamadas a subrutinas. El uso de llamadas de retorno es característico de la programación orientada a eventos, que modela comportamientos asíncronos. Es común también en la gestión de dispositivos de entrada/salida. No obstante, nos detendremos ahora a estudiar sus propiedades porque los resultados obtenidos serán directamente aplicables a una importante discusión posterior, en el apartado 4.3.8 *Llamadas de retorno y herencia*, sobre las dificultades y serias implicaciones de este tipo de llamadas durante la herencia de implementación.

- ❖ **Def. 2-57.** Una llamada de retorno o *callback* es una llamada a subrutina que un código proveedor realiza sobre un código cliente.

Por ejemplo, en un entorno gráfico se puede registrar una subrutina para ser llamada en caso de que se pulse un botón determinado. El código de cliente corresponde con la subrutina registrada, y el código proveedor es la interfaz gráfica. Este es el caso de la mayoría de los entornos gráficos como Win32 de Windows [Microsoft, 2003g], Carbon de MacOS [Apple, 2002c], Motif [OSF, 1991], o XView [Heller, 1993] que implementa el entorno OPEN LOOK [Sun, 1989]. En ellos las llamadas desde el entorno gráfico a la aplicación se producen a través del registro dinámico de funciones de llamada de retorno. Las llamadas de retorno introducen nuevas aserciones que no son necesarias en una ejecución cliente/proveedor simple.

- *Definición de interfaz implícita.* El registro de una subrutina de retorno no suele ser comprobable por el compilador, tan solo el número y tipo de parámetros de ésta.
- *Asincronía.* Se puede recibir, por ejemplo, la notificación de *botón pulsado* sobre un botón de la interfaz gráfica.

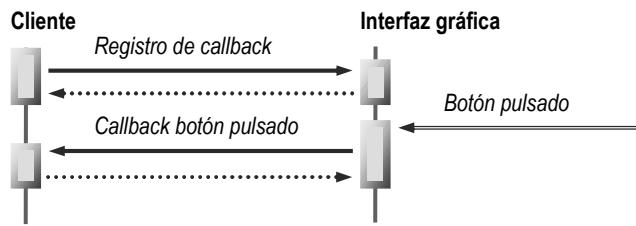


Figura 2-13 Asincronía en llamadas de retorno.

- **Temporalidad.** Supongamos que el entorno gráfico permite registrar dos subrutinas para un botón de la interfaz gráfica: una para notificar que se ha pulsado un botón del ratón y otra para notificar que se ha soltado. En este caso, el entorno gráfico siempre hace dos llamadas de retorno en el mismo orden: *botón pulsado* y *botón soltado*.

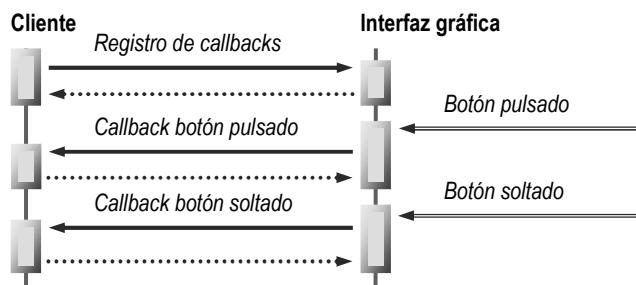


Figura 2-14 Temporalidad.

- **Información de estado intermedio del proveedor.** Cuando se recibe una notificación mediante una llamada de retorno, el proveedor no suele empaquetar toda la información necesaria para que cualquier cliente pueda tomar decisiones al respecto. Al contrario, por motivos de eficiencia sólo envía un mínimo de información al cliente. El cliente entonces ha de efectuar llamadas adicionales al proveedor para obtener más información acerca de la notificación. En este escenario, el proveedor debe ofrecer un estado intermedio consistente para el cliente, y ese estado se convierte temporalmente en variables compartidas entre el proveedor y el cliente, cuyos problemas son equivalentes a los encontrados en variables globales. Para aliviarlo, el proveedor debe de proporcionar una *copia* del estado intermedio al cliente —para que éste no necesite realizar llamadas adicionales— o recibir sólo llamadas adicionales de lectura. La eficiencia de ambas opciones depende tanto del número de variables del estado intermedio como del número de llamadas posibles de petición de información de estado.

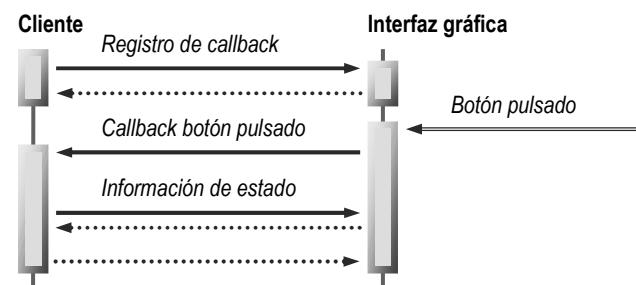


Figura 2-15 Información de estado intermedio del proveedor.

- **Llamadas de retorno encadenadas.** Si el cliente interroga al proveedor durante una llamada de retorno haciendo una llamada al proveedor, esta última llamada puede dar lugar a una nueva llamada

de retorno *antes* de que termine el proceso de la primera llamada de retorno. Este escenario ocurre habitualmente cuando el cliente induce una modificación —conocida o no— en el estado del proveedor, que a su vez dispara una llamada de retorno adicional. El proveedor debe mostrar también un estado consistente para el cliente, aun cuando todavía espere el resultado de una llamada de retorno. En este caso ofrecer un estado consistente se complica, dado que el estado del proveedor puede cambiar y éste no puede simplemente suministrar una copia del estado al cliente. Debe, por el contrario, instar al cliente a no guardar localmente información sobre el proveedor y pedir siempre ésta al proveedor en modo lectura.

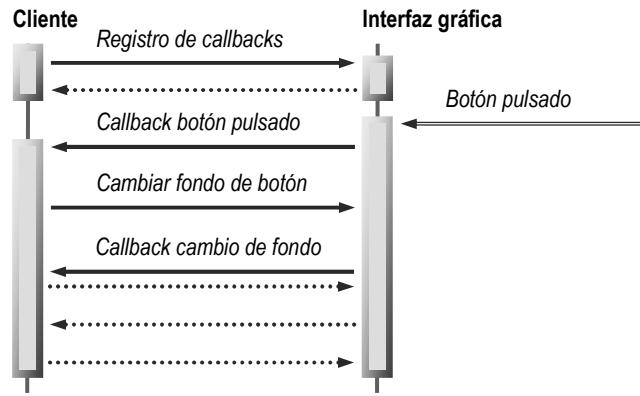


Figura 2-16 Llamadas de retorno encadenadas.

Las llamadas entre proveedores y clientes complican el flujo de ejecución de un programa y hacen más difícil la comprobación de la exactitud de los algoritmos asociados a cada subrutina. Muchas veces, el código asociado a las subrutinas del proveedor no es conocido, por lo que sólo la información de interfaz con las subrutinas está disponible. Si la interfaz de esas subrutinas no da información para poder identificar todas las nuevas aserciones adicionales identificadas, éstas se convertirán en parte de la funcionalidad implícita de las subrutinas. La asincronía y la temporalidad son relativamente fáciles de reconocer y comprobar, pero no las aserciones acerca del estado intermedio del proveedor o las llamadas de retorno encadenadas. Por tanto, es difícil asegurar cuándo una llamada es correcta o no.

En todo caso, sí es posible evaluar dónde se encuentran algunas dificultades relevantes. El estado intermedio del proveedor en llamadas de retorno no debe dar pie a aserciones implícitas por parte del cliente o, de lo contrario, el número y cantidad de éstas aumentará muy significativamente. Aserciones implícitas son:

- *Suponer que el estado del proveedor no cambia durante una llamada de retorno.* Tal suposición permite al cliente guardar temporalmente los valores obtenidos del proveedor. Si no es correcta, esos valores pueden ser erróneos.
- *Tener acceso a las estructuras internas del proveedor en una llamada de retorno.* Las aserciones introducidas son equivalentes a las asociadas a estructuras de datos. Si estas estructuras son modificables, es necesario añadir también las aserciones asociadas a las variables globales, junto con los problemas que esto conlleva.
- *Mantenimiento de la temporalidad.* La ejecución de múltiples llamadas de retorno no tiene por qué garantizar la temporalidad. Puede que algunas ocurran en momentos inesperados dependiendo de la implementación interna del proveedor, transformándose en llamadas asíncronas.

Es posible evitar estas aserciones implícitas aportando al cliente la interfaz suficiente para poder obtener la información que necesita sin necesidad de asumirla implícitamente. Una desventaja de esta opción es que es menos eficiente. Para mantener un estado consistente en el proveedor, es necesario que éste tenga constancia de todos los intentos de modificación de su estado. El proveedor puede proporcionar referencias de sólo lectura por motivos de eficiencia, pero únicamente admitir modificaciones a través de una interfaz adicional. Así, el proveedor tiene siempre la posibilidad de mantener consistente su estado interno y evitar las dos

primeras clases de aserciones implícitas anteriores. El mantenimiento de la temporalidad es más complejo y no es fácil intuir una forma general que impida que se rompa. El cliente puede evitar depender de la temporalidad si codifica sus algoritmos asumiendo que las llamadas de retorno son siempre asíncronas.

Una llamada a subrutina normal es más simple que una llamada de retorno porque no presenta asincronía ni temporalidad ni llamadas de retorno encadenadas, aunque sí ha de tener cuidado con los parámetros por referencia, especialmente en llamadas recursivas. Las llamadas de retorno modifican sustancialmente la semántica asociada a una subrutina, dando lugar a comportamientos asíncronos que no son programables con subrutinas normales. Los entornos gráficos, así como cualquier sistema que deba reaccionar ante eventos asíncronos, se pueden diseñar con mayor simplicidad usando llamadas de retorno, si bien es necesario tener en cuenta cuáles son sus puntos fuertes y débiles.

En resumen, el análisis anterior pone de manifiesto que en la actualidad hasta los programas más simples realizan un número extremadamente elevado de aserciones locales. Muchas de ellas pueden obviarse porque se suponen correctas por construcción. El diseño y desarrollo de herramientas que garanticen aserciones correctas es la base para reducir el número que hay que tener presente en un programa. Los compiladores, ensambladores y enlazadores son ejemplos de estas herramientas. El resto de las aserciones no resueltas automáticamente—o aquellas que sí lo son pero asumen erróneamente algo—se perciben como complejidad de los programas. Interdependencias generalmente implícitas que no se tienen en cuenta lo suficiente en el diseño y desarrollo. Es más, las aserciones locales tienden a expandirse de forma combinatoria, por lo que su identificación y férreo control son indispensables para dar viabilidad a la construcción de programas más complejos.

2.8 Recapitulación

En este capítulo se ha definido con precisión el concepto de componente que va a usarse en este trabajo, con el objetivo de evitar posibles ambigüedades de interpretación. La organización de sistemas informáticos con componentes trae consigo la esperanza de reducir costes en su desarrollo a la vez que se aumenta la confianza en el producto resultante. Se ha puesto de manifiesto cómo poco a poco esa organización se va asentando en los sistemas informáticos actuales, usando como ejemplo las aplicaciones para Internet. Estudios en ingeniería del *software* han puesto énfasis en esta organización y propuesto métodos y guías para fomentarla. La ingeniería de dominios, las familias de sistemas y las líneas de productos son los ejemplos más notables, dando una perspectiva de los avances en este campo. Las herramientas para poner en práctica esas ideas son los modelos de componentes. COM, COM+, JavaBeans, EJB, CORBA, CCM y servicios *web* son los modelos actuales más conocidos. Modelos que intentan fomentar más la separación entre procesos y productos, y la división de tareas de creación de componentes, ensamblado, configuración y despliegue final. En ese punto del capítulo concluye la descripción del estado del arte acerca de los componentes.

La distinción entre procesos y productos existente en la actualidad es muy difusa y no se comprende bien. Las técnicas usadas en la construcción de los sistemas informáticos a veces socavan las ideas organizativas de más alto nivel vistas antes en el capítulo, e introducen complejidad adicional que hace difícil acometer con efectividad la separación entre procesos y productos. Por esa razón se estudia con detalle el concepto de interdependencia y las aserciones locales. Se propone una nueva descripción formal de las dependencias entre fragmentos de código y se estudian cualitativamente. Se caracteriza formalmente con más precisión el principio de encapsulación, la ruptura de la encapsulación, el reemplazo de componentes, y los serios inconvenientes asociados con el reemplazo. Problemas que en general únicamente son discutidos difusamente. Los resultados obtenidos permiten abordar con mayores garantías una revisión crítica de las técnicas básicas de construcción de programas, realizando un análisis de aserciones locales. Se destaca que la construcción y ensamblado de programas usando reglas que limiten la proliferación de aserciones locales ayuda a crear programas más correctos, con menor interdependencia y, en última instancia, facilita la

construcción de programas más complejos. Se sientan así las bases de discusión posterior, y prepara el camino para el próximo capítulo, que introduce a grandes rasgos la solución propuesta en este trabajo.

3

XC. Una propuesta

3.1 Introducción

Este capítulo presenta una perspectiva del deseo de usar técnicas basadas en componentes desde el punto de vista del diseño de un lenguaje de programación. Como ya se ha revisado en la introducción, existen ciertas similitudes y parecidos entre los objetivos de un modelo de componentes y los objetivos de un lenguaje de programación moderno, orientado a la construcción robusta de sistemas informáticos. En el capítulo anterior se ha discutido la importancia de la posibilidad de reemplazar elementos *software*, y qué implicaciones tiene que el reemplazo sea posible. En este capítulo analizaremos qué estructuras *software* han de tenerse en cuenta, qué conceptos adicionales hay que evaluar, y bosqueja la solución propuesta: la definición de un modelo de componentes soportado directamente por un lenguaje de programación. El modelo, como se verá, es independiente del lenguaje. La construcción del modelo dentro del marco de un lenguaje de programación simplificará algunas partes importantes de su implementación, y pondrá en relieve que los conceptos hoy asociados a componentes podrían formar parte de la definición de lenguajes y simplificar la interacción con componentes.

En primer lugar se introducen algunas definiciones complementarias, relativas también a la posibilidad de reemplazo, pero ahora desde el punto de vista del diseño de lenguajes: modularidad, interfaz de módulo e independencia. Se ven posteriormente varias premisas de diseño que es necesario evaluar para asegurar que no nos desviamos de las guías obtenidas en el estudio de las condiciones de reemplazo y las técnicas de ensamblado. Luego se esbozan las características generales del modelo de componentes y cómo éste puede ser implementado directamente por un lenguaje de programación. Se define entonces con más detalle cuáles son los objetivos del lenguaje propuesto, XC, muchas de cuyas características van más allá de la definición de un modelo de componentes y de las facilidades de programación de sistemas, pero que son esenciales para la implementación creíble y consistente de un lenguaje. Por último se evalúa una tabla comparativa del prototipo construido respecto a otros lenguajes: C, C++, Java, C#, Objective-C, Smalltalk-80 [Goldberg & Robson, 1989], SELF [Ungar & Smith, 1991], Eiffel [Meyer, 1997] y Modula-3.

3.2 Conceptos básicos

Cuando se plantea un modelo de componentes desde el punto de vista de un lenguaje de programación, es preciso dar forma a los conceptos abstractos asociados a los componentes: partición de programas, estructura jerárquica, definición de interfaz, composición de componentes o posibilidad de reemplazo. Siguiendo el principio de encapsulación estudiado en el capítulo anterior (Def. 2-34), el

primer aspecto a tener en cuenta es la separación del código de un programa en diferentes partes. Los programas deben poder dividirse en partes más pequeñas y manejables para esconder complejidad a otros fragmentos.

- ❖ **Def. 3-1.** Un *módulo* es una partición identificada de un programa, cuyo objetivo es resolver una funcionalidad dada. Una definición más completa se da en (Def. 3-16).

Un módulo es un tipo especial de elemento *software*, en donde se definen otras estructuras más pequeñas como subrutinas y objetos. Si en un módulo se pueden definir submódulos, entonces se dota de estructura jerárquica a las particiones de un programa. También facilitan la asignación de las particiones a equipos de trabajo separados, potencialmente en paralelo. Es posible aplicar a los módulos las definiciones generales dadas en el apartado 2.7 *Interdependencia*. Según (2-19) la funcionalidad de un módulo se divide en funcionalidad pública y funcionalidad interna. La funcionalidad pública es aquella accesible por otros módulos.

- ❖ **Def. 3-2.** *Interfaz o protocolo de módulo*. Es una especificación de la funcionalidad pública de un módulo y cómo acceder a ella.
- ❖ **Def. 3-3.** Una interfaz o protocolo de módulo es *implícita* si no se encuentra escrita en ningún sitio.
- ❖ **Def. 3-4.** Una interfaz o protocolo de módulo es *explícita* si está especificada en algún lugar.

Es deseable que la funcionalidad de un protocolo explícito se pueda comprobar automáticamente. Las definiciones anteriores son paralelas a las dadas en (2-22), en donde se expresa que la funcionalidad pública se puede dividir a su vez en explícita e implícita. En otras palabras, si M es el elemento *software* en estudio:

$$(3-1) \quad FP(M) = FPe(M) \cup FPi(M)$$

- ❖ **Def. 3-5.** Una interfaz o protocolo de módulo es *parcialmente explícita* si la especificación explícita de un protocolo de módulo no es completa, siendo implícito parte de su protocolo. Es decir, existe funcionalidad pública implícita:

$$(3-2) \quad FP(M) \neq FPe(M) \text{ si } FPi(M) \neq \emptyset$$

- ❖ **Def. 3-6.** *Implementación de módulo*. Es la funcionalidad interna de un módulo que expresamente no se desea hacer pública más aquella funcionalidad que es pública implícitamente. Intuitivamente, la implementación de un módulo contiene el resto de la funcionalidad que no es parte explícita de su interfaz o protocolo. Se deduce a partir de (2-19) y (2-22).

$$(3-3) \quad Imp(M) = FI(M) \cup FPi(M) = F(M) - Fe(M)$$

La definición (Def. 3-6) reconoce la imperfección de las interfaces de módulo, debida a la existencia de funcionalidad implícita. El principio de encapsulación pretende reducir la interdependencia entre distintos elementos *software*. Es obvio que la interdependencia cero no es siempre útil, porque impediría la comunicación entre módulos. Tampoco lo es la interdependencia indiscriminada, porque en ese caso no se esconde complejidad. Reducir la interdependencia entre módulos a una funcionalidad pública limitada y controlada es el camino más adecuado. Funcionalidad que se representa con los protocolos de módulo y, si es explícita, sirve para que los módulos se pongan de acuerdo en cuáles son sus dependencias. Pero, dado que normalmente estos protocolos suelen ser únicamente parcialmente explícitos, hay que tener en cuenta las complicaciones debidas a la funcionalidad implícita y a la posible ruptura de encapsulación.

La relación (2-23) ejemplifica la meta que se desea obtener con los protocolos de módulo: si la funcionalidad implícita es nula, entonces la funcionalidad pública de un módulo corresponde con la funcionalidad explícita, es decir, su protocolo de módulo. Tal y como expresan Britton y Parnas:

“La meta general de la descomposición en módulos es la reducción de los costes del *software* que permita a los módulos ser diseñados y revisados independientemente. [...] Cada estructura de módulo debería ser lo suficientemente simple para que ésta pueda ser comprendida completamente; debería ser posible cambiar la implementación de otros módulos sin el conocimiento de la implementación de otros módulos y sin afectar el comportamiento de esos módulos; [y] la facilidad de realización de un cambio en el diseño debería mantener una relación razonable con la posibilidad de que el cambio sea necesario”. [Britton & Parnas, 1981; p. 2].

Esta meta recibe el nombre de *módulos independientes*.

- ❖ **Def. 3-7.** Un módulo es *independiente* si su implementación no depende de detalles de implementación de otros módulos, tan sólo de sus interfaces con ellos. Se define de forma general mediante (2-23). Por tanto, siguiendo (3-3):

$$(3-4) \quad Im\ p(M_{ind}) = FI(M_{ind}) = F(M_{ind}) - Fe(M_{ind}) \quad \text{con} \quad FPi(M_{ind}) = \emptyset$$

La propiedad de independencia es deseable, porque da la libertad para cambiar la implementación de un módulo sin que el cambio afecte a la implementación de otros módulos. Se dice entonces que presenta *modularidad*.

- ❖ **Def. 3-8.** Un cambio en la implementación de un módulo es *modular* si el cambio se mantiene encapsulado dentro del módulo y no es visible en otros módulos.

Las definiciones (Def. 3-7) y (Def. 3-8) son similares a las definiciones clásicas de *acoplamiento* y *cohesión*, que tienen su origen en el diseño estructurado y pueden encontrarse por ejemplo en [Booch, 1994; pp. 136-137]. A la propiedad de independencia se la denomina también *desacoplamiento* o *acoplamiento débil*. Se define *acoplamiento* como:

Def. 3-9. “Acoplamiento es la medida de la fuerza de asociación establecida por una conexión de un módulo a otro. El acoplamiento fuerte complica un sistema, dado que un módulo es más difícil de entender, cambiar o corregir por sí mismo si éste está altamente interrelacionado con otros módulos. La complejidad puede ser reducida mediante el diseño de sistemas con el acoplamiento más débil posible entre módulos.” [Stevens, Myers & Constantine, 1979; p. 209].

Un concepto muy relacionado con el acoplamiento es la *cohesión*.

Def. 3-10. “Cohesión mide el grado de conectividad entre los elementos de un módulo simple [...]. La forma menos deseable de cohesión es la cohesión coincidental, en la que abstracciones completamente no relacionadas son situadas en la misma clase o módulo. Por ejemplo, considérese una clase incluyendo las abstracciones de perros y naves espaciales, cuyos comportamientos están muy poco relacionados. La forma más deseable de cohesión es la cohesión funcional, en la que todos los elementos de una clase o módulo trabajan juntos para proporcionar algún comportamiento bien definido.” [Booch, 1994; p. 137].

Ambos conceptos ayudan a Booch a definir la modularidad como:

Def. 3-11. “Modularidad es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesionados y débilmente acoplados.” [Booch, 1994; p. 57].

En este apartado se da una nueva definición de módulos independientes (Def. 3-7) en términos del concepto de interdependencia del apartado 2.7, para poner énfasis en cómo la interdependencia modela las propiedades finales de los módulos. La definición (Def. 3-11) corresponde con la definición de módulos independientes dada en (Def. 3-7) en el sentido de estar débilmente acoplados, pero no en si los módulos están cohesionados. La cohesión, si bien muy importante desde el punto de vista del diseño, es una propiedad ortogonal al hecho de que dos fragmentos de código de uno o más módulos tengan interdependencias.

3.3 Premisas de diseño de XC

Cuando se acomete el diseño de un lenguaje de programación, una de las primeras preguntas que se debe formular es: ¿cuál es el tipo de programas que se pretende construir con él? La respuesta a

esta pregunta modelará características básicas del incipiente lenguaje y le conferirá de ciertas propiedades que harán más sencillo escribir programas del tipo o tipos elegidos. Recordemos que, en nuestro caso, el objetivo es diseñar una herramienta que asista en la creación de componentes y con ellos ser capaces de elaborar sistemas más complejos.

Precisando más, el contexto de aplicación que se quiere dar al nuevo lenguaje es facilitar la construcción de sistemas informáticos de al menos tamaño medio mediante componentes. Se entiende por *sistemas informáticos de tamaño medio* aquellos que requieren actualmente del orden de 10^5 líneas de código C, y que generalmente implican a equipos de trabajo coordinados durante un tiempo relativamente largo. Los sistemas informáticos de estos tamaños suelen tener requisitos de bajo nivel, propios de programación de sistemas, que involucran aspectos de intercomunicación de procesos, gestión de memoria, distribución, acceso al sistema operativo o control de multitarea, por poner algunos ejemplos. Lenguajes adaptados a este tipo de requisitos son C, C++, Objective-C, Modula-3 o Component Pascal [Oberon, 1997]. En muchos casos, los requisitos de un sistema informático incluyen la necesidad de realizar programación de bajo nivel para obtener las prestaciones deseadas, puesto que la naturaleza genérica de los sistemas operativos y bibliotecas no siempre es adaptable a los problemas particulares de cada aplicación. Ejemplos de ello lo encontramos en aplicaciones de gestión de bases de datos, manipulación intensiva de gráficos o sistemas distribuidos. Las cuatro premisas que se estudian a continuación centran las decisiones fundamentales de diseño del nuevo lenguaje propuesto. Existe una quinta premisa (Def. 4-9), cuyo enunciado se postergará hasta el siguiente capítulo, cuando se estudie la propiedad de monotonía en el apartado 4.3.4.

La primera premisa introduce el concepto de módulos independientes visto anteriormente en un programa:

- ❖ **Def. 3-12. Premisa 1.** Un programa se debe poder descomponer en la medida de lo posible en módulos independientes que se comunican a través de interfaces de módulo.

La búsqueda de módulos independientes es esquiva, puesto que las interfaces de módulo suelen ser a lo sumo parcialmente explícitas, como muestra (3-3). Esta premisa, al ser la primera, constata la importancia de acercarnos a la propiedad de independencia de los módulos sobre otras propiedades del lenguaje. También hace explícita la existencia de interfaces de módulo que sean verificables por separado de su implementación.

Ciertos aspectos de la propiedad de independencia están muy relacionados con la *calidad*. Cuanto más se especifica el comportamiento de un módulo, más seguro se está de su funcionamiento y en qué condiciones éste es adecuado. Una especificación de interfaz más explícita es más exacta y completa, y como resultado probablemente más cara. En cambio, una especificación menos formal es más barata inicialmente, pero a su vez menos exacta y quizás más cara a largo plazo. Este razonamiento nos lleva a intuir que el grado de independencia de un módulo está parcialmente controlado por la calidad de la interfaz explícita del mismo.

La segunda premisa pone de manifiesto el deseo de poder realizar cambios incrementales independientes.

- ❖ **Def. 3-13. Premisa 2.** La implementación de un módulo debe poder variar de forma independiente siempre que no se modifique su interfaz o protocolo.

El protocolo de un módulo independiente describe toda la funcionalidad que otros módulos deben tener en cuenta. Nótese que la posibilidad de efectuar cambios modulares depende directamente de la calidad y completitud del protocolo explícito publicado hacia otros módulos. El cambio es independiente si otros módulos no son conscientes de ese cambio. Si una modificación cambia el protocolo implícito, tal modificación no será independiente y otros módulos pueden llegar a tener dependencias con el cambio. Un cambio de este tipo no es modular según la definición (Def. 3-8).

La variación de la implementación de un módulo puede parecer en principio sencilla, pero no lo es. Las posibilidades de variación de un módulo dependen estrechamente de la definición de los tipos de datos

de cada módulo y en cómo se produzca la variación, por ejemplo usando sustitución de bibliotecas o mediante técnicas de herencia. Si las técnicas de variación no son modulares, entonces se corre el riesgo de ruptura de encapsulación con los problemas de interdependencia asociados.

La tercera premisa está muy relacionada con la segunda y la posibilidad de cambio modular. Es más sutil, pero no por ello menos importante:

- ❖ **Def. 3-14.** *Premisa 3.* La semántica en un módulo cliente no debe cambiar si cambia la implementación de su módulo proveedor y este último respeta su propio protocolo.

Esta premisa es consecuencia de la lectura de la segunda premisa desde el punto de vista del módulo cliente. Si el cambio realizado en el módulo proveedor es modular, entonces la semántica en el lugar de llamada no debería cambiar. Es decir, en el lugar de llamada siguen siendo verdad las mismas aserciones acerca de la funcionalidad pública del proveedor. Mantener la semántica del código cliente es crítico para que un cambio sea efectivamente modular en el módulo proveedor. Si la semántica del código cliente no cambia, entonces no debe haber necesidad de modificar o revisar ese código cuando hay un cambio modular en el módulo proveedor. Mantener la semántica del código cliente es equivalente a mantener la funcionalidad pública dada por el proveedor antes del cambio.

La cuarta premisa destaca que no se debe presuponer que se tiene acceso a todo el código de un programa.

- ❖ **Def. 3-15.** *Premisa 4.* No es posible disponer de todo el código fuente de los módulos que constituyen un programa, tan sólo de sus interfaces. Tampoco es posible disponer de todo el código en tiempo de ejecución en presencia de bibliotecas dinámicas.

Pretender tener acceso a todo el código fuente que forma un programa no es realista a medida que aumenta su complejidad y tamaño. Implícitamente los sistemas de enlazado de bibliotecas estáticas y dinámicas siguen directamente esta premisa: están pensados para ensamblar código sin necesidad de tener acceso a los fuentes. Las bibliotecas dinámicas imponen una limitación adicional: En tiempo de ejecución sólo parte del código de una aplicación estará cargado en un momento dado.

En general, un programa usará módulos de terceras partes para disminuir los costes y obtener funcionalidades sobre las que no se tiene la experiencia o el tiempo suficiente para replicarlas. Adicionalmente, toda aplicación enlazará con bibliotecas preexistentes del sistema operativo o de terceros. Estas bibliotecas disponibles no son en general objeto de esta premisa, puesto que hoy en día se desarrollan con lenguajes como C o C++, cuyas reglas para la descomposición modular son débiles.

3.4 El modelo de componentes de XC

En el capítulo 2, se ha definido un componente (Def. 2-11) como un programa que sigue un modelo de componentes, que puede ser independientemente desplegado y componerse sin modificación de acuerdo con un estándar de composición. Existen así tres aspectos clave en la definición de componentes: *el modelo de componentes*, *el despliegue independiente* y *el mecanismo de composición*. Un modelo de componentes es definido en (Def. 2-13) como un conjunto de reglas de composición e interacción a través de interfaces que han de seguir los componentes, siendo la interfaz (Def. 2-9) una abstracción del comportamiento de un componente. El despliegue independiente corresponde con la posibilidad de modificación de un componente independientemente a otros, es decir, con la posibilidad de sustituir un componente por otra versión del mismo en un programa o añadir un nuevo componente a un conjunto existente. Las reglas de composición se refieren a los mecanismos de llamada entre componentes y cómo un componente hace referencia a otro, bien porque lo contiene, bien porque lo desea usar. El resultado de la composición, definido en (Def. 2-10), es un nuevo componente con un comportamiento distinto al de los originales.

Ciertas propiedades de los componentes son similares a las de los módulos independientes. Ambos deben tener una interfaz y una implementación, ambos deben poder modificarse y desplegarse separadamente, y los módulos pueden servir para limitar la visibilidad de las llamadas entre componentes. Las definiciones asociadas con componentes tienen su origen en una analogía con componentes físicos en otras disciplinas. En cambio, las definiciones asociadas a módulos surgen a partir de conceptos de programación. Aun cuando ambos conjuntos de definiciones tienen orígenes dispares, existen similitudes importantes en el tipo de propiedades deseables para ellos. Es posible obtener provecho de esas similitudes para reducir la complejidad de la implementación de componentes y módulos. Un módulo, como se verá en la definición (Def. 3-16), es una estructura sintáctica de un lenguaje de programación que puede ser usada para implementar componentes y conferirles automáticamente parte de sus propiedades. Szyperski describe la relación entre módulos y la tecnología de componentes como sigue:

“La modularidad no es un nuevo concepto, y ciertamente es un prerequisito para la tecnología de componentes. Desafortunadamente, la vasta mayoría de soluciones *software* de hoy no son siquiera modulares. [...] Adoptar la tecnología de componentes requiere la adopción de los principios de independencia y dependencias explícitas controladas. La tecnología de componentes lleva inevitablemente a soluciones modulares. Los beneficios de ingeniería del *software* pueden ser suficientes para justificar la inversión inicial en tecnología de componentes, incluso cuando los mercados de componentes no se prevén a medio plazo.” [Szyperski, 1998; p. 33].

Los componentes tienen ciertas propiedades que no pueden describirse bien únicamente con módulos. Por ejemplo, por conveniencia a la hora de manipular componentes desde un lenguaje de programación, una referencia a un componente debe describirse mediante un tipo de datos concreto en un lenguaje de programación. De esta manera se permite al compilador del lenguaje hacer comprobaciones automáticas acerca del uso del componente a partir de la interfaz que este último ofrezca. Los modelos de componentes actuales como COM, CORBA o EJB incluyen código especial adicional que trata a las referencias a componentes como tipos de datos dentro de lenguajes de programación. Este código es superfluo si está implícitamente incorporado a cómo un lenguaje de programación manipula los componentes. Un módulo, por el contrario, no es usualmente un tipo de datos⁵, sino un contenedor de otras declaraciones y definiciones, que da unas reglas de ámbito y un espacio de nombres separado.

En definitiva, para facilitar la interacción con componentes, éstos deben asemejarse a tipos de datos del lenguaje. Esta conclusión amplía el problema original de soportar un modelo de componentes a la cuestión de cuál debe ser el sistema de tipos del lenguaje, incluyendo qué estructuras de datos se representan a través de tipos de datos. Dada su importancia y debido a las distintas opciones posibles —muchas de las cuales deberán ser descartadas— se le dedicará un capítulo entero más adelante.

Los lenguajes orientados a objetos proporcionan hoy en día una buena abstracción acerca de cómo implementar un tipo de datos. Dependiendo del lenguaje y las características particulares de cada modelo de objetos, esta abstracción es mejor o peor. Ahora bien, estos modelos son buenos candidatos para implementar un tipo de datos de componentes porque representan más sencillamente la imagen en tiempo de ejecución de los componentes. Modelos de componentes actuales como COM, CORBA o EJB siguen esta misma línea de razonamiento. Las reglas de composición son implementables con mayor facilidad a partir de un modelo de objetos, tal y como también han demostrado los modelos de componentes existentes. La construcción de componentes compuestos se puede realizar almacenando referencias a esos componentes en un nuevo componente contenedor, y la interacción de unos componentes con otros es posible con llamadas a las operaciones que definen. Estas llamadas, sin embargo, precisan una mayor atención de la que habitualmente se da a las operaciones asociadas a los objetos. Una definición cuidadosa ayudará a simplificar los mecanismos de llamada a componentes, si los mecanismos de llamada a objetos se diseñan teniendo en cuenta el potencial uso de componentes.

⁵ Un contraejemplo es Eiffel, donde el concepto de *clase* incluye además el concepto de módulo. En el capítulo 5 estudiaremos por qué es interesante que ambos conceptos se mantengan separados.

3.4.1 Implementación del modelo

¿Cómo se implementa el modelo de componentes? Las propiedades de un componente son en cierto modo abstractas, relacionadas más con lo que se espera que un componente sea desde el punto de vista de su programación. Sobre esas propiedades existen diferentes opciones de implementación. Hemos visto por un lado que es interesante asociar parte de las propiedades de un componente a las propiedades de los módulos. Por otro lado, hemos visto también que es conveniente que un componente se describa como un tipo de datos, y que éste pueda implementarse mediante objetos. Por último, hemos visto que los objetos son buenos candidatos para implementar los mecanismos de composición.

La Figura 3-1 esquematiza los elementos básicos de un modelo de componentes: interfaz de componente, implementación, mecanismos de composición —llamadas y referencias a componentes— y procedimientos de despliegue independiente. Se construyen a partir de interfaces, módulos, objetos y el soporte en tiempo de ejecución del lenguaje.

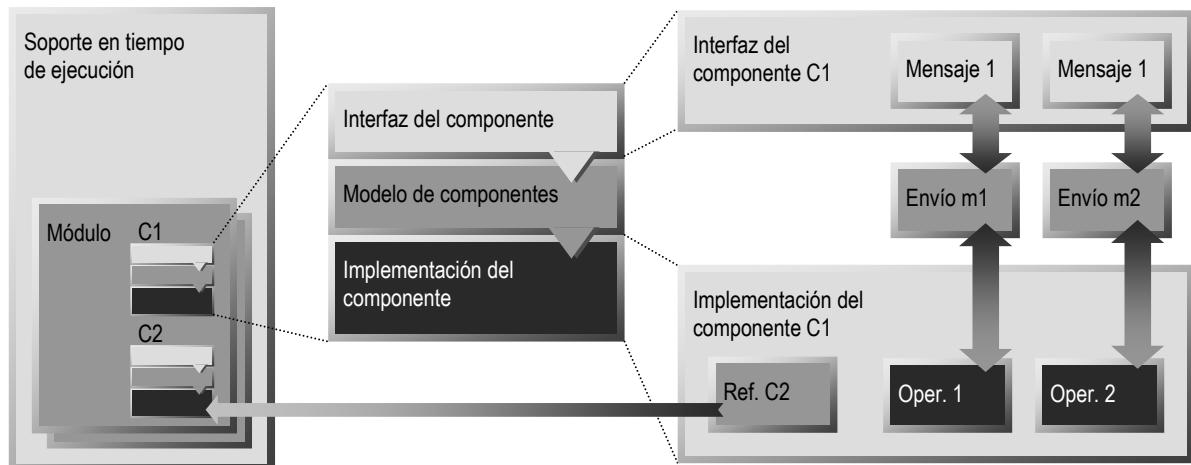


Figura 3-1 Implementación de componentes.

El modelo de componentes, en gris medio, gestiona cómo los componentes están organizados. Las definiciones de componentes, su interfaz e implementación, se registran mediante módulos. Las llamadas se gestionan mediante el envío de mensajes a las implementaciones de componentes. La composición se implementa con referencias a otros componentes registrados en el soporte en tiempo de ejecución.

El modelo de componentes se implementa dentro del soporte en tiempo de ejecución, en la definición de módulos, en la definición de los mecanismos de llamada y referencia de componentes, y en la definición del modelo de objetos subyacente, que es el que controla la implementación final de las operaciones de un componente. Además, el modelo de componentes está directamente soportado por la sintaxis y semántica del lenguaje de programación.

El soporte en tiempo de ejecución del lenguaje se encarga de la carga y registro de módulos, así como de todos los elementos declarados dentro de un módulo, como por ejemplo los componentes. El módulo es la unidad básica para desplegar componentes a otros sistemas. Cada componente está descrito por una interfaz y una implementación, enlazados por un mecanismo de envío de mensajes. Este mecanismo se encarga de traducir una petición realizada por un cliente a un protocolo de un componente proveedor, con la operación asociada en el proveedor.

3.4.2 Módulos

Los módulos son útiles para describir parte de las propiedades de los componentes porque resuelven muchos aspectos sintácticos relacionados con la definición y el uso de componentes, y son una unidad manejable con relativa comodidad dentro de un programa.

- ❖ **Def. 3-16.** Un *módulo* es una estructura sintáctica de un lenguaje de programación que crea una partición sintáctica de un programa, un espacio de nombres, un lugar donde declarar elementos de un programa relacionados (cohesionados), unas reglas de ámbito para la resolución de referencias a otras partes del programa, y una unidad de compilación y carga separada.

En el aspecto sintáctico, quizá la ventaja más importante sea dar una organización jerárquica a un programa. La Figura 3-2 muestra una imagen general del papel de los módulos en el nuevo lenguaje y en el modelo de componentes. Los módulos constan de una declaración o interfaz y una implementación. Dentro de un módulo se pueden declarar y definir submódulos. Las declaraciones sirven para identificar qué símbolos de un programa son visibles a otras partes, correspondiendo con la funcionalidad explícita. El lenguaje incluye reglas de ámbito que aseguran que un módulo sólo puede depender de la información proveniente de las interfaces de otros módulos.

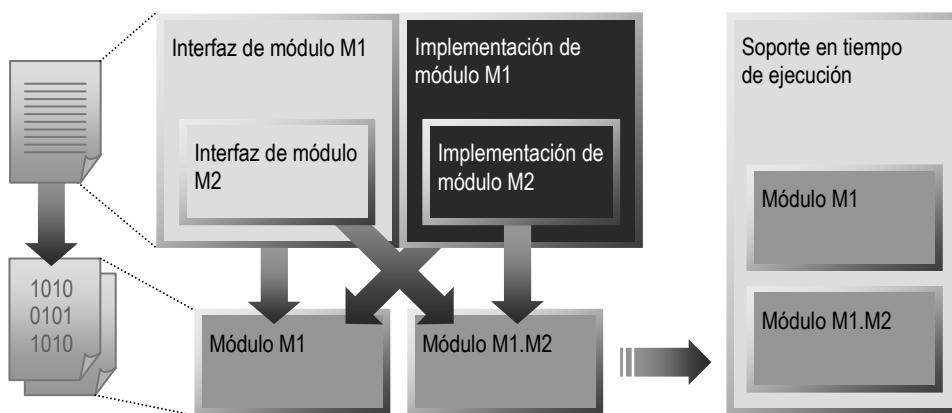


Figura 3-2 Estructura modular.

La estructura modular sintáctica facilita la organización jerárquica de un programa complejo y aplicar reglas de ámbito y visibilidad. Cada módulo es compilado independientemente. Es cargado y registrado también de forma independiente dentro del soporte en tiempo de ejecución.

Los módulos son compilados separadamente, para que se puedan cargar y registrar independientemente dentro de un programa en ejecución. El proceso de carga de un módulo comprueba que todos los módulos dependientes se encuentren ya correctamente registrados. Los módulos son una mejor unidad de carga que una clase u objeto por dos razones: (1) un módulo puede cargar de una sola vez varios componentes u objetos relacionados que deben cargarse en sincronía, y (2) un módulo comprueba dependencias con otros módulos con un nivel de granularidad más apropiado —grupos de declaraciones relacionadas— en vez de comprobar las dependencias de cada elemento de un módulo por separado. Una vez que un módulo ha sido registrado, éste se vuelve accesible a otros módulos a través del soporte en tiempo de ejecución.

3.4.3 El modelo de objetos subyacente

Otra parte de las propiedades de un componente puede expresarse con naturalidad mediante un modelo de objetos. En primer lugar, daremos unas definiciones preliminares acerca de qué es un objeto y un modelo de objetos.

- ❖ **Def. 3-17.** Un *objeto* es una abstracción de una estructura de datos y las operaciones que manipulan esa estructura.
- ❖ **Def. 3-18.** Una *clase de objetos* es una plantilla que describe las propiedades e implementación de un conjunto de objetos similares. En algunos lenguajes la clase puede ser un objeto separado. No todos los lenguajes orientados a objetos soportan clases.
- ❖ **Def. 3-19.** En un lenguaje que tenga clases, un objeto se construye a partir de una clase mediante un proceso llamado *instanciación*.
- ❖ **Def. 3-20.** En un lenguaje que no tenga clases, un objeto se construye a partir de otro mediante la copia o *clonación* de este último.
- ❖ **Def. 3-21.** Un *modelo de objetos* es una descripción de las propiedades de los objetos y cómo se implementan éstos en un lenguaje de programación. Una propiedad que describe el modelo de objetos es, por ejemplo, si usa clases o no.
- ❖ **Def. 3-22.** Un *lenguaje orientado a objetos* es un lenguaje que directamente soporta en su sintaxis y semántica un modelo de objetos.

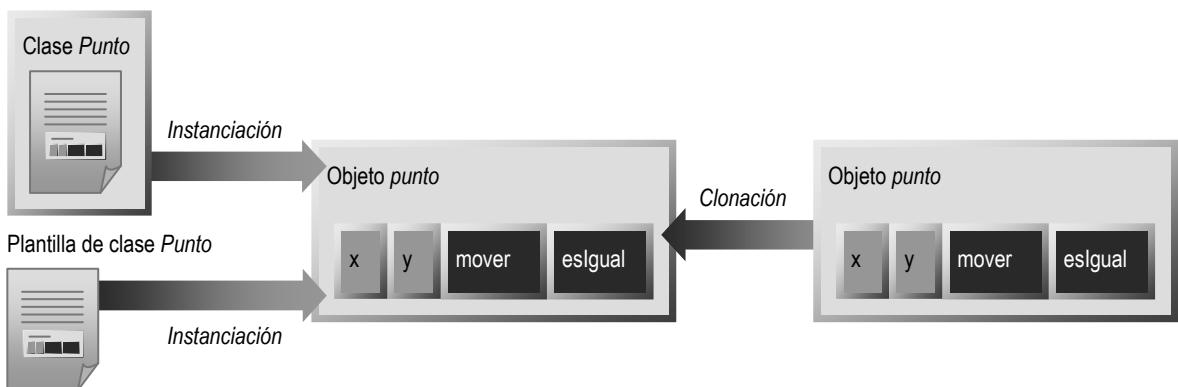


Figura 3-3 Objetos en un modelo de objetos.

Un objeto se instancia con una clase en lenguajes basados en clases. La clase puede ser un objeto o sólo una plantilla. Un objeto se construye como una copia de otro en lenguajes que no soportan directamente clases.

La Figura 3-3 ilustra las definiciones anteriores. La abstracción punto contiene las coordenadas del punto y operaciones que manipulan las coordenadas. Pueden ser instanciados a partir de clases que definen las características de estos puntos, o clonados a partir de otros puntos preexistentes, dependiendo del modelo de objetos. El resultado es siempre un nuevo objeto.

En los modelos actuales de componentes, es habitual que exista una correspondencia entre componentes y los objetos soportados por lenguajes de programación. Esta relación es debida a que tanto los componentes como los objetos describen unos datos y definen un conjunto de operaciones para manipular esos datos. Técnicamente, un componente se puede implementar sin necesidad de usar objetos. Es suficiente con que facilite una interfaz de composición y se pueda desplegar separadamente en forma binaria. Una subrutina, un objeto, un módulo o incluso una aplicación completa puede considerarse un componente [Szyperski, 1998; p. 4]. Una ventaja de los objetos es que dan una implementación por defecto que simplifica la implementación de componentes, dotando a la manipulación de componentes de un mayor nivel de abstracción.

La Figura 3-4 ilustra este último punto. En general, la estructura básica de un componente sigue siendo un conjunto de procedimientos y definiciones de datos empaquetados en una biblioteca estática o dinámica. Como técnica de implementación, la biblioteca —aunque necesaria— es insuficiente, porque sus elementos no están relacionados expresamente. Usando programación estructurada, un componente se puede construir agrupando un conjunto de procedimientos que manipulen alguna

estructura de datos. Estos procedimientos se hacen públicos a través de la biblioteca. El defecto obvio de esta opción es que tanto la estructura de datos como los procedimientos no están explícitamente interrelacionados y la relación debe mantenerse manualmente. Los modelos de objetos precisamente realizan automáticamente la asociación entre procedimientos y datos relacionados. La visión orientada a objetos hace explícitas esas relaciones, aumentando el nivel de abstracción. En la Figura 3-4 puede verse la correspondencia entre objetos y la biblioteca. El modelo de objetos esconde esta última relación, eliminando parte de la complejidad percibida del conjunto y haciendo más práctico el uso de componentes. Igualmente, es conveniente que los datos de los componentes —y por extensión los datos de los objetos que los implementan— se mantengan privados. Para ello, los objetos deben poder manipularse únicamente mediante operaciones que abstraigan los datos. El conjunto de las operaciones aplicadas a un grupo de datos particular constituye un tipo de datos. La visión de tipos de datos tiene mayor abstracción que aquella donde los objetos no protejan la manipulación directa de los datos. La abstracción con tipos de datos establece la interfaz del componente, haciéndola más independiente de una implementación particular. En la Figura 3-4 se observa que el tipo de datos no hace referencia directa a los datos, quedando éstos escondidos y simplificando más las relaciones que se han de tener en cuenta al tratar con componentes. Los tipos de datos simplifican la construcción y manipulación de componentes compuestos mediante el uso de referencias a otros componentes, ya que sólo es necesario crear o usar referencias a los tipos de datos asociados a los componentes, y son controlables por los lenguajes de programación.

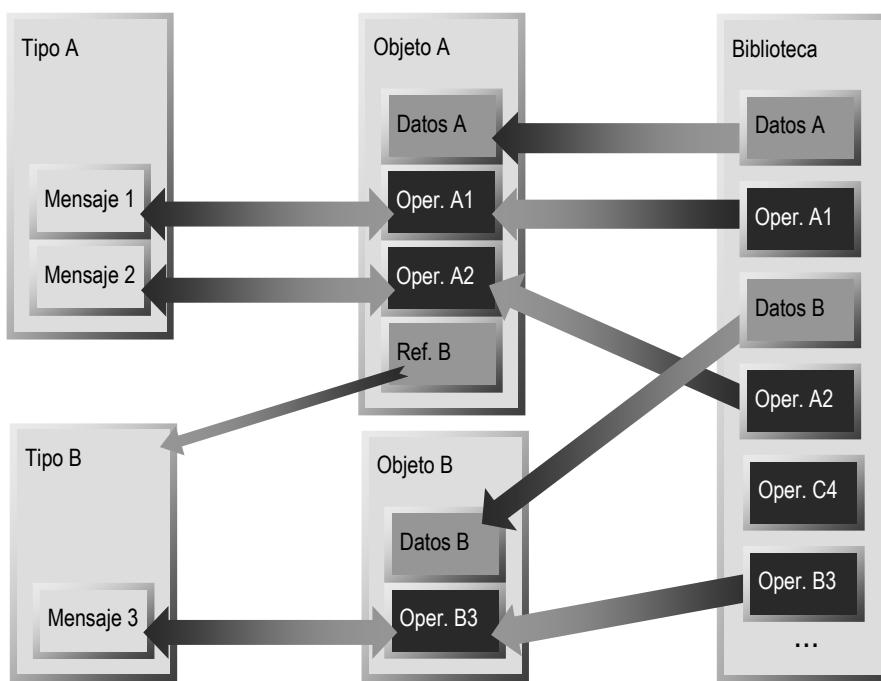


Figura 3-4 El modelo de objetos subyacente.

Un componente se define en programación estructurada con una biblioteca, pero sus elementos no están explícitamente relacionados. Un punto de vista orientado a objetos de los componentes es más elaborado, relacionando los elementos de la biblioteca automáticamente. Una perspectiva de tipos de datos trata a los componentes como entes funcionales de más alto nivel aún, que esconde la implementación y el acceso a los datos.

Si un componente se implementa con un objeto, ¿cuál es entonces la diferencia entre un componente y un objeto? La diferencia más importante es que un componente no tiene un estado, pero un objeto sí. Si un componente se implementa con objetos, el objeto —u objetos— se corresponde con la imagen en tiempo de ejecución del componente. Lo que implica que generalmente existe una copia de un componente en una aplicación y, en cambio, pueden existir muchas copias de los objetos que representan un componente en tiempo de ejecución [Szyperski, 1998; pp. 30-31]. Además, un

componente no tiene porqué estar implementado con objetos, ni los objetos han de implementar siempre componentes. Esta circunstancia da lugar a implementaciones de modelos de objetos que no tienen en cuenta los problemas y características asociadas a los componentes, haciendo más compleja la implementación de componentes usando esos objetos.

Por ejemplo, en C++ la modificación de una clase de objetos muchas veces requiere la recompilación del programa receptor de la nueva clase modificada, dificultando el despliegue final de componentes modificados. Muchos lenguajes orientados a objetos ni siquiera proporcionan un concepto explícito de interfaz o protocolo de un objeto, como por ejemplo C++, Eiffel, Smalltalk-80 o SELF. En C++ y Eiffel la definición del tipo de objeto —una *clase* de objetos— sirve también como declaración. En Smalltalk-80 y SELF no existe siquiera el concepto de declaración como tal, tan solo un conjunto de definiciones compiladas disponibles directamente en el soporte en tiempo de ejecución. En Java sí existe el concepto de interfaz, aunque una definición de una clase de objetos actúa también como declaración, mezclándose ambos conceptos. Esta situación es patente en la especificación de componentes JavaBeans vista en el apartado 2.5.2.

3.4.4 Envío de mensajes

El envío de mensajes es un concepto esencial de un modelo de objetos y muchos modelos de componentes. Equivale a una llamada a subrutina en programación estructurada. La nomenclatura usada a continuación fue introducida originalmente por Smalltalk-80 [Goldberg & Robson, 1989; pp. 6-8]. En los modelos de objetos y modelos de componentes que usen correspondencias a objetos, se les atribuye conceptualmente una identificación única y la ilusión de reaccionar ante eventos, como una metáfora con los objetos y componentes del mundo real. El centro de atención es el objeto o componente. La Figura 3-5 representa esta idea.

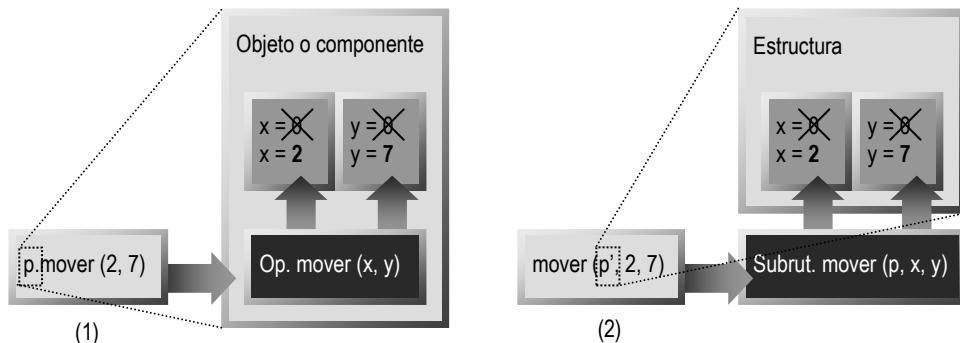


Figura 3-5 Énfasis en programación orientada a objetos y en programación estructurada.

Durante una operación de movimiento de un punto, en (1) el objeto p es el elemento relevante, en (2) es la subrutina mover. La estructura p' es sólo un parámetro más.

En programación orientada a objetos (1), si se desea mover un objeto gráfico p a otra localización, el objeto recibe un *mensaje* de movimiento indicando las coordenadas de una nueva posición. El objeto reacciona ante ese evento cambiando a la posición indicada. En programación estructurada (2), por el contrario, una subrutina de movimiento es la encargada de actualizar la posición de una estructura de datos p' pasada como parámetro. Aquí el énfasis se encuentra en la subrutina. El esquema de razonamiento orientado a objetos es muy natural y cercano a la experiencia diaria, a la vez que efectivo a la hora de analizar programas orientados a objetos o compuestos por componentes. Obviamente, es la implementación particular del modelo de objetos o componentes la que se encarga de hacer creíble la ilusión de tener entes que reaccionan ante mensajes, de la misma forma que un entorno de ventanas hace verosímil la ilusión de estar manipulando ventanas que se puedan mover y apilar como en un escritorio.

- ❖ **Def. 3-23.** Un *mensaje* describe una operación de un objeto o un componente, incluyendo parámetros y valor de retorno opcionales.
- ❖ **Def. 3-24.** El *envío de mensajes* es un procedimiento de petición de ejecución de una operación a un objeto o un componente.
- ❖ **Def. 3-25.** Un *receptor* es un objeto o un componente que recibe mensajes.
- ❖ **Def. 3-26.** Un *método* es una implementación de una operación en un objeto⁶. Si un componente se implementa con objetos, entonces un método también corresponde con una operación de un componente.
- ❖ **Def. 3-27.** La *resolución de un método* es un procedimiento que intenta encontrar un método asociado a una operación de un objeto o un componente, a partir de la petición de ejecución de esa operación mediante un mensaje.
- ❖ **Def. 3-28.** Un objeto o componente *responde a un mensaje*, si el procedimiento de resolución encuentra un método asociado al mensaje pedido.
- ❖ **Def. 3-29.** La *invocación de un método* es el procedimiento de ejecución de un método en el contexto de un objeto o componente particular.

Los eventos corresponden con el envío de mensajes a los objetos o componentes. La Figura 3-6 muestra la relación entre las definiciones asociadas con los mensajes, su envío y su resolución.

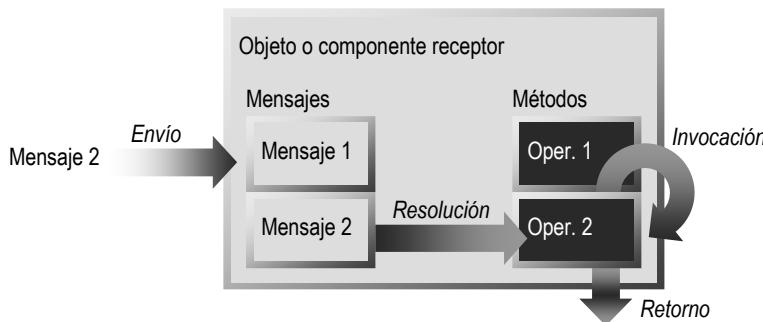


Figura 3-6 Elementos de un envío de mensajes.

Un mensaje se envía al objeto receptor, que resuelve primero el método exacto que ha de ejecutar y luego invoca el método en cuestión.

El procedimiento de envío de mensajes se aplica tanto a objetos como a componentes, existiendo múltiples implementaciones diferentes del mismo que equilibran flexibilidad y eficiencia. Por ejemplo, en el caso más eficiente el procedimiento de envío de mensajes puede degenerar a una llamada a subrutina, siempre que se asegure que existe una relación 1:1 entre un mensaje y una operación. En un modelo de objetos o de componentes muchas veces no se puede asumir esa relación. Un mismo mensaje puede hacer referencia a dos o más operaciones, como por ejemplo dos implementaciones distintas del mismo protocolo de un objeto o componente. En este caso la relación es 1:n.

Existe, no obstante, una diferencia significativa en la definición de un procedimiento de envío de mensajes para componentes que no es aplicable por igual a un lenguaje orientado a objetos. En un modelo de componentes, un componente debe poseer una interfaz independiente de una implementación, necesaria para poder cambiar posteriormente un componente por otro distinto o por una nueva versión de forma independiente. En muchos modelos de objetos no existe esta restricción. Como se vio en el apartado anterior, muchos lenguajes de programación —C++, Eiffel, Smalltalk-80 y SELF eran ejemplos— no incluyen el concepto de interfaz o protocolo de objetos explícitamente. En estos lenguajes, la implementación de un modelo de componentes va más allá de las posibilidades dadas por el lenguaje y debe construirse sobre él. La existencia de un protocolo separado tiene mucho

⁶ En C++ se llama a un método *función miembro*.

que ver con el modo en que se realiza la asignación de tipos de datos a los componentes en un lenguaje orientado a objetos. La Figura 3-7 ilustra este importante aspecto.

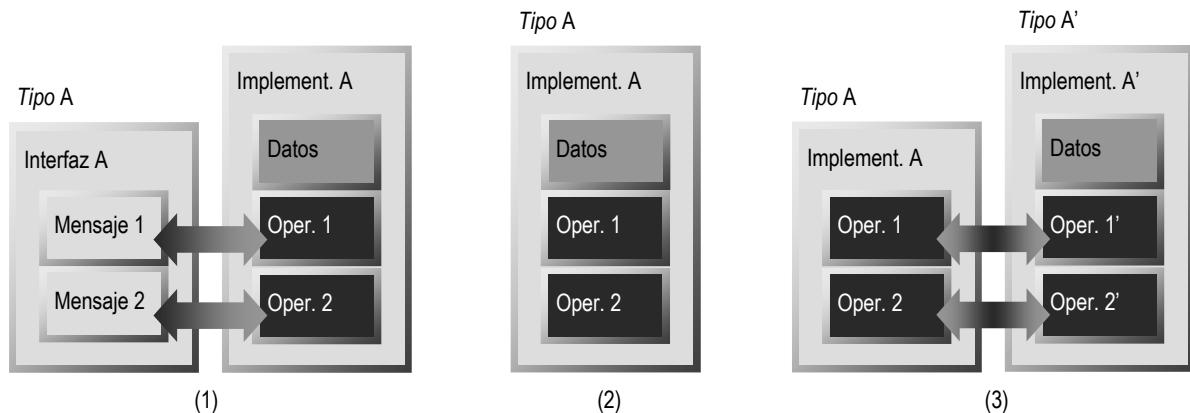


Figura 3-7 Envío de mensajes en objetos y componentes.

Una definición explícita de interfaz asigna un tipo al protocolo independientemente de la implementación (1). Una definición implícita de interfaz obliga a renunciar a la implementación independiente (2) o a construir una implementación específica únicamente para mantener el tipo de la interfaz (3).

Si el lenguaje soporta interfaces o protocolos explícitos, entonces existe la posibilidad de asignar el tipo de datos del componente a la interfaz, manteniendo clara la separación entre interfaz e implementación, mostrado con (1) en la Figura 3-7. Si no existe esta separación, entonces los tipos de datos son proporcionados por clases de objetos. Pero al corresponder una clase con una descripción de la implementación de un objeto, se pierde la distinción entre interfaz e implementación desde el punto de vista de los tipos, como se ve en (2). Para recuperarlo es necesario crear una clase adicional que haga las veces de interfaz, y conectar la implementación de esta nueva clase con la implementación real del componente (3). La emulación de la interfaz en (3) es incompleta, porque tanto **A** como **A'** definen tipos de datos, mientras que en (1) sólo existe el tipo **A**.

Se puede objetar al razonamiento anterior que la herencia en lenguajes con clases permite tener distintas implementaciones para un mismo tipo **A**. La herencia se verá en mayor detalle en el capítulo 4, pero ahora se puede dar una breve explicación más intuitiva a porqué la herencia no resuelve bien el problema de los tipos. La herencia admite que dos clases **A** y **B** posean el mismo tipo **A**, si la clase **B** hereda de la clase **A**. En *herencia simple* sólo se puede heredar de una clase, por lo que se limita mucho la posibilidad de una implementación independiente de **B**. Obligatoriamente **B** ha de heredar de **A**, no pudiendo **B** obtener ventajas de implementación heredadas de otra clase **C** quizás más adecuada. En *herencia múltiple*, se admite que una clase herede de varias clases. Entonces, en principio puede parecer que **B** sólo necesita heredar de **A** y **C** simultáneamente para resolver el problema de los tipos. La desventaja de esta opción es que la herencia múltiple trae consigo sus propias complicaciones, como el uso de clases *mixin* intermedias, o el hecho de que se están heredando implementaciones, no sólo interfaces.

La configuración de doble implementación de la opción (3) de la Figura 3-7 es en realidad muy habitual. Es usada por ejemplo en los modelos de componentes CORBA y COM. La implementación adicional suele ser automatizable si se dispone de alguna descripción de la interfaz que ha de cumplir, y recibe el nombre de *proxy*, *stub* o *skeleton*. EJB usa una variante que mezcla doble implementación e interfaces explícitas.

3.4.5 Tipos de extensiones

Los componentes, al igual que los objetos, son entes que requieren modificaciones con el tiempo, necesarias para adaptarlos a nuevos requisitos o a cambios en los requisitos originales. Una

posible solución a esta necesidad de cambio es sustituir un componente u objeto por otro completamente distinto. Se trata de una aproximación drástica al problema que lleva asociado un coste y, si bien puede ser útil algunas veces, es más razonable pensar que parte del componente u objeto original se puede reutilizar para disminuir el coste del cambio. En este último caso hablamos de *extensiones* a un componente o un objeto. El modelo de componentes y el modelo de objetos deben soportar algún tipo de extensión de este último estilo. Obviamente, en ambos modelos es factible aplicar también la solución drástica.

3.4.5.1 Extensiones planificadas y no planificadas

Las extensiones las podemos clasificar en dos tipos: no planificadas y planificadas. De las primeras existen cuatro tipos importantes: herencia, composición, categorización y extensiones ortogonales.

- ❖ **Def. 3-30.** Una extensión es *no planificada* si en el diseño de un elemento *software* no especifica que una extensión puede ocurrir, pero permite que ocurra.

Las extensiones no planificadas son puntos de variabilidad en un programa no conocidos completamente. Un puntero a función, el registro de una llamada de retorno, una biblioteca dinámica cargada en tiempo de ejecución, o un controlador de dispositivo de un sistema operativo, son tipos de extensiones no planificadas. Estas extensiones son más flexibles que las planificadas. La razón es sencilla: potencialmente adaptan programas existentes a situaciones desconocidas cuando se diseñaron. Su principal ventaja es también su mayor inconveniente. El problema de las extensiones no planificadas es que no se sabe qué es lo que van a hacer. Es posible que una extensión sea completamente incompatible con el elemento *software* o sistema original, por lo que resulta imperativo asegurar que tal incompatibilidad no se produce. Pero, como se vio en el apartado 2.7.1.6 *Reemplazo de elementos software en la práctica*, se trata de algo bastante más fácil de decir que de hacer.

- ❖ **Def. 3-31.** Una extensión es *planificada* si expresamente en el diseño de un elemento *software* se prevé una extensión y se proveen los mecanismos adecuados para que ésta ocurra.

Una extensión planificada es un punto de variabilidad conocido [Czarnecki & Eisenecker, 2000; p. 75-76], un elemento opcional al sistema que muchas veces puede verificarse con antelación. En un compilador por ejemplo, extensiones planificadas son las opciones de compilación. Las extensiones planificadas y su estudio explícito están ganando aceptación en los sistemas informáticos en los que se aplica la técnica de modelado de funciones o *feature modeling* de ingeniería de dominios [Czarnecki & Eisenecker, 2000; pp. 82-130], como por ejemplo las líneas de productos [Clements & Northrop, 2002]. Una analogía típica usada en estos ámbitos acerca de los puntos de variabilidad se hace con los extras de un vehículo: el climatizador, los tipos de llantas o las tapicerías, son puntos de variación conocidos y probados de un vehículo.

3.4.5.2 Herencia y composición

Los modelos de objetos se han hecho muy populares por facilitar un modo ingenioso de extensión de la implementación de una objeto o clase: la herencia.

- ❖ **Def. 3-32.** La *herencia* en un lenguaje orientado a objetos es un procedimiento que permite a un objeto o una clase recibir características de otro objeto o clase preexistente. A partir de ese momento el nuevo objeto o clase puede realizar cambios incrementales sobre las características heredadas.
- ❖ **Def. 3-33.** Se dice que un objeto o clase *deriva* o *hereda* de otro si ambos están relacionados por la herencia y el primero obtiene características del segundo.

La herencia es un mecanismo de extensión no planificada muy potente, útil y complejo, que evita comenzar desde cero la definición de un nuevo objeto o clase. Si los cambios que se precisan son

pocos, se puede reusar casi toda la implementación heredada y—más importante—el coste percibido del proceso es inicialmente muy bajo. Ahí radica su popularidad. La herencia cuenta con variantes de muchos tipos, cada una con sus características particulares, ventajas e inconvenientes. Sin entrar en detalles que se verán en el capítulo 4, es interesante resaltar ahora que a la herencia se la considera *reuso de caja blanca* [Gama, Helm, Johnson & Vlissides, 1995; p. 19], porque los detalles de implementación heredados acaban estando presentes durante la construcción del objeto o clase derivada.

Una alternativa a la herencia es la composición de objetos. Este estilo de reutilización es llamado *reuso de caja negra* [Gama, Helm, Johnson & Vlissides, 1995; p. 19]. Por sus características es muy apropiado para los modelos de componentes. En el capítulo 4 se verá que no es buena idea en general que un modelo de componentes soporte herencia de implementación como mecanismo fundamental de extensión, puesto que la herencia muestra detalles de la implementación heredada [Snyder, 1986a; pp. 40-44]

3.4.5.3 Categorización

Desde el punto de vista de un modelo de componentes, existe una tercera opción: extender el propio componente directamente con funcionalidad adicional, extensión que trata al componente original como encapsulado. Esta aproximación también se puede aplicar a los objetos. Reciben el nombre de categorías por analogía con una agrupación informal de métodos en las clases de Smalltalk-80, que originalmente tenía ese nombre.

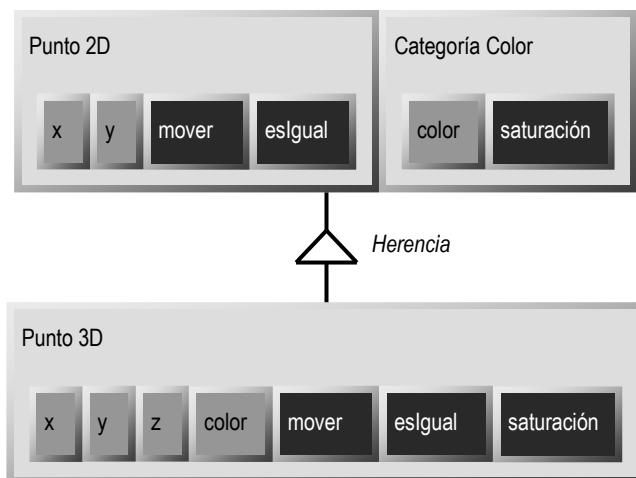


Figura 3-8 Extensiones no planificadas: herencia y categorías.

La herencia genera nuevas descripciones de objetos, las categorías extienden descripciones existentes.

La categorización se encuentra implementada en Objective-C [NeXT, 1995] y en el lenguaje Tom [Schoenmakers, 2000], donde recibe el nombre de extensiones. En Smalltalk-80, CLOS [Keene, 1989] y en lenguajes basados en prototipos como SELF o Javascript (ECMAScript) [ECMA, 1999], se simulan modificando dinámicamente la estructura del objeto o clase. La Figura 3-8 esquematiza los conceptos de herencia y categorías, ambas extensiones no planificadas. La herencia define incrementalmente nuevos objetos o clases, las categorías facilitan la definición incremental de un objeto o clase existente.

Por último, conviene hacer notar que las técnicas de extensiones no planificadas como la herencia o las categorías sirven para implementar extensiones planificadas, siempre que quede claro cuáles son las extensiones válidas y no aceptar cualquier extensión posible. Es una estrategia muy usada en ingeniería de dominios.

3.4.5.4 Extensiones ortogonales

Existe otro tipo de extensiones que llamaremos *ortogonales*. Corresponden con funcionalidad extra que no afecta directamente a la funcionalidad de un elemento *software*. Un ejemplo servirá para introducir este concepto. Supongamos un vector que almacena información en una aplicación con un solo hilo de ejecución o *thread*. Su implementación es muy sencilla en este caso. Ahora supongamos que se desea usar el mismo vector entre varios hilos de ejecución clientes. Si tenemos acceso al código fuente del vector es posible poner bloqueos en las operaciones de actualización. Si no tenemos acceso al mismo, parece que no queda más remedio que poner los bloqueos en el código cliente que accede al vector, haciéndolo más complejo. Las extensiones ortogonales dan una tercera opción, que mantiene intacto el código original del vector y el código cliente en varios hilos de ejecución.

- ❖ **Def. 3-34.** Una extensión *ortogonal* admite la inclusión de un comportamiento en un elemento *software* que no forma parte de la funcionalidad original del mismo, y que *no* modifica el resultado obtenido por la funcionalidad original.

Las extensiones ortogonales son extensiones no planificadas. En los modelos de componentes las extensiones ortogonales tienen especial relevancia, porque por definición son extensiones *independientes* de la funcionalidad original dada por un componente, una cualidad muy apreciada en un modelo de componentes. Las extensiones ortogonales se han estudiado bastante en el contexto de la metaprogramación o programación con metaobjetos.

- ❖ **Def. 3-35.** Un *metaobjeto* es un objeto que intercepta y puede modificar el comportamiento de las operaciones de objetos normales, sin que estos últimos se den cuenta de ello.

El código original del elemento *software* que se ve modificando no prevé la nueva funcionalidad, que es añadida a posteriori. Las extensiones ortogonales reciben también el nombre de *protocolos de metaobjetos* [Kiczales, Rivières & Bobrow; 1991], si bien éstos en particular sí permiten la modificación del comportamiento de los objetos originales. Desde un criterio más modular, es preferible que la semántica de las extensiones ortogonales *no* afecte al comportamiento original de los objetos que extienden, para evitar interdependencias. La Figura 3-9 ilustra una extensión ortogonal mediante metaobjetos.

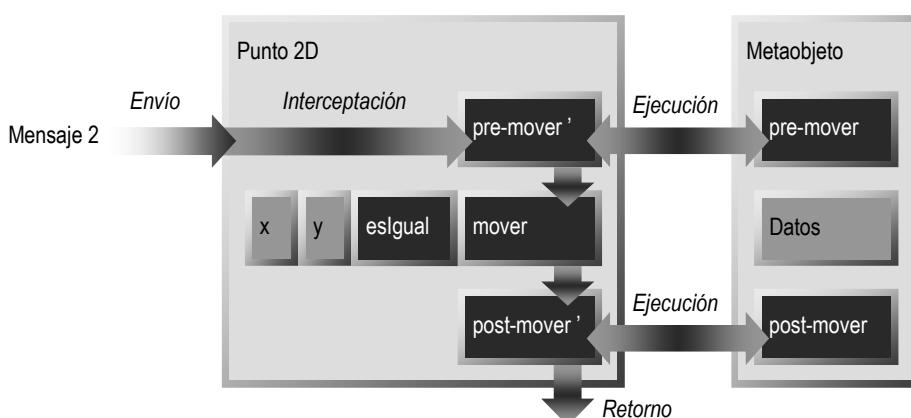


Figura 3-9 Extensiones ortogonales.

La extensión modifica el comportamiento de una operación de un objeto o clase interceptando algunas operaciones antes y después de ejecutarse. La interceptación permite ejecutar las acciones ortogonales del metaobjeto.

El metaobjeto introduce la nueva funcionalidad ortogonalmente mediante la interceptación del envío de mensajes a un objeto normal. El metaobjeto puede entonces realizar acciones adicionales a las realizadas originalmente por el objeto normal. Ejemplos sencillos de acciones ortogonales son contar el número de veces que se ha invocado el método *mover* en *Punto 2D*, o guardar las posiciones inicial y final de movimiento del punto para reproducirlas posteriormente. Nótese cómo la nueva funcionalidad

no modifica ni el propósito ni el resultado de la operación mover original. Estas cualidades hacen a las extensiones ortogonales idóneas para ser aplicadas a componentes. Sin una técnica de extensiones ortogonales, el código de cuenta de invocaciones y de almacenamiento de posiciones tendría que haberse incluido directamente en el método `mover`. Ejemplos más sofisticados de extensiones ortogonales incluyen persistencia, seguridad, serialización o soporte de múltiples hilos de ejecución.

Otra técnica de extensiones ortogonales son los *aspectos* [Kiczales et alii, 1997]. Los aspectos son un refinamiento de los protocolos de metaobjetos, en donde la funcionalidad ortogonal no tiene porqué implementarse mediante objetos completos, sino únicamente con métodos adicionales. Cada tipo de funcionalidad ortogonal es un aspecto. Por ejemplo, el tratamiento de excepciones comunes en múltiples partes del código de un programa puede centralizarse en un aspecto o distribuirse a lo largo de múltiples fragmentos de código por todo el programa. La ventaja de describir aspectos es que la funcionalidad cubierta por cada aspecto se encuentra centralizada en un único sitio y el código original queda más limpio. Los aspectos actualmente consienten cambios en el comportamiento original de los objetos o clases que modifican.

3.5 Programación de sistemas en XC

Este tipo de programación aparece a menudo a medida que los programas se hacen más complejos. También es importante para programas de bajo nivel como controladores de dispositivos, que pueden ser pequeños en extensión, pero que demandan características no habituales en una programación normal. Acceder a características del sistema operativo, bibliotecas, realizar optimizaciones o simplemente trabajar con un nivel de abstracción más bajo son otros ejemplos. Impedir el acceso a esas funcionalidades es un obstáculo para el uso práctico de cualquier lenguaje de propósito general, y que, en última instancia, puede llevar a desestimar el uso del lenguaje completamente por falta de idoneidad. El objetivo inicial de XC es dar soporte a la programación de sistemas sin degradar su capacidad para describir código de alto nivel. No pretende ser un lenguaje apto para entornos más exigentes, como sistemas empotrados o sistemas en tiempo real.

Las características que facilitan la programación de sistemas en XC son:

- Tipos integrales y booleanos optimizados. Ocupan un registro de máquina.
- Operaciones básicas óptimas: aritméticas, lógicas, binarias y de desplazamiento. Manipulan directamente registros de máquina.
- Compilación de tipos básicos, operaciones básicas y sentencias de control como bucles y sentencias condicionales a sus equivalentes en C, sin pérdida de eficiencia.
- Funciones con expansión en línea.
- Conversiones de tipo (*type casts*) no comprobadas por el compilador en código no seguro.
- Manipulación del tipo `Pointer` y aritmética de punteros en código no seguro. Direccionamiento de memoria de bajo nivel.
- Llamadas a funciones C sin sobrecarga con tipos básicos. Denominadas *funciones nativas*.
- Sintaxis de objetos para estructuras en C. Llamados *objetos nativos*. Pueden tener implementación mixta nativa y en XC. Tienen control explícito de creación y liberación de memoria.
- Modificador `volatile` en declaración de variables.

Aspectos de muy bajo nivel, como el acceso a dispositivos de entrada/salida o la gestión de interrupciones deben realizarse en ensamblador, y tener un registro de activación de función C para poder ser llamados como funciones nativas desde XC. Los saltos largos (`longjmp`) son accesibles a través de las bibliotecas básicas de C y usando funciones nativas, al igual que las funciones básicas de gestión de memoria.

Las llamadas a bibliotecas estáticas y dinámicas son la vía más habitual para acceder a las abstracciones fundamentales del sistema operativo. Gestión de procesos, hilos de ejecución, sincronización o acceso al sistema de ficheros son posibles usando funciones y objetos nativos. La funcionalidad nativa expone al lenguaje de programación código existente, evitando rescribirlo y simplificando la intercomunicación entre sistemas existentes o *legacy code* y los escritos en el nuevo lenguaje. También sirve para facilitar la manipulación desde dentro del lenguaje de código quizá optimizado escrito en otros lenguajes de programación como C o ensamblador.

Para reducir la complejidad percibida, debe ser posible crear objetos y componentes cuya implementación sea nativa pero cuya manipulación se realice desde dentro del lenguaje sin complicaciones adicionales. La sobrecarga de exponer la funcionalidad nativa al lenguaje debe minimizarse, así como los mecanismos de conversión de datos entre la funcionalidad nativa y los objetos o componentes del lenguaje. Si el lenguaje es capaz de generar código eficiente, entonces la necesidad de construir objetos nativos expuestos al lenguaje disminuye, pero no se elimina totalmente. En particular es especialmente crítico en tipos de datos básicos como los números enteros. Una implementación de enteros ineficiente puede incluso no ser reparable dentro del ámbito del lenguaje. Por ejemplo, simplemente no es práctico que una suma de dos enteros requiera una llamada a función, aunque ésta sea nativa.

Si la implementación de objetos y componentes es eficiente, y también lo es el acceso a funcionalidad nativa, entonces el lenguaje puede ser utilizado para simplificar la construcción de abstracciones consideradas usualmente de bajo nivel y facilitar la programación de sistemas. Por ejemplo, una tarjeta de red *plug-and-play* de un ordenador portátil que se conecte y desconecte dinámicamente precisa cargar y descargar su controlador de dispositivo, que registre los niveles inferiores de red y los elimine al desconectarse. Un nivel superior de red, como por ejemplo TCP/IP, no conoce a qué dispositivo de red se va a conectar, menos aún si éste cambia dinámicamente. Tan sólo conoce su protocolo. Una forma de resolver la carga dinámica de dispositivos en C consiste en emular mediante programación explícita el proceso de envío de mensajes mediante el registro y uso de punteros a funciones, pero el procedimiento es complejo y propenso a errores. Si la implementación es lo suficientemente eficiente, los conceptos de componentes, objetos e interfaces vistos en los apartados anteriores son aplicables igualmente al diseño de un sistema de carga dinámica de dispositivos de red, simplificando el código resultante.

3.6 Objetivos principales de XC

El estudio de cómo incorporar un modelo de componentes a un lenguaje de programación tiene el fin de allanar el camino hacia la construcción de sistemas informáticos basados en componentes mediante la simplificación de los conceptos en juego. Simplificación que se produce al unir los conceptos habitualmente otorgados a los componentes con aquellos que ha de resolver un lenguaje de programación, a menudo redundantes. Si el lenguaje prototipo es muy sencillo, no podrá evaluarse si en la práctica es factible el enfoque propuesto en este trabajo. Antes al contrario, la definición de un ámbito y unos objetivos realistas para la implementación del lenguaje es condición para evaluar seriamente la viabilidad de la solución presentada, y su utilidad como herramienta. A continuación se revisan las cualidades más importantes a tener en cuenta en el diseño de un lenguaje que vaya más allá de un mero ensayo: consistencia, robustez, escalabilidad y eficiencia.

3.6.1 Consistencia

Un lenguaje de programación describe una máquina virtual que presenta abstracciones de más alto nivel que una máquina real. Las abstracciones resuelven y esconden multitud de problemas de bajo nivel comunes, a la vez que suministran un marco de trabajo y de organización más cercano a la forma de pensar de las personas. Un lenguaje es consistente si provee un conjunto de abstracciones

claras, comprensibles y fáciles de usar. Un lenguaje poco consistente es también menos práctico. La consistencia es un factor técnico muy importante para la viabilidad de un lenguaje. Si bien puede complicar la implementación del mismo, los resultados obtenidos son mucho más satisfactorios y el lenguaje se describe con conceptos más simples.

Una abstracción que posea muchos casos especiales poco relacionados, o cuya manipulación requiera distintas técnicas dependiendo del caso, no es una abstracción consistente. Un ejemplo común son los números enteros en muchos lenguajes orientados a objetos. En lenguajes como Java o C++ existen dos variedades fundamentalmente distintas de números enteros: *enteros primitivos* y *objetos*. Los números enteros primitivos se representan con una palabra del procesador por razones de eficiencia. Los números enteros que son objetos se representan en cambio con una estructura de objeto. Ambos poseen distintas propiedades: los primeros son eficientes, pero no son objetos; los segundos no son eficientes, pero tienen todas las propiedades de los objetos. El resultado es que ambas variantes de números enteros no son intercambiables en expresiones y obliga a incluir manualmente código particular que identifique cada variante. Esta sobrecarga es significativa, por ejemplo, durante la manipulación de estructuras del lenguaje mediante reflexión. Smalltalk-80, Eiffel, C# y XC proporcionan en cambio una visión unificada de todos los números, manejándolos con eficiencia a la vez que dotándolos de todas las propiedades de los objetos. Es decir, si los objetos tienen un método `print` es posible escribir:

```
Integer value;
Object object;

value.print;
object.print;
```

La abstracción de números es más consistente y simple, porque no precisa la codificación de casos especiales. La implementación por parte del compilador es un poco más compleja: básicamente la información de tipo permite al compilador seleccionar una implementación eficiente para el tipo `Integer`.

La consistencia afecta también a aspectos sintácticos del lenguaje. Cada abstracción de un lenguaje posee una representación sintáctica. Un lenguaje es más consistente si da una única representación sintáctica para un concepto, independientemente de que ese concepto pueda implementarse de distintas maneras. Para entender su valor es ilustrativo revisar algunos casos prácticos. Un ejemplo obtenido de [Joyner, 1996; p. 26] es el operador de acceso a campos de una estructura u objeto. En C y C++ existen dos operadores de acceso a campos: ‘.’ y ‘->’ que se diferencian sólo en aspectos de implementación: dependen de si la variable declarada es una estructura o un puntero a una estructura.

```
struct Point { int x; int y; } point, *ptrPoint;
int a = point.x;
int b = ptrPoint->x;
```

El compilador *conoce* si una variable es un puntero o no, puesto que impide aplicar el operador erróneo a la variable. Una mejor opción de diseño habría sido simplemente tener un único operador de acceso y seleccionar internamente la implementación del acceso basándose en la información del tipo de la variable que el compilador de todas formas *ya tiene*. El uso de un único operador de acceso es más consistente porque representa el concepto de acceso con mayor simplicidad, sin casos especiales.

Otro ejemplo es el operador de ámbito ‘::’ de C++. El ámbito se gestiona separadamente al acceso a campos y expresamente se hace una distinción sintáctica. En la práctica, el concepto que se usa es el de jerarquía. Una definición de variable puede estar dentro de un ámbito y ésta tener un campo que a su vez tiene varios punteros a otros campos. En C++ se accedería al último campo como:

```
Ambito1::Ambito2::var.campo1->campo2
```

que es una representación compleja para un concepto muy simple: un acceso jerárquico. Esta deficiencia es subsanada en lenguajes como Java, C# o XC escribiendo una expresión más natural que representa la jerarquía:

```
Ambito1.Ambito2.var.campo1.campo2
```

Proporcionar conceptos demasiado simples es también fuente de inconsistencias. Por ejemplo Smalltalk-80 y LISP [Steele, 1990] son minimalistas en el uso de algunos conceptos, particularmente los sintácticos. Smalltalk-80 relega todas las operaciones a paso de mensajes, incluidas las sentencias de control, para exponer el mínimo de conceptos que soporten la programación orientada a objetos, desde los elementos más primitivos a grandes aplicaciones [Goldberg & Robson, 1989; p. ix]. Así, en Smalltalk-80 la operación matemática $2 + 3 * 4$ no produce el resultado esperado 14 —conocido desde los primeros cursos escolares— sino 20, que corresponde con $(2 + 3) * 4$. El ahorro conceptual en la *implementación* del compilador de Smalltalk-80 rompe las reglas aritméticas usuales en favor del receptor del mensaje, que siempre va delante del mensaje enviado. Como la evaluación de una expresión de mensajes es de izquierda a derecha, a no ser que se introduzcan paréntesis el mensaje de suma se encuentra antes que el de multiplicación y se ejecuta primero. Haciendo ligeramente más complejo el compilador, es posible identificar esas expresiones y, manteniendo las mismas abstracciones de mensajes, compilar las llamadas con la semántica usual de la aritmética. La ausencia de estructuras de control explícitas y simuladas mediante mensajes es otro ejemplo de economía de conceptos durante la implementación del compilador, que afecta a la legibilidad de los programas escritos en Smalltalk-80. Por ejemplo:

```
(a > b) ifTrue: [ ... ] ifFalse: [ ... ]
```

Una crítica similar se ha hecho tradicionalmente a LISP. Usa únicamente listas como sintaxis:

```
(if (> a b) ( ... ) ( ... ))
```

Contrastan con la más elegante de Pascal:

```
if a > b then ... else ... end
```

o la inusualmente clara de C⁷:

```
if (a > b) ... else ...
```

Otra vez, se trata de un mero aspecto sintáctico resoluble sin dificultad por el compilador. La sintaxis de mensajes de múltiples argumentos de Smalltalk-80 es más clara que la tradicional con estilo de función de C++ o Java. En Smalltalk-80 y Objective-C —que usa un estilo similar— un mensaje complejo suele poder leerse con mayor comodidad. Por ejemplo, en Objective-C un método complejo del protocolo del objeto básico se escribe como:

```
[object performSelector: sel withObject: arg afterDelay: delay inModes: modesArray];
```

En cambio, en Java o C++ se escribiría como:

```
object.performSelector (sel, arg, delay, modesArray);
```

La sintaxis de envío de mensajes en Objective-C tiene el inconveniente de incluir corchetes alrededor de cada mensaje, que en expresiones complejas degenera en una sucesión de corchetes similar a la encontrada en expresiones de LISP. En XC se usa la siguiente sintaxis:

⁷ C en general no es precisamente un ejemplo de buena sintaxis, si bien la del `if` es bastante clara. En un significativo porcentaje, la sintaxis es una cuestión de gustos y hábitos por lo que no tiene mucho sentido extenderse. XC hereda la sintaxis de las estructuras de control y operadores de C porque está más orientado a usuarios de lenguajes de la familia de C, si bien elimina muchas de las construcciones más problemáticas. Particularmente no elegantes pero existentes en XC son los operadores binarios y lógicos, la sentencia `for` y la sentencia `switch`. Se han mantenido por (in)consistencia con el resto de estructuras de control y operadores de C.

```
object.performSelector: sel withObject: arg afterDelay: delay inModes: modesArray;
```

Otros aspectos sintácticos se discuten en el capítulo 5.

La manipulación sintáctica de abstracciones debe ser también ortogonal [Pratt & Zelkowitz, 1999; p. 13]. Es decir, los diferentes elementos sintácticos deben combinarse con naturalidad, existiendo el mínimo posible de casos especiales. Por ejemplo, si una expresión puede ser un *array*, entonces es natural que, independientemente de la expresión, se le pueda aplicar el operador [] de acceso a elementos de un *array*.

Un último aspecto asociado con la consistencia es el número de abstracciones importantes soportadas por el lenguaje. Smith y Ungar, creadores del lenguaje SELF exponen este punto con claridad. La Figura 3-10 resume esa idea.

“A medida que se añaden más funcionalidades a un lenguaje, el programador puede hacer más cosas inmediatamente. Pero la complejidad crece con cada nueva funcionalidad: debe ser definido cómo interactúan entre ellos los elementos fundamentales del lenguaje, por lo que el crecimiento de la complejidad puede ser combinatorio. Tal complejidad hace al lenguaje básico más difícil de aprender, y lo vuelve más difícil de usar, al forzar al programador a elegir entre varias opciones de implementación, una elección que quizás deba ser revisada más tarde.” [Smith & Ungar, 1995; p. 309]

El añadido de abstracciones a un lenguaje ha de ser compatible con las abstracciones existentes. Cada abstracción nueva debe ser evaluada con cuidado para ver si genera problemas imprevistos en las abstracciones existentes. Por ello, la regla informal que conviene seguir es mantener un número pequeño de abstracciones, pero no demasiado pocas, para no caer en los defectos de LISP o Smalltalk-80.

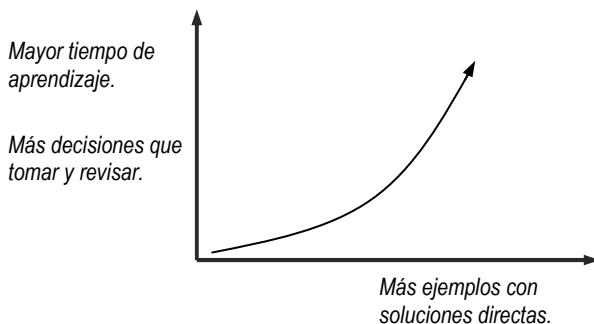


Figura 3-10 Tensión entre número de abstracciones y complejidad del lenguaje.

Cuantas más abstracciones haya, existen más ejemplos con soluciones elegantes, pero también hay más decisiones que hay que tomar y revisar durante el diseño del lenguaje, y más tiempo de aprendizaje requieren sus usuarios. Adaptado de [Smith & Ungar, 1995; p. 309].

La consistencia es una propiedad sutil, puesto que sólo se echa de menos cuando está ausente. Sin embargo, es una de las propiedades más importantes de un lenguaje, porque de manera intangible guía y simplifica el razonamiento de los programadores con la ayuda de abstracciones coherentes.

Algunos aspectos del lenguaje que mejoran su consistencia son:

- **Ayudas sintácticas y abstracciones ortogonales.** El lenguaje da un número pequeño de abstracciones consistentes y ortogonales entre sí. El compilador incluye ayudas sintácticas para hacer más legible y claro el código, sin prejuicio de las abstracciones del lenguaje. Se verán en el capítulo 5.
- **Abstracción homogénea basada en objetos.** Todos los tipos de datos que maneja XC son objetos y se manipulan siempre igual. El mismo esquema sirve para manejar componentes.
- **Orientación a objetos basada en prototipos.** XC no está basado en clases, sino en objetos prototípico. Son objetos autosuficientes a partir de los que se pueden construir otros objetos

nuevos. Los prototipos simplifican las relaciones internas del modelo de objetos y se adaptan con más naturalidad a un modelo de componentes.

3.6.2 Robustez

Un lenguaje es robusto si facilita la incorporación de cambios sin incurrir en nuevos errores. Existen errores algorítmicos que no se pueden prevenir cuando se realiza un cambio. Pero existen también varias clases de errores comunes que pueden ser comprobados automáticamente y simplifican las labores de programación, notablemente cuando se efectúan cambios. Un lenguaje poco robusto no facilita esa labor.

Los lenguajes que no tienen comprobación de tipos en tiempo de compilación, o que admiten la declaración implícita de variables son lenguajes que fomentan más la existencia de errores que su prevención. Un lenguaje que no realiza esas comprobaciones es indudablemente más simple de implementar, pero el ahorro en una implementación se transforma en costes de depuración de *todos* los programas que se crean con esa herramienta. Es más, es posible que simples y obvios errores sintácticos permanezcan ocultos durante mucho tiempo si determinada rama de un algoritmo no es ejecutada. Una variable mal escrita puede dar lugar a la declaración de una variable nueva en vez de usar una existente. Un parámetro de menos —o del tipo erróneo— quizá pase desapercibido o modificar sustancialmente el comportamiento de un algoritmo, comportamiento que sólo es patente cuando se ejecuta el bloque de código afectado y sólo entonces.

La robustez afecta también al dinamismo de un programa. La construcción dinámica de objetos con posibilidad de añadir o borrar campos arbitrariamente, así como la herencia dinámica, son altamente sensibles a pequeños errores. La eliminación o modificación dinámica de un método de un objeto puede afectar a la corrección de todos los fragmentos de código que usan objetos del mismo tipo y llaman a ese método. En general estos fragmentos confían en que el método original siga presente. El acto de eliminarlo rompe esa asercción e introduce *a partir de ese momento* interdependencia que no es factible comprobar antes de que suceda. Algo similar ocurre con los cambios de herencia dinámicos, sólo que en este caso se modifican, añaden o eliminan bloques de métodos y campos de una vez, y se afecta a grupos de tipos de datos —todos los tipos heredados— simultáneamente. En resumen: modifican dependencias dinámicamente que aumentan la complejidad del comportamiento de los programas. Complejidad que no se puede prever ni reducir a priori por su naturaleza dinámica, es decir, fomentan la proliferación de cambios no modulares y no verificables.

La falta de robustez afecta en última instancia al tamaño viable de programas escritos en esos lenguajes. Esencialmente el problema se puede diagnosticar como un aumento de interdependencias. La acción de declarar implícitamente una variable nueva por un error tipográfico adquiere interdependencias con el comportamiento del algoritmo original sin indicación expresa al programador. La eliminación de un parámetro en una función genera interdependencias directamente con todas las llamadas realizadas a esa función. Si parte del código no está disponible, ¿cómo se puede eliminar la posibilidad de error? Si un código erróneamente elimina un método no deseado o de otro objeto, ¿cómo es posible estar seguro de que ese error es detectable?

La robustez se traduce en confianza: muchos pequeños cambios de mantenimiento se pueden realizar sin problemas, asegurando al menos que han pasado todos las comprobaciones previas. Si la estructura global y local de cada parte del programa está escrita estáticamente, es viable comprobarla cada vez que se realizan cambios. La estructura estática, si bien menos flexible, permite razonar con mayor confianza y exactitud acerca de los programas en ejecución. Por ejemplo, nunca surge la duda de si se ha cambiado un método por otro, o si el padre de herencia es el correcto o no.

Características del lenguaje que soportan robustez son:

- *Compilación y comprobación de tipos estricta.* El lenguaje es compilado para mejorar su velocidad y tiene comprobación estricta de tipos para detectar inconsistencias durante la creación o la realización de cambios en los ficheros fuente.
- *Declaración de tipos independiente de su implementación.* La declaración de un tipo de datos en XC se mantiene completamente independiente de su definición. Un tipo de datos define un protocolo. Varias implementaciones distintas pueden seguir un mismo protocolo, sin necesidad de que estén relacionadas por la herencia. Un protocolo define un tipo de datos, una implementación no. Esta separación es necesaria para desligar a los clientes de un tipo de datos de una implementación determinada, y porque las declaraciones son más estables que las implementaciones.
- *Monotonía.* Es una propiedad que intenta preservar la corrección del código en presencia de cambios incrementales aditivos. La monotonía se estudia en el apartado 4.3.4.

3.6.3 Escalabilidad

Un lenguaje es escalable si facilita la construcción de programas de gran tamaño sin perder el control. La pérdida de control es resultado de la complejidad interna de un programa, que a su vez es una percepción de las interdependencias reales entre distintas partes de código. La pérdida de control se produce cuando no se está seguro de los efectos colaterales asociados a un cambio.

La escalabilidad se consigue mediante la construcción de módulos lo más independientes posibles y la inclusión de mecanismos de extensión que preserven la modularidad. Los módulos deben limitar la interdependencia con otros módulos, para que se pueda mantener el siguiente razonamiento intuitivo: *un cambio local sólo debe producir efectos colaterales locales, no globales.*

Propiedades del lenguaje que soportan escalabilidad son:

- *Módulos independientes.* XC incluye soporte de módulos jerárquicos con compilación independiente y publicación selectiva de símbolos a otros módulos. Los módulos se cargan en el soporte en tiempo de ejecución por separado.
- *Objetos encapsulados.* El uso de variables de objeto privadas da al lenguaje un acceso uniforme al estado y comportamiento de cada objeto.
- *Herencia.* Los tipos de datos o protocolos admiten la herencia múltiple. Los objetos permiten herencia simple. La herencia de implementación es más modular y opaca: es posible cambiar aspectos privados de un antecesor, como cambiar un cálculo por un acceso a variable, sin tener que recomilar un descendiente. Igualmente es posible cambiar el tamaño de un antecesor, o añadir nuevos mensajes sin necesidad de recomilar a sus descendientes.
- *Objetos extensibles.* Los protocolos y los objetos son extensibles con nuevas categorías. Esta característica facilita la adaptabilidad del código existente a situaciones y requisitos nuevos o cambiantes. La extensión se produce modularmente.
- *Soporte de extensiones ortogonales.* Facilita la metaprogramación. Son una abstracción adecuada para interceptar llamadas modularmente y añadir nuevo código sin afectar a la funcionalidad principal de un objeto.

3.6.4 Eficiencia

Un lenguaje es eficiente si las abstracciones que implementa también lo son. La eficiencia es una propiedad práctica. Dos algoritmos que produzcan el mismo resultado, como por ejemplo una ordenación, quizás sean equivalentes respecto al resultado, pero no tienen por qué serlo respecto al tiempo que tardan en terminar la ordenación. El tiempo de cómputo, así como la memoria son recursos valiosos que hay que evaluar con cuidado. Si una nueva funcionalidad es más lenta y ocupa más memoria, hay que sopesar las ventajas aportadas por la nueva funcionalidad y las desventajas asociadas a su implementación. En cualquier caso, una funcionalidad interesante puede resultar poco práctica y caer en desuso si su implementación es muy lenta. Un ejemplo son las llamadas a métodos en objetos. Si la llamada es muy lenta, dado el alto número de métodos que se ejecutan en un programa, puede ocurrir que el lenguaje se desestime por no ser lo suficientemente rápido. Otro ejemplo es la manipulación de tipos de datos habituales como números enteros o *arrays*.

Conciliar eficiencia y nueva funcionalidad requiere a veces nuevas estructuras en el soporte en tiempo de ejecución del lenguaje. Otras veces precisa un mejor análisis por parte del compilador o ayuda por parte del programador. Si un lenguaje produce sistemáticamente código ineficiente, es probable que su uso sea más limitado debido a que no cumpla las expectativas apropiadas.

Características del lenguaje que soportan eficiencia son:

- *Compilación a código C.* C es un lenguaje de programación estructurado a medio camino entre el ensamblador y los lenguajes de alto nivel. La compilación a código C hace más portable el código generado, incluso sin la presencia del compilador original de XC en la plataforma de destino.
- *Optimización de código a la velocidad de C.* En aquellas circunstancias donde se ha requerido del compilador que genere código optimizado, éste se ejecuta a la velocidad de C. Por ejemplo, la manipulación de objetos de tipos integrales y *arrays* de tipos integrales se realiza a la misma velocidad que en C.
- *Definición de funciones y objetos nativos.* Es posible definir con facilidad y sin necesidad de crear *wrappers* extra, tanto funciones como objetos nativos. Son usados para crear interfaces con sistemas existentes y publican esas interfaces al lenguaje.
- *Modificadores de objetos.* Por defecto, los objetos son extensibles mediante herencia u ortogonalmente. Los modificadores de objetos son indicaciones al compilador que le ayudan a optimizar su representación y el envío de mensajes.

3.7 Esquema de la solución propuesta

Cómo cumplir las cuatro propiedades anteriores: consistencia, robustez, escalabilidad y eficiencia, a la vez que construir un lenguaje que soporte un modelo de componentes y facilite la programación de sistemas, es el objetivo del resto de los capítulos de este trabajo. Actualmente, la versión de XC es la 0.7. Si bien la definición de XC no es todavía completa, se considera que la investigación de los aspectos principales de definición e implementación de un lenguaje con esas características se encuentran resueltos y probados con la construcción de un compilador prototípico para el lenguaje. El camino elegido para la presentación se muestra en la Figura 3-11. El enfoque es incremental, correspondiendo a grandes rasgos con el proceso de resolución.

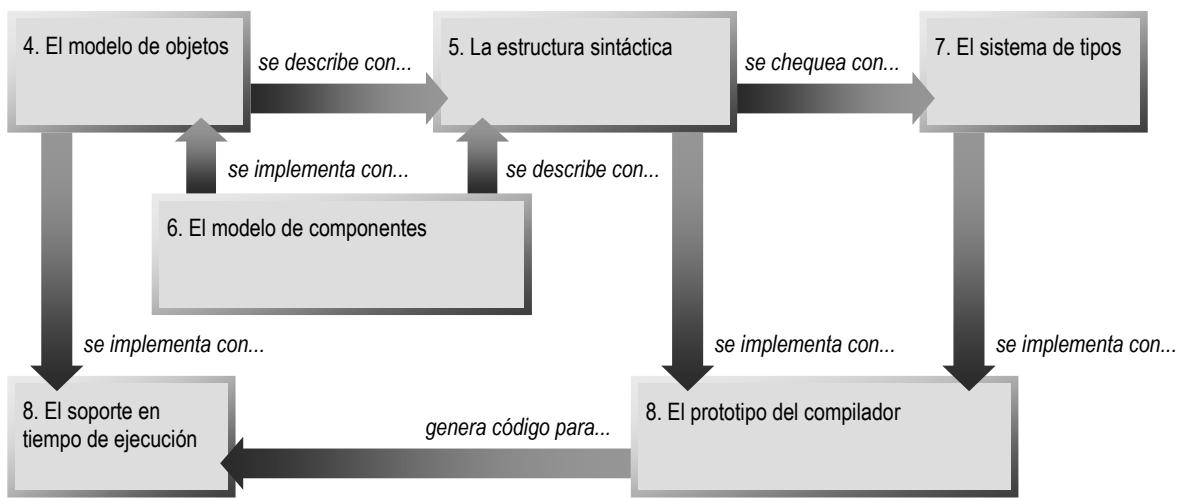


Figura 3-11 Guía general de los próximos capítulos.

Describe la definición del lenguaje XC en sus aspectos más importantes. El número al principio indica el número de capítulo.

3.8 Breve comparación con otros lenguajes

Quizá una buena forma de poner en perspectiva la definición de XC y su ámbito de aplicación consiste en evaluar las similitudes y diferencias generales con otros lenguajes. La Tabla 3-1 muestra algunos lenguajes importantes y las características más relevantes de ellos. Algunas pocas características incluyen el carácter ‘?’ indicando que no se han podido constatar, si bien se indica cuál es la considerada más probable.

La comparación de la Tabla 3-1 no pretende ser exhaustiva. Aun así, es posible obtener una idea del tipo de lenguaje diseñado en este trabajo. Se han seleccionado C++, Objective-C, C# y Java por ser lenguajes derivados de C, los tres primeros con clara vocación para la programación de sistemas. XC toma de C la parte principal de su sintaxis, con la salvedad de la sintaxis de envío de mensajes. Objective-C deriva además de Smalltalk-80. XC deriva en parte también de SELF y Objective-C. Su estructura modular se inspira en Modula-3. Eiffel es probablemente el lenguaje con clases más completo y, por tanto, de obligada referencia. Una breve explicación de las características comparadas es la siguiente:

- *Tipo de lenguaje.* Todos los lenguajes elegidos se incluyen en la familia de lenguajes orientados a objetos. Algunos de ellos: C++, Objective-C o Modula-3 son extensiones de los lenguajes estructurados o procedurales C y Modula-2. Soportan la declaración de estructuras de datos procedurales y de objetos, por lo que reciben el nombre de lenguajes híbridos. El resto de los lenguajes admite únicamente estructuras de datos usando objetos.
- *Variante de orientación a objetos.* Dentro de los lenguajes orientados a objetos existen varias vías posibles para implementar los objetos. Los lenguajes basados en clases se centran en la definición de una plantilla usada para crear objetos. En los lenguajes basados en clases con metaobjetos, las clases no son sólo plantillas, sino también objetos. Los lenguajes basados en prototipos son un poco más generales y simples conceptualmente: definen siempre objetos. Los objetos en este caso se crean clonando objetos prototipo, formando familias de objetos afines. Si se usa siempre un mismo objeto prototipo para realizar la clonación, se consigue una funcionalidad muy similar a la obtenida con clases.

	XC	C++	Java	C#	Objective-C	Smalltalk-80	SELF	Eiffel	Modula-3
Tipo de lenguaje	Orientado a Objetos	Híbrido (OO, Proc)	Orientado a objetos	Orientado a Objetos	Híbrido (OO, Proc)	Orientado a Objetos	Orientado a Objetos	Orientado a Objetos	Híbrido (OO, Proc)
Variante de OO	Prototipos	Clases	Clases	Clases	Clases y metaobjetos	Clases y metaobjetos	Prototipos	Clases	Clases
Sistema de tipos	Interfaces	Clases, primitivos	Clases, Interfaces, primitivos	Clases, interfaces, primitivos	Clases, interfaces, primitivos	Clases	Prototipos	Clases	Clases, Interfaces, primitivos
Momento de comprobación de tipos	Estático, dinámico	Estático, dinámico limitado	Estático, dinámico	Estático, dinámico	Estático, dinámico	Dinámico	Dinámico	Estático, dinámico	Estático, dinámico
Herencia de implementación	Simple, estática, opaca	Múltiple, estática	Simple, estática	Simple, estática	Simple, estática	Simple, estática	Múltiple, dinámica	Múltiple, estática	Simple, estática
Herencia de interfaces	Múltiple, estática, nombrada	No	Múltiple, estática, estructural	Múltiple, estática, estructural	Múltiple, estática, nombrada	No	No	No	Múltiple, estática, estructural
Tipos genéricos	Sí, sustitución restrictivos exactos	Sí, paramétricos generales polimórficos	No	No	No	No	No	Sí, paramétricos restrictivos polimórficos	Sí, paramétricos restrictivos polimórficos
Comprobación de tipos genéricos	Estático sin expansión, modular	Estático con expansión	No	No	No	No	No	Estático con expansión	Estático con expansión, modular?
Tipos covariantes	Sí, exactos	No	No	No	No	No	No	Sí, generales	No
Compilación	Compilado	Compilado	Compilado	Compilado	Compilado	Compilado	Compilado	Compilado	Compilado
Máquina virtual	No	No	Sí, JIT	Sí, JIT/No	No	Sí, JIT	Sí, JIT	No	No
Soporte en tiempo de ejecución	Sí	Limitado	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Soporte de componentes	Sí	No	Medio	Medio	Medio	Bajo	Bajo	Medio	Medio
Envío de mensajes	Selectores	Tabla virtual	Selectores	Tabla virtual?	Selectores	Selectores	Selectores	Tabla virtual	Tabla virtual
Extensiones de implementación	Categorías, estática	No	No	No	Categorías, estática	No	Dinámica	No	No
Extensiones de interfaces	Categorías, estática	No	No	No	No	No	No	No	No
Extensiones ortogonales	Sí, estática	No	No	No	No	No	Dinámica	No	No
Módulos	Sí, explícitos	Medio, explícitos	Sí, implícitos	Sí, implícitos	Bajo, explícitos	No	No	No	Sí, explícitos
Excepciones	No, en estudio	Sí	Sí	Sí	Sí (Biblioteca C)	No	No	Sí	Sí
Soporte de threads	No, en estudio	Sí	Sí	Sí	Sí (Biblioteca)	Sí	Sí	Sí (Biblioteca)	Sí
Atributos	No	Pragmas	No	Sí	Pragmas	No	No	Sí	Pragmas
Recolector de basura	Sí	No	Sí	Sí	Sí, configurable	Sí	Sí	Sí	Sí
Soporte código no seguro	Sí	No	No	Sí	No	No	No	No	Sí

Tabla 3-1 Características principales de XC junto a otros lenguajes populares.

- *Sistema de tipos.* Indica qué elementos del lenguaje son usados para construir tipos de datos. Tradicionalmente los lenguajes basados en clases sólo generan nuevos tipos creando nuevas clases. Los tipos primitivos son tipos soportados directamente por el lenguaje que no representan objetos. En C++ son los tipos originales de C que no son estructuras. En Java son los tipos integrales y los caracteres. En lenguajes que tienen algún soporte de interfaces o protocolos es posible también generar nuevos tipos a partir de ellos. Los lenguajes basados en prototipos suelen distinguir tipos de datos a partir de los objetos prototípicos, aunque cualquier objeto de una misma familia es válido. XC, en cambio, usa sólo un sistema de tipos con interfaces, que separa de forma clara interfaz e implementación.
- *Momento de comprobación de tipos.* La comprobación de tipos estática es realizada por un compilador. Aquellos lenguajes que tienen soporte en tiempo de ejecución guardan suficiente información de tipo para hacer comprobaciones dinámicamente durante la ejecución de un programa. C++ incluye soporte limitado de comprobación de tipos en tiempo de ejecución.
- *Herencia de implementación.* Este es el tipo de herencia tradicional de los lenguajes basados en clases, en los que nuevos tipos son generados con nuevas clases. La herencia de implementación ayuda a reutilizar directamente el código de otras clases u objetos al definir una nueva clase u objeto. Si la herencia es simple, existe una única clase u objeto padre. Si es múltiple, pueden haber varios padres. Si la herencia es estática el padre o padres no pueden variar en ejecución. Si es dinámica, es aceptable que una clase u objeto cambie de padre —y por tanto de implementación— en tiempo de ejecución. La herencia es opaca si no requiere información de implementación de los padres.
- *Herencia de interfaces.* Llamada también herencia de subtipos o *subtyping inheritance* es aplicable sólo a lenguajes que tengan alguna construcción que identifique interfaces. La herencia de interfaces reutiliza las declaraciones de otras interfaces. Durante este tipo de herencia no se hereda ninguna implementación. Es habitual que se admita herencia múltiple, porque no causa dificultades. La definición estática de una interfaz indica que ésta varía en tiempo de ejecución. La comprobación de tipos sobre interfaces heredadas puede ser por nombre o estructural. Si es por nombre se consideran distintas dos interfaces que tengan las mismas operaciones pero diferente nombre. La comprobación estructural, por el contrario, las considera iguales.
- *Tipos genéricos.* Son definiciones de tipos incompletas que incluyen variables de tipo. Estas variables deben ser sustituidas por otros tipos para dar lugar a un tipo completamente definido. Los tipos paramétricos aceptan cualquier otro tipo para completar un tipo genérico y se instancian cuando se declaran variables del tipo genérico con los parámetros adecuados. Con sustitución de tipos las variables de tipo se instancian durante la definición de una nueva clase u objeto, con o sin herencia. Los tipos restrictivos sólo aceptan subtipos de un tipo base que ha de cumplir cada parámetro. Las variables de tipos polimórficas aceptan subtipos en las sustituciones de tipo, las variables de tipo exactas no.
- *Comprobación de tipos genéricos.* Se refiere a cómo los tipos genéricos son comprobados por el sistema de tipos. Las comprobaciones estáticas son realizadas por el compilador. Si se produce comprobación con expansión, el compilador genera nuevo código por cada tipo final distinto instanciado. Si la comprobación es sin expansión, no es necesaria la generación de nuevo código. Las comprobaciones modulares esconden la implementación de un tipo genérico a otros módulos.
- *Tipos covariantes.* Son tipos que varían automáticamente a medida que se generan subtipos. Estos tipos verifican en tiempo de compilación métodos de clases u objetos que contengan como parámetro esa misma clase u objeto. Ejemplos típicos son las operaciones binarias de comparación, aritméticas o lógicas. Los tipos covariantes exactos representan tipos fijos. Los tipos covariantes generales representan tipos y cualquiera de sus subtipos. Los tipos covariantes generales pueden dar lugar a errores de comprobación de tipos.
- *Compilación.* Indica si un lenguaje es compilado o interpretado.

- *Máquina virtual*. Señala si junto a la definición del lenguaje se provee una máquina virtual que ejecute el código de ese lenguaje, y si la máquina virtual soporta compilación en demanda o JIT (*Just In Time*).
- *Soporte en tiempo de ejecución*. Es código de ayuda en la ejecución de programas, suministrado por el lenguaje, y que implementa parte de la funcionalidad dinámica que provee el lenguaje. Facilita la comprobación de tipos en tiempo de ejecución y puede llegar a contener mucha información acerca del programa en ejecución, como por ejemplo todas las clases o tipos de objetos, incluidos sus métodos, parámetros y tipos de los parámetros.
- *Soporte de componentes*. Es una medida cualitativa de las facilidades nativas de cada lenguaje para soportar el concepto de componentes, sus interfaces y composición. Cuanto menor es el soporte nativo, más código es necesario para construir un modelo de componentes sobre un lenguaje. Los lenguajes que soportan interfaces explícitas se encuentran a medio camino de representar componentes con facilidad.
- *Envío de mensajes*. Indica la técnica de implementación usada en el proceso de resolución de envío de mensajes a objetos. La técnica de tabla virtual es más eficiente, pero no modular. La técnica de selectores es un poco menos eficiente, pero es modular.
- *Extensiones de implementación*. Se refiere a la posibilidad de extender objetos preexistentes sin necesidad de crear nuevas clases u objetos. Extensiones habituales son añadido de métodos o nuevas variables de instancia. La extensión estática indica que se produce y comprueba en tiempo de compilación. La extensión dinámica acepta cambios en la estructura de un objeto en tiempo de ejecución, incluyendo el añadido o borrado de variables y métodos.
- *Extensiones de interfaces*. Son nuevas declaraciones de mensajes añadidas a interfaces preexistentes. Si se efectúa estáticamente, la extensión se comprueba en tiempo de compilación.
- *Extensiones ortogonales*. Estas extensiones dan soporte a la programación con metaobjetos o a la programación con aspectos. Si es estática, la extensión se comprueba en tiempo de compilación. Si es dinámica, es posible realizar la extensión en tiempo de ejecución.
- *Módulos*. Indica si un lenguaje contiene un nivel organizativo superior a las clases o prototipos. Los módulos pueden dar estructura jerárquica. Si son explícitos existe una declaración específica de interfaz. Si son implícitos, se obtiene la funcionalidad de la interfaz directamente a partir de la definición de la implementación. El soporte medio o bajo indica que han de usarse convenciones para obtener modularidad: como separar ficheros fuente y escribir cabeceras.
- *Excepciones*. Se refiere a si un lenguaje soporta directamente algún tipo de mecanismo de excepciones. Objective-C posee un esquema basado en bibliotecas de C. El mecanismo de excepciones de XC se encuentra actualmente en estudio.
- *Soporte de threads*. Señala si un lenguaje posee alguna construcción que facilite la programación concurrente. El soporte directo de programación concurrente en XC se encuentra en estudio.
- *Atributos*. Son anotaciones al código fuente que dan al compilador o al soporte en tiempo de ejecución información adicional durante la compilación o ejecución de un fragmento de código. Sólo Eiffel y C# tienen algún mecanismo de este tipo. Una variante más primitiva de atributos son los *pragmas* o instrucciones especiales de compilación.
- *Recolector de basura*. Sirve para gestionar automáticamente la gestión de memoria dinámica pedida en tiempo de ejecución. El algoritmo final de recolección de basura se encuentra aún en estudio.
- *Soporte de código no seguro*. El código no seguro es código que realiza operaciones de bajo nivel que no se pueden controlar por completo con un sistema de tipos. El soporte de código no seguro —original de Cedar [Swinehart, Zellweger, Beach & Hagmann, 1986]— obliga al programador a marcar explícitamente los fragmentos de código en los que se realizan este tipo de operaciones potencialmente peligrosas y no comprobadas por el compilador. El compilador prohíbe usar las funcionalidades no seguras en fragmentos seguros de código. De esta forma se

facilita la identificación automática y el aislamiento del código no seguro a zonas muy delimitadas.

3.9 Recapitulación

En este capítulo se han asentado las bases de los capítulos posteriores, introduciendo los conceptos fundamentales necesarios y las relaciones clave entre ellos. En primer lugar se han definido las premisas de diseño del nuevo lenguaje, que guiarán en la elección de la funcionalidad final del nuevo lenguaje, y su idoneidad como herramienta para la construcción de componentes. Luego se ha estudiado con qué técnicas se puede implementar un modelo de componentes que siga las directrices dadas en el capítulo 2, y cómo se puede simplificar la descripción del modelo de componentes usando módulos, interfaces, objetos y paso de mensajes, incluyendo las relaciones entre ellos. También se han revisado técnicas de extensiones planificadas y no planificadas de componentes. Se han evaluado cuáles deben ser los objetivos del lenguaje si además pretende soportar programación de sistemas, y descrito con más detalle los objetivos del nuevo lenguaje y su audiencia, poniendo sobre la mesa características cualitativas importantes que ha de tener un lenguaje de programación serio: consistencia, robustez, escalabilidad y eficiencia. Por último, un esquema de las relaciones entre los temas tratados en los próximos capítulos, y una breve comparativa de características relevantes sirve para poner en contexto el nuevo lenguaje con respecto a otros lenguajes conocidos.

4

El modelo de objetos

4.1 Introducción

Este capítulo introduce varios de los conceptos más importantes del nuevo lenguaje, dejando la modularidad y otros aspectos sintácticos para el siguiente capítulo. Se estudia primero el modelo de objetos subyacente usado para definir el modelo de componentes. El estudio de las características de la herencia sirve para describir con más precisión las decisiones de diseño que se encuentran detrás del modelo de objetos final. Debido al papel que la herencia juega en los modelos de objetos, y a la necesidad de profundizar en sus diferentes propiedades, la selección final de las características de la herencia se realiza en un apartado de discusión separado. Luego se estudian los mecanismos de envío de mensajes, evaluando los más importantes y seleccionando una técnica apropiada para poder definir sobre ella la composición de componentes. Por último, se revisan las técnicas de reflexión y metaprogramación, cómo afectan al modelo de objetos y cómo el modelo de objetos puede ayudar a simplificar esos conceptos.

Un modelo de objetos tiene tres elementos principales. La definición incremental mediante herencia de los objetos, cómo interactúan entre sí enviándose mensajes, y si es posible controlar elementos del modelo de objetos usando mecanismos de reflexión. Se evalúan las aportaciones anteriores en este campo para poco a poco ir definiendo el modelo de objetos final.

4.2 De prototipos a clases

Las personas representan el conocimiento a partir de generalizaciones sobre las experiencias diarias. Un ejemplo obtenido de [Lieberman, 1986; p. 214] sirve para introducir esta noción:

“Cuando una persona tiene una experiencia en una situación particular, digamos acerca de un elefante particular llamado Clyde, hechos acerca de Clyde pueden ser a menudo útiles cuando se encuentra a otro elefante, digamos uno llamado Fred. [...] Podemos considerar a Clyde como representante del concepto de *elefante prototípico*. Si le hago la pregunta: “Piense acerca de un elefante”, sin ninguna duda la imagen mental de algún elefante particular aparecerá en su mente, completo con sus características: color gris, trompa, etc. Si Clyde era el elefante más familiar para usted, el elefante prototípico podría ser la imagen de Clyde. Si le hago una pregunta como: “¿Cuántas patas tiene un elefante?”, una posible respuesta a la pregunta es asumir que la respuesta es la misma que cuántas patas tiene Clyde, a menos que exista una buena razón para suponer otra cosa”.

A partir del conocimiento particular de algunos ejemplares de elefante, se extrae el conocimiento acerca de un concepto de elefante más genérico, que es usado para responder a las preguntas previas. Es decir, se produce una clasificación de propiedades acerca de ciertos ejemplares de elefante en un elefante ideal que es una mezcla de los elefantes conocidos. Es difícil saber de antemano cuál es el

conjunto esencial de características para un concepto, puesto que los límites de un concepto son difusos. Por ejemplo, a la pregunta: “¿De qué color es un elefante?”, la respuesta habitual es: “Gris”. Sin embargo, existen también elefantes blancos. Una persona que supiese de la existencia de tales elefantes respondería en cambio: “Gris o blanco”. Es decir, los conceptos se modelan a partir de experiencias y nuevas experiencias pueden modificar a su vez los conceptos, añadiendo o quitando propiedades significativas. A partir del conocimiento de un conjunto de ejemplares es posible inferir también las características relevantes del grupo, que permite dar respuesta a si una propiedad es más común que otra, tal como el color de los elefantes.

Una simplificación del proceso anterior consiste en obviar a todos los ejemplares y empezar directamente con un conjunto inicial de propiedades que se suponen generales a un concepto. El caso general describe más concisamente muchos casos particulares, simplificando y mejorando a la vez nuestro entendimiento sobre un conjunto de casos quizá antes dispar. La abstracción, clasificación y generalización son técnicas usadas por muchas ciencias para hacer más comprensible el conjunto de experiencias que estudian. Construyen un modelo simplificado de la realidad, que intenta explicarla a partir de unos pocos conceptos elementales. Ambos modos de trabajo: el análisis de casos particulares y las generalizaciones que implican abstracción y clasificación, son convenientes y no excluyentes para modelar problemas en sistemas informáticos. Si bien la organización del conocimiento se construye de prototipos a clases, en informática el proceso fue el inverso, proponiéndose primero lenguajes basados en clases y luego lenguajes basados en prototipos.

4.3 Herencia

La herencia describe la recepción de características que forman parte de otros entes. En informática se refiere a cómo ciertas características de un programa son recibidas por otros programas (Def. 3-32). La razón para transferir esas características es obtener una descripción incremental de los programas a partir de otros existentes, con la esperanza de que tal proceso ahorre esfuerzo de programación [Nygaard & Dahl, 1978; p. 258]. Una descripción preliminar de la herencia se da en el apartado 3.4.5.2 *Herencia y composición*.

El deseo de poder simular procesos del mundo real mediante un programa dio lugar a las ideas iniciales de *clase*, *subclase*, *objeto* y al concepto de *prefijos* (*prefixing*) que más tarde evolucionó al de herencia [Nygaard & Dahl, 1978; pp. 258-259]. Fueron implementadas por primera vez en SIMULA-67 [Dahl, Myrhaug & Nygaard, 1968], un lenguaje diseñado para realizar simulaciones. Las características transferidas por la herencia son de dos tipos: *conceptuales* y de *implementación*. Las primeras corresponden con abstracciones y la estructura lógica del programa. Las segundas hacen referencia a la implementación de las abstracciones. Las clases facilitan la organización jerárquica de programas a través de la relación clase/subclase que da la herencia, donde las subclases son una especialización de las clases [Dahl & Hoare, 1972]. La Figura 4-1 resume los conceptos fundamentales asociados con clases y herencia.

La descripción de una clase en SIMULA-67 se hace en el código fuente. Describe una abstracción, mientras que los objetos o *instancias* son entes concretos durante la ejecución. La clase detalla un conjunto de *propiedades*, y *acciones* o *métodos* asociados, que son especializables mediante subclases para crear objetos concretos [Nygaard & Dahl, 1978; p. 260]. Las propiedades tienen varios sinónimos: *variables de objeto*, *variables de instancia*, *campos*, o *slots*. La especialización en subclases permite crear y heredar campos, así como heredar métodos, modificar métodos heredados o crear nuevos métodos.

Las ideas de SIMULA-67 dieron un paso decisivo al influir en el desarrollo de Smalltalk a principios de la década de 1970, junto con la visión de Alan Kay de tratar de construir un sistema orientado a objetos uniforme [Goldberg & Robson, 1989; p. x]. Se quería crear un sistema de información donde un usuario pudiese acceder, manipular y modificar información para que el sistema creciese a medida que

lo hiciesen las ideas del usuario [Goldberg & Robson, 1989; p. vii]. La primera versión comercial data de 1980. A partir de esa fecha se había asentado el fundamento para el desarrollo de nuevos lenguajes orientados a objetos. Notablemente, en los primeros lenguajes orientados a objetos el concepto de clase era central, siendo fuente de inspiración para lenguajes posteriores basados también en clases.

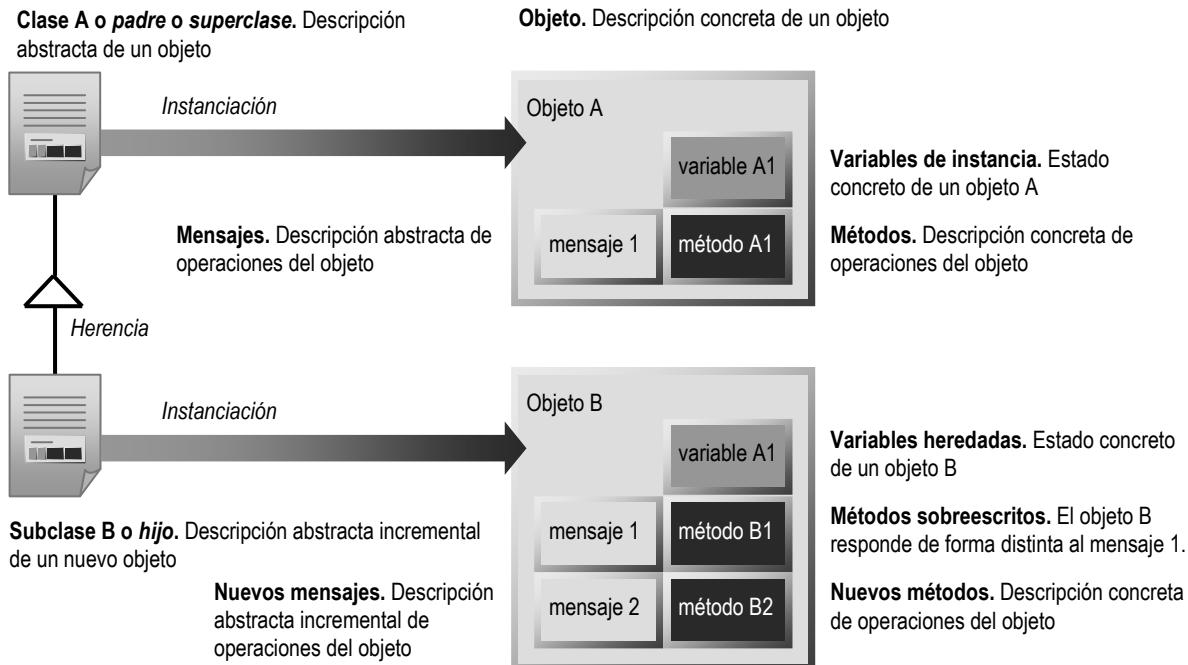


Figura 4-1 Conceptos básicos tradicionales de herencia.

La clase A describe objetos con una variable de instancia y un método que responde a un mensaje. La subclase B describe incrementalmente modificaciones a la clase A, creando objetos con propiedades ligeramente distintas. Nótese cómo se hereda estado (variables de instancia), conceptos (mensajes asociados a objetos) e implementaciones (métodos que implementan los conceptos).

Smalltalk-80 llevó más allá las ideas de orientación a objetos de SIMULA-67. Todo —incluidas las clases— eran objetos. Las clases debían ser objetos porque eran entes manipulables desde el entorno de desarrollo [Goldberg & Robson, 1989; p. 40]. Si las clases eran objetos y todos los objetos eran instancias de alguna clase, entonces las clases debían ser también instancias de alguna clase. Una clase cuya instancia es otra clase recibe el nombre de *metaclase*, que literalmente quiere decir clase de clases. En versiones anteriores, Smalltalk sólo tenía una metacategoría. Eso obligaba a que todas las clases tuvieran el mismo protocolo, pues éste estaba descrito en la metacategoría. Para facilitar que las clases tuvieran diferentes protocolos de inicialización, cada clase debía tener una metacategoría distinta. Smalltalk-80 se encargaba de crear las metacategorías apropiadas automáticamente [Goldberg & Robson, 1989; pp. 70-71].

La Figura 4-2 muestra estas ideas empleando dos clases muy sencillas: la clase A define una variable de instancia v1 y un método m1. La clase B define un nuevo método m2, heredando la definición de A. Los mensajes a los que responden los métodos se han obviado por simplicidad. En SIMULA-67 las clases no se representan mediante objetos. Los objetos creados almacenan todas las propiedades y métodos. En Smalltalk-80, en cambio, las variables sí se guardan pero los métodos no, los métodos pertenecen a las clases. Como puede verse, en muy poco tiempo los conceptos de orientación a objetos se encontraban ya muy elaborados.

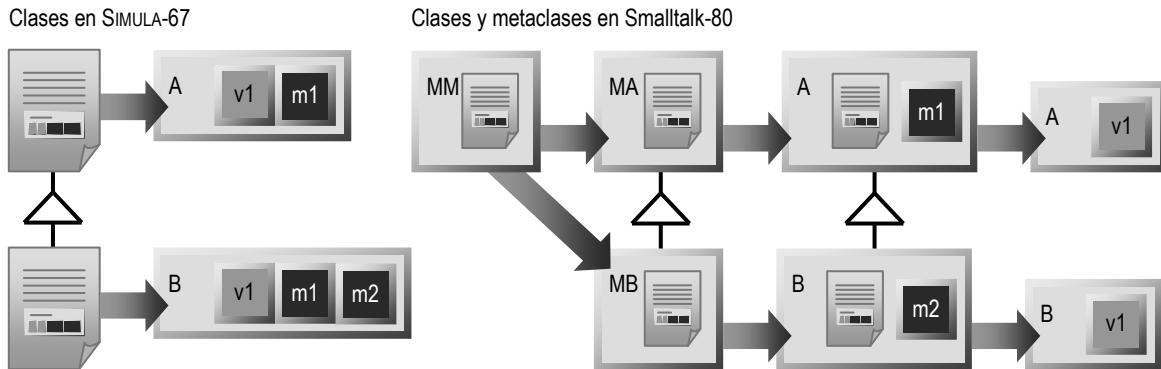


Figura 4-2 Clases en SIMULA-67 y en Smalltalk-80 (simplificado).

En SIMULA-67 las clases son entes abstractos que sólo existen en el código fuente y se encargan de instanciar objetos. En Smalltalk-80, todas las clases y metaclasses son también objetos. Una metaclass MM instancia todas las demás metaclasses. Cada metaclass MA y MB a su vez instancia una clase cada una. Las clases A y B instancian los objetos finales.

Es natural que el concepto de clase apareciese durante la evolución de un lenguaje dedicado a simulaciones, puesto que los modelos de simulación se crean a partir de conceptos matemáticos más cercanos a las abstracciones. Como se vio en el apartado 4.2, la obtención de conocimiento funciona ciertamente a la inversa: a partir de experiencias particulares se extraen conceptos más generales. Por esa época ya existían algunos tipos de sistemas que trabajaban mejor con objetos prototípicos: sistemas de visualización y manipulación de objetos como ThingLab [Bornig, 1981; p. 359] o lenguajes de actores [Hewitt, 1977], [Lieberman, 1981]. Extender la idea de prototipos al diseño de lenguajes motivó originalmente la propuesta de lenguajes orientados a objetos basados en prototipos [Bornig, 1986], [LaLonde, Thomas & Pugh, 1986], [Lieberman, 1986]. Las metaclasses de Smalltalk-80 eran un concepto arduo de entender y de enseñar, siendo una de las causas de la búsqueda de modelos más sencillos [Bornig, 1986; p. 36]. El aspecto más interesante de las propuestas de lenguajes basados en prototipos es que no requieren clases, planteando un modelo de objetos más simple. Los objetos prototípico contienen también variables y métodos. Se pueden clonar para obtener nuevos objetos y luego modificar si se considera necesario añadiendo, modificando o borrando variables de instancia y métodos. Junto a la descripción de cómo eran los objetos prototípico, se concibieron técnicas para implementar la herencia en esos modelos.

4.3.1 Delegación y concatenación

Existen dos vías para transferir las características de la herencia: mediante características compartidas o mediante copia de ellas. De ahí surgen las dos técnicas principales para implementar la herencia: *delegación* y *concatenación* [Taivalsaari, 1996; pp. 459-461].

- ❖ **Def. 4-1.** La *delegación* es una técnica de implementación de la herencia donde las características transferidas son *compartidas* por quien las transfiere y quien las recibe.
- ❖ **Def. 4-2.** La *concatenación* es otra técnica de implementación de la herencia donde las características transferidas son *copiadas* por quien las recibe.

Nos centraremos primero en los objetos prototípico por ser más simples. [Lieberman, 1986] asoció la idea del modelo de objetos prototípico con una implementación: *delegación*. La delegación es una técnica muy sencilla para compartir información entre varios objetos. Un objeto posee unas variables de instancia y métodos definidos. Si se envía un mensaje a un objeto y éste no sabe responder, entonces *delega* el mensaje en otro objeto. La delegación fue usada originalmente para modelar lenguajes de actores. La otra opción de implementación de herencia consiste en copiar todas las características compartidas y construir un objeto nuevo que no tenga relación con el objeto original.

concatenando las características⁸. Este era el mecanismo original de *prefixing* o concatenado de SIMULA-67.

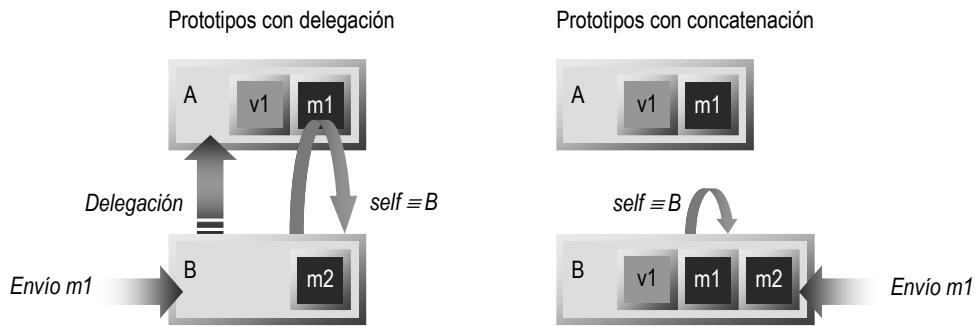


Figura 4-3 Prototipos con delegación y concatenación.

Con *delegación* la descripción del objeto B es incremental. Cuando se envía el mensaje m1 a B, se delega en A, pero self sigue asignado al objeto B aunque el método que se ejecuta es de A. Con *concatenación* los objetos están completamente separados. Cuando se envía el mensaje m1 a B, es B quien directamente responde, self siempre está asignado a B. Adaptado de [Taivalsaari, 1996; p. 460].

La Figura 4-3 ilustra las diferencias entre delegación y concatenación aplicada a objetos prototipo. En delegación el estado total del objeto B se encuentra repartido entre los objetos A y B. Esta configuración da lugar a algunos comportamientos curiosos. Si por ejemplo el objeto A representa una ventana prototipo en una interfaz gráfica y todas las ventanas delegan en A el color, entonces se puede modificar el color de todas las ventanas de un entorno gráfico cambiando únicamente el color de la ventana prototipo [Myers, Giuse & Zanden, 1992; pp. 187-188].

El comportamiento de la pseudo-variable self juega un papel importante en el modo en que se realiza la herencia. La delegación siempre vincula la pseudo-variable self con el objeto que recibe el mensaje. Así, en la Figura 4-3 el objeto A posee el método m1, pero al estar self vinculado con B, se acceden a los datos de B en vez de a los de A. Por eso parece que m1 pertenece a B. El resultado práctico es que el método m1 es compartido por A y B, actuando A como padre de B. La delegación sirve para que, en principio, un objeto pueda obtener funcionalidad de cualquier otro objeto, siempre y cuando delegue en él.

- ❖ **Def. 4-3.** La delegación *explícita* requiere que la llamada a un objeto externo se realice manualmente en el código.
- ❖ **Def. 4-4.** La delegación *implícita* usa ciertas variables de instancia o *slots* en el objeto que delega, que indican cuál es el objeto —u objetos si se trata de herencia múltiple— en los que se pretende delegar.

Siguiendo el mismo ejemplo, en caso de realizar concatenación, el método m1 pertenece físicamente a B, por lo que no se hacen búsquedas en otros objetos y self siempre está vinculado con B. La delegación ahorra espacio de memoria, en cambio, la concatenación produce objetos más independientes. Debe hacerse notar que cualquiera de las dos opciones de implementación caracteriza con igual expresividad el concepto de prototipo del apartado 4.2. Las dos técnicas crean objetos equivalentes para el propósito de inferir características a partir de ejemplares.

Es posible ver la herencia entre clases como un mecanismo particular de delegación, donde los objetos delegan en las clases la búsqueda de métodos y las clases delegan en las superclases los métodos que no han sobrescrito. De hecho, la herencia funciona exactamente así en Smalltalk-80. En un lenguaje compilado, todo el proceso puede ser resuelto por el compilador. La Figura 4-4 ilustra un

⁸ Concatenación no implica una implementación con un registro continuo, únicamente una implementación con copia. Por ejemplo, un diccionario donde se copian características heredadas es otra forma de implementar concatenación.

ejemplo simplificado de Smalltalk-80 para el envío de un mensaje a un objeto. Las clases y objetos usados son los mismos que en ejemplos anteriores. Los objetos en Smalltalk-80 no guardan directamente sus propios métodos, éstos se encuentran compartidos en los objetos de clase. Las clases tampoco guardan todos los métodos, sino sólo aquellos que son nuevos o han sido redefinidos. Cuando se envía un mensaje a un objeto, éste delega en su clase la búsqueda del método apropiado que responde al mensaje. Si no se encuentra ninguno, la clase delega a su vez en su superclase la búsqueda del método para ejecutar. Una vez encontrado, el método se ejecuta usando la pseudo-variable `self` vinculada al objeto original que recibió el mensaje. En la práctica, las clases de Smalltalk-80 no están delegando mensajes constantemente, sino que mantienen una caché de los métodos encontrados para mejorar el rendimiento. El mismo esquema se repite entre clases y metaclasses puesto que son también objetos.

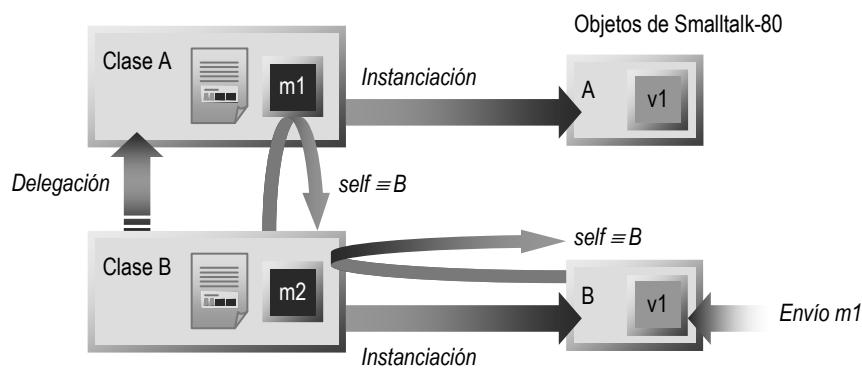


Figura 4-4 Herencia en Smalltalk-80 (simplificado).

Las clases delegan los mensajes enviados a los objetos. Cuando se envía el mensaje `m1` al objeto `B`, éste delega en su clase la búsqueda del método asociado. Si la clase `B` no lo encuentra, ésta a su vez delega en la clase `A`. En el proceso, `self` siempre se encuentra vinculado al objeto, no a la clase.

En Smalltalk-80 los objetos de clase tienen un comportamiento similar a los prototipos con delegación. Desde este punto de vista cada objeto prototipo es una clase, puesto que tiene potencialmente una vida independiente del resto de los objetos. De hecho, fue esta relación la que usó [Stein, 1987; p. 138] para rebatir las ideas de [Lieberman, 1986; pp. 217-219] acerca de que la técnica de delegación era más expresiva que la herencia. Stein mostró que ambas técnicas eran igualmente expresivas si se usan clases para representar objetos prototipo. La Tabla 4-1, adaptada de [Taivalsaari, 1996; p. 461], identifica características importantes de cada opción.

	Delegación	Concatenación
Estrategia de combinación de registros	Compartido, discontinuo y con referencias	Copiado, continuo, sin referencias
Interfaces	Dependientes	Autocontenidos
Jerarquía de herencia	Preservada	Aplanada

Tabla 4-1 Delegación frente a concatenación.

La estrategia de combinación de registros construye objetos compartidos si usa delegación, u objetos continuos si usa concatenación. Las interfaces de los objetos heredados se encuentran distribuidas entre los elementos compartidos en delegación y son autocontenidas en concatenación. La jerarquía de herencia se preserva explícitamente en delegación, mientras que en concatenación se aplana. El modo de implementación usado es independiente de si el lenguaje está basado en clases o en prototipos. La Tabla 4-2 resume la opción tomada por algunos lenguajes conocidos y el nuevo lenguaje propuesto:

Clases	Prototipos
Delegación	Smalltalk-80
Concatenación	SIMULA-67, BETA, C++, Java, Modula-3, C++, Objective-C
	SELF
	Kevo, Cecil, XC

Tabla 4-2 Opciones de implementación de herencia.

La técnica de concatenación es más apropiada para lenguajes compilados, ya que da lugar a representaciones más eficientes. Kevo [Taivalsaari, 1992], no obstante, es un lenguaje interpretado. La delegación suele usarse para compartir código y datos, siendo más habitual en lenguajes interpretados. La delegación tolera que cambios realizados en los ancestros se propaguen automáticamente a los descendientes, puesto que éstos dependen explícitamente de la definición de aquellos.

4.3.2 Herencia de implementación y herencia de tipos o interfaces

Las características que se transfieren por herencia tienen siempre dos caras. La primera es la herencia *conceptual*, que afecta a la modelización del programa que se esté construyendo. El sistema de tipos define generalmente qué se está heredando, es decir, qué interfaces de tipos de datos se quieren reutilizar.

- ❖ **Def. 4-5.** La *herencia de tipos o interfaces* transfiere descripciones de las operaciones de un tipo de datos a otro, permitiendo describir incrementalmente un nuevo tipo de datos.

La segunda cara concierne a la herencia de implementación de tipos de datos. Si se define un nuevo tipo de datos incrementalmente a partir de otro, es posible que parte de la implementación del tipo original sea reutilizable en el tipo derivado. Por ejemplo, en la Figura 4-3 y la Figura 4-4 se intenta reutilizar el método heredado `m1`.

- ❖ **Def. 4-6.** La *herencia de implementación* transfiere la implementación de las operaciones de un tipo de datos a otro, permitiendo implementar un nuevo tipo de datos incrementalmente.

Muchos lenguajes han unificado ambas variantes de herencia. Los intentos originales datan de SIMULA-67 [Nygaard & Dahl, 1978; p. 261] con la esperanza de simplificar la gestión de la relación entre la herencia y el sistema de tipos. Desafortunadamente, tal unificación genera ciertos problemas serios que se verán en el capítulo 7. Por esa razón, desde ahora se hace la distinción entre ambas variantes de herencia.

Un aspecto importante de la herencia pasado por alto a menudo es que, en el proceso de recibir características mediante herencia, se están aprovechando conceptos o implementaciones que quizás fueron ideadas en otros contextos. Si parte de la estructura de herencia se planifica de una vez, el contexto es usualmente el mismo. No suele haber problemas serios a la hora de factorizar declaración e implementación para resolver unos requisitos dados. Sin embargo, cuando posteriormente se quiere aplicar la herencia para simplificar la resolución de nuevos requisitos en un contexto diferente, con frecuencia la herencia no es tan efectiva como lo fue inicialmente. Porque en este segundo caso *no sólo* se están factorizando conceptos y código, se están también adaptando a nuevos requisitos quizás parcialmente nuevos o contradictorios. Las dependencias existentes en el código original respecto a los requisitos iniciales resueltos pueden hacer más difícil la adaptación, puesto que el diseño original no conoce cuál es el ámbito actual y quizás no haya previsto un diseño acomodado al mismo. Un ejemplo se encuentra en el apartado 5.3.3 *Extensión a tipos y objetos existentes*.

4.3.3 Polimorfismo

Literalmente, polimorfismo significa “*con múltiples formas*”. El polimorfismo es una de las propiedades más interesantes de los lenguajes orientados a objetos, si bien existen ciertas variantes que no son exclusivas de ellos.

- ❖ **Def. 4-7.** Una variable es *polimórfica* si puede contener referencias a objetos de diferentes tipos, de ahí la analogía con múltiples formas.

El polimorfismo surge con naturalidad en lenguajes que poseen herencia. Supongamos dos tipos de datos **A** y **B**, donde **B** hereda de **A** y **A** define un método `print`. Entonces, en el siguiente fragmento de código escrito en Java, las expresiones que involucran a la variable `var` son expresiones polimórficas válidas:

```
A aobj = new A();
B bobj = new B();
A var = aobj;
var = bobj;
```

La variable `var` de tipo **A** es polimórfica porque puede contener objetos de tipo **A** o de tipo **B**. Sin entrar en detalles sobre los sistemas de tipos que se verán en el capítulo 7, en la práctica quiere decir que el código que maneja una variable contenidoiendo objetos de tipo **A** puede manejar también objetos de tipo **B** usando la misma variable, *sin* necesidad de realizar modificaciones en el código. Por ejemplo, la siguiente expresión polimórfica es válida tanto si `var` está asignada al objeto `aobj` como si lo está a `bobj`.

```
var.print();
```

Es decir, el código teóricamente se puede *reusar* en nuevos contextos sin modificación. Implícitamente es capaz de operar con objetos cuya funcionalidad no se ha previsto en el momento de escribirlo, como sería el caso si la clase **B** se escribiese tiempo después que la clase **A**. Facilitar que un mismo código maneje nuevos tipos de datos para reusar código es sugestivo, pero es una historia muy diferente si el resultado obtenido es coherente o no. Este último extremo se verá también en el capítulo 7.

El polimorfismo funciona en coordinación con el envío de mensajes o enlace dinámico. Un objeto de tipo **B** se puede usar en lugar de otro de tipo **A** si el primero entiende el mismo conjunto de mensajes que el segundo. Ese conjunto está garantizado —al menos sintácticamente— si el tipo **B** hereda de **A** y no se eliminan o renombran mensajes durante la herencia. Si bien el polimorfismo es conceptualmente muy interesante, no siempre funciona con la misma simplicidad que su definición da a entender. Por ejemplo, si un lenguaje tiene únicamente tipos dinámicos, entonces no es posible comprobar a priori si un mensaje enviado a un objeto almacenado en una variable puede ser reconocido. Las variables en este caso son polimórficas, dado que pueden contener objetos de muchos tipos de datos —quizá de *cualquier* tipo de datos— pero recae únicamente en el programador la responsabilidad de elegir los objetos adecuados a los que se han de enviar los mensajes. Esto es lo que ocurre en Smalltalk-80, SELF o Javascript. Si por el contrario se desea comprobar el envío de mensajes en tiempo de compilación con un sistema de tipos, entonces no siempre se puede asegurar que todos los mensajes van a ser respondidos. Este extremo se verá también con mayor detenimiento en el capítulo 7. La postura de este trabajo es contraria a la existencia única de tipos dinámicos, puesto que toda comprobación que no realice un compilador ha de trasladarse al usuario del lenguaje, que ha de evaluar caso por caso si es o no apropiado el envío de un mensaje a un objeto.

Existen además otras variedades de polimorfismo que no se van a estudiar aquí por no estar relacionadas directamente con la herencia. El polimorfismo *ad hoc* se produce cuando se definen métodos o funciones con igual nombre pero con distintos parámetros. A este tipo de polimorfismo se le llama también *sobrecarga* u *overload*. Una variante interesante la constituyen los *multimétodos* que se verán en el apartado 4.4.4.1. El *polimorfismo paramétrico* es una variante de tipos genéricos que se estudiará en el capítulo 7.

4.3.4 Monotonía

Cuando un tipo de datos extiende a otro mediante herencia, el objetivo es crear un nuevo tipo de datos parecido al tipo del que se hereda, pero con ciertos cambios incrementales. Por ejemplo, puede incluir más propiedades, nuevos métodos, o redefinir alguno de los métodos que ya estaban definidos en el tipo original.

- ❖ **Def. 4-8.** Un cambio realizado en un tipo de datos usando herencia es *monótono creciente* o simplemente *monótono* cuando se realizan cambios incrementales que son exclusivamente aditivos en su interfaz. En otras palabras, *no* se renombran o eliminan mensajes⁹.

La monotonía es una propiedad muy importante que ha de seguir la herencia desde el punto de vista de ingeniería. Aunque no siempre es reconocida, el extracto mostrado a continuación es un buen resumen de las razones para ello:

“Para los propósitos de ingeniería del *software* (en vez de los de representación del conocimiento), una característica de una buena jerarquía de herencia es la monotonía de propiedades. Esto es, si sabemos que una propiedad es verdad para una clase, es deseable que la propiedad también sea verdad para sus descendientes. Este criterio surge de la observación de que la inversión hecha en el estudio de una clase en la jerarquía de herencia no debería ser devaluada cuando se estudian clases descendientes”. [Forman & Danforth, 1999; p. 79]¹⁰.

La monotonía asegura sintácticamente que un objeto de un tipo de datos **B** que hereda de otro tipo de datos **A** *al menos* recibe los mismos mensajes que un objeto cuyo tipo sea **A** [Palsberg & Schwartzbach, 1990; p. 153]. La monotonía está muy relacionada con el polimorfismo. Facilita el reuso teórico de código polimórfico asegurando que **B** recibe los mensajes declarados en el tipo **A** sin modificar o recompilar el código original que accede a objetos de tipo **A**.

La monotonía es condición necesaria para reusar código polimórfico sin tener que hacer modificaciones o recomplilaciones. Lo es porque si se permitiera el renombrado o borrado de mensajes, entonces el código original no modificado que accede a objetos de tipo **A** podría aún enviar alguno de los mensajes originales afectados, que ahora no sería reconocido. No es condición suficiente para reusar código polimórfico porque bien (1) el tipo heredado debería ser un subtipo válido, o bien (2) al menos el tipo heredado debería satisfacer la propiedad de encaje o *matching*. Ambas condiciones se estudian en mayor profundidad en el capítulo 7.

La idea de monotonía también es aplicable al añadido de nuevas funciones en una biblioteca tradicional. Téngase en cuenta que, desde este punto de vista, añadir nuevas funciones a una biblioteca tradicional tiene un comportamiento monótono. No requiere recomplilar a ninguno de los clientes. Por estas razones, introduciremos una nueva premisa de diseño del lenguaje que ayude a la construcción de programas más modulares.

- ❖ **Def. 4-9. Premisa 5.** Una modificación monótona de un módulo no debe requerir recomplilar módulos clientes.

4.3.5 Herencia sintáctica y semántica

Hasta este momento se han revisado características sintácticas de la herencia. Muchas veces se advierte que las condiciones para obtener subtipos de un tipo dado no son sólo sintácticas, sino

⁹ Matemáticamente la monotonía creciente se define como sigue. Sean P y Q dos conjuntos ordenados, una función $f: P \rightarrow Q$ es monótona creciente si $x \leq y$ en P implica que $f(x) \leq f(y)$ en Q . En este caso se considera el conjunto P de clases u objetos relacionados por la herencia, Q el conjunto de mensajes declarados en todos los elementos de P , y $f()$ la función que identifica los mensajes distintos heredados o declarados en cada elemento de P . Esta definición difiere de [Forman & Danforth, 1999; p. 42] en que no tiene en cuenta el árbol de herencia de metaclasses, si bien es muy similar en concepto.

¹⁰ Las propiedades referidas en la cita corresponden con metaclasses, que codifican *adjetivos* de clases como transaccionalidad o persistencia. Este tipo de adjetivos se expresan en XC mediante categorías de protocolos y prototipos. Independientemente de la implementación exacta de los adjetivos, la cita es aplicable igualmente en el contexto de este trabajo.

también semánticas, pero los estudios de tipos y subtipos se suelen concentrar en los aspectos sintácticos. Una razón para ello es que los aspectos semánticos de los subtipos de datos son mucho más complejos de evaluar. También se debe a que existen serias dificultades únicamente en la identificación de subtipos sintácticos válidos.

- ❖ **Def. 4-10.** La herencia es considerada *sintáctica* si el lenguaje sólo comprueba aspectos sintácticos de los nuevos tipos de datos generados por la herencia. Los aspectos sintácticos se refieren al nombre de los tipos y subtipos, y a las declaraciones de sus variables de instancia, mensajes y argumentos.
- ❖ **Def. 4-11.** La herencia es *semántica* si, además de la comprobación de aspectos sintácticos, facilita la comprobación de propiedades semánticas relacionadas con los tipos de datos. Son invariantes, precondiciones o poscondiciones —globales o parciales— aplicables a cada tipo de datos en algún momento de su ejecución.

La validez sintáctica y semántica de la herencia es importante para validar el uso del polimorfismo en los programas. Si no se garantiza que los tipos heredados son compatibles con los tipos originales que se sustituyen en expresiones polimórficas, entonces se están introduciendo errores en los programas. La monotonía sirve para facilitar que sintácticamente dos tipos de datos sean compatibles, pero incluso si sintácticamente lo son, todavía queda por recorrer el camino que asegure la compatibilidad semántica. La validez semántica se revisa en el apartado 7.3.4 *Comprobación semántica de tipos*.

4.3.6 Herencia estática y dinámica

Otra propiedad de la herencia se relaciona con el momento en que ésta se produce, distinguiéndose entre herencia estática y dinámica.

- ❖ **Def. 4-12.** La herencia es *estática* si la relación entre las clases u objetos de los que se hereda se mantiene fija a lo largo de la ejecución de un programa.
- ❖ **Def. 4-13.** La herencia es *dinámica* si la relación entre las clases u objetos de los que se hereda puede variar a lo largo de la ejecución de un programa.

La herencia estática define con más precisión los tipos de datos. Al ser estática, un compilador es capaz de comprobar si el envío de mensajes es correcto, teniendo muchas oportunidades de realizar optimizaciones que saquen partido del conocimiento del árbol de herencia. La herencia dinámica facilita el cambio de los tipos de datos dependiendo del contexto de ejecución. Una desventaja de la herencia dinámica es que resulta muy difícil realizar comprobaciones en tiempo de compilación que ayuden a encontrar posibles errores, aunque no es impedimento para conseguir importantes optimizaciones en demanda como las realizadas por SELF [Chambers, 1992a].

La herencia estática es más rígida, siendo objeto de críticas por partidarios de los lenguajes basados en prototipos, que ven una necesidad en el control dinámico de las características compartidas, particularmente en programación experimental [Stein, Lieberman & Ungar, 1987; p. 43]. No obstante, también reconocen que la evolución hacia sistemas más estables precisa la sustitución —idealmente automatizada por el entorno de desarrollo— de las características más dinámicas como la herencia por otras más estáticas [Stein, Lieberman & Ungar, 1987; p. 44].

4.3.7 Herencia simple y múltiple

La herencia introduce también una relación entre quien proporciona unas características y quien las hereda. De forma natural identifica como *padre* al origen de las características trasmitidas y como *hijo* al receptor de ellas. Pueden existir, además, características que se consideren estrictamente privadas y que no se permitan trasmisir.

- ❖ **Def. 4-14.** La herencia es *simple* si un hijo sólo puede obtener características de un padre.
- ❖ **Def. 4-15.** La herencia es *múltiple* si un hijo puede obtener características de más de un padre.

4.3.7.1 Herencia simple y múltiple de tipos o interfaces

La herencia simple de tipos necesita sólo el añadido de definiciones de mensajes. La herencia múltiple tiene una fuente de ambigüedad: la herencia de una misma declaración de mensaje —con igual nombre y tipo de argumentos— desde dos tipos distintos. La Figura 4-5 muestra variantes de esta situación. En herencia simple el mensaje *m1* es declarado en *A* y redeclarado en *B*. En herencia múltiple el mensaje *m1* es declarado por separado en *A* y *B*, y luego redeclarado en *C*. El mensaje *m3* es declarado en *B* y redeclarado en *C*. El resto de los mensajes se hereda con normalidad. Si se unifican las declaraciones de mensajes de ambos tipos en una sola, es posible eliminar la ambigüedad. También se puede considerar a los tipos de los que se pretende heredar incompatibles. Si se unifican, se están obviando las posibles diferencias semánticas entre los dos tipos originales y que, de algún modo, la implementación del nuevo tipo heredado es responsable de hacerlos compatibles. La segunda opción prohíbe la definición de un nuevo tipo por posible incompatibilidad semántica. Generalmente se usa la primera opción.

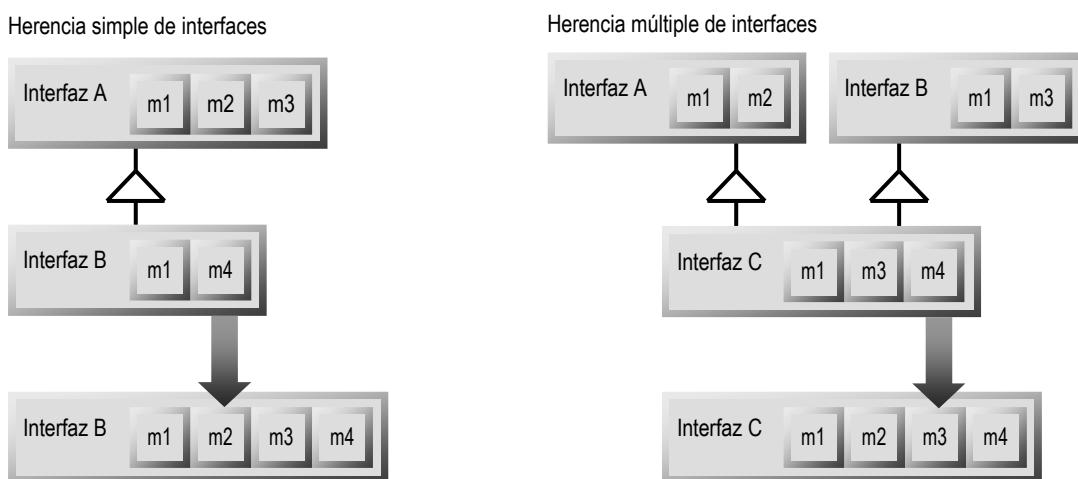


Figura 4-5 Herencia simple y múltiple de interfaces

En herencia simple *B* hereda el mensaje *m1* de *A*. En herencia múltiple el mensaje *m1* está heredado de *A* y *B*, mientras que *m3* está heredado de *B*. Tanto en herencia simple como en herencia múltiple, lo habitual es resolver la ambigüedad de los mensajes heredados mezclando las definiciones de mensajes en una sola.

Aunque no se mantenga la propiedad de monotonía, todavía puede tener sentido recibir parte de la declaración de un tipo de datos para la construcción incremental de un nuevo tipo. En este caso hay que observar que *no* es conveniente que exista una relación polimórfica entre variables que contengan ambos tipos de datos, puesto que no se puede garantizar siquiera que sintácticamente todos los mensajes enviados tendrán respuesta. En un lenguaje como Smalltalk-80 se puede argumentar que el lenguaje nunca garantiza esa propiedad y que, por tanto, podría llegar a tener sentido. De hecho muchas clases estándar de Smalltalk-80 si bien son monótonas, inhabilitan mensajes que son respondidos por sus antecesores [Cook, 1992; p. 2]. Eiffel también aprueba la eliminación de métodos durante la herencia manteniendo la relación polimórfica entre ellos. Propone un mecanismo adicional llamado *catcalls* para identificar potenciales envíos erróneos que demanden comprobaciones en tiempo de ejecución [Meyer, 1997; pp. 636-639].

Un aspecto más sutil de la herencia está relacionado con la semántica, incluso cuando sólo la sintaxis está en juego: si se define una abstracción *pila* con los mensajes *push* y *pop*, no tiene demasiado sentido heredar únicamente la operación *push* de una pila y no la operación *pop*. La monotonía asegura un mínimo de coherencia semántica al no dissociar operaciones relacionadas. Sin embargo, Eiffel consiente exactamente este tipo de herencia parcial, siendo la coherencia resultante responsabilidad del programador.

4.3.7.2 Herencia simple de implementación

A diferencia de la herencia de tipos, la herencia de implementación plantea algunas dificultades prácticas más serias. La herencia de implementación recibe definiciones de campos de clases u objetos y definiciones de métodos asociados a mensajes. No sólo se adoptan conceptos, sino cómo éstos se encuentran implementados. La Figura 4-6 muestra cómo se realiza la herencia simple de implementación. Se heredan campos o variables de instancia y métodos. Durante el proceso, deberán resolverse algunas ambigüedades, que variarán dependiendo de si la herencia usa concatenación o delegación.

Veamos en primer lugar la herencia de campos o variables de instancia con concatenación. La recepción por herencia de variables ha de reservar espacio para ellas en los hijos. La primera dificultad surge con la redefinición de un misma variable en una clase u objeto hijo. Es el caso de $v1'$ redefiniendo $v1$ y $v2'$ haciendo lo propio con $v2$ en la Figura 4-6.

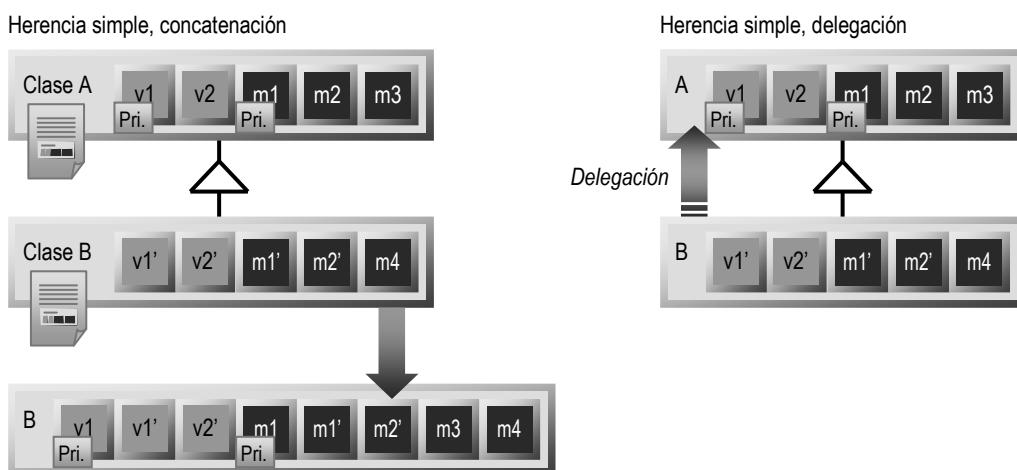


Figura 4-6 Herencia simple de implementación con concatenación y delegación.

Se heredan conceptualmente variables de instancia y métodos. Las variables de instancia privadas deben heredarse también. El método privado $m1$ en A debe heredarse conceptualmente en B para que B pueda ser usado en contextos donde A se usa (y que pueden llamar al método $m1$ privado). En delegación, la estructura de objetos es más simple a costa de que A comparta estado e implementación con B. Nótese cómo la estructura de delegación es equivalente a la estructura de clases si éstas se representasen con objetos.

Se tienen tres opciones: (1) mantener ambas definiciones, (2) esconder la definición heredada y (3) indicar un error o aviso. La primera opción tiene sentido si la definición es privada — $v1$ en la figura— y por ello teóricamente no visible en los hijos. No tiene sentido unificar definiciones privadas como hace Flavors [Moon, 1986] ya que puede dar lugar a errores sutiles relacionados con estado compartido que no se debiera compartir [Snyder, 1986a; p. 40]¹¹. Si la variable de instancia es pública, como ocurre con $v2$, mantener ambas definiciones genera una ambigüedad. Más habitual es usar la segunda opción en este caso: la nueva definición pública esconde la definición heredada. El razonamiento para admitir que una variable redefinida esconde a una variable heredada es evitar que un cambio de nombre de esa variable en el antecesor requiera la recompilación de sus descendientes. Sin embargo, si la variable renombrada es también pública probablemente afecte semánticamente a los descendientes de todas formas, por lo que no mostrar ningún error o aviso puede ser engañoso. De ahí la tercera opción, hacer explícito el error o aviso.

La herencia de métodos usando concatenación también se describe en la Figura 4-6. Solucionar el proceso de recepción de métodos durante la herencia es equivalente a especificar el algoritmo de

¹¹ Si no fuese así, el código polimórfico que accediese al objeto B como si fuese un objeto A fallaría si quisiera leer la variable privada $v1$ de A sobre el objeto B. Es decir, la variable de instancia privada debe estar disponible físicamente en B.

resolución de métodos usado. En Smalltalk-80 y la mayoría de los lenguajes con herencia simple, los métodos definidos en los descendientes sobrescriben los métodos definidos en sus antecesores. Es el caso de `m2'` redefiniendo `m2`. Si un descendiente no sobrescribe un método se selecciona el último método heredado, empezando por el antecesor más cercano, como ocurre con `m3`. El descendiente también puede definir nuevos métodos —`m4` en la figura—. Para tener acceso a un método de un padre sobrescrito se usa la cláusula `super`. Nótese que el método privado `m1` heredado de `A` debe estar conceptualmente disponible en el objeto hijo `B`, aunque `m1'` redefina el mismo identificador¹².

Para ejecutar métodos de otros antecesores más lejanos algunos lenguajes admiten nombrarlos directamente, mientras que otros sólo aceptan la llamada al padre usando `super`. Esta última opción tiene la ventaja de no exponer el árbol de herencia directamente en el código de los descendientes. En ambos casos la llamada a un antecesor debe hacerse explícitamente en el código del método que sobrescribe. Si no se realiza esa llamada, es posible que un método sobrescrito tenga un comportamiento erróneo por no haber ejecutado la funcionalidad del antecesor. Para evitar esa posibilidad, BETA [Kristensen, Madsen & Møller-Pedersen, 1987] utiliza otra técnica. Asegura que los métodos definidos en los antecesores siempre se ejecutan, proporcionando una cláusula `inner` para llamar a los hijos. El mecanismo de herencia es similar, sólo la dirección de incorporación de funcionalidad varía. En Smalltalk-80 y la mayoría de los lenguajes los nuevos métodos son favorecidos, reemplazando métodos heredados. Los nuevos métodos se ejecutan siempre, decidiendo incorporar o no la funcionalidad heredada a través de `super`. En BETA los métodos originales son favorecidos y siempre se ejecutan. Ellos deciden si aprueban la extensión de su funcionalidad usando la cláusula `inner` [Bracha & Cook, 1990; p. 306]. Smalltalk-80 favorece extensiones no planificadas, pues la llamada a `super` es opcional. BETA favorece extensiones planificadas, al ser opcional la llamada a `inner`.

Se puede observar también en la Figura 4-6 que la herencia por delegación obvia todo el proceso de construcción del objeto `B` por concatenación. Es notable que la estructura de los objetos por delegación sea equivalente a la relación entre clases cuando se usa concatenación. Las reglas para el acceso a las variables o métodos heredados dependen exclusivamente del proceso de delegación: buscar la definición en el objeto actual `y`, si no se encuentra, buscarla en sus antecesores, empezando por el más directo. Este proceso no da lugar a ambigüedades. Por ejemplo, en `B` se accede directamente a la variable `v2'`, quedando la variable `v2` de `A` escondida. Se puede referenciar a `v2` desde `B` si se delega expresamente en ella. Generalmente, los lenguajes que usan delegación sólo soportan tipos dinámicos, no teniendo tipo las variables. En ellos no es factible la existencia de variables o métodos privados¹³. El procedimiento de delegación usado con los métodos es exactamente el mismo, por lo que no es necesario extenderse más. El atractivo de la delegación es que, por un lado, el proceso de seleccionar la variable o método heredado sigue unas reglas muy simples y, por otro, que el resultado es conceptualmente más sencillo. La desventaja de esta técnica es que es absolutamente dependiente del estado compartido entre objetos.

4.3.7.3 Herencia múltiple de implementación

La herencia múltiple de implementación parece en principio una extensión natural de la herencia simple, ya que facilitaría la combinación de diferentes implementaciones para una

¹² Al igual que con las variables privadas, si no fuese así, el código polimórfico que accediese al objeto `B` como si fuese un objeto `A` fallaría si ejecutase el método privado `m1` de `A` sobre `B`. Al contrario que con las variables, el método privado no tiene que estar disponible físicamente en los descendientes —aunque sí debe estarlo conceptualmente—. Al ser privado, el método no se puede sobreescritir y su dirección es siempre fija. Por tanto, `A` y `B` pueden acceder sin ambigüedad al código asociado al método privado.

¹³ Veamos porqué. Si la variable `v1` de `A` es privada, desde `B` no será posible acceder a ella si el proceso de delegación comprueba que el tipo de la variable que contiene el objeto `B` es de tipo `B`. Es decir, hace falta la comprobación del tipo de la variable para bloquear la delegación a `v1`. Código polimórfico que accediese al objeto `B` usando una variable de tipo `A` podría leer la variable privada `v1` de `A` si el proceso de delegación comprueba que el tipo de la variable usada es realmente `A` y no `B`, aunque el objeto sea de tipo `B`. Si no se asocia tipo a las variables no hay modo de distinguir entre ambos accesos polimórficos.

programación incremental más flexible y expresiva. Como veremos, no es exactamente el caso. La Figura 4-7 muestra diferentes opciones de herencia múltiple de implementación mediante concatenación. La figura no pretende ser exhaustiva, ilustrando la configuración de objetos resultantes de aplicar las reglas de herencia múltiple según la interpretación que dan algunos lenguajes que la soportan. El objetivo es tener una idea de las decisiones que han de tomarse al extender la herencia simple a múltiple y ver cómo aumentan las distintas alternativas disponibles. Procederemos análogamente al apartado anterior, describiendo primero el proceso de herencia de variables de instancia y luego de métodos.

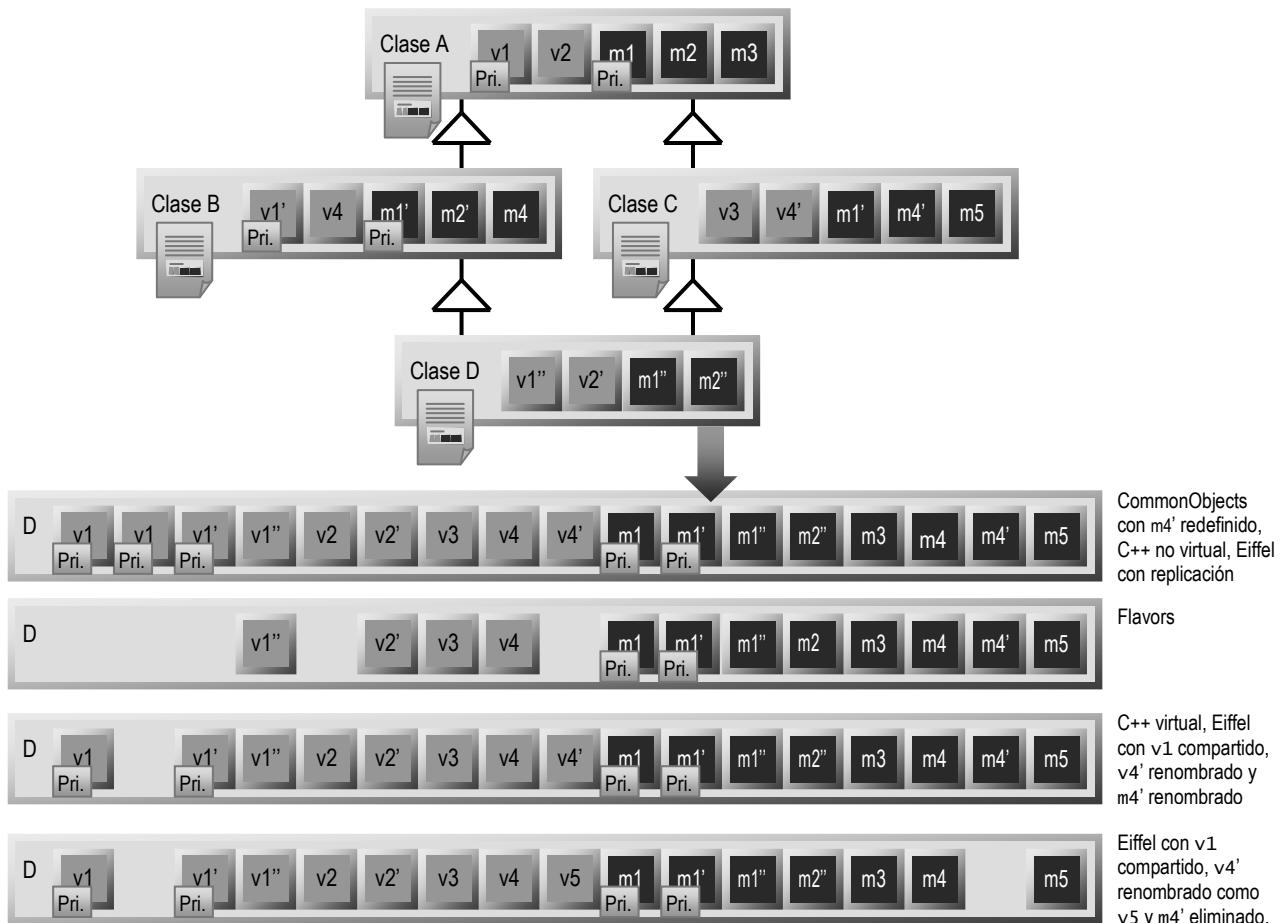


Figura 4-7 Herencia múltiple de implementación por concatenación.

Se heredan conceptualmente variables de instancia y métodos. Las variables privadas deben heredarse también. Diferentes variantes de objetos son posibles dependiendo del lenguaje usado u opciones dentro de algunos lenguajes como C++ o Eiffel.

La variable de instancia v1 de A es privada, se define de nuevo en B como privada y finalmente en D como pública, heredándose desde dos ramas distintas. En este caso se ha de decidir si los datos del antecesor son únicos o si se multiplican por el número de padres. CommonObjects [Snyder, 1986b] no mezcla nunca antecesores comunes de distintas ramas, manteniendo ambas. C++ permite decidir si se mezclan o no: en herencia *no virtual* se duplica la variable privada v1 de A porque se hereda de dos sitios distintos, en herencia *virtual* se hereda sólo una de las variables v1 de A. En Flavors se mezclan las definiciones independientemente de que sean privadas o públicas.

La variable v2 de A se redefine en D como v2' escondiendo la v2 original que, de todas formas, ha de tener su espacio reservado por si se desea acceder a ella directamente desde código polimórfico. La variable v3 se hereda de C sin mayores problemas. La variable v4 se hereda simultáneamente de B y C,

producíendose una ambigüedad. La manera de resolverla varía también de lenguaje en lenguaje. Flavors mezcla ambas definiciones. C++ y Eiffel la tratan como una ambigüedad y consideran que se produce un error, debiendo seleccionarse cada variable mediante código. CommonObjects siempre las duplica.

La herencia de métodos mostrada en la Figura 4-7 prioriza a los nuevos métodos redefinidos sobre los métodos heredados, ya que es la interpretación dada por todos los lenguajes del ejemplo. Los métodos privados `m1` y `m1'` deben estar disponibles en `D` para permitir polimorfismo¹⁴. El método `m1''` en `D` se define por primera vez público y no supone problema alguno. El método `m2''` en `D` redefine el método `m2'` de `B`, que a su vez redefine el método `m2` de `A`. Sólo se almacena el método `m2''`, los demás están disponibles a través de `super`¹⁵. El método `m3` se hereda por dos ramas diferentes. El método se combina en Flavors y C++. En CommonObjects y Eiffel se debe seleccionar un único método heredado. El método `m4` se hereda de `B` y `C` independientemente. En CommonObjects y Eiffel es un error de ambigüedad la presencia de un segundo método `m4'`, que debe ser bien borrado bien renombrado. En C++ y Flavors ambos están disponibles. Por último, el método `m5` se hereda sin problemas.

Eiffel posee probablemente la implementación más flexible y quizá también eficiente de herencia múltiple. Su sintaxis hace frente a múltiples conflictos posibles de variables de instancia y métodos heredados, permitiendo eliminarlos y renombrarlos, replicarlos o mantenerlos compartidos. Una descripción detallada de los mecanismos de herencia múltiple de Eiffel se encuentra en [Meyer, 1997].

En caso de usar delegación y herencia múltiple, la decisión de tener variables compartidas o no depende únicamente de la delegación en un objeto común o en varios objetos separados. Factorizando adecuadamente los objetos y delegando en ellos es posible obtener objetos compuestos funcionalmente equivalentes a los mostrados en la Figura 4-7 para la concatenación, si bien con mayor cantidad de información compartida. Por brevedad no nos extenderemos en su explicación, ya que los conceptos manejados son similares a los ya vistos en la herencia múltiple usando concatenación.

4.3.7.4 Linearización de herencia múltiple

La selección del método heredado que se ejecutará primero depende a veces del orden de declaración de los mismos. Por ejemplo, en la Figura 4-7 el método `m4` en `D` se hereda de dos padres distintos: `B` y `C`. Si no se prioriza ningún parente, entonces un envío del mensaje `m4` es ambiguo. En C++ ha de seleccionarse por código de qué parente se va a obtener la implementación de `m4`. En CommonObjects y Eiffel deberá renombrarse alguno de los `m4` para evitar ambigüedad, o seleccionar uno de ellos y eliminar el otro. Por el contrario, en Flavors, Loops [Bobrow & Stefik, 1983] o CLOS se priorizan los padres, produciéndose una *linearización* de los mismos. La priorización afecta también a cómo se tratan a otros ancestros, existiendo reglas diferentes en los tres lenguajes.

La Figura 4-8 muestra las linearizaciones resultantes de algunas clases. La ventaja de la linearización es que el procedimiento de resolución de métodos se hace equivalente al de herencia simple, evitando posibles ambigüedades. Una desventaja es que al programador le resulta más difícil intuir qué método acabará ejecutándose en una jerarquía compleja. Otras desventajas más comprometidas se ven en el apartado siguiente.

¹⁴ Los métodos `m1` y `m1'` no ocupan espacio físico. Por la misma razón que en herencia simple, no necesitan ser redefinidos al ser su dirección fija y conocida.

¹⁵ Nótese que una variable polimórfica de tipo `A` que contenga un objeto de tipo `D` y al que se envíe el mensaje `m2`, usará el método `m2''` para responder a él. Por tanto, en un objeto de tipo `D` sólo el método `m2''` debe almacenarse.

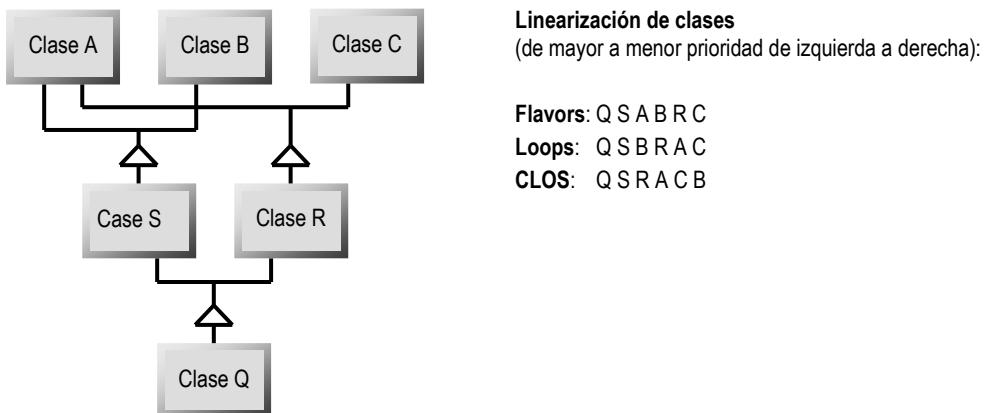


Figura 4-8 Algoritmo de linearización de herencia.

Flavors, Loops y CLOS poseen tres formas distintas de priorizar clases antecesoras, aun cuando los tres son variantes de LISP y muy similares en otros aspectos. Adaptado de [Kiczales, Rivières & Bobrow, 1991; p. 81].

4.3.7.5 Mixins

[Bracha & Cook, 1990] propusieron los *mixins* o *subclases abstractas*, que generalizan la modelización de jerarquías con herencia simple y múltiple. Los *mixins* intentan organizar las diferentes variedades de herencia simple y múltiple en un marco común único, a la vez que hacen frente a un problema serio de la linearización de herencia múltiple: [Snyder, 1986a; pp. 43-44] identificó problemas de encapsulación entre clases linearizadas, que afecta a los padres esperados según el árbol de herencia. Por ejemplo, en la Figura 4-8 puede comprobarse que en Flavors los métodos del antecesor A tienen preferencia sobre los de R en la clase Q, aunque A es un parente de R. En Loops la clase S ve cómo se ejecutan los métodos de R antes que los de su antecesor directo A, cuando se acceden desde Q, situación que no se manifiesta si se accede desde S. En CLOS ocurre exactamente lo mismo con la clase S cuando se accede desde Q.

[Bracha & Cook, 1990] diferencian entre dos tipos de clases, aquellas de las que se pueden instanciar objetos y aquellas de las que no: las clases *mixin*. Se espera únicamente de ellas que proporcionen métodos y variables de instancia para ser heredadas —mezcladas— por otras clases. Los *mixins* factorizan un grupo de métodos para que no se produzcan los problemas de definición múltiple de métodos heredados de varias clases. Un *mixin* puede heredar de otros *mixins*, pero no de clases instanciables. Una clase instanciable puede heredar de múltiples *mixins* pero a lo sumo de una sola clase instanciable. Por ejemplo, es posible representar la Figura 4-7 con diferentes configuraciones de *mixins*, algunas de las cuales se presentan en la Figura 4-9.

Los *mixins* *MixinBC* y *MixinAC* de la Figura 4-9 son subclases no instanciables, por ello reciben también el nombre de *subclases abstractas*. Los *mixins* hacen frente a la falta de encapsulación de la linearización de la herencia definiendo en cada *mixin* una relación explícita con sus padres directos, evitando las variaciones de antecesores durante la resolución de métodos desde distintas clases. En caso de que un mismo mensaje aparezca en dos *mixins*, la clase instanciable decide la priorización adecuada en la implementación de sus métodos, seleccionando el orden de mensajes apropiado y quedando explícito en el código.

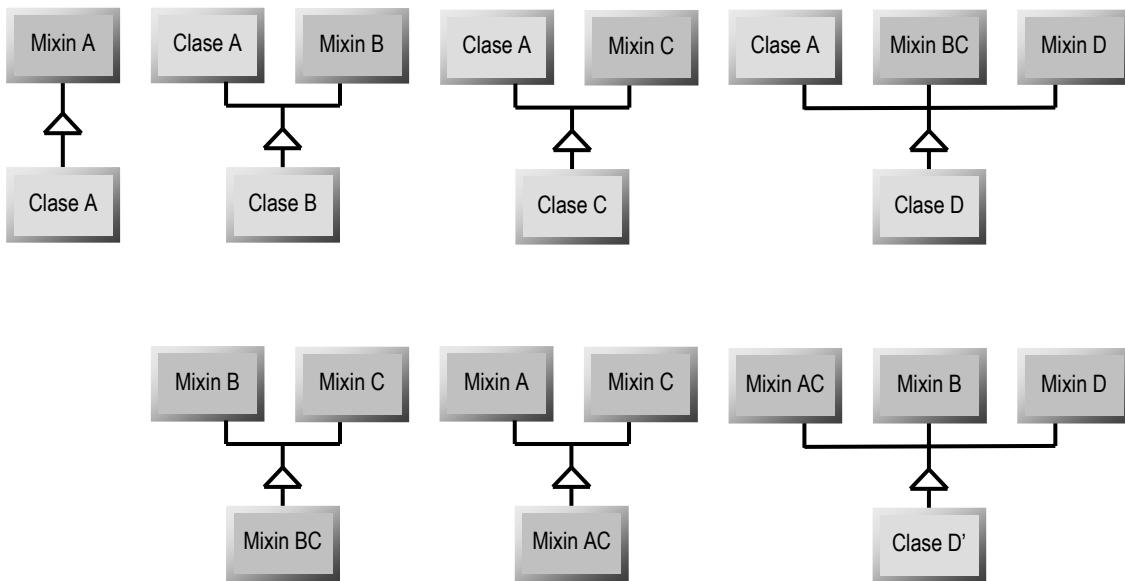


Figura 4-9 Mixins como una forma disciplinada de herencia múltiple.

El ejemplo está basado en la Figura 4-7. Cada *mixin* implementa un protocolo. Cuando varias clases necesitan el mismo protocolo heredan del mismo *mixin* . Así se reusa la implementación de cada *mixin* evitando padres múltiples. Varias soluciones son posibles dependiendo de la factorización de los protocolos. En el ejemplo se muestran dos posibles clases D a partir de distintos *mixins* . Una clase instanciable hereda a lo sumo de otra clase instanciable y múltiples *mixins* . Los *mixins* sólo heredan de otros *mixins* .

4.3.8 Llamadas de retorno y herencia

En el apartado 2.7.2.5 *Aserciones de llamadas de retorno* se hizo una introducción a las llamadas de retorno y a los problemas y aserciones que implica usar ese tipo de llamadas a subrutinas. En herencia se producen llamadas similares a las llamadas de retorno, aunque sus implicaciones son más profundas y los comportamientos resultantes más complejos. Una discusión más completa se expone en [Szyperski, 1998; pp. 57-68, pp. 104-109]. Las llamadas de retorno o *callbacks* (Def. 2-57) invierten quién realiza las llamadas a subrutinas. En las llamadas a subrutinas habituales es el cliente el que llama al proveedor. En las llamadas de retorno es el proveedor quien llama al cliente. En presencia de la herencia, hay dos formas importantes de llamadas de retorno.

La primera variante asocia el código de una clase u objeto padre con el código proveedor, y el código de una clase u objeto hijo con el código cliente. Una llamada de retorno aparece cuando el código del hijo hace una llamada a un método heredado, y éste ejecuta un método redefinido por el hijo. Consideremos la definición siguiente de clases escritas en Java, donde las clases del ejemplo se han mantenido muy sencillas para ilustrar mejor la naturaleza de las interacciones.

```

class A extends Object
{
    private unsigned printNameCount;

    A()
    {
        this.printNameCount = 0;
    }

    void printName()
    {
        this.printNameCount++;
    }
}
  
```

```

        System.out.println("A");
    }

void print()
{
    this.printName();
}

class B extends A
{
    void printName()
    {
        System.out.println("B");
    }

    void print()
    {
        super.print();
    }
}

```

La clase **B** extiende a la clase **A**, sobrescribiendo los métodos **printName** y **print** definidos en **A**. El código siguiente de un cliente final externo produce la impresión de la cadena “**B**” en la consola:

```

B bobj = new B();
bobj.print();

```

El envío del mensaje **print** a **bobj** ejecuta el método **print** definido en **B**, que a su vez hace una llamada al código heredado **print** de **A** con la cláusula **super**. El código de **A** envía el mensaje **printName** a sí mismo a través de **this**. Al ser **bobj** un objeto de tipo **B**, el método ejecutado finalmente es el definido en **B**, siendo esta última llamada una llamada de retorno. La Figura 4-10 representa las llamadas realizadas.

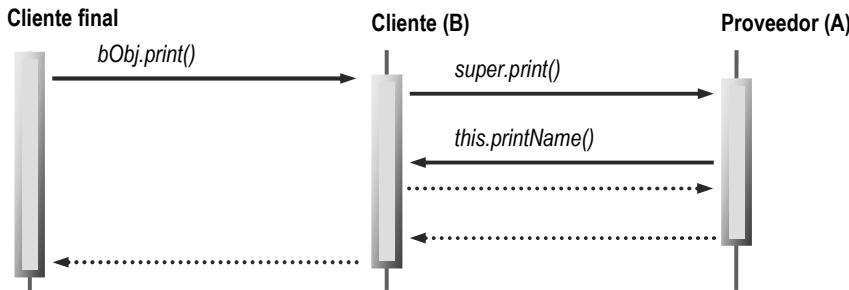


Figura 4-10 Llamadas de retorno realizadas desde los padres.

La segunda variante ve las llamadas desde el punto de vista complementario. Se asocia el código de una clase u objeto padre con el código cliente, y el código de una clase u objeto hijo con el código proveedor. Una llamada de retorno aparece cuando el código del parent hace una llamada a un método sobrescrito, y éste ejecuta un método no redefinido por el hijo o un método empleando **super**. Usando el mismo ejemplo anterior, el siguiente código genera una llamada de este tipo:

```

A aobj = new B();
aobj.print();

```

En este caso, el mensaje **print** enviado a **aobj** es respondido por el método asociado definido en **B**. El resultado en la consola es otra vez la cadena “**B**”, siguiendo el mismo conjunto de llamadas que en el ejemplo anterior. La Figura 4-11 muestra la cadena de llamadas:

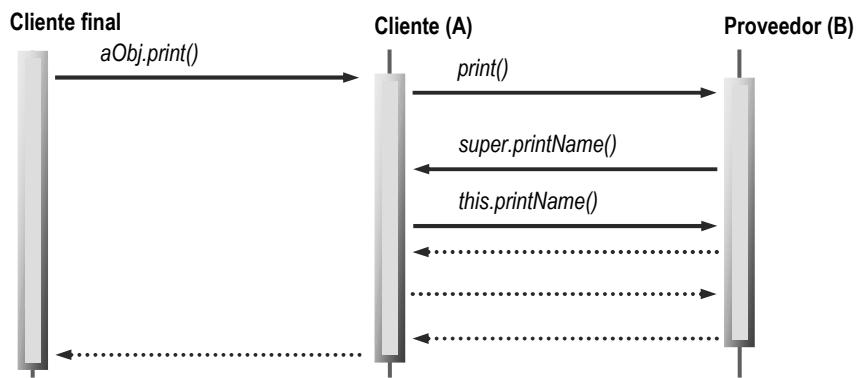


Figura 4-11 Llamadas de retorno realizadas desde los hijos.

Algunas veces el comportamiento descrito es el esperado —si se conoce la existencia de la clase **B**— pero otras veces pudiera no serlo —si la clase **B** no existiera cuando se creó la clase **A**—. En particular, nótese que la llamada a `printName` de **B** no actualiza el contador privado `printNameCount` de **A**. La modificación obvia en el método `printName` de **B** no funciona como se esperaría:

```
void printName()
{
    super.printName();
    System.out.println("B");
}
```

porque tiene como efecto secundario la impresión adicional en la consola de la cadena “**A**”. Tal y como está escrito el código, no se puede actualizar el contador privado e imprimir únicamente la cadena “**B**”. Sería necesario rescribir la clase **A** para que tenga en cuenta que la variable de instancia privada `printNameCount` fuese accesible desde clases descendientes. Pero esa modificación no tiene porqué ser esperada o planificada si no se considera desde un principio que la clase **A** pudiese tener clases derivadas. De hecho, objetos de tipo **A** funcionan sin problemas si **A** no tiene descendientes. La clase **A** pudo haber pasado satisfactoriamente sus pruebas de funcionalidad sin advertir el problema potencial de la variable privada.

La diferencia entre ambos tipos de llamadas de retorno es sutil, pero importante. En el primer caso el centro de atención —el código cliente— es la clase hija **B**. En el segundo caso es la clase padre **A**. En el primer caso el código cliente recibe llamadas de retorno provenientes de sus ancestros, en el segundo las recibe de sus descendientes. En ambos casos el código que es el foco de atención puede ver modificado su comportamiento debido a la interacción con sus ascendientes o descendientes. El cambio de centro de atención es importante porque identifica problemas diferentes: en el primer caso, se reconocen interacciones con ascendientes existentes, en el segundo se valoran interacciones con quizá futuras clases descendientes. Por ejemplo, desde el segundo punto de vista, al escribir el código de la clase **B** habría que tener en cuenta posibles interacciones que surgiesen de la existencia de una futura clase **C** que heredase de **B**. Pero es muy posible que esas interacciones no se puedan prever a priori sin conocer el cometido de las futuras clases.

En el apartado 2.7.2.5 se identificaron varias modalidades de aserciones locales asociadas con las llamadas de retorno. Puesto que la herencia genera también llamadas de retorno, esas aserciones son también aplicables a la herencia:

- *Definición implícita de un protocolo de llamadas de retorno.* Corresponde con todos los métodos potencialmente sobreescrribibles por clases u objetos derivados. El protocolo de llamadas de retorno es bidireccional si la clase u objeto no es una raíz u hoja de una jerarquía de herencia. A diferencia de las llamadas de retorno entre bibliotecas procedurales tradicionales, las que ocurren durante herencia no se presentan siempre explícitamente en el código, sino que muchas veces surgen implícitamente a partir de la definición de nuevos descendientes.

- **Asincronía.** En presencia de múltiples ascendientes y descendientes es difícil identificar cuándo un método dado va a ser ejecutado. Su comportamiento se parece más a una llamada asíncrona que a una llamada síncrona secuencial. Nótese cómo con herencia no es necesario introducir eventos externos, solamente nuevas definiciones de ascendientes o descendientes.
- **Temporalidad.** Los ascendientes o descendientes son capaces de realizar llamadas de retorno en determinado orden, cuya causa directa es la implementación interna de los mismos. Dependiendo del orden de llamadas de ascendientes o descendientes puede ser fuente de serios problemas si esa funcionalidad es implícita. Otro origen de temporalidad surge cuando dos métodos en una misma clase u objeto cooperan para resolver una funcionalidad común. A veces existe temporalidad en el orden que se espera que éstos se ejecuten. Tal orden no tiene porqué garantizarse si existen descendientes que sobrescriban algunos métodos.
- **Información de estado intermedio del proveedor.** Hemos visto que una clase u objeto puede hacer tanto de proveedor como de cliente, dependiendo del foco de atención. En herencia, la información de estado no privado del proveedor está disponible para todas las clases u objetos presentes y futuras que formen parte del mismo árbol de herencia. No existe envío de información al cliente porque su acceso es implícito a través de la pseudo-variable `self` o `this`. De modo efectivo, el estado no privado es compartido entre proveedores y clientes, muchas veces sin ningún tipo especial de control, que introduce dependencias sutiles y profundas entre los distintos miembros de una jerarquía. En el ejemplo anterior, incluso el valor de la variable de instancia privada `printNameCount` se ve afectado por la interacción entre padre e hijo.
- **Llamadas de retorno encadenadas.** Este tipo de llamadas —la variante más problemática de las llamadas de retorno— son más la norma que la excepción cuando se usa la herencia. Ni siquiera es necesaria la modificación del estado compartido a través de `self` o `this`. Las llamadas de retorno son implícitas a la propia definición de los métodos, por lo que éstas se disparan únicamente por la definición adicional de métodos sobrescritos. Las llamadas encadenadas hacen más difícil dar un estado consistente, puesto que éste puede cambiar imprevisiblemente por medio de redefiniciones en los hijos.

Las aserciones locales de llamadas de retorno en presencia de la herencia tienen un impacto más profundo que en bibliotecas procedurales tradicionales con llamadas de retorno. Pocas veces se identifican estos problemas, asociándose —a menudo de forma simplista— herencia con reusabilidad de código y elogiando las virtudes del código polimórfico. A la luz de las aserciones anteriores debe quedar claro que reusar código heredado en métodos es mucho más complejo que reusar el código de una función o procedimiento tradicional no polimórfico. El código de los métodos incluye implícitamente llamadas polimórficas que pueden desencadenar llamadas de retorno presentes y futuras, debidas a la definición de nuevos antecesores o descendientes, y que generan interdependencias implícitas. Algo que no ocurre en las funciones y procedimientos tradicionales. Las interacciones de las llamadas de retorno en la herencia pueden volver incorrecto código que originalmente no lo era, como ocurre con el método `printName` de la clase `A` del ejemplo anterior al definir una nueva clase `B`. ¿Qué comportamiento de `B` quedará invalidado cuando se cree una nueva clase `C` que herede de `B`? ¿Qué se podrá hacer si el código de la clase `B` o de la clase `A` no está disponible en ese momento?

El origen de estos cambios de comportamiento está en el *cambio implícito de los algoritmos* descritos en el código durante las llamadas polimórficas, que potencialmente puede introducir errores. Así, la reusabilidad que da la herencia está íntimamente relacionada con la posibilidad de disponer del código fuente de las clases u objetos intervenientes. De manera que, si se descubren anomalías de funcionamiento debidas a nuevas clases u objetos, el código existente pueda ser refactorizado para acomodar los nuevos comportamientos que surgen de las nuevas definiciones. Por esa razón se denomina a veces a la herencia *reuso de caja blanca* [Gamma, Helm, Johnson & Vlissides, 1995; p. 19], [Szyperski, 1998; p. 137], cuya “virtud” más sobresaliente es su notable *falta de modularidad*.

Un último aspecto relevante a tener en cuenta es la similitud entre el comportamiento de bibliotecas procedurales que registran llamadas de retorno, y árboles de herencia cuya profundidad es 2—elementos raíz o padres y elementos derivados un solo nivel o hijos—. Si el nuevo código cliente ataña únicamente a clases u objetos que heredan de los padres, y de aquellos no se pueden heredar nuevas clases u objetos, entonces las llamadas de retorno que surgen de la definición de nuevos descendientes de los hijos no se presentan en la práctica. En estas condiciones es más sencillo construir clases u objetos que funcionen correctamente, ya que se elimina un conjunto completo de interdependencias implícitas. Los métodos definidos en los hijos contienen sólo llamadas de retorno desde los padres, no desde futuros descendientes. Las llamadas de retorno desde los padres son equivalentes a las llamadas de retorno de una biblioteca procedural. A medida que se añaden más niveles de herencia, bien a los padres bien a los hijos clientes, el escenario se transforma paulatinamente en otro donde las llamadas de retorno provenientes de ascendientes y descendientes se vuelven cruciales, incrementando la complejidad percibida de la jerarquía de herencia.

4.3.9 El problema de la clase base frágil

Se trata de un problema serio de falta de modularidad que aparece durante la evolución de un diseño orientado a objetos que incluye herencia [Szyperski, 1998; pp. 102-104]. Se resume en que la nueva versión de una clase antecesora puede ser incompatible con sus descendientes. El problema se agrava con clases u objetos básicos que tienen muchos descendientes. Para simplificar la exposición, supongamos únicamente una clase antecesora o base **A** y una clase descendiente **B** que hereda de la anterior directa o indirectamente.

4.3.9.1 El problema sintáctico de la clase base frágil

La primera mitad del problema es sintáctico. Se refiere a la compatibilidad binaria del descendiente compilado **B** cuando aparece una nueva versión de su antecesor **A**. Una actualización modular del antecesor **A** no debería obligar a la recompilación del descendiente **B**. Algunos cambios pueden ser sintácticos, como mover la implementación de un método entre dos ascendientes de **B**, o añadir algunas variables de instancia privadas. Nótese que los cambios pueden tener también consecuencias semánticas, pero por ahora las ignoraremos. Si cambios sintácticos en los antecesores precisan la recompilación de los descendientes, entonces se produce el problema sintáctico de la clase base frágil. Se denomina así porque las clases base o raíz de herencia suelen ser las más afectadas. Un grupo importante de modificaciones sintácticas son los cambios monótonos referidos en la quinta premisa de diseño (Def. 4-9). Estos cambios pueden llegar a producirse sin caer en el problema sintáctico de la clase base frágil si se efectúan con cuidado. Por ejemplo, el modelo de objetos SOM de IBM hace frente al aspecto sintáctico construyendo las tablas de métodos en tiempo de ejecución y organizando la estructura de objetos con diccionarios [Forman & Danforth, 1999; p. 16].

4.3.9.2 El problema semántico de la clase base frágil

La segunda mitad del problema es semántico. ¿Cómo puede un descendiente compilado **B** seguir siendo compatible con la semántica de las operaciones de una nueva versión de su antecesor **A**? Es posible que la nueva versión de **A** sólo modifique la implementación de métodos existentes. Es decir, no realizar modificaciones a su interfaz. Se trata del mismo problema que intenta abordar la tercera premisa de diseño (Def. 3-14). La aparición de llamadas de retorno durante la herencia complica la obtención de nuevas versiones de ascendientes compatibles con descendientes quizás desconocidos para sus desarrolladores. Existen varios caminos para hacer frente a estas dificultades. Algunos de ellos intentan especificar mejor cuál es la interfaz entre ascendientes y descendientes, también llamada *interfaz de especialización* [Lamping, 1993]. La especificación de esa interfaz pretende esconder el código de los ascendientes para hacer más modular el uso de la herencia, en la misma dirección que la cuarta premisa de diseño (Def. 3-15), que constata la imposibilidad de tener todo el código fuente disponible. Su objetivo es aumentar el protocolo explícito entre ambos y así disminuir las

dependencias implícitas que surgen del uso de la herencia. Cuánta información debe estar disponible en esa interfaz de especialización y cuál es su comportamiento con sucesivas versiones es todavía tema de investigación. El apartado 7.3.4 *Comprobación semántica de tipos* estudia algunas ramificaciones interesantes. Otras opciones intentan disminuir la dependencia con la herencia, puesto que ésta pierde cierta relevancia al aplicarla a entornos distribuidos. Por ejemplo, el modelo de componentes COM prohíbe la evolución de las interfaces de componentes. Nuevas versiones de interfaces generan siempre nuevas interfaces. Un componente puede ser compatible con varias versiones de una interfaz, siempre y cuando implemente cada una por separado.

4.3.10 Discusión

A lo largo de este apartado se han introducido características conceptuales y de implementación de la herencia, incluyendo sus variedades más importantes. El resto del apartado se dedica a una revisión crítica sobre esas características, seleccionando aquellas más apropiadas para los objetivos del nuevo lenguaje en proceso de definición.

La herencia no es un mecanismo simple. Sus consecuencias van más allá del mero hecho de reaprovechar protocolos existentes o su implementación. En cierto sentido, ha existido durante muchos años una confianza en las posibilidades de la herencia —y la orientación a objetos por extensión— como mecanismo director de la reusabilidad que no ha acabado de fraguar. En la práctica, la adopción de la tecnología orientada a objetos en proyectos tiene alta complejidad, baja compatibilidad con métodos existentes, dificultades de implementación incremental, y falta de reconocimiento de resultados y beneficios, aspectos estudiados en [Fichman & Kemerer, 1997; p. 48]. Igualmente, el mismo estudio sobre empresas que adoptaban la nueva tecnología pone de manifiesto que la capacidad de reuso es elusiva:

“Ninguno de los cuatro casos estudiados manifestaron un reuso consistente con la visión de la orientación a objetos, donde los desarrolladores construyen sistemas principalmente ensamblando componentes existentes. Cuando el reuso ocurrió, fue en sus niveles más básicos (clases de cadenas, contenedores de datos, clases de fecha y medidas de tiempos) o en sus aspectos menos óptimos (portar código, reaprovechar código durante una reescritura). Aunque estas vías de reuso pueden ser valiosas, una organización puede conseguir esos resultados sin adoptar la orientación a objetos.” [Fichman & Kemerer, 1997; p. 54]

[Jacobson, Griss & Johnsson, 1997; p. 37] reconocen que el reuso es difícil. Para mejorar las condiciones de reuso proponen resolver correctamente al menos los siguientes cinco factores interdependientes: visión, arquitectura, gestión organizativa, financiación y procesos de ingeniería del software, en el contexto de una familia de aplicaciones, descritas en el apartado 2.4.2. En cierto sentido, las mismas condiciones son aplicables a un desarrollo procedural, no orientado a objetos. Hacer énfasis en ellas es signo de que esas condiciones han de cumplirse en mayor grado.

4.3.10.1 Encapsulación y herencia

Resulta de especial importancia distinguir entre herencia de tipos o interfaces y herencia de implementación a la hora de hablar sobre su relación con la encapsulación. La herencia de tipos o interfaces expresamente recibe características —declaraciones de mensajes— de sus padres, por lo que inherentemente no está encapsulada según la definición (Def. 2-35). Se trata de funcionalidad explícita conocida y, por tanto, potencialmente comprobable en tiempo de compilación. Esa información es la única que se pretende usar para comunicar módulos mediante herencia, siguiendo la primera premisa de diseño del lenguaje (Def. 3-12). Como caso particular, si la herencia de tipos es además monótona, entonces una extensión realizada en un ancestro no invalida a ninguno de sus descendientes, tan sólo añade nuevos mensajes reconocibles, y no es necesario recompilar ninguno de los módulos dependientes del protocolo heredado.

La herencia de implementación, en cambio, debe mantenerse encapsulada si se sigue la segunda premisa de diseño del lenguaje (Def. 3-13). Un primer punto de conflicto surge debido a que la

herencia de implementación no mantiene la encapsulación definida en (Def. 2-35), como hemos visto en el apartado 4.3.8 *Llamadas de retorno y herencia*. La presencia de antecesores o descendientes implícitamente puede causar llamadas a otros módulos que quizá afecten al comportamiento del módulo original mediante llamadas de retorno, siendo ésta la razón por la que se dice que la herencia permite únicamente reuso de caja blanca. La herencia crea un nuevo tipo de cliente: Además del usual que usa una funcionalidad dada, existe aquél que hereda esa funcionalidad [Snyder, 1986a; p. 39].

Un segundo punto de conflicto en herencia de implementación aparece con la herencia de variables de instancia. Si un descendiente tiene acceso privilegiado a las variables heredadas de sus ancestros, como es habitual en muchos lenguajes orientados a objetos, entonces aumenta el protocolo externo de los antecesores, que implícitamente incluye información de las variables de instancia [Snyder, 1986a; p. 40], [Lieberman, 1986; p. 218]. El acceso directo a la estructura interna de los objetos añade implícitamente aserciones estudiadas en el apartado 2.7.2.3 *Aserciones de estructuras, registros y arrays*, si el lenguaje implementa un objeto como un registro continuo¹⁶, como es el caso de C++. En particular, implícitamente la construcción del registro de un objeto incluye las aserciones siguientes que codifican:

- Todos los desplazamientos de las variables de instancia del descendiente.
- El orden en el código fuente de todas las variables de instancia del descendiente.
- Todos los desplazamientos de las variables de instancia heredadas de todos los ascendientes.
- El orden en el código fuente de todas las variables de instancia heredadas de todos los ascendientes.

Los ascendientes se convierten en permeables según la definición (Def. 2-40) y se rompe la encapsulación. Así, un cambio en el orden de las variables de instancia de alguno de los ascendientes —un cambio monótono que no modifica la funcionalidad del ascendiente— puede exigir la recompilación de los descendientes, resultado que va en contra de la segunda y quinta premisas de diseño del lenguaje (Def. 3-12) y (Def. 4-9). Igualmente ocurre con un cambio monótono que añada una nueva variable de instancia. Por tanto, este trabajo se decanta por no acceder directamente a las variables de instancia heredadas ni a las variables de instancia heredables por futuros descendientes, utilizando en su lugar métodos de acceso y escritura, aun cuando incurran en una ligera pérdida de eficiencia. El beneficio se considera superior a sus inconvenientes en el caso general. La pérdida no tiene porqué ser real en la práctica. Por ejemplo, las recomendaciones de diseño en C++ para Taligent ponen de manifiesto este problema, advirtiendo que para clases no triviales debe añadirse espacio para expansiones futuras con punteros opacos `void *` [Taligent, 1994; p. 62]. La misma técnica es usada en modelo de objetos de Xt escrito en C para implementar extensiones de *widgets* [Nye & O'Reilly, 1993; p. 200]. Evidentemente, en algunas situaciones particulares de programación de sistemas, los inconvenientes pueden hacer inviable el caso general. El nuevo lenguaje debe proveer vías para mantener la eficiencia sin incurrir en sus inconvenientes. El apartado 5.4 *Protocolos, prototipos y sus variantes* y el apartado 8.7 *Optimización* examinan estos aspectos con mayor detenimiento. La herencia de métodos y cómo afecta a la encapsulación se estudia en el apartado 4.4 *Mecanismos de envío de mensajes*.

La visibilidad explícita del árbol de herencia es también una fuente de falta de encapsulación. [Snyder, 1986a; pp. 40-41] argumenta acertadamente que el árbol de herencia no debería formar parte del protocolo explícito, puesto que el cambio de algún antecesor podría invalidar a sus clientes. Si los antecesores superiores al padre son visibles al hijo, éste podría tener dependencias explícitas con alguno de ellos que podrían ser invalidadas si el padre decidiese cambiar de quién hereda su implementación. Esta situación es más palpable con la herencia múltiple de implementación. Por ejemplo, en C++ un método heredado por dos ramas de herencia se considera ambiguo. Para eliminar la ambigüedad es necesario identificar al ancestro apropiado del que se escogerá el método. La

¹⁶ Si un objeto se implementa como un diccionario, entonces no se codifican los desplazamientos de variables de instancia en los descendientes.

eliminación de la ambigüedad introduce una dependencia explícita sobre el árbol de herencia. En herencia simple no existen esas ambigüedades. Un mensaje enviado usando la cláusula `super` o `inner` no depende más que de su padre o hijo inmediato respectivamente, escondiendo el árbol de herencia. Si es posible seleccionar arbitrariamente un método de un ancestro calificando el método apropiadamente como es el caso de C++, se está favoreciendo la presencia de dependencias explícitas con el árbol de herencia. Por esa razón XC opta por usar la cláusula `super` para las referencias al ascendiente directo. Un aspecto muy relacionado con la visibilidad del árbol de herencia y la encapsulación son las jerarquías de herencia inversas, que se estudiarán en el apartado 7.3.3.

4.3.10.2 Delegación o concatenación

La elección de delegación como mecanismo para implementar la herencia tiene un serio inconveniente: la falta de modularidad que aparece por el mero hecho de compartir información entre distintos objetos. Un cambio en cualquiera de ellos afecta a todos los demás automáticamente, quizás inadvertidamente. Esta propagación automática ha sido tradicionalmente mostrada por los partidarios de los lenguajes basados en prototipos como una ventaja [Lieberman 1986; p. 222]. A la luz del análisis de aserciones locales del apartado 2.7 *Interdependencia*, este trabajo duda de la conveniencia de tal funcionalidad. Antes al contrario, la propagación de cambios debe ser supervisable para evitar la propagación de errores e interdependencias no deseadas. El sistema de tipos y una implementación adecuada son herramientas que ayudan en la tarea de limitar la propagación de interdependencias.

Si no se usa delegación, entonces el mecanismo de implementación de herencia debe ser la concatenación. Esta técnica no comparte tanta información con sus ascendientes, pero tampoco asegura siempre modularidad. Si la codificación del árbol de herencia se realiza en un registro continuo, puede dar lugar al problema inverso: la falta de adaptación del código antiguo a un cambio, que obligue la recompilación de todos los descendientes para acomodar el nuevo cambio. Problema que debe solventar la implementación de los objetos del lenguaje.

4.3.10.3 Herencia simple o múltiple

La elección entre herencia simple o múltiple debe realizarse con cuidado. A finales de los años 80 y principios de los años 90 estaba extendido el sentimiento de que los lenguajes deberían incorporar herencia múltiple. Sin embargo, hoy en día la elección no es tan clara como demuestran las opciones tomadas por lenguajes más modernos como Java, Modula-3, Ada-95 o C#. La herencia múltiple de tipos o interfaces es un sustituto de la herencia múltiple de implementación para muchos problemas, y su implementación en un lenguaje es bastante más simple. Su deficiencia principal —y también su mayor ventaja— es que no especifica ningún aspecto de implementación de la funcionalidad declarada, quedando como responsabilidad del programador.

Con herencia múltiple de implementación es necesario elegir entre diferentes estrategias de herencia de variables de instancia y métodos, revisadas en el apartado 4.3.7.3 *Herencia múltiple de implementación*. No está claro cuál de ellas es la mejor, puesto que cada una tiene su utilidad. Abarcar todas o casi todas las estrategias como hace Eiffel parece ser el único camino de implementarla correctamente, a costa de complicar los conceptos asociados con la herencia. La estrategia de linearización de herencia múltiple para simplificar su implementación vista en el apartado 4.3.7.4 tiene también desventajas notables en la claridad del código y en la simplicidad del razonamiento sobre cuál será el siguiente método a ejecutar en una llamada polimórfica. La aparición de la estrategia de los *mixins* analizada en el apartado 4.3.7.5 es en sí misma una manifestación de que el diseño orientado a objetos con herencia múltiple de implementación es complejo, siendo necesaria una aplicación más disciplinada de la herencia. [Szyperski, 1998; p. 101] indica un inconveniente llamativo de la herencia mediante *mixins*. Si los *mixins* no son instanciables, entonces no podrían heredar de una clase de una biblioteca, puesto que otros *mixins* podrían también hacerlo y duplicarían la funcionalidad heredada, lo que daría lugar a las complicaciones habituales de heredar múltiples veces los mismos métodos. Dificultades adicionales aparecen cuando se tiene en cuenta que los *mixins*

intentan muchas veces modelar adjetivos heredables, aunque no es el único modo de hacerlo. Este aspecto se verá con más detalle en el apartado 4.5.4 *Metaclases*.

Con todo, quizás la objeción más significativa a la herencia múltiple surge del análisis de interdependencia del apartado 2.7, y la relación entre herencia y llamadas de retorno del apartado 4.3.8. Al tratarse de reuso de caja blanca, quizás la herencia no deba ser el mecanismo básico de construcción de sistemas complejos, porque las interdependencias implícitas propias de la interacción con llamadas de retorno limitarán en última instancia el tamaño manejable de los programas cuya estructura esté fuertemente basada en la herencia. Esta es una percepción común de complejidad que puede encontrarse en [Gamma, Helm, Johnson & Vlissides, 1995; p. 19] y [Szyperski, 1998; p. 102]. En su lugar deben enfatizarse otras técnicas más modulares. ¿Quiere decir esto que la herencia no es útil? Probablemente aún sí lo sea. La herencia ha demostrado ser una buena técnica de implementación de caja blanca que reduce notablemente la duplicidad de código y facilita el mantenimiento de subsistemas no muy complejos sobre los que se tiene acceso al código fuente. El comportamiento de jerarquías de herencia de dos o tres niveles es similar a bibliotecas con llamadas de retorno tradicionales, cuya complejidad está acotada significativamente al evitar el uso de llamadas polimórficas. Como hemos visto, estas características se difuminan al aumentar la complejidad y profundidad del árbol de herencia. Este trabajo considera prudente limitar el poder de la herencia como mecanismo básico organizativo, fortaleciendo y fomentando en cambio el uso de módulos y protocolos. Conceptos que facilitarán la construcción de un modelo de componentes para dar estructura a los sistemas informáticos. La herencia de tipos se mantiene múltiple para mejorar la capacidad de modelización conceptual mediante protocolos y porque parece introducir muchas más ventajas que inconvenientes. Es una abstracción más simple que separa claramente la declaración e implementación de tipos de datos. Reconociendo la utilidad de la herencia de implementación como técnica de reuso de caja blanca para factorizar código fuente, y sopesando las dificultades descritas anteriormente, se selecciona para el nuevo lenguaje la herencia simple como compromiso entre complejidad conceptual y flexibilidad en la implementación.

4.3.10.4 Herencia estática o dinámica

La siguiente decisión relacionada con la herencia se refiere a cuándo se resuelve el árbol de herencia. La herencia estática no cambia durante la ejecución de un programa y tiene la ventaja de ser comprobable en tiempo de compilación. Por el contrario, la herencia dinámica admite cambios en el árbol de herencia en tiempo de ejecución. Este tipo de herencia surge con naturalidad en modelos de prototipos con delegación. Es posible, por ejemplo, obtener la siguiente funcionalidad:

“Dado que los objetos en SELF heredan de objetos, y puesto que no hay clases que fueren una estructura, los objetos pueden fácilmente alterar sus padres en tiempo de ejecución. Esta capacidad de herencia dinámica resulta ser un excelente vehículo para implementar objetos. Su comportamiento varía completamente a medida que cambia a lo largo de un pequeño conjunto de estados. Por ejemplo, una ventana que pueda ser expandida o iconificada es fácilmente implementable en SELF cambiando el padre de la ventana y heredando comportamiento bien del ícono, bien de una ventana abierta.” [D. Ungar en Smith, 1994; p. 106].

Facilitar el ejemplo de la ventana incurre en un alto coste. La imposibilidad de saber estáticamente a qué conjunto de métodos responde una ventana en un momento dado y, por tanto, la imposibilidad de saber con certeza si los mensajes enviados van a tener respuesta o no. En ese sentido Taivalsaari hace una crítica constructiva a los lenguajes basados en prototipos:

“Los partidarios de sistemas basados en prototipos han errado al hacer publicidad de los beneficios conceptuales de los prototipos, y han destacado en cambio curiosidades técnicas. Si bien funcionalidades como compartir de forma no planificada *slots* de datos, herencia dinámica, y padres priorizados son a menudo útiles, todos ellos son cuestionables e incluso peligrosos en el desarrollo de software serio a gran escala”. [A. Taivalsaari en Smith, 1994; p. 111].

Es habitual que los lenguajes dinámicos sean considerados más apropiados para construir primeras versiones de sistemas complejos que deberán rescribirse en otros lenguajes. Implícitamente en esta

afirmación se dice que, de algún modo, estos lenguajes alcanzan su límite de complejidad antes que otros lenguajes menos dinámicos, en los que se pueden construir programas más grandes y complejos. No es difícil identificar la fuente de la complejidad: se trata de la ruptura de la encapsulación y propagación combinatoria de dependencias promovida por los propios mecanismos básicos de los lenguajes. Hemos visto ya algunos ejemplos. La herencia implementada mediante delegación implícitamente vincula las modificaciones realizadas en el objeto delegado con el objeto que delega en él. La herencia implementada con concatenación puede exponer la estructura de la herencia a otros objetos si no se implementa con cuidado.

Este trabajo no comparte las ideas sobre herencia dinámica. Se trata de una funcionalidad cuya flexibilidad se intuye demasiado alta. Es difícil imaginar un sistema de comprobación automática que evite cometer errores durante la definición de relaciones dinámicas de herencia. Un cambio de herencia que afecte a los mensajes que puede recibir un objeto es equivalente a un cambio de tipos. Tal cambio sólo debería ser legal si estuviese controlado. Lenguajes dinámicos como SELF no disponen de herramientas que verifiquen ese cambio y, por tanto, se trata actualmente de una funcionalidad con mucho riesgo potencial. La herencia estática, si bien no tan flexible, tiene la propiedad de ser más previsible. Abre las puertas a comprobaciones de tipos asociadas al árbol de herencia que descubran potenciales errores durante la modificación de cualquier elemento del mismo. No obstante, incluso la herencia estática se puede complicar si se aumenta en exceso la flexibilidad.

Consideremos por ejemplo el caso de CLOS. La necesidad de compatibilidad con otras variantes orientadas a objetos de LISP —Flavors y Loops— fue una de las causas de las importantes investigaciones en protocolos de metaobjetos [Kiczales, Rivières & Bobrow, 1991; p. 78]. Las variantes eran parecidas a CLOS en muchos sentidos, pero cada una de ellas implementaba la herencia múltiple con diferente priorización de los padres (véase la Figura 4-8). CLOS necesitaba algún modo de cambiar el algoritmo de herencia múltiple usado y hacerlo compatible con el código de las variantes, para portar con menor esfuerzo el código escrito en las variantes y así facilitar la adopción de CLOS. La solución elegida fue consentir la modificación del algoritmo de linearización de herencia aplicado a cada clase.

El coste de la compatibilidad de CLOS con Flavors y Loops es el aumento de la complejidad de las abstracciones básicas del lenguaje. Las abstracciones de un lenguaje deben ser relativamente sencillas para que la atención de la programación se centre en la complejidad del problema a tratar, no en la complejidad del lenguaje usado. Si la linearización de herencia puede variar en un programa dependiendo de qué tipos de objetos se estén usando cada vez, resulta más difícil evaluar cambios o posibles errores. Ciertos comportamientos heredados serán correctos para unos objetos pero no para otros dependiendo del tipo de linearización usado. En cierto sentido es similar a que sea posible añadir o borrar métodos arbitrariamente a un objeto: no se pueden seguir entonces unas reglas fijas de razonamiento que —al menos— garanticen los métodos existentes en un objeto, sino que éstas dependen del contexto donde se ejecuten. En el caso de múltiples linearizaciones, evaluar qué método heredado finalmente se ha ejecutado depende de la linearización particular elegida, y ésta depende del tipo de objeto usado. CLOS permite llamadas polimórficas, por lo que un fragmento de código puede lidiar potencialmente con objetos cuya resolución de métodos sea distinta. Las abstracciones deben también ser consistentes. No facilita la consistencia un lenguaje que admite linearizaciones arbitrarias que varían de programa en programa o incluso dentro de un mismo programa. [Kiczales, Rivières & Bobrow, 1991; pp. 78-79] reconocen que una excesiva flexibilidad en el cambio de linearización es perjudicial y a veces inviable, optando por asumir el coste de cambios limitados en la linearización de herencia para favorecer la compatibilidad. Los *mixins* son más sencillos de usar y más disciplinados que las linearizaciones implícitas de CLOS porque, al menos, la linearización queda reflejada explícitamente en el código que combina *mixins*. En definitiva, y para limitar la flexibilidad de la herencia, el nuevo lenguaje selecciona herencia estática comprobable en tiempo de compilación.

4.3.10.5 Lenguajes basados en clases o en prototipos

Un último aspecto importante a tener en cuenta en la elección de las características del modelo de objetos y la herencia tiene que ver con la selección de clases o prototipos como abstracción básica para organizar los objetos. Esta distinción ha dado lugar a acalorados debates sobre qué modelo de objetos es más adecuado. En el apartado 4.2 *De prototipos a clases* se introdujo la motivación para tener objetos prototípicos y tener clases. En los apartados subsiguientes se analizó la herencia y su implementación desde ambos puntos de vista —objetos prototípicos y clases— que pone de manifiesto cómo ambas opciones intentan resolver muchas veces los mismos problemas mediante técnicas similares. Incluso Smalltalk-80 usa un híbrido de prototipos y clases, empleando las clases para instanciar objetos y objetos prototípico para implementar las clases.

Los lenguajes basados en clases que no poseen información de reflexión en tiempo de ejecución son conceptualmente más simples que aquellos que sí la poseen. Sin embargo, la información en tiempo de ejecución es crítica para ciertos comportamientos dinámicos o genéricos muy comunes que si no están presentes deben ser replicados en código, aumentando innecesariamente la complejidad de los programas. Los conceptos de reflexión se ven posteriormente en el apartado 4.5 *Reflexión y arquitecturas meta-nivel*. Si bien la información en tiempo de ejecución es importante, no es estrictamente necesario tener que implementar las clases o metaclasses como objetos, consiguiéndose en ese caso un equilibrio entre flexibilidad y simplicidad del modelo de objetos. Java es un lenguaje que posee esas características. Pero Java introdujo sus mecanismos de reflexión un poco tarde en la versión 1.1, cuyos efectos negativos más importantes son la presencia de tipos primitivos y objetos, y un rendimiento relativamente pobre de los mecanismos de reflexión. C# y la recientemente introducida versión 1.4 de Java mejoran estos aspectos.

Los prototipos tienen a su vez algunos defectos conocidos. En [Borning, 1986; p. 39] se enumeran varios. En un mundo sin clases ¿cuál es el número entero prototípico? ¿Cómo se describen estructuras de datos estándar como pilas o colas? No parece que haya una pila o cola prototípica, es más natural hablar de pilas o colas en general. Existe también el peligro de modificar inadvertidamente un prototípico, puesto que tiene el mismo protocolo que el resto de los objetos. Por otro lado, la clonación no siempre es la operación más eficiente. ¿Qué ocurre si se quieren cambiar todos los campos del nuevo objeto? La copia inicial de la clonación no sería necesaria.

En SELF —un lenguaje que emplea prototipos— se usan unos objetos especiales llamados *traits* para implementar propiedades compartidas entre grupos de objetos. Los objetos normales delegan en los *traits* las propiedades y métodos compartidos. Los *traits* no responden a mensajes como se podría esperar en un lenguaje donde todos los objetos son teóricamente iguales. Por ejemplo, no responden bien a mensajes de acceso a propiedades no compartidas, puesto que *no tienen* tales propiedades por definición, si bien los métodos de los *traits* sí esperan manejarlas [D. Ungar en Smith, 1994; p. 107]. En cierto sentido, los *traits* son objetos especiales en un modelo de objetos que se supone que intenta evitar la existencia de objetos especiales como las clases o las metaclasses.

Es habitual que los modelos de prototipos puedan eliminar métodos dinámicamente. Esta es una característica que no preserva la monotonía, obliga a revisar el código de los clientes, y precisa recompilar los clientes si el lenguaje es compilado y no se desea tener errores de mensajes sin respuesta. El problema inverso también ocurre. Por ejemplo, un prototípico se puede *corromper* al eliminar un método. A partir de entonces todos los objetos descendientes dejan de tener ese método. Como comenta [D. Ungar en Smith, 1994; p. 107] al manejar únicamente entes concretos y no abstractos como las clases: “Lo concreto puede ser una espada de doble filo”.

Otra complicación identificada por los diseñadores de lenguajes de prototipos es el problema del *parentesco familiar*. Los objetos con herencia por delegación tienden a tener un cierto parecido con sus hijos, pero los padres e hijos suelen jugar papeles muy diferentes, así que la idea de parentesco no parece ser siempre la apropiada [R. Smith en Smith, 1994; p. 103].

También se ha descubierto cómo surge estructura en varios lenguajes que no poseen una jerarquía de clases que imponga orden [R. Smith *en Smith*, 1994; p. 103]. Algunos lenguajes de prototipos como SELF tienen menos garantías estructurales que un lenguaje con clases, porque son demasiado flexibles. Por ejemplo, si un método requiere el acceso a una variable de instancia, la variable no está garantizada en todos los objetos descendientes [D. Ungar *en Smith*, 1994; p. 107]. Una estructura de herencia más rígida garantizaría estas restricciones estructurales. Lo mismo ocurre con las reglas de ámbito. Un modelo de objetos demasiado uniforme hace difícil encontrar un lugar consistente donde especificar y forzar las reglas de ámbito, tal y como ocurre en NewtonScript [W. Smith *en Smith*, 1994; p. 109]. Kevo propone una doble estructura paralela. La primera es gestionada de forma automática por el entorno de desarrollo e invisible al usuario, y que mantiene relaciones entre familias de objetos clonados (*clone families*). La segunda es una estructura organizativa jerárquica similar a los directorios y gestionada por el usuario [Taivalsaari, 1992; pp. 11-14].

Se puede observar que los lenguajes basados en clases proveen muchas veces un modelo de objetos que se complica con la presencia de metaclasses, heredado del modelo de objetos original de Smalltalk-80. Muchos lenguajes basados en clases que no poseen clases implementadas como objetos, adolecen en cambio de información suficiente en tiempo de ejecución. Por otro lado, los modelos de prototipos presentan un modelo más simple, pero muchas veces su implementación ha sido demasiado flexible. El problema de fondo quizás sea el deseo de diferenciación entre los lenguajes basados en clases y prototipos, si bien ambos modelos están muy relacionados e intentan resolver los mismos problemas, a veces incluso con técnicas muy similares de implementación. La diferenciación aparente nace de la aplicación de distintas técnicas de implementación en cada caso. La elección de implementación mediante delegación y herencia dinámica—típicas de los lenguajes de prototipos más relevantes—aumentan las diferencias percibidas entre lenguajes basados en clases y prototipos. Pero esas discrepancias son superficiales. El análisis efectuado en este capítulo, manteniendo las premisas de diseño del lenguaje descritas en el apartado 3.3, acerca más que aleja las dos perspectivas.

Este trabajo se decanta por hacer una síntesis de ambas perspectivas —clases y objetos— pretendiendo no añadir ningún concepto superfluo al modelo de objetos y haciendo énfasis especial en las características modulares. A la vez, intenta ofrecer razonable flexibilidad siempre que sea comprobable en tiempo de compilación. El modelo de prototipos es conceptualmente más sencillo que el modelo de clases, puesto que cada objeto puede ser usado como prototipo para crear nuevos objetos. Elimina el concepto de clase y la tentación de añadir metaclasses porque deja de resultar natural. Por otro lado, si se usa un sistema de tipos fuerte, compilación y herencia estática, en la práctica la definición de un objeto prototipo es poco diferente a la definición de una clase. El lenguaje protege la definición de los objetos prototipo para evitar que cambios dinámicos afecten a los razonamientos que puedan hacerse a partir del estudio del código fuente. El objeto prototipo definido en el código fuente realiza las funciones de una clase en tanto en cuanto se utilice para clonar nuevos objetos, aunque también pueden usarse otros objetos para obtener otros valores por defecto. Si se desea, el concepto de clase surge con naturalidad a partir de los objetos prototipo: una clase es simplemente un objeto prototipo que se usa a menudo para describir otros objetos. Como se verá en el apartado 5.2.1.4 *Creación e inicialización de objetos*, tampoco existen diferencias prácticas en cómo son creados los objetos, diferencia que se ha considerado tradicionalmente notable entre las dos perspectivas. Se trata más bien de discusiones filosóficas: ¿Qué es mejor: copiar un objeto instanciado de una clase o clonar un objeto? Probablemente la respuesta a esta pregunta no tenga relevancia práctica, ya que dan el mismo resultado y una implementación eficiente es similar, si no igual. Es una indicación de que quizás sea más sencillo unificar ambas perspectivas.

4.4 Mecanismos de envío de mensajes

En el apartado 3.4.4 *Envío de mensajes*, se hizo una introducción al envío de mensajes y a su importancia en los modelos de objetos y componentes. Sin embargo, no siempre se ha prestado

suficiente atención al proceso de envío, eligiéndose durante el diseño de algún nuevo lenguaje o modelo de componentes un modo de implementación que existiese previamente. En este apartado estudiaremos varios mecanismos de envío de mensajes desde el punto de vista del análisis de aserciones locales e interdependencia vistos en el apartado 2.7 *Interdependencia*. Su análisis se ha trasladado a este capítulo para poder introducir con mayor profundidad primero los conceptos de orientación a objetos y herencia.

4.4.1 La resolución de métodos

La herencia en lenguajes orientados a objetos impone la semántica que ha de seguir el proceso de resolución de métodos. En el apartado 4.3 se han visto las razones por las que XC usa herencia simple de implementación. Por ello, el resto de esta discusión se centrará en mecanismos de envío de mensajes con herencia simple, si bien buena parte de ella es extrapolable directamente a lenguajes con herencia múltiple.

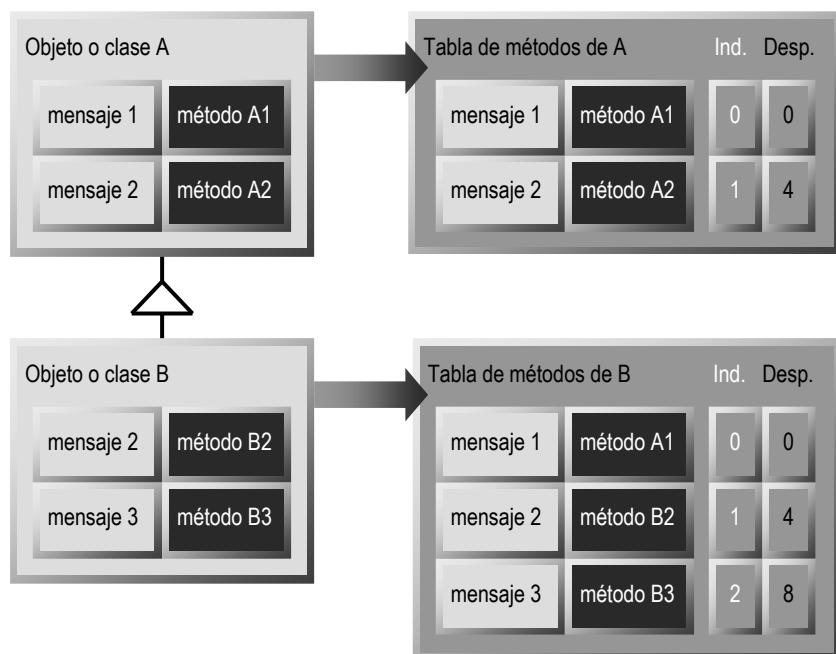


Figura 4-12 Tablas de métodos.

La definición de clases u objetos indica los métodos asociados a cada mensaje. La herencia describe cómo construir la tabla de todos los métodos a los que responde una clase u objeto, para que el procedimiento de resolución de métodos sea más rápido. La tabla de métodos de B ya ha resuelto que el objeto B responde al mensaje 1 con el método heredado A1 y responde al mensaje 2 con el método redefinido B2. A cada entrada de la tabla se le pueden asociar índices y desplazamientos para un procesador de 32 bits.

La resolución de métodos es un procedimiento que sirve para encontrar el método asociado a un mensaje. En herencia simple, debido a la existencia de un solo parente, la selección del método correcto no tiene ambigüedades: hace falta únicamente la inspección de qué método responde a un mensaje comenzando desde la clase u objeto receptor del mensaje. Si no se encuentra el método, se repite el proceso buscando en los antecesores correspondientes. El resultado es una lista de métodos —heredados o no— asociados uno a uno con una lista de mensajes. El proceso se resume en la Figura 4-12.

- ❖ **Def. 4-16.** Una *tabla de métodos* asocia uno a uno el conjunto de mensajes a los que responde un objeto receptor con una lista de métodos para ser invocados como respuesta.

Si la herencia es dinámica, el contenido de la tabla de métodos puede variar en tiempo de ejecución. Se corre entonces el riesgo de enviar mensajes y no recibir respuesta. El procedimiento de resolución se debe realizar bien durante el envío de cada mensaje, bien generando una nueva tabla de métodos cuando la herencia cambia. XC tiene herencia estática para realizar todas las comprobaciones en tiempo de compilación y evitar el envío de mensajes sin respuesta.

Una opción intuitiva para implementar la tabla de métodos es utilizar un diccionario, donde a cada nombre de mensaje se le asocia un método. Precisamente fue esa la implementación original de Smalltalk-80 [Goldberg & Robson, 1989; p. 420-421]. También fue elegida por el modelo de objetos SOM [IBM, 1994] escrito sobre C++ [Forman & Danforth, 1999; p. 19]. El acceso con diccionarios es relativamente costoso, así que resulta interesante la búsqueda de mecanismos más eficientes. Es posible optimizar el acceso usando un *array*, que es un diccionario degenerado. Es la opción elegida por C++ [Stroustrup, 1994; pp. 74-76] y Objective-C [Cox & Novobilsky, 1991; pp. 151-156]. Se asigna a cada mensaje un índice y la tabla de métodos se implementa como un *array* que almacena directamente los métodos. Cuando se envía un mensaje se traduce el mensaje por su índice asociado y se accede al *array* en la posición identificada por el índice del mensaje.

4.4.2 Tablas virtuales

La técnica de tablas virtuales de métodos proviene de SIMULA-67, y es usada también por C++ y Eiffel [Zendra, Colnet & Collin, 1997; p. 127]. Su característica más sobresaliente es ser muy eficiente. La definición de clases mediante herencia simple precisa implícitamente una ordenación de métodos, desde la clase raíz hacia las clases derivadas. Esta ordenación se muestra también en la Figura 4-12. La ordenación de métodos se traduce en desplazamientos sobre un *array* que contiene directamente los punteros a los métodos: en la Figura 4-12 el *mensaje1* corresponde con el primer elemento del *array*, el *mensaje2* con el segundo elemento y así sucesivamente.

El procedimiento de resolución de métodos en C++ lo soluciona directamente el compilador. Los objetos de C++ cuando usan herencia simple contienen como primer campo (desplazamiento cero) el puntero a la tabla virtual de métodos. Cuando se envía un mensaje a un objeto en C++, el compilador identifica el índice asignado al mensaje y genera la llamada al método como una indirección sobre la tabla virtual. Así, en C++ y siguiendo el ejemplo de la Figura 4-12, para el envío del *mensaje2* a un objeto *bobj* cuya clase es *B*, desde un código cliente se escribe:

```
bobj->mensaje2();
```

Si la declaración de *mensaje2* fuese *virtual* y si el compilador de C++ usase el procesador hipotético de 32 bits del apartado 2.7.2, implementaría un procedimiento de resolución de métodos similar al siguiente¹⁷:

```
MOV R1, bobj      ; Carga la dirección del objeto bobj (this)
MOV R2, [R1]       ; La tabla virtual está en desplazamiento cero del objeto
PUSH R1           ; this es el primer y único parámetro de llamada.
CALL [R2+4]        ; Llamada al método B2: 2º índice de la tabla virtual es el
                  ; desplazamiento 4 para un procesador de 32 bits.
```

El aspecto más importante del ejemplo anterior es la última línea: el desplazamiento del método asociado a *mensaje2* se codifica directamente en el código cliente. El mismo código funciona correctamente si la variable *bobj* contuviese un objeto de clase A. La clase A también responde a *mensaje2*, y el desplazamiento del método asociado en su tabla virtual es el mismo. Así es cómo C++ implementa el polimorfismo asociado a la herencia simple: un mismo mensaje tiene igual desplazamiento en todas las subclases. En este ejemplo se puede comprobar que el código generado es perfectamente aplicable tanto a objetos de clase A como de clase B.

¹⁷ Nótese que por claridad se está obviando el uso de optimizaciones en el ejemplo.

Ahora consideremos potenciales problemas de este esquema. En primer lugar, los desplazamientos de tabla virtual asociados a los métodos de `B` dependen del número de mensajes declarados en `A`: `mensaje2` está declarado originalmente en `A`, tiene índice 1 y desplazamiento 4; `mensaje3` está definido únicamente en `B` así que toma índice 2 y desplazamiento 8, la siguiente entrada libre en la tabla virtual. Hasta aquí todo está bien. Supongamos que `A` declara un mensaje nuevo —digamos `mensaje0`— y se le asigna el índice cero, entonces `mensaje0` tiene desplazamiento cero en la tabla virtual. Pero eso quiere decir que `mensaje2` tendría asignado el índice 2 y desplazamiento 8. Igualmente, y puesto que `B` también hereda a `mensaje0`, el índice asociado a `mensaje3` pasa a ser 3. Se puede ver que el nuevo desplazamiento de `mensaje2` es diferente al especificado en el código cliente del ejemplo anterior, que usa un objeto de la clase `B` y supone que `mensaje2` se encuentra en el desplazamiento 4. Si no se recompila el código cliente, éste estaría accediendo erróneamente a `mensaje1`. Es decir, añadir un método en la clase `A` invalida el código cliente que accede a objetos de clase `B` porque éstos heredan de `A` y deben recompilarse. Este es claramente *un comportamiento no modular* de la implementación mediante tablas virtuales que codifican desplazamientos en el código cliente.

Es más, dependiendo de dónde se ha insertado `mensaje0` en la clase `A`, los índices asignados a los mensajes son diferentes. Si se inserta el último, adquiere el índice 2 en lugar del cero. Pero eso no es todo: sin necesidad de añadir ningún `mensaje0`, si se cambia el orden actual de declaración entre `mensaje1` y `mensaje2`, los índices asignados a ambos también cambian y, por tanto, sus desplazamientos dentro de la tabla virtual. Es decir, un cambio de posición de métodos en el código fuente en C++ —que no incluye en absoluto nueva funcionalidad— *tampoco es modular*.

Desde el punto de vista de la interdependencia, lo que está ocurriendo es que el compilador añade por su cuenta aserciones en el cliente acerca de la estructura de la tabla de métodos del proveedor, que en el ejemplo era `B`. La información introducida indica que el método asociado a un mensaje se va a encontrar en un desplazamiento dado. Pero la tabla de métodos codifica mucha más información implícitamente:

- Todos los métodos del proveedor.
- El orden en el código fuente de todos los métodos del proveedor.
- Todos los métodos heredados por el proveedor.
- El orden en el código fuente de todos los métodos heredados por el proveedor.

Toda esa información son aserciones locales que hace el cliente acerca del proveedor, y forman parte de la funcionalidad pública del proveedor, *aun cuando no se especifique en ningún sitio*. La técnica de tablas virtuales es, según (Def. 2-40), *permeable*, puesto que rompe la encapsulación no sólo del proveedor, sino de todos los ascendientes de los que éste último hereda. Es un ejemplo de transmisión combinatoria de dependencias entre distintos fragmentos de código.

4.4.3 Selectores

La técnica de selectores se originó con Smalltalk-80 [Goldberg & Robson, 1989; pp. 420-421]. La implementación es diferente, pues se trata de un lenguaje que se ejecuta en una máquina virtual, pero las propiedades generales de los selectores se mantienen. En [Driesen, Hözlé & Vitek, 1995; pp. 259-260] las llaman STI (*Selector Table Indexing*).

- ❖ **Def. 4-17.** Un *selector* identifica a un mensaje únicamente mediante un índice único asociado al mensaje.

Los selectores mantienen una relación 1:1 entre índices y mensajes. En cambio, en la técnica de tablas virtuales, un mismo índice puede identificar a distintos mensajes. Por ejemplo, dos clases no relacionadas por herencia tendrán diferentes mensajes con índice cero. Es decir, con tablas virtuales no hay relación 1:1 entre índices y mensajes en general, aunque sí se mantiene la relación 1:1 entre índices y mensajes si las clases están relacionadas por herencia simple.

Asignar selectores únicos es equivalente a tener una tabla de métodos cuyo tamaño es el número máximo de mensajes diferentes. El índice del selector sirve entonces para realizar una indirección en esa tabla de métodos y encontrar con rapidez el método asociado a un mensaje. Para clases u objetos no relacionados mediante herencia, el número de mensajes distintos puede ser muy significativo. Al exigir la propiedad de identificación única a los selectores, una parte importante del espacio de las tablas de métodos se encuentra vacía. La Figura 4-13 ilustra estas particularidades.

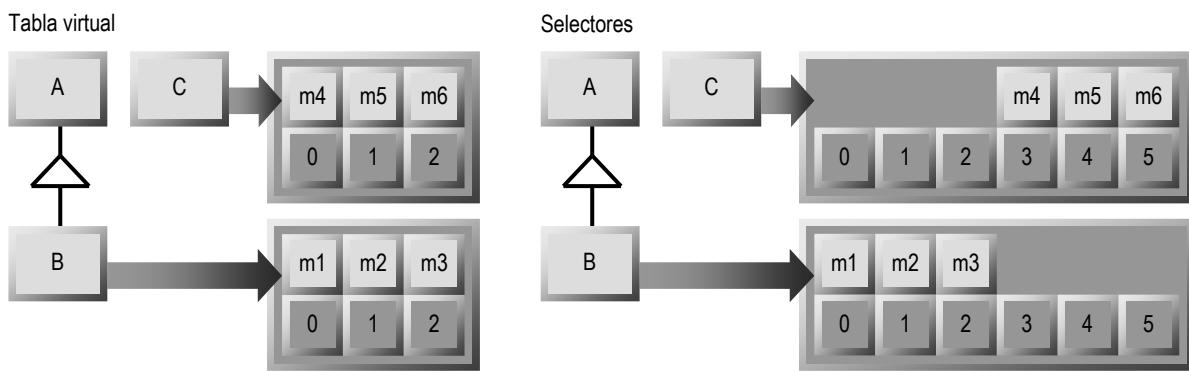


Figura 4-13 Tablas de métodos para las técnicas de tabla virtual y selectores.

A y B están tomados de la Figura 4-12. Definen los mensajes m1, m2 y m3. C define nuevos mensajes m4, m5 y m6. Con tablas virtuales, las tablas de métodos están compactadas y un mismo índice representa varios mensajes. Con selectores existen más entradas en la tabla que mensajes, para que un índice identifique cada mensaje únicamente.

En principio, mantener selectores únicos de mensaje parece poco modular. ¿Cómo se asegura la elección de identificadores únicos si partes de código se compilan por separado? Todos los fuentes de un programa deberían estar disponibles para poder realizar correctamente la elección de selectores. Es más, cualquier cambio en un programa obligaría a reconsiderar todos los fuentes otra vez para regenerar la lista de selectores. Para evitar estas dificultades, la elección de selectores no ha de realizarse durante la compilación. Debe existir un procedimiento de registro de mensajes que permita asignar a cada uno de ellos un selector único en tiempo de ejecución¹⁸.

El índice del selector asociado a un mensaje varía de hecho de una ejecución a otra, ya que el índice asignado a un selector depende del orden de registro de los mensajes, y éste no se puede asegurar en presencia de bibliotecas dinámicas. Si bien el índice del selector puede variar, en cambio la relación entre un índice de selector y un mensaje sigue siendo 1:1. Esta relación es la usada para acceder a la tabla de métodos. El procedimiento de resolución de envío de mensajes usa una *función mensajera*. Por ejemplo, el envío del mensaje `mensaje1` al objeto `obj` con los argumentos `arg1 ... argn` se traduce en una llamada a la función mensajera. A esta función se le pasa como parámetros el objeto receptor, el selector asociado al mensaje a enviar y los parámetros del mensaje. En C puede escribirse como:

```
sendmsg (obj, selmsg1, arg1 ... argn);
```

donde la función mensajera tiene la siguiente declaración en C:

```
Object * sendmsg (Object *self, Selector *sel, ...);
```

El índice del selector se ha inicializado en tiempo de ejecución durante el registro de mensajes con selectores. El algoritmo de la función mensajera accede al índice dentro del selector, que contiene el desplazamiento dentro de la tabla de métodos donde se encuentra el método asociado al mensaje enviado. Una vez encontrado, la función mensajera efectúa la llamada al método pasando como parámetros el objeto receptor y los argumentos del mensaje. Pasar una referencia al selector del

¹⁸ En [Driesen, Hözle & Vitek, 1995; p. 257, pp. 259-260] se clasifica la técnica de selectores como estática. Con una implementación estática, los índices de los selectores deben conocerse todos en tiempo de compilación, perdiendo prácticamente toda su utilidad.

mensaje es la esencia de la técnica de selectores: *permite no codificar en el cliente el desplazamiento dentro de la tabla de métodos*. Esta diferencia con la técnica de tablas virtuales es sutil pero fundamental. La técnica de selectores es más modular porque evita la fuente de dependencia de las tablas virtuales. No fuerza la recompilación del código cliente cuando se añaden nuevos mensajes que no afectan al cliente en sí. Por supuesto, tampoco precisa recompilación cuando se reordenan mensajes en el código fuente.

4.4.4 Otras técnicas de resolución de métodos

Las tablas virtuales y los selectores son mecanismos comunes de resolución de métodos. Existen por supuesto otras técnicas de resolución, cada una con sus ventajas e inconvenientes. La técnica de multimétodos es probablemente la más importante e interesante, ya que el mecanismo de resolución trasciende a las propiedades del lenguaje. Por su relevancia se revisa a continuación. Otras técnicas notables son los algoritmos de coloración de mensajes para tablas de métodos, de desplazamiento de tablas de métodos, y las cachés polimórficas en línea. Estudios más extensos de las mismas se encuentran en [Driesen, Hölzle & Vitek, 1995] y en [Holst & Szafron, 1997]. Por estar muy relacionadas con aspectos de implementación, su revisión se retrasa hasta el apartado 8.4 *Implementación del envío de mensajes*.

4.4.4.1 Multimétodos

Los multimétodos son listas de métodos asociadas a un mismo mensaje que difieren únicamente en el tipo de los parámetros. Téngase presente que los multimétodos son diferentes a la sobrecarga de funciones o métodos. Los multimétodos son originarios de CommonLoops [Bobrow *et alii*, 1986]. Otros lenguajes que poseen multimétodos son CLOS y Dylan [Apple, 1995]—donde reciben el nombre de *funciones genéricas* o *generic functions*— y Cecil [Chambers, 1992]. Constituyen un modo más general de envío de mensajes que el estudiado en los apartados anteriores, en donde el método elegido depende del tipo de *todos* los parámetros, no únicamente del receptor del mensaje. Por esa razón, los multimétodos también reciben el nombre de *envío múltiple* o *multiple dispatch*, en contraposición con el envío de mensajes tradicional, de *envío simple* o *single dispatch*.

Con envío simple, el procedimiento de resolución de métodos usa el tipo de datos del objeto receptor para seleccionar el método asociado a cada mensaje enviado. Los otros argumentos de los mensajes enviados no participan en el procedimiento de resolución. Sin embargo, existen ciertos tipos de mensajes comunes que no tienen un receptor claro. Por ejemplo, las operaciones aritméticas entre enteros involucran a dos números enteros. Cuando se usa envío simple, debe elegirse artificialmente a un receptor de la operación aritmética, aunque los dos números son candidatos igualmente válidos. La misma situación aparece cuando se envían mensajes donde el receptor y los argumentos son polimórficos. Dibujar objetos gráficos en diferentes dispositivos es un ejemplo de este tipo. El mensaje de dibujado ha de tratar simultáneamente con objetos (rectángulos, círculos, ...) y dispositivos (impresoras, pantallas, ...), ambos polimórficos [Ingalls, 1986; pp. 347-348]. Por ejemplo, supongamos que `GraphicalObject` representa objetos gráficos y `Port` dispositivos de salida. Usando Java, la siguiente llamada polimórfica:

```
gobj.displayon (port);
```

donde `gobj` es un objeto derivado de `GraphicalObject` y `port` es un objeto derivado de `Port`, se trata de una llamada especializada tanto en `gobj` como en `port`. El envío simple selecciona el objeto gráfico adecuado de la variable polimórfica `gobj`, pero no resuelve la selección del puerto de salida para dibujar la figura, que debe programarse explícitamente. Para una hipotética clase `Rectangle` se tendría en Java:

```
class Rectangle extends GraphicalObject
{
    // ...
```

```
void displayOn (Port port)
{
    if (port instanceof Printer)
        { // ... }
    else if (port instanceof Display)
        { // ... }

    // ...
}
```

La técnica de *reenvío de mensajes* o *double dispatch* descrita en [Ingalls, 1986] puede usarse para resolver estos envíos con múltiples argumentos polimórficos en lenguajes que únicamente soporten el envío simple. La idea consiste en incluir envíos de mensaje adicionales que seleccionen polimórficamente cada argumento. En el caso anterior, el método `displayOn` de `Rectangle` debe seleccionar polimórficamente el puerto de salida:

```
void displayOn (Port port)
{
    port.displayRectangle (this);
}
```

y en cada clase derivada de `Port` debe añadirse un método de presentación por objeto gráfico, que describe cómo se presenta cada objeto gráfico en cada dispositivo de salida. Recuérdese que los envíos de mensajes siguen siendo simples:

```
class Printer extends Port
{
    void displayRectangle (Rectangle gobj)
    { // ... }

    void displayCircle (Circle gobj)
    { // ... }

    // ...
}
```

Igualmente, el dispositivo `Display` debe incluir su propio conjunto de métodos de presentación. El reenvío de mensajes tiene dos inconvenientes: solucionar el problema proceduralmente y aumentar su complejidad con el número de parámetros polimórficos. ¿Es posible gestionar mejor los mensajes con múltiples argumentos polimórficos?

El envío múltiple es una respuesta a esa pregunta. Resuelve los casos anteriores de un modo más elegante, declarativamente. Todos o algunos de los argumentos de un mensaje pueden participar en la selección final del método que se ejecuta. La selección del método exacto usa la especialización de varios argumentos. Usando el ejemplo anterior, en una hipotética extensión a Java con multimétodos, deben definirse los métodos siguientes para los dispositivos `Printer` y `Display`, y para los objetos gráficos `Rectangle` y `Circle`:

```
void displayOn (Rectangle gobj, Printer port)          // multimétodo
{ // ... }

void displayOn (Circle     gobj, Printer port)          // multimétodo
{ // ... }

void displayOn (Rectangle gobj, Display port)           // multimétodo
{ // ... }

void displayOn (Circle     gobj, Display port)           // multimétodo
{ // ... }
```

El resultado es, en principio, más claro, natural y simple. El mensaje `displayOn` tiene asociados múltiples métodos. La selección del método apropiado depende del tipo de ambos parámetros y el código se escribe directamente en los métodos adecuados sin necesidad de mensajes intermedios. Ciertamente es menos propenso a errores que la técnica de reenvío. Nótese que *no* se está definiendo un conjunto de funciones sobrecargadas al estilo de C++. Son distintas definiciones de los métodos asociados a un multimétodo llamado `displayOn`. La diferencia fundamental entre ambas es que la

selección de la función sobrecargada en C++ se realiza comparando los tipos de los argumentos en tiempo de compilación, mientras que con multimétodos la comparación se efectúa en tiempo de ejecución. La implementación original de multimétodos se detalla en [Bobrow *et alii*, 1986]. Una descripción de la implementación de multimétodos para CLOS puede encontrarse en [Kiczales & Rodríguez, 1990]. La correspondiente de Cecil se halla en [Chambers & Chen, 1999].

Como se comentó antes, no existe un claro receptor del mensaje `displayon`, dado que ambos argumentos lo son por igual. Por eso resulta embarazoso incluir la definición de un multimétodo dentro de alguna de las clases de los receptores. Si fuese así, se estaría dando preferencia a alguna de ellas. [Chambers, 1992b; pp. 35-36] caracteriza bien el problema: los multimétodos fomentan un estilo de codificación más descentralizado, más parecido a la programación funcional. De hecho, la implementación de multimétodos usa técnicas similares a los encajes de patrones o *pattern matching* de programación funcional. El envío simple fomenta la agrupación natural de métodos con datos y la abstracción de datos. Los multimétodos, al no pertenecer a ninguna clase, no fomentan la encapsulación. Es más, ciertamente van en contra de ella, puesto que un multimétodo debe tener acceso a la representación interna de sus argumentos. Esta interpretación es más natural en lenguajes creados sobre LISP, donde el lenguaje raíz no fomenta la encapsulación.

Para volver a centrar la atención en la modularidad y la encapsulación, [Chambers, 1992b; p. 37] propone cambiar la interpretación de los multimétodos. En vez de considerar que se encuentran fuera de las definiciones de clases, supone que los multimétodos se encuentran en *todas* las definiciones de las clases asociadas a los tipos de sus argumentos. Interpreta que en envío simple se envía un mensaje al objeto receptor, mientras que en envío múltiple se envía el mensaje a todos los argumentos, que son los receptores. [Chambers & Leavens, 1994] proponen un sistema de módulos para Cecil que fomente la encapsulación y la comprobación estática de tipos, a la vez que de soporte a los multimétodos. No obstante, existen todavía algunas dificultades importantes relacionadas con los multimétodos.

En primer lugar, los multimétodos deben seleccionar cuál es el método más específico a ejecutar, pero no siempre existe tal método, produciéndose una ambigüedad. Resulta crucial en una implementación de multimétodos conocer cómo se eliminan esas ambigüedades, análogamente a cómo en herencia múltiple se resuelven las ambigüedades de variables de instancia y métodos heredados. CLOS y CommonLoops solventan las ambigüedades ordenando los multimétodos y dando prioridad a los argumentos de la izquierda. El inconveniente de la resolución automática de ambigüedades es similar al que se produce por la linearización de clases en herencia múltiple, pudiendo dar lugar a errores difíciles de detectar. Cecil prohíbe los métodos ambiguos, presentando un error [Chambers, 1992b; pp. 45-46]. En envío simple no existe ese tipo de ambigüedades. Un ejemplo se presenta un poco más adelante, en el tercer punto de discusión.

En segundo lugar, dado que un multimétodo puede precisar el acceso a la representación interna de sus argumentos, se produce un problema de pérdida de encapsulación. Un desarrollador es capaz de tener acceso a la representación interna de otras clases únicamente porque declara multimétodos con argumentos de esas clases. Los lenguajes que no soporten módulos tienen en este caso una seria desventaja. Cecil sí soporta módulos, y exige la creación de listas de multimétodos con privilegios especiales para hacer frente a este problema, pero espera que sea el entorno de desarrollo el que ayude a definir esa lista [Chambers, 1992b; p. 50], lista por otro lado no trivial.

En tercer lugar, y más seriamente, los multimétodos pueden dar lugar a incompletitud o inconsistencias que generen nuevas ambigüedades, debidas a la combinación de multimétodos desarrollados independientemente. Consideremos el caso siguiente inspirado en [Leavens & Millstein, 1998; pp. 380-381] usando una hipotética extensión de Java con multimétodos, donde los multimétodos se sitúan fuera de las definiciones de clase:

```
package A;
class GraphicalObject
{ // ...
class Port
```

```

{ // ...
void displayOn (GraphicalObject gobj, Port port)      // multimétodo
{ // ...
}

```

Supongamos otro paquete que incluye al primero, que extiende los objetos gráficos, y que define una especialización del multimétodo `displayOn`:

```

package B;
import A;

class Rectangle extends GraphicalObject
{ // ...

void displayOn (Rectangle gobj, Port port)      // multimétodo
{ // ...
}

```

Y también un tercero similar que extiende los dispositivos:

```

package C;
import A;

class Printer extends Port
{ // ...

void displayOn (GraphicalObject gobj, Printer port) // multimétodo
{ // ...
}

```

Los módulos `B` y `C` son compatibles con `A` por separado. Con todo, el envío del siguiente mensaje es ambiguo en un código externo, porque no existe ningún método más específico para el mensaje `displayOn`:

```

import A, B, C;
// ...
Rectangle rectangle = new Rectangle();
Printer printer = new Printer();
displayOn (rectangle, printer);           // llamada a multimétodo

```

Los métodos `displayOn (GraphicalObject, Printer)` y `displayOn (Rectangle, Port)` son ambiguos. Ambos son candidatos igualmente válidos, ya que son los más específicos que se encuentran para el multimétodo `displayOn`. Cecil muestra un error en este caso, mientras que otros lenguajes como CLOS dependen de la ordenación de los argumentos para seleccionar implícitamente el método más aplicable. En CLOS la regla es de izquierda a derecha, por lo que seleccionaría `displayOn (Rectangle, Port)`. Pero es posible que no exista ningún método apropiado. Por esa razón, el comportamiento de Cecil es más correcto al no enmascarar posibles errores. Para solventar este problema [Chambers, 1994; pp. 12-13] propone extensiones de módulo que resuelvan las nuevas ambigüedades que surjan de la importación de múltiples módulos, que en la práctica es similar a la definición de un nuevo módulo que elimine las ambigüedades:

```

package D;
import A, B, C;

void displayOn (Rectangle gobj, Printer port)      // multimétodo
{ // ...
}

```

La solución no es del todo satisfactoria, porque la responsabilidad de conseguir que los módulos sean compatibles recae en el desarrollador que usa los módulos en conflicto. Usuario que probablemente no tenga los conocimientos adecuados para hacerlos compatibles si los módulos importados son de terceros. La limitación es más importante si se considera en el ámbito de un lenguaje que fomente la construcción de componentes, donde la importación de múltiples módulos de terceros se supone muy habitual. El proceso de composición no puede depender de la resolución de los conflictos que surgen por la importación de multimétodos. La importación debe funcionar sin sorpresas. Si no se usan multimétodos este tipo de problemas no aparece. Estas potenciales dificultades hacen aconsejable no

introducir el concepto de multimétodos en el lenguaje, y optar por resolver menos elegantemente los mensajes con múltiples argumentos polimórficos con la técnica de reenvío o *double dispatch*.

4.4.5 Discusión

La separación modular entre el cliente y el proveedor por el mecanismo de envío de mensajes da lugar a una implementación natural de los protocolos, que por definición son independientes de implementación. Un sistema de envío de mensajes modular es directamente una implementación de protocolos. Cobra por ello doble importancia la elección de un sistema de envío de mensajes modular, porque mantiene las premisas de diseño del lenguaje y, a la vez, evita la implementación de un soporte de protocolos adicional.

La técnica de tablas virtuales es más sencilla y eficiente en memoria, al no necesitar ningún procedimiento de registro y ya que, al reusar índices para clases no relacionadas, elimina todos los posibles huecos en las tablas de métodos. La eficiencia y simplicidad de las tablas virtuales tiene un coste: la codificación de los desplazamientos dentro de las tablas de métodos en el código cliente. Debe codificarse así precisamente porque se ha realizado la compactación de la tabla de métodos, perdiendo los índices de los mensajes su carácter único. La compactación demanda que, cuando se añade un nuevo mensaje a una clase u objeto base, se recalculen todos los desplazamientos. En cambio, si no se produce compactación, el índice del selector sí es significativo. Cada mensaje tiene asociado un selector único y el añadido de un nuevo mensaje no afecta al selector asignado a mensajes anteriores, siendo éste un comportamiento más modular y monótono, puesto que no requiere revisar los desplazamientos.

La modularidad trae consigo un coste también: la técnica de selectores es más compleja. Necesita la ayuda del compilador y mayor cuidado durante su implementación, en particular para la función mensajera. No obstante, es una técnica más modular y, por tanto, más acorde con el espíritu de diseño de XC. Los detalles de implementación de los selectores en XC se ven en el apartado 8.4 *Implementación del envío de mensajes*.

Existen otras técnicas modulares de envío de mensajes. La técnica de selectores es una implementación eficiente de diccionarios. Una implementación de la tabla de métodos directamente con diccionarios —donde cada método se identifica con una función *hash* sobre la firma del mensaje— si bien menos eficiente, también es modular por la misma razón que la técnica de selectores. Los desplazamientos dentro de la tabla de métodos tampoco se codifican en el cliente, se calculan a través de la función *hash*. Otro ejemplo de implementación modular del envío de mensajes es el especificado por la máquina virtual de Java. La máquina virtual usa códigos de operación de invocación de métodos que incluyen un índice al *constant pool* de la clase en el *runtime*, que no codifica el desplazamiento del método dentro de la clase, ya que la estructura interna de un objeto dentro del *runtime* de Java no es conocida por el compilador [Lindholm & Yellin, 1999; pp. 376-377]. El esquema es similar al elegido por Smalltalk-80, que utiliza un desplazamiento sobre la zona de literales o *literal frame* que acompaña al código de cada método compilado [Goldberg & Robson, 1989; pp. 420-421]. En Java, la implementación final realizada por la máquina virtual no está definida por el lenguaje. Una técnica usada por las máquinas virtuales saca también partido del invariante de los selectores. Una vez obtenido el método asociado a un selector, si todas las clases derivadas de una misma clase dada mantienen el invariante de posición de métodos de las tablas virtuales, entonces es posible modificar el código de llamada tras descubrir el índice del selector, realizando en este caso una implementación con tablas virtuales [Mitchell, 2003; pp. 407-408]. Nótese que esto es posible porque la máquina virtual puede modificar dinámicamente el código intermedio o *bytecode* original que ejecuta, usando técnicas de compilación dinámica similares a las ensayadas inicialmente con SELF [Chambers, 1992]. En Smalltalk-80 la implementación de la tabla de métodos usa directamente un diccionario.

4.5 Reflexión y arquitecturas meta-nivel

El concepto de *reflexión* en lenguajes de programación fue introducido a principios de la década de 1980 [Rivières & Smith, 1984]. Un lenguaje de programación es *reflectivo* si presenta un protocolo que permite manipular estructuras internas del lenguaje desde el propio lenguaje. Este protocolo está formado por *operadores reflectivos* o *reflective operators* y se encuentran definidos en el soporte en tiempo de ejecución o *runtime*. Lenguajes con esas características existían con anterioridad, como es el caso de Smalltalk-80. Smith los llama *lenguajes de programación reflectivos*. El concepto de reflexión se puede aplicar a otro tipo de programas, no sólo a lenguajes de programación. Por ejemplo, muchas bases de datos relacionales hacen accesible sus propiedades internas —como índices, tablas o usuarios— a través de tablas manipulables por el lenguaje SQL. No se extenderá la discusión a esos sistemas, que no son objetivo de este trabajo, sino que se mantendrá dentro de los límites de los lenguajes de programación.

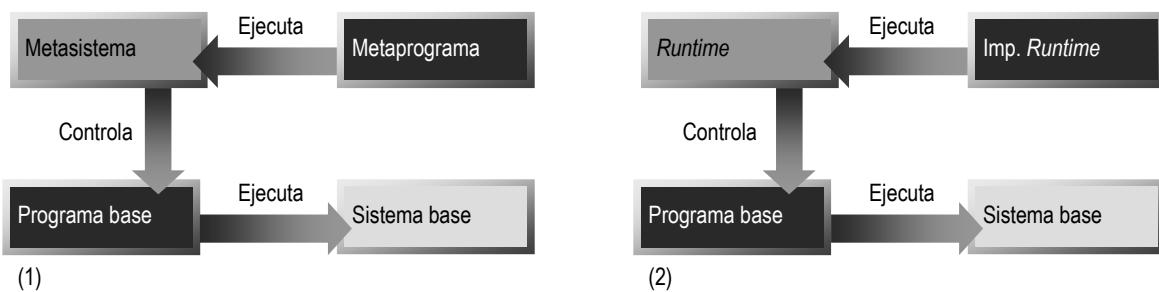


Figura 4-14 Programas y metaprogramas en lenguajes de programación.

Tradicionalmente se han construido programas base que ejecutan sistemas informáticos. (1) Un metaprograma define un metasistema que controla un programa base para monitorizar su ejecución. (2) En lenguajes de programación el metaprograma es la implementación del soporte en tiempo de ejecución o *runtime* del lenguaje, y el metasistema es el soporte en tiempo de ejecución en sí.

Las funciones u operadores que interrogan qué facilidades proporciona un lenguaje —como puede ser la lista de funciones disponibles— son operadores especiales llamados *reflectivos*. Demarcan una separación entre un *programa base* y un *metaprograma*. El primero emplea los operadores usuales de un lenguaje. El segundo usa, además, los operadores reflectivos. El metaprograma es capaz de interrogar el estado y afectar a la ejecución del programa base. La relación entre programas y metaprogramas se presenta en la Figura 4-14.

Esta separación de roles de los programas se puede extender aún más, observando que un metaprograma es también un programa y, por tanto, es concebible construir un meta-metaprograma que afecte a la ejecución del metaprograma. Lo mismo se puede decir del meta-metaprograma. Esta cadena de metaprogramas se denomina *metaregresión*, que potencialmente puede ser infinita [Rivières & Smith, 1984; pp. 332-334]. Si un lenguaje describe sus operadores reflectivos en el mismo lenguaje, entonces se dice que el lenguaje es *metacircular*, puesto que la descripción del lenguaje está escrita en el mismo lenguaje. Una exposición más completa se encuentra en [Abelson, 1996].

Si bien puede considerarse la línea de razonamiento anterior sugestiva, la diferenciación centrada en programas que afectan a otros programas lleva consigo la introducción de un número considerable de nuevos conceptos de programación: programa base, metaprograma, meta-metaprograma, operadores reflectivos, regresión, regresión infinita o metacircularidad, cuyo significado sea quizás más filosófico que práctico. Se puede hacer una argumentación similar, por ejemplo, sobre la relación entre un programa, un depurador, el sistema operativo y el depurador del sistema operativo. En un contexto literario se puede llegar a argumentar que una novela cuya trama fuese la escritura de otra novela pudiese ser considerada una meta-novela.

Las ideas anteriores no son más que una reformulación de una arquitectura muy conocida. Resulta mucho más simple hacer constar que es interesante que un programa puede afectar a otro, y que para ello es importante la existencia de un protocolo definido para ese propósito, tal y como ocurre con naturalidad entre un programa cliente y otro servidor. Así se evita incorporar nueva nomenclatura para definir conceptos existentes. El soporte en tiempo de ejecución de un lenguaje se implementa en la práctica como una biblioteca que se ejecuta usualmente en el mismo espacio de direcciones que el programa base. Exactamente igual a la relación entre una biblioteca gráfica de un entorno de ventanas y un programa con presentación gráfica. Desde este punto de vista, la arquitectura de un lenguaje de programación reflectivo es equivalente a un programa servidor cuyo cliente es el programa base, como se ve en la Figura 4-15.

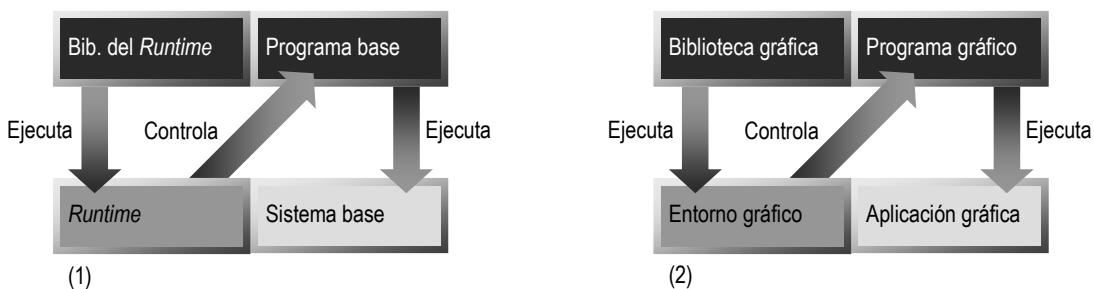


Figura 4-15 Interpretación de la metaprogramación como programas cliente/servidor.

La organización de un lenguaje de programación reflectivo (1) es similar a la de un entorno gráfico (2). El entorno gráfico hace las veces de servidor. La aplicación gráfica es cliente del entorno gráfico y su ejecución es controlada por el entorno gráfico. Esta interpretación es más simple y evita introducir nuevos conceptos.

4.5.1 Implementaciones abiertas

El concepto de *implementaciones abiertas* u *open implementations* es introducido en [Rao, 1991] y en [Kiczales, Rivières & Bobrow, 1991]. Un lenguaje tiene una implementación abierta si da al menos dos interfaces de programación: un *protocolo de base* o *base-level* que contiene la funcionalidad del lenguaje, y un *metaprotocolo* o *meta-level* que revela aspectos de cómo funciona el protocolo de base. Kiczales da algunos argumentos interesantes acerca de su razón de ser:

“La imagen de abstracción sobre la que se asienta la ingeniería del *software* no soporta la realidad de la práctica: sugiere que las abstracciones esconden su implementación, cuando la evidencia es que esto no es generalmente posible. Esta discrepancia entre los conceptos básicos fundamentales y la práctica parece estar en el fondo de un número de problemas de portabilidad y complejidad”. [Kiczales, 1992; p. 1]

Un ejemplo obtenido de [Kiczales, 1992; p. 2] es la construcción de una hoja de cálculo de 100x100 celdas usando una interfaz gráfica. Una implementación que use una ventana por cada posible celda es muy sencilla conceptualmente y usa al máximo las abstracciones del entorno de ventanas, pero es inviable en la práctica por su lentitud y recursos usados.

“Lo que está claro es que existe una discrepancia básica entre nuestra imagen existente de abstracción y la realidad de la programación del día a día. Decimos que diseñamos claras y poderosas abstracciones que esconden sus implementaciones, y que luego usamos esas abstracciones sin pensar sobre su implementación para construir funcionalidad de más alto nivel. Pero la realidad es que la implementación no puede ser siempre escondida, las particularidades de sus prestaciones pueden percibirse indirectamente de muchas formas. De hecho, el programador de un cliente es bien consciente de ellas, y está limitado por ellas tanto como lo está por la abstracción misma”. [Kiczales, 1992; p. 2]

Las implementaciones abiertas se han aplicado a la construcción de entornos de soporte en tiempo de ejecución para lenguajes [Kiczales, Rivières & Bobrow, 1991], compiladores [Mendhekar, Kiczales & Lamping, 1994] o sistemas operativos [Yakote, 1992], por poner unos ejemplos.

Una de las metainterfaces dadas por una implementación abierta es el protocolo de reflexión. Otras metainterfaces manipulan y modifican el comportamiento del protocolo de base. [Kiczales, Rivières & Bobrow, 1991] aborda el tema de la construcción de metainterfaces en el contexto de lenguajes orientados a objetos, usando un subconjunto de CLOS como ejemplo. Éstas se construyen definiendo nuevos objetos que controlan los objetos usuales del lenguaje. Por ejemplo, define objetos para describir métodos, argumentos y clases. Se llaman *metaobjetos*, puesto que controlan el comportamiento de los objetos normales o base. Los metaobjetos ejecutan *metamétodos*. Los metaobjetos y los metamétodos son iguales a los objetos y métodos normales. La única diferencia radica en su propósito. A los protocolos o interfaces de estos objetos los denomina *protocolos de metaobjetos*.

Un lenguaje reflectivo tiene usualmente dos conjuntos de operadores en su interfaz o protocolo. El primero es usado para leer las estructuras internas del lenguaje, como por ejemplo la lista de métodos implementados por un objeto. El segundo es empleado para modificar las estructuras internas, como el añadido de un nuevo método. Se dice entonces que se produce una *reificación* o *reification* del cambio. La reificación es denominada también *introspección* o *instrospection*. Una vez el cambio se realiza, afecta a la ejecución en el programa de base, es decir, se produce una *reflexión* o *reflection* de los cambios en el programa de base. La Figura 4-16 ilustra estas ideas.

Cuando se usan capacidades reflectivas, a veces ocurre que parte de las estructuras internas de un lenguaje son reificadas (*reified*) y reflejadas (*reflected*) a la vez. Por ejemplo, en un lenguaje basado en clases, un objeto puede guardar explícitamente una referencia a su propia clase en una variable de instancia. Esta situación puede dar lugar a problemas si la clase del objeto puede cambiar en el lenguaje. A este comportamiento se le denomina *superposición reflectiva* o *reflective overlap*. Volviendo a usar la analogía con entornos gráficos, la reificación es similar a la modificación de elementos del entorno gráfico por parte de la aplicación, o a la cesión del control del flujo del programa al entorno gráfico. La reflexión es análoga a la presentación de los elementos modificados en la aplicación, o a las llamadas de retorno que la aplicación recibe para responder a eventos. Como los términos reificación y reflexión se refieren a conceptos ya conocidos, en general no serán usados para no complicar la nomenclatura.

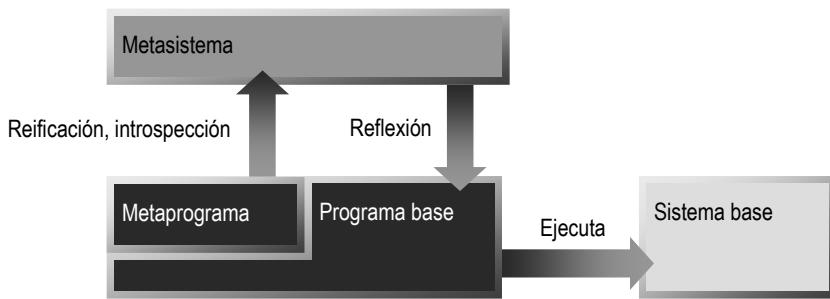


Figura 4-16 Reificación y reflexión en metaprogramación.

Si un programa base es capaz de modificar las estructuras definidas en su metasistema (reificación), entonces está realizando metaprogramación. Parte del programa base es en realidad un metaprograma. Las modificaciones hechas en el metasistema se reflejan en la ejecución del programa base.

Se denomina *metaprogramación* a la programación de metaprogramas. Un metaprograma es un programa que manipula otros programas, por ejemplo un entorno de desarrollo, o un depurador. También se llama usualmente metaprogramación a la programación usando metainterfaces o protocolos de metaobjetos, como las interfaces de reflexión. Las *arquitecturas metanivel* son todas aquellas que involucran metaprogramación para organizar el diseño de un programa.

4.5.2 Metaobjetos

En metaprogramación existen dos variantes de objetos: los objetos normales o base y los metaobjetos. A estos últimos se les permite interceptar las operaciones ejecutadas por los objetos base. Por ejemplo, pueden interceptar un método para dar transaccionalidad a un objeto base que no la había previsto originalmente. Antes de ejecutar el método del objeto, el metaobjeto ejecuta un metamétodo anterior que guarda el estado del objeto base, luego se ejecuta el método del objeto base y finalmente un metamétodo posterior se encarga de comprobar si la ejecución del método base fue correcta o no. Si fue incorrecta, puede restaurar el estado anterior guardado del objeto base. Los metaobjetos son capaces así de extender ortogonalmente el objeto original o base (véase la Figura 3-9).

La definición de un metaobjeto está muy ligada con la definición del objeto que controla o supervisa. Sus metamétodos deben recibir al menos los mismos argumentos que recibe el método base y, además, una referencia al objeto base, necesarios para tener información de contexto suficiente. También deben tener cierto acceso privilegiado a la estructura del objeto que supervisan para poder efectuar los cambios vistos en el ejemplo de transaccionalidad.

4.5.3 Envío de mensajes a metaobjetos

Los metaobjetos evitan mezclar el código del metaobjeto con el código original del objeto base situando su código en metamétodos. El proceso que hace la llamada a los metamétodos antes y después del método base se llama *intercepción* o *interception*. Los métodos que se ejecutan antes se denominan usualmente métodos *anteriores* o *before*, y los que se ejecutan después reciben el nombre de métodos *posteriores* o *after*.

Flavors y CommonLoops poseen este tipo de métodos [Bobrow *et alii*, 1986; p. 20], al igual que CLOS, sucesor entre otros de Flavors. También estaban presentes originalmente en C con Clases. Recibían el nombre de métodos de llamada o *call* y de retorno o *return*, pero luego no acabaron formando parte de C++ [Stroustrup, 1994; pp. 57-58]. CLOS también tiene métodos *around*, que se ejecutan *en vez de* del método que sustituyen.

La implementación tradicional del envío de mensajes a metaobjetos —cuyo esquema general es también el descrito en la Figura 3-9— usa un preprocesador para insertar las llamadas a metamétodos en medio del código original. Por ejemplo, si consideramos un método base *baseMethod* de una clase *A*, que suponemos escrito un hipotético lenguaje con soporte de metaprogramación similar a C++:

```
class A : metaclass MA
{
    // ...
    void baseMethod (int arg1, ... , int argn)
    { // ... }
};
```

donde la clase *A* indica que *MA* es su metacategoría, y *baseMethod* es un método normal con varios argumentos. La definición de los metamétodos se describiría con una metacategoría separada¹⁹:

```
metaclass MA
{
    // ...
    before void baseMethod (int arg1, ... , int argn)
    { // ... }
    after void baseMethod (int arg1, ... , int argn)
    { // ... }
};
```

Cuando en el código del programa base se encuentra una llamada al método base:

¹⁹ Las metacategorías se analizan con más detalle en el apartado 4.5.4.

```
A obj;
obj.baseMethod (arg1, ... , argn);
```

el compilador del hipotético lenguaje efectúa una sustitución funcionalmente equivalente al siguiente código en C:

```
before_baseMethod (metaself, self, arg1, ... , argn);
baseMethod (self, arg1, ... , argn);
after_baseMethod (metaself, self, arg1, ... , argn);
```

El parámetro `metaself` hace referencia al metaobjeto y el parámetro `self` al objeto base. No siempre se envían todos los argumentos del método base a los métodos anteriores y posteriores por razones de eficiencia, pero conceptualmente son necesarios para que los métodos anteriores y posteriores tengan acceso al contexto apropiado de ejecución del método base. Otros aspectos de eficiencia se ven en el apartado 8.4 *Implementación del envío de mensajes*.

Una variante interesante es la propuesta por Open C++ a partir de la versión 2 [Chiba, 1995]. Al contrario que la versión 1 [Chiba, 1993]—similar en espíritu al esquema explicado antes—Open C++ v.2 define un protocolo de metaobjetos durante la compilación, no durante la ejecución. Los metaobjetos traducen el código original anotándolo con las acciones adicionales definidas en los metaobjetos. Éstos pueden verse también como un conjunto sofisticado de macros de traducción [Chiba, 1998], cuyo resultado es código fuente en C++ que finalmente es compilado a código objeto. La estructura sintáctica del programa base es visible a los metaobjetos en Open C++, para que éstos puedan modificarla durante la traducción al programa final en C++. Es decir, en Open C++ v.2 los metaobjetos describen macros que transforman el código fuente original en Open C++ a código C++ estándar. Por ejemplo, un metaobjeto en Open C++ puede encargarse de generar la llamada al metamétodo `before_baseMethod` del ejemplo anterior, al recibir como parámetro la cadena con la llamada al método original `baseMethod`. Para ello concatena la llamada al metamétodo antes de la llamada al método base. Nótese que la manipulación se produce estrictamente con cadenas de texto, porque es el código fuente original el que está siendo modificado por el metaobjeto. El protocolo de metaobjetos se aplica únicamente durante la compilación, por lo que no hay sobrecarga en el código final, salvo la decisión de realizar aquellas llamadas adicionales —como la llamada al metamétodo anterior— que se consideren necesarias.

4.5.4 Metaclases

En algunos lenguajes basados en clases, éstas tienen representación en tiempo de ejecución mediante objetos. Ocurre con Smalltalk-80, CLOS, Objective-C o en extensiones a C++ como SOM. Los objetos son una técnica de implementación relativamente natural para describir internamente los métodos y variables de clase. Si una clase se representa con un objeto, ¿cuál es la clase de una clase? Algunos lenguajes—entre ellos Smalltalk-80 y Objective-C—responden a esa pregunta introduciendo el concepto de *metaclase*: una clase de clases. Otros lenguajes basados en clases—C++ y Java son dos de ellos—obvian el problema ya que ni siquiera una clase está representada por un objeto²⁰. Ciertos lenguajes, como Smalltalk-80 u Objective-C, gestionan automáticamente las metaclases, convirtiéndose más bien en una particularidad de la implementación. En otros lenguajes—CLOS y SOM son ejemplos—las metaclases se pueden controlar explícitamente.

Las metaclases establecen un árbol de herencia paralelo al de las clases, siendo su cometido influir en el comportamiento de las clases que definen. Las metaclases corresponden con un nivel de abstracción superior al de las clases, pues parametrizan cómo se comportan las clases. La parametrización se torna interesante cuando se modelan propiedades o *adjetivos* generales a varias clases: seguridad, soporte de múltiples hilos de ejecución o definición de clases abstractas. Se dice también que estas propiedades

²⁰ A partir de Java 1.1 existe un objeto que gestiona la representación de la información de compilación de una clase Java, pero no es un objeto clase.

son *ortogonales* a la funcionalidad principal. El nombre *adjetivo* se inspira en que son propiedades no fundamentales para la clases, pero sí convenientes en ciertas situaciones.

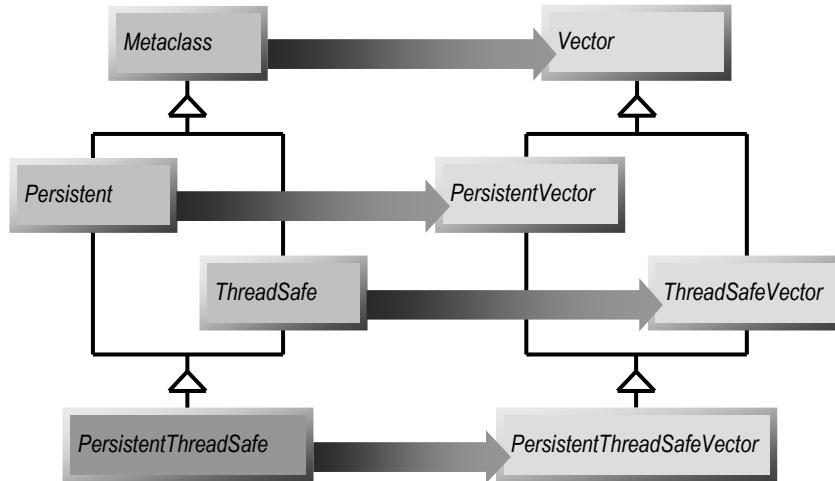


Figura 4-17 Metaclasses y adjetivos.

Las metaclasses implementan propiedades comunes a varias clases, factorizando su código. Al ser propiedades compartidas por múltiples clases —pero no esenciales— reciben el nombre de adjetivos.

La Figura 4-17 muestra varias metaclasses que implementan adjetivos, y varias clases **vector**, que varían en funcionalidad no esencial. Son instancias de metaclasses y obtienen sus propiedades adicionales de ellas. Las metaclasses sirven para factorizar el código común de persistencia o soporte de varios hilos de ejecución. El código de la clase **vector** es el mismo, heredándose de una clase común **vector**. Su metaclass es la metaclass general **Metaclass**, que factoriza el código común a todas las clases.

La gestión de metaclasses genera ciertas dificultades de *compatibilidad* al añadir nuevos adjetivos. Sin entrar en demasiados detalles, las metaclasses deben garantizar que, cuando se combinan dos clases por herencia para generar una tercera, se salvaguardan las posibles llamadas entre las metaclasses asociadas a las clases combinadas [Graube, 1989]. SOM genera metaclasses derivadas intermedias automáticamente para mantener compatibilidad entre las metaclasses. Es decir, se encarga de hacer heredar por la tercera clase todos los métodos de las metaclasses de las dos clases heredadas, para que todos los mensajes entre metaclasses sean resueltos satisfactoriamente. A esta propiedad la denominan *herencia de restricciones de metaclass* o *inheritance of metaclass constraints* [Forman & Danforth, 1999; p. 44]. En la Figura 4-17 la combinación de las clases **PersistentVector** y **ThreadSafeVector** para crear la clase **PersistentThreadSafeVector** precisa la construcción de una metaclass asociada **PersistentThreadsafe** si se quieren evitar los problemas de compatibilidad. SOM construye esta metaclass automáticamente, pero otros lenguajes con metaclasses no lo hacen y debe añadirse manualmente.

Existen otros problemas sutiles que aparecen con las metaclasses. Por ejemplo, SOM garantiza la compatibilidad de metaclasses en parte porque mantiene monotonía (Def. 4-8) al heredar metaclasses, pero su solución genera a su vez nuevos inconvenientes. Por ejemplo, si una metaclass codifica la propiedad de abstracción de clases, todas sus submetaclasses son siempre abstractas. Otras dificultades son también analizadas en [Bouraqadi-Sââdani, Ledoux & Rivard, 1998; p. 85]. [Forman & Danforth, 1999; p. 53] argumentan que en SOM algunos adjetivos como “*una clase debe ser abstracta*” no constituyen buenas metaclasses, y que es mejor mantener la propiedad de monotonía que permitir adjetivos no monótonos. Como alternativa, sugieren crear una metaclass que codifique “*una clase puede ser abstracta*” y que cada clase indique si quiere ser abstracta o no.

Una crítica significativa que se puede hacer a las metaclasses es que éstas se relacionan también mediante herencia —generalmente múltiple— mezclándose interfaz e implementación. Se puede argumentar que al ser la herencia una técnica de reuso de caja blanca, se están generando interdependencias implícitas entre *todas* las clases a través de las propiedades que suministran sus metaclasses. Se puede objetar también que un único árbol o grafo de herencia que mezcle interfaz e implementación en los objetos de base ya genera suficientes molestias, para tener en cuenta otros árboles o grafos paralelos que, además, vuelven a mezclar ambos conceptos.

Una crítica constructiva debe sugerir alguna alternativa. Los metaobjetos que *no* son metaclasses —y que, por tanto, no instancian objetos— son una opción viable, ya que ofrecen un modelo para ciertas propiedades implementables con metaclasses, como seguridad, persistencia o soporte de múltiples hilos de ejecución. Sin embargo, es difícil imaginar cómo estos metaobjetos podrían dar un modelo convincente para la propiedad de abstracción de clases. A su favor tienen ser conceptualmente más simples, puesto que el comportamiento de las clases de base no puede verse modificado de forma no modular a través de perturbaciones relacionadas con la herencia. Ahora bien, los metaobjetos introducen sus propios inconvenientes: pueden demandar un árbol de herencia paralelo que hay que mantener explícitamente y acceso privilegiado a los objetos que supervisan. Curiosamente, el autor no conoce ningún equivalente a un “metaprototipo” en lenguajes basados en prototipos, si bien el modelo de objetos es tan flexible como el de clases. XC propone una alternativa distinta, que se verá en el apartado 4.5.6.

4.5.5 Aspectos

Una evolución de las ideas de metaprogramación son los *aspectos* [Kiczales *et alii*, 1997], que muchas veces describen propiedades similares a los adjetivos vistos con anterioridad. Los aspectos surgen de la observación siguiente. Supongamos una jerarquía de clases a la que se desea proporcionar capacidades de impresión. Se pretende que todos los objetos de todas las clases se puedan imprimir a sí mismos, por lo que se debe añadir un método `print` a cada clase de la jerarquía. La propiedad de impresión se distribuye entonces por todas las clases de la jerarquía. Pero, como tal, es una propiedad que tiene su propia identidad, y pierde su carácter independiente al ser repartida por múltiples clases. Es decir, no se fomenta la factorización de código de esas propiedades. En casi todos los lenguajes orientados a objetos no existen facilidades que ayuden a factorizar código como el de la impresión. Son lenguajes que no soportan aspectos. Un aspecto es un *corte transversal* o *cross-cut* de una funcionalidad de un programa [Kiczales *et alii*, 1997; p. 220], en el sentido de estar distribuido por muchas clases. Corresponde con un punto de vista complementario al principal dado por el árbol de herencia. La programación que tiene en cuenta el uso de aspectos recibe el nombre de *programación orientada a aspectos* o *aspect-oriented programming*.

Muchos otros tipos de código son susceptibles de ser interpretados como aspectos. Un ejemplo son los adjetivos vistos en el apartado 4.5.4 *Metaclasses*. Es posible modificar la estructura estática de un programa usando aspectos. Los aspectos son en la actualidad un campo de investigación interesante y prometedor, aunque todavía existen muy pocos lenguajes que lo soporten. El lenguaje con aspectos más conocido es AspectJ [Kiczales *et alii*, 2001], una extensión con aspectos de Java. Es capaz de añadir nuevas variables de instancia y métodos a una clase Java existente o, incluso, cambiar la jerarquía de clases. En este sentido la funcionalidad que provee se acerca a la de Open C++ v. 2, vista en el apartado 4.5.3, y que define un protocolo de metaobjetos en tiempo de compilación. En ambos casos, la modularidad se encuentra comprometida, pues este tipo de metaprogramación precisa el acceso al código fuente original. Los aspectos también pueden modificar la estructura dinámica de un programa. En AspectJ es posible interceptar ciertos momentos de ejecución de un programa —llamados puntos de unión o *join points*— antes, después, o en vez de ejecutar el código original, al estilo de los metamétodos *before*, *after* y *around* de CLOS. Los puntos de unión son llamadas a métodos, ejecución de métodos, acceso a variables de instancia, inicialización estática de variables o manejo de excepciones. Cada aspecto se emplea en un punto de corte o *pointcut*, que selecciona qué

condiciones de los puntos de unión deben cumplirse para aplicar el aspecto. Los puntos de corte son una combinación de puntos de unión mediante operadores de encaje de patrones o *matching* y operadores lógicos. Un problema serio de AspectJ es que modifica el código fuente de los programas en Java para incorporar las modificaciones oportunas indicadas por los aspectos. La cuarta premisa de diseño de XC (Def. 3-15) impide el conocimiento explícito del código fuente, pues se trata de un comportamiento no modular. Más investigación es necesaria en este importante sentido.

4.5.6 Discusión

El mero hecho de querer dotar a XC de un soporte en tiempo de ejecución pone sobre la mesa la necesidad de la definición de un soporte de metaprogramación. Java ha demostrado que añadir tardíamente el soporte en tiempo de ejecución al lenguaje es perjudicial para él. La presencia de tipos primitivos y objetos, o la baja eficiencia de la biblioteca de reflexión de Java durante varios años son síntomas de ello. Es posible diseñar sistemas excesivamente flexibles con capacidades de metaprogramación. Smalltalk-80 y CLOS son dos claros ejemplos. Puesto que XC es un lenguaje compilado, resulta deseable limitar las capacidades de metaprogramación a aquellas aptas para ser comprobables en tiempo de compilación. Por esa razón, el soporte para programación de metaobjetos en XC intenta reunir simplicidad de conceptos, comprobación de tipos en tiempo de compilación, y razonable flexibilidad.

XC se inspira en las ideas descritas por [Kiczales, Rivières & Bobrow, 1991]. XC evita el uso de los términos metaprograma y metaprogramación, puesto que el acceso a las facilidades de reflexión del lenguaje son muy habituales por las bibliotecas de base. De hecho, todos los programas escritos en XC son en realidad metaprogramas, pues desde los objetos más básicos se hereda y usa funcionalidad de reflexión e introspección, con lo que se pierde el “presunto” beneficio que existiese en la distinción explícita entre programa y metaprograma. Además, con la reflexión y la metaprogramación se ha desarrollado una nomenclatura que es a veces oscura, como la metacircularidad o la regresión infinita. XC adopta un vocabulario más claro y cercano, para simplificar el razonamiento sobre metaprogramación con el uso de abstracciones más sencillas. El concepto de implementaciones abiertas se considera importante, puesto que pone de manifiesto la idoneidad de planificar un protocolo dual: uno para los clientes, y otro de más bajo nivel para obtener un control de grano más fino acerca de la implementación, que la expone de forma controlada.

XC es un lenguaje basado en prototipos y no maneja clases —aunque sí las simula a efectos prácticos—. Resulta entonces natural que XC no soporte tampoco metaclases. En XC se opta en cambio por un novedoso cambio de perspectiva que evita introducir el concepto de metaclases y de metaobjetos como entes que supervisan la ejecución de objetos base. En vez de depender de metaobjetos para monitorizar o controlar la ejecución de los objetos base, como se muestra en la Figura 3-9, en XC se usan extensiones de objetos con categorías, usando metamétodos anteriores y posteriores. La Figura 3-8 introduce gráficamente la idea de la extensión de objetos con categorías, que añade propiedades a un objeto sin necesidad de usar herencia. El resultado es una mayor economía de abstracciones básicas en el lenguaje y una simplificación de los conceptos en juego, a la vez que proporciona importante flexibilidad.

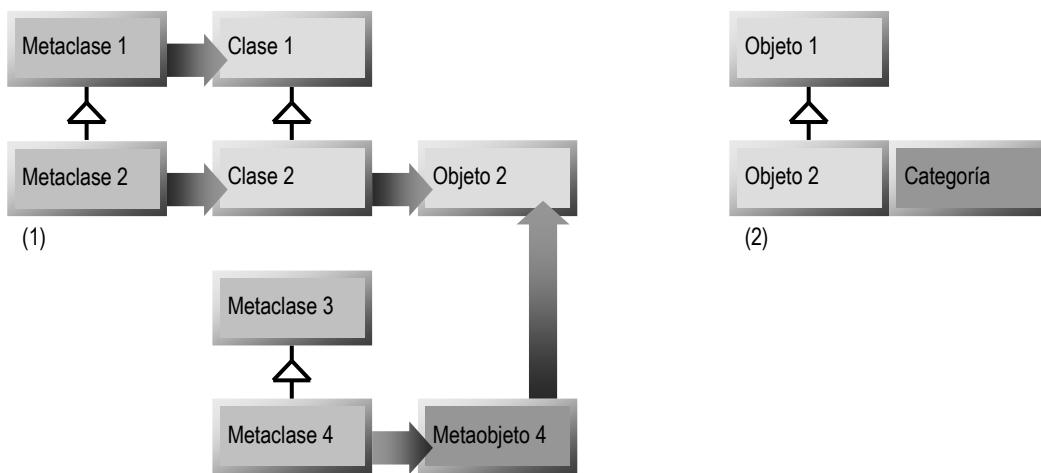


Figura 4-18 Metaprogramación con metaclases frente a metaprogramación en XC.

Usando metaclases existen potencialmente tres árboles de herencia relacionados con un objeto dado (1). En XC sólo existe un árbol de herencia entre objetos prototípicos y extensiones sobre los propios objetos usando categorías (2).

La Figura 4-18 presenta el esquema usado por lenguajes con soporte de metaclases y el de XC. La combinación de objetos prototípicos y extensiones directas con categorías en XC usa un único árbol de herencia. Los lenguajes que definen metaclases usan hasta tres árboles de herencia simultáneamente para controlar un objeto dado. La herencia entre las clases del objeto, la herencia entre las metaclases que instancian metaobjetos, y la herencia entre las metaclases de las clases —si este tipo de metaclases existe—. Los lenguajes que tienen metaclases explícitas han de hacer frente al problema de la compatibilidad visto en el apartado 4.5.4 y a la necesidad de soportar herencia múltiple. La economía de conceptos hace mucho más sencilla la aplicación de técnicas de metaprogramación en XC. No obstante, actualmente XC no soporta programación con aspectos, puesto que no está claro cómo puede preservarse la modularidad. Los apartados 5.3.3 *Extensión a tipos y objetos existentes* y 5.3.4 *Extensiones ortogonales* explican los pormenores de la definición de categorías en XC.

Es ilustrativo advertir cómo una misma funcionalidad se puede implementar usando diferentes técnicas. SOM, por ejemplo, incorpora una metaclase para identificar métodos anteriores y posteriores [Forman & Danforth, 1999; pp. 173-189], similares a los que provee XC mediante el lenguaje. También soporta clases abstractas con metaclases [Forman & Danforth, 1999; pp. 224-226] —protocolos en XC— o métodos finales [Forman & Danforth, 1999; pp. 226-227] —una palabra reservada en XC—. SOM usa otra vez la técnica de metaclases para proporcionar otros adjetivos —que son requisitos no funcionales u ortogonales— gestionados con métodos anteriores y posteriores en XC, como las invariantes [Forman & Danforth, 1999; pp. 215-220] o el soporte de hilos de ejecución [Forman & Danforth, 1999; pp. 189-191]. En estos últimos casos SOM usa la metaclase que gestiona métodos anteriores y posteriores para dar el soporte básico a los adjetivos. Los lenguajes tradicionales de metaobjetos interceptan las llamadas de los objetos para implementar esos adjetivos. Open C++ usa metaclases en tiempo de compilación para codificar los mismos adjetivos y AspectJ emplea aspectos con el igual fin. XC usa categorías y metamétodos anteriores y posteriores.

SOM y otros lenguajes con soporte de metaclases como CLOS son claramente más flexibles, y son probablemente capaces de incorporar más conceptos nuevos a sus modelos de objetos, a costa de ser más dinámicos. Open C++ y AspectJ son lenguajes compilados, en los que es posible crear transformaciones complejas imposibles en XC, pero precisan el acceso al código fuente. XC, en cambio, intenta mantener un marco conceptual tan simple como sea posible, manteniendo la modularidad y la comprobación de tipos en tiempo de compilación. El soporte de metaprogramación se obtiene incorporando sólo dos conceptos nuevos: los metamétodos anteriores y posteriores y la ordenación de categorías. Ambos se verán en el apartado 5.3.4 *Extensiones ortogonales*. También

reduce la importancia de otros conceptos considerados básicos como los metaobjetos y la relevancia de la herencia. En XC no se programan generalmente metaobjetos —salvo en el soporte en tiempo de ejecución del lenguaje— ni existen ramas paralelas de herencia de implementación de metaclasses. Tampoco son necesarios conceptos como la metacircularidad o la regresión infinita. En XC todos los programas son metaprogramas al igual que lo son todos los de Smalltalk-80, pues todos los objetos siguen el protocolo `Object` que contiene métodos reflectivos. Del mismo modo que en Smalltalk-80, no se usa el calificativo de metaprograma, porque no aporta ningún significado distintivo respecto de cualquier otro programa escrito en XC²¹.

XC opta por dar un conjunto de abstracciones flexibles que se mantienen bajo el control del compilador y con una semántica bien definida. La elección puede interpretarse como más tradicional y estática. Es difícil imaginar otra opción que asegure una semántica bien definida en todos los programas de XC. Un ejemplo de flexibilidad limitada es el soporte para métodos anteriores y posteriores: no se da la posibilidad de cambiar arbitrariamente el orden de éstos como hace CLOS, sólo se aprueba con las declaraciones de ordenación de categorías.

4.6 Recapitulación

En este capítulo se ha descrito la definición de las propiedades principales del nuevo lenguaje. Equilibrar las características de un modelo de objetos sencillo pero a la vez flexible es el objetivo del capítulo, cuyo resultado es un modelo de objetos con nuevas e interesantes propiedades. El estudio de los mecanismos de herencia y el contraste con las guías de diseño del lenguaje vistas en el capítulo anterior sugieren limitar la visibilidad de la herencia, usar prototipos en vez de clases, usar concatenación para implementar la herencia, comprobar estáticamente la relación de herencia, y tener herencia simple de implementación y herencia múltiple de interfaces. Se hace énfasis en que la herencia de implementación no es un mecanismo modular, y que debe restringirse. Mantener y fomentar la propiedad de monotonía facilita las extensiones modulares. Con ello se define con precisión la estructura de los objetos del nuevo lenguaje.

El envío de mensajes a objetos se estudia desde el punto de vista de la modularidad, muy poco habitual en la literatura. Se critican mecanismos conocidos de envío de mensajes, sus pros y sus contras. Y se da una justificación para preferir la técnica de selectores sobre otras técnicas por sus propiedades modulares. Por último, el estudio de las arquitecturas metanivel y la reflexión dan pie para presentar uno de los elementos más novedosos del modelo de objetos de XC: la metaprogramación usando extensiones con categorías en vez de usar metaobjetos externos y/o metaclasses, que simplifica notablemente los diseños de objetos en el lenguaje, y en particular las relaciones de herencia de implementación, que son inherentemente no modulares.

²¹ En una extensión a C++, en cambio, sí puede tener sentido hacer énfasis en las propiedades de un metaprograma en comparación con C++ estándar.

5

La estructura sintáctica

5.1 Introducción

Este capítulo es un resumen de la estructura sintáctica del lenguaje. La sintaxis de un lenguaje es similar a la carrocería de un coche. No es esencial pero dice mucho acerca de él. Como todas las carrocerías, algunos aspectos de ellas son cuestión de gustos y preferencias, mientras que otros reflejan características importantes. XC está orientado a programación de sistemas y aplicaciones de tamaño medio con componentes. Los lenguajes que más han influenciado la sintaxis son por este orden: C, Smalltalk-80, Objective-C, C++, Java y Modula-3. C y Smalltalk-80 tienen sintaxis radicalmente diferentes, Objective-C se inspira en ambos y hace un intento no demasiado afortunado de usar una doble sintaxis sin mezclarlas. C++ es un referente importante porque intenta compatibilizar orientación a objetos y eficiencia. Java se fundamenta en C y C++, simplificando algunas construcciones y recuperando conceptos de Smalltalk-80. Las estructuras de control y los operadores de XC son muy similares a los de C. Las expresiones de mensaje de XC se parecen más a las de Smalltalk-80. De Modula-3 se ha recuperado la organización modular y los protocolos de módulos separados. Las palabras clave prefijo se inspiran en las de Java. Algunos ejemplos se usan a lo largo de este texto para ilustrar puntos relevantes. Una visión más real de un programa en XC se encuentra en el Apéndice B, que contiene un ejemplo de declaración y definición de módulo completa.

En primer lugar se revisa la representación sintáctica del modelo de objetos: los protocolos y los prototipos. Las construcciones sintácticas notables incluyen una discusión de su necesidad, con ejemplos de su utilidad práctica. Las extensiones de herencia y categorías se explican después. Luego se estudian los módulos, por qué existen, por qué se ha elegido esta estructura particular, y se comparan diferentes implementaciones en varios lenguajes. A continuación se repasa la organización jerárquica del lenguaje usando módulos y protocolos de módulo. Con el mismo estilo se revisan finalmente otras conveniencias sintácticas, relacionadas principalmente con la consistencia del lenguaje.

5.2 Organización general de programas

Para mantener consistencia, es importante proporcionar un número pequeño pero flexible de conceptos o abstracciones usados para modelar programas más complejos.

“Un lenguaje de programación da un marco conceptual para pensar sobre algoritmos, y una manera de expresar esos algoritmos. El lenguaje debe ser una ayuda al programador mucho antes de la etapa de codificación. Debe ofrecer un conjunto de conceptos claro, simple y unificado, que puedan ser usados como primitivas en el desarrollo de algoritmos. Para este fin, es deseable tener un mínimo número de conceptos

distintos, con las reglas para su combinación siendo tan simples y tan regulares como sea posible. La sintaxis de un lenguaje afecta a la facilidad con la que un programa puede ser escrito, probado y posteriormente entendido y modificado. La comodidad de lectura es un aspecto central aquí.” [Pratt & Zelkowitz, 1999; p. 12].

Snyder resume en pocas palabras la misma idea de claridad y simplicidad:

“Simplicidad es una meta principal ya que construcciones complejas probablemente no sean bien entendidas, usadas de forma efectiva, o no usadas sin más”. [Snyder, 1986b; p. 19].

Siguiendo las guías de consistencia del apartado 3.6.1, XC se asienta en las siguientes abstracciones básicas:

- *Módulos*. Dan jerarquía, un protocolo explícito de acceso a módulos externos, y asocian explícitamente declaraciones y definiciones relacionadas.
- *Protocolos*. Describen el sistema de tipos del lenguaje independiente de la implementación como conjuntos de mensajes nombrados e inherentemente modulares.
- *Prototipos*. Definen un modelo de objetos simple usado para implementar tipos de datos y componentes, incluyendo datos privados y las operaciones sobre esos datos.
- *Envío de mensajes*. Es el nexo de unión entre protocolos y su implementación: asocia modularmente los mensajes descritos en los protocolos con los métodos implementados en los prototipos.

Junto con esas cuatro abstracciones fundamentales, se definen los siguientes mecanismos de extensión:

- *Herencia múltiple de protocolos*. Permite definir un nuevo protocolo incrementalmente a partir de otros existentes.
- *Herencia simple de implementación*. Facilita la definición incremental de una implementación a partir de otra. La herencia no es un mecanismo modular y, por tanto, debe usarse con moderación. Pero es también un valioso mecanismo de factorización de código dentro de un mismo módulo o entre módulos muy relacionados.
- *Categorías de protocolos*. Añaden nuevos mensajes a protocolos existentes.
- *Categorías de prototipos*. Se usan para añadir modularmente nuevos métodos y datos asociados a nuevos mensajes.
- *Métodos anteriores y posteriores*. También llamados *metamétodos*, son una técnica para realizar modularmente extensiones ortogonales al código original y sustituyen a la programación con metaobjetos.

Las categorías y los metamétodos dan una flexibilidad notable al lenguaje. Exactamente por esa razón es primordial simplificar en la medida de lo posible las interacciones con estas construcciones, usando el compilador para restringir y supervisar su uso. Otras características del lenguaje se construyen a partir de las descritas anteriormente. Son en general facilidades sintácticas, que simplifican la escritura y lectura del código. No añaden conceptos nuevos.

- *Expresiones de operador y estructuradas*. Son conveniencias estrictamente sintácticas del lenguaje. Corresponden siempre con mensajes encadenados.
- *Propiedades*. Declaran y definen métodos de lectura y escritura sobre variables de objetos y esconden el acceso directo a los datos privados de los objetos.

Por último, el sistema de tipos —estudiado en el capítulo 7— ayuda a construir expresiones, declaraciones y definiciones más ricas comprobables en tiempo de compilación. Para ello define:

- *Variables de tipo*. Sustituyen un tipo de datos por otro para asegurar comprobaciones en tiempo de compilación. No generan copias de código.
- *Tipos covariantes*. Es un tipo especial de variable de tipo, que facilita la comprobación en tiempo de compilación de una importante clase de mensajes.

5.2.1 Introducción al modelo de objetos

En este apartado se examina el modelo de objetos básico y la sintaxis de algunas construcciones elementales del lenguaje. Las decisiones más importantes para la definición del modelo ya se han examinado en capítulos anteriores. El objetivo ahora es tener una imagen mental del modelo definido que sirva de guía para abordar los próximos capítulos.

5.2.1.1 Expresiones de mensaje

Las expresiones de mensaje en XC se parecen mucho a las expresiones de mensaje en Smalltalk-80, SELF y Objective-C. En vez de usar la tradicional notación de función, XC usa los mensajes con palabras clave o *keyword messages* de Smalltalk-80, más legibles. Las expresiones de mensaje en XC se identifican con el operador '.', separando cada argumento con una palabra y seguida con ':'. Los mensajes sin argumentos no tienen el carácter ':' al final. Algunos mensajes de un hipotético prototipo `File` pueden ser:

```
File file = File.clone;
file.open: name forMode: "r+";
file.close;
```

5.2.1.2 Protocolos

Existe una separación estricta entre declaración e implementación en XC. La separación es necesaria para poder construir con garantías un modelo de objetos que cumpla las premisas de diseño del apartado 3.3. Los protocolos representan la parte declarativa del modelo.

- ❖ **Def. 5-1.** Un *protocolo* define un conjunto de mensajes y sus argumentos asociados agrupados por un nombre común.

Los protocolos son aplicables a la programación de sistemas con componentes, y ésta puede beneficiarse de una implementación común eficiente de los protocolos que no hay que replicar. Por ejemplo, el protocolo de acceso a dispositivos entre un proceso de usuario y del núcleo de sistema operativo en Mac OS X es descrito por un objeto protocolo de dispositivo cuya estructura es similar a las interfaces COM de Microsoft [Apple, 2002b; pp. 42-43]. A través del mismo protocolo un proceso de usuario puede acceder a distintos dispositivos, como unidades de disco SCSI, que se representan internamente como componentes distintos asociados al protocolo. Los protocolos de XC dan un soporte natural a este tipo de construcciones.

Los protocolos juegan un papel muy importante en XC. Las interfaces de componentes se construyen a partir de protocolos. Las interfaces de módulo se especifican también mediante conjuntos de protocolos. Por último, los tipos de datos se definen *exclusivamente* con protocolos.

5.2.1.2.1 Mensajes y propiedades de protocolos

Los mensajes y las propiedades de protocolos son las abstracciones básicas de interacción entre diferentes partes de un programa.

- ❖ **Def. 5-2.** Un *mensaje* describe una operación de un objeto, que identifica una funcionalidad dada.
- ❖ **Def. 5-3.** Una *firma de un mensaje* define el nombre de una operación, los argumentos y protocolos de los argumentos, y el protocolo del valor de retorno.

El siguiente código define un protocolo:

```
protocol Point2D
{
    Integer x;
    void setX: Integer value;
    Integer y;
```

```

void      setY: Integer value;
Boolean isEqual: Point2D otherPoint;
}

```

A partir de un protocolo no se puede diferenciar entre datos calculados y datos almacenados. Por ejemplo, la declaración anterior del protocolo `Point2D` define el mensaje `x` y el mensaje `setX`: para manipular la coordenada `x` de un punto bidimensional. Las parejas de mensajes como la anterior se denominan *propiedades*.

Def. 5-4. Una *propiedad de un protocolo* define una pareja de mensajes cuyo objetivo es acceder o modificar el valor un objeto descrito por el protocolo asociado a la propiedad.

La definición del protocolo `Point2D` anterior puede modificarse para hacer más explícita esa relación:

```

protocol Point2D
{
    property Integer x;
    property Integer y;

    Boolean isEqual: Point2D otherPoint;
}

```

Nótese que *no* es posible inferir a partir del protocolo si la implementación de las coordenadas `x` o `y` se realiza reservando memoria en un objeto `Punto2D`, manteniéndose una separación clara entre interfaz e implementación. Una propiedad es parecida a la construcción `field` de Cecil [Chambers, 1992b; pp. 42-43]. El acceso a las propiedades tiene una sintaxis similar a otras expresiones de mensaje:

```

Point2D p = Point2D.clone;
Integer x = p.x;

```

La sintaxis del envío de mensajes sin paréntesis esconde la implementación de la propiedad `x`: puede ser un cálculo o un valor almacenado. La sentencia de asignación natural usa implícitamente el mensaje de acceso de escritura de una propiedad. Las dos expresiones siguientes son equivalentes:

```

p.y = p.x;
p.setY: p.x;

```

Las expresiones de mensajes en XC pueden encadenarse sin necesidad de paréntesis superfluos tan comunes en C++ o Java:

```

unsigned xStrLength = p.x.asString.length;

```

5.2.1.3 Prototipos

Como hemos visto, el modelo de objetos de XC está basado en prototipos. Los prototipos definen una implementación completamente independiente para un conjunto de protocolos dado.

❖ **Def. 5-5.** Un *objeto prototipo* o *ejemplar* describe un conjunto de datos privados y las operaciones —quizá públicas— que acceden y manipulan esos datos.

Cada prototipo ha de implementar o adoptar al menos un protocolo. Por defecto adopta un protocolo con su mismo nombre. En el siguiente ejemplo el objeto prototipo `Point2D` adopta automáticamente el protocolo `Point2D`.

```

prototype Point2D
{ // ... }

```

La definición anterior es equivalente a:

```

prototype Point2D implements: Point2D
{ // ... }

```

La lista de protocolos implementados por un prototipo aparece tras la palabra clave `implements`:. Protocolos y prototipos pueden tener el mismo identificador porque el lenguaje mantiene sus espacios de nombres separados. Tener nombres iguales de protocolo y prototipo facilita razonamientos similares a los que se producen en lenguajes basados en clases, sin necesidad de mezclar técnicamente ambos conceptos. Un prototipo puede implementar más de un protocolo:

```
prototype Figure2D implements: List, Array
{ // ... }
```

5.2.1.3.1 Métodos

Son operaciones asociadas a los datos que forman parte de un objeto y pueden efectuar llamadas a otros métodos del mismo u otros objetos.

- ❖ **Def. 5-6.** Se llaman *métodos* a los procedimientos de manipulación de los datos privados de un objeto, e implementan una funcionalidad dada.
- ❖ **Def. 5-7.** Una *firma de un método* define el nombre de una operación, los argumentos y protocolos de los argumentos, y el protocolo del valor de retorno.

Cada método tiene una firma que lo caracteriza y diferencia de cualquier otro método. Cada método de un prototipo se asocia a un mensaje de un protocolo que tenga la misma firma. Por tanto, un mensaje identifica una funcionalidad y un método la implementa. Siguiendo el ejemplo anterior, la firma de la definición del método `isEqual`: del prototipo `Point2D` queda asociada a la firma del mensaje `isEqual`: del protocolo `Point2D`—adoptado automáticamente—porque ambas son iguales.

```
prototype Point2D
{
    // ...

    Boolean isEqual: Point2D otherPoint
    { // ... }
}
```

5.2.1.3.2 Propiedades y propiedades compartidas

Los datos de un objeto se definen con *propiedades* de un prototipo. Los datos internos de un objeto se llaman habitualmente también *campos*, *slots* o *variables de instancia*, pudiéndose usar cualquiera de ellos como sinónimos.

- ❖ **Def. 5-8.** Una *propiedad de prototipo* encapsula un dato y el acceso y actualización de ese dato.

Los datos son siempre privados para esconder la estructura interna de un objeto a sus clientes y eliminar interdependencias. Si los clientes que acceden a un objeto dependieran del conocimiento de la posición o la estructura de un dato particular del mismo, tal objeto no podría cambiar su implementación sin que involucrarse también cambios en los clientes, porque éstos conocerían las posiciones relativas de los datos dentro del objeto accedido. Por tanto, esconder la estructura interna permite no recompilar los clientes cuando se efectúen cambios en la implementación del objeto accedido, manteniendo la modularidad. El acceso al valor de un dato usa métodos equivalentes a los definidos por las propiedades de un protocolo. Las propiedades definen el protocolo de *posibles* propiedades de la implementación. Ésta es libre de elegir implementar una propiedad de un protocolo mediante una propiedad de implementación o algún otro cómputo. En el ejemplo que aparece a continuación, la implementación del prototipo `Point2D` opta por definir las coordenadas como datos privados.

```
prototype Point2D
{
    property Integer x;
    property Integer y;

    Boolean isEqual: Point2D otherPoint
    { // ... }
```

}

- ❖ **Def. 5-9.** Una *propiedad de prototipo compartida* es una propiedad que puede ser accedida por todos los objetos de una misma familia (Def. 5-15).

Como cualquier otra propiedad, una propiedad compartida incluye los métodos de acceso y escritura al valor de la propiedad, que es estrictamente privado. Una propiedad compartida es heredada cuando se obtiene un prototipo derivado. Sin embargo, el espacio de almacenamiento de la propiedad no se comparte entre ambos, puesto que un prototipo heredado crea una nueva familia de objetos. La herencia entre prototipos se estudia con más detalle en el apartado 5.3.2 *Herencia simple de implementación*.

```
prototype Point2D
{
    shared property familyMembersCount;
    // ...
}
```

5.2.1.3.3 Prototipos y clases

La programación tradicional con clases de muchos lenguajes orientados a objetos surge con naturalidad del modelo de prototipos. Nótese que esto es posible debido a las características estáticas de la herencia y al uso de concatenación para implementar herencia. En muchos casos, existe un objeto prototipo usado como referencia para crear nuevos objetos de su misma familia. Ese objeto hace el papel de una *clase* en un lenguaje orientado a objetos basado en clases. A partir de las definiciones anteriores se pueden obtener las definiciones usuales de los modelos de objetos basados en clases, aplicables en XC debido a la forma en que implementa el modelo de objetos.

- ❖ **Def. 5-10.** Una *clase* es equivalente a un objeto prototipo usado preferentemente para realizar la clonación.
- ❖ **Def. 5-11.** Una *instancia* es equivalente a un objeto clonado a partir de un objeto prototipo.
- ❖ **Def. 5-12.** Una *variable de instancia* es equivalente a la definición de una propiedad de un objeto prototipo.
- ❖ **Def. 5-13.** Una *variable de clase* es equivalente a una propiedad compartida de un objeto prototipo.

No obstante, y a diferencia de un lenguaje basado en clases, cualquier objeto de una misma familia sirve para crear nuevos miembros de esa familia. La aproximación es similar en este sentido a la de Clyptic Script [M. Lentczner en Smith, 1994; pp. 104-105] y Agora [Steyaert, 1994], donde el concepto de clase se implementa directamente con un objeto prototipo.

5.2.1.4 Creación e inicialización de objetos

La creación habitual de objetos en lenguajes con prototipos es mediante clonación. Con clases se usa la instanciación. Más allá de las diferencias filosóficas, ambos procedimientos hacen exactamente lo mismo: crear objetos. Como veremos, su implementación debe ser también similar.

- ❖ **Def. 5-14.** La *clonación* o *copia* crea nuevos objetos a partir de otros existentes, usados como prototipos. El resultado de la clonación es la creación de un nuevo objeto igual al objeto clonado. Cualquier objeto puede servir como prototipo de otros.
- ❖ **Def. 5-15.** Una *familia* de objetos es el conjunto de objetos clonados a partir de objetos prototipos que poseen las mismas características.

Muchas veces hace falta la obtención de un nuevo objeto con ciertos valores por defecto. La estrategia a seguir es clonar un prototipo que tenga esos valores. Los prototipos expresan mejor el conocimiento acerca de los valores por defecto [Lieberman, 1986; p. 215]. Por ello, la clonación de nuevos objetos es posible a partir de cualquier ejemplar. La clonación en XC no comparte parte de la información con el

objeto original como proponía [Lieberman, 1986; pp. 217-219], sino que se efectúa una copia para evitar interdependencias implícitas entre el objeto original y el clonado, aun cuando implique una penalización en el tiempo de creación del nuevo objeto y ocupe más memoria. La realización de una copia superficial o en profundidad queda a discreción de la implementación del método `clone` de cada objeto, dejando a juicio del diseñador equilibrar eficiencia y ausencia de estado compartido. Se opta por esta solución intermedia porque hacer únicamente copias en profundidad no es práctico, y compartir información siempre en el lenguaje genera excesivas dependencias implícitas.

Otras veces se sabe con seguridad que el nuevo objeto clonado va a cambiar muchas o todas sus propiedades, por lo que clonar un objeto existente simplemente no es eficiente. Desperdicia el tiempo de una inicialización que se ha de sobreescibir. Ya en las primeras propuestas de lenguajes de prototipos se subraya esta dificultad [Bornning, 1986; p. 39].

Para reconciliar ambos modos de trabajo, la clonación se divide en *creación* e *inicialización*, identificados respectivamente con los mensajes `alloc` e `init`. Esta convención se denomina *protocolo de clonación de objetos*. Su implementación por defecto en `Object` es:

```
Self clone
{
    return self.alloc.init;
}
```

donde `self` es una palabra clave que representa el tipo de la pseudo-variable `self`, el mensaje `alloc` se encarga de la obtención de memoria para el objeto, e `init` de la inicialización por defecto. La separación permite que una implementación eficiente realice únicamente la llamada a `alloc`, evitando la sobrecarga de la inicialización por defecto. Si se necesitan parámetros de creación o inicialización, se usan variantes de estos mensajes con parámetros. Por ejemplo, un prototipo `Counter` sencillo puede inicializarse con un método `init`: que reciba como parámetro el valor del contador.

```
prototype Counter extends: Object
{
    property Unsigned counter;

    Self init: Unsigned value
    {
        self.counter = value;
        return self;
    }

    Self init
    {
        return self.init: 1;
    }
}
```

El método `clone` de `Counter` es heredado de `Object` y envía los mensajes `alloc` e `init`. Para mantener el protocolo de clonación de objetos, el prototipo `Counter` debe definir un método `init` sin parámetros que dé correctamente una inicialización por defecto a la propiedad `counter` cuando reciba un mensaje de clonación.

```
Counter c = Counter.clone;
```

En la sentencia anterior, el contador `c` se inicializa con uno, ya que el método heredado `clone` llama al método heredado `alloc` y al método sobreescrito `init` que inicializa a uno la propiedad `counter`. El esquema resultante final es muy similar al encontrado en lenguajes basados en clases tanto en espíritu como en objetivos prácticos.

5.3 Extensiones

En este apartado se revisan las extensiones a protocolos y prototipos, introducidas informalmente en el apartado 3.4.5 *Tipos de extensiones*. Siguen las guías de diseño identificadas en el

capítulo 4. La herencia múltiple de protocolos, la herencia simple de prototipos, las extensiones por categorías de protocolos y prototipos y las extensiones ortogonales se estudian con ejemplos de su utilidad práctica.

5.3.1 Herencia múltiple de tipos

Los tipos en XC se definen mediante protocolos. Un protocolo puede heredar de varios protocolos previamente definidos. La comprobación realizada es estrictamente sintáctica. Si dos protocolos definen un mismo mensaje, entonces éste se considera único, siendo responsabilidad del nuevo tipo de datos resolver cualquier ambigüedad semántica. La herencia es estática y monótona. Por ejemplo, el protocolo `Figure2D` hereda los mensajes definidos en `List` y `Array` para manipular figuras en dos dimensiones como listas enlazadas o *arrays*. La lista de protocolos heredados aparece después de la palabra clave `extends`:

```
protocol Figure2D extends: List, Array
{ // ... }
```

5.3.2 Herencia simple de implementación

Una implementación no define *nunca* un nuevo tipo de datos. La herencia de implementación es simple, estática y monótona. Se usa concatenación como técnica de implementación de herencia. La lista de prototipos heredados aparece después de la palabra clave `extends`:. La lista de protocolos implementados aparece tras la palabra clave `implements`:. Para facilitar la factorización de la implementación, un objeto prototipo puede heredar a lo sumo de otro objeto prototipo. Se admite únicamente herencia simple de implementación para mantener sencillo el modelo de objetos. El próximo ejemplo muestra una posible implementación del objeto prototipo `Figure2D`. Al usar herencia simple, el objeto prototipo no puede heredar la implementación de `List` y `Array` simultáneamente.

```
prototype Figure2D extends: List implements: Array
{
    // Implementación del protocolo Array
}
```

Puesto que el protocolo `Figure2D` no expone el árbol de herencia de sus implementaciones, más adelante podría cambiarse la implementación de `Figure2D` para heredar de `Array` en vez de `List` sin afectar a sus clientes, que dependen únicamente del protocolo.

```
prototype Figure2D extends: Array implements: List
{
    // Implementación del protocolo List
}
```

5.3.3 Extensión a tipos y objetos existentes

Los objetos pueden llegar a ser entes bastante complejos. Para ayudar en la definición y estructuración de objetos complejos, es posible dar la descripción de un objeto por partes. Cada una de estas partes o *categorías*, establece una funcionalidad dada. A veces también es conveniente extender objetos existentes para incorporar nueva funcionalidad no prevista inicialmente. Un objeto se extiende añadiendo una nueva categoría a ese objeto. Desde el punto de vista de implementación existen dos herramientas de construcción de objetos:

- *Prototipos*. Implementan la definición de objetos prototípicos. El árbol de herencia de implementación se mantiene entre prototipos.

- *Categorías de extensión.* Añaden funcionalidad a los prototipos. La funcionalidad añadida sólo es visible para el código que conoce a esas categorías, mejorando la modularidad dentro de los objetos.

La construcción de objetos por partes tiene un cierto parecido con los *objetos partidos* o *split objects* propuestos en [Bardou & Dony, 1996], donde cada categoría corresponde con una *pieza* de un objeto partido. Tanto las categorías como las piezas tratan una vista u aspecto diferente de un objeto, permitiendo la descripción incremental de un objeto complejo a partir de diferentes fragmentos.

Pero las semejanzas terminan aquí. Los objetos partidos son de naturaleza muy dinámica: es posible añadir, modificar y borrar piezas, al igual que campos y métodos de piezas. El envío de mensajes debe identificar el objeto receptor y la pieza que debe recibir el mensaje, puesto que los campos y los métodos están siempre asociados a cada pieza. Los objetos partidos vinculan la pseudo-variable `self` al objeto partido al completo, y una segunda pseudo-variable llamada `thisviewpoint` a cada pieza, que no tiene *status* de objeto. También proponen una estrategia de resolución de métodos para enviar mensajes al objeto completo, sin tener que especificar cada pieza particular [Bardou & Dony, 1995].

Un modelo más simple de vistas de objetos se encuentra en las extensiones realizadas por NeXT a Objective-C [NeXT, 1995]. Una clase es capaz de recibir una extensión —también llamada *categoría*— después de haber sido creada. La categoría añade nuevos métodos pero no variables de instancia. El protocolo de la clase extendida se actualiza con los nuevos mensajes a los que responde la categoría. No se introduce la noción de una segunda pseudo-variable, por lo que las clases extendidas mantienen una identidad simple. Una categoría únicamente precisa la declaración de la clase a la que extiende, si bien tiene acceso a todas las variables definidas en la clase extendida, inclusive aquellas declaradas privadas. Desafortunadamente, el mecanismo de añadido de métodos a la clase principal no tiene una semántica bien definida: Los métodos añadidos por la categoría sobrescriben la tabla de métodos de la clase original, perdiéndose la posibilidad de llamar al método sustituido. Si varias categorías sobrescriben un mismo método, aquél perteneciente a la última categoría cargada es el único que prevalece. En presencia de carga dinámica existe otro vacío semántico. Dado que no se puede asegurar siempre el orden de carga, o que una nueva categoría sobrescriba un método existente previamente, el conjunto final de métodos que prevalecen en una clase quizás varíe de una ejecución a otra.

La opción elegida por XC es similar a la de Objective-C, pero con una semántica más modular. Al igual que en Objective-C, se permite que un objeto prototipo reciba una extensión de implementación o de declaración con una categoría. La extensión se produce conociendo únicamente el protocolo del prototipo que es extendido. El objeto resultante sigue siendo un objeto simple, no un objeto partido. XC admite que una categoría cree variables de instancia. Prohibe el acceso directo a las variables del objeto original por ser un comportamiento no modular. Prohibe también que una categoría declare un método asociado a un mensaje declarado por el prototipo, para evitar el problema semántico del añadido de métodos en Objective-C. La categoría sí acepta nuevos métodos anteriores y posteriores a cada mensaje del prototipo original o de cualquier otra categoría conocida de ese prototipo. Por último, la categoría puede indicar el orden relativo donde se insertará respecto al prototipo o a otra categoría conocida. Este orden se denomina *linearización de categorías* y se discute en el apartado 5.3.4 *Extensiones ortogonales*.

Las categorías solventan algunos problemas recurrentes de la programación orientada a objetos. Tradicionalmente en programación orientada a objetos se ha supuesto y confiado que muchos de los beneficios de reusabilidad surgen gracias a la herencia. La construcción de grupos de clases relacionadas o *frameworks* que cooperan para resolver una funcionalidad dada debe ayudar a factorizar aquella funcionalidad común presente. La herencia debe facilitar la especialización del *framework* a las necesidades particulares de cada cliente. Esta creencia está fuertemente arraigada y ha sido la estrategia pilar de importantes esfuerzos de desarrollo como [Taligent, 1995; pp. 27-39]. Sin embargo, el beneficio real es menor. Una seria dificultad relacionada con la utilidad práctica de los *frameworks* en lenguajes orientados a objetos tradicionales es la siguiente:

Consideremos un grupo de clases dedicado a la edición de textos que proporciona una clase `MultiLangString` para manejar cadenas en varios lenguajes. El *framework* emplea consistentemente la clase `MultiLangString` para soportar multilenguaje en todas sus clases. Por ejemplo, la clase `EditorBuffer` del *framework* puede definir un método que devuelva la cadena multilenguaje asociada a una línea dada. Usando sintaxis Java:

```
class EditorBuffer
{
    // ...
    MultiLangString textAtLine (Unsigned line)
    { // ... }
}
```

Imaginemos ahora que la clase `MultiLangString` no admite la conversión entre una cadena multilenguaje y una cadena con formato del lenguaje C. ¿Cómo se puede dar soporte a esta nueva funcionalidad? Cuando se usa herencia como mecanismo de extensión, debe crearse una subclase de `MultiLangString` que añada el método de conversión. Suponiendo que el tipo `CString` representa una cadena en formato C, se puede definir la subclase `MyMultiLangString` que implemente los métodos de conversión adicionales y herede el resto de la clase `MultiLangString` original:

```
class MyMultiLangString extends MultiLangString
{
    MyMultiLangString (MultiLangString mlstr)
    { // ... }

    MyMultiLangString (CString cstr)
    { // ... }

    CString ascString ()
    { // ... }
}
```

La extensión funciona correctamente con objetos de clase `MyMultiLangString`. También se puede usar en expresiones polimórficas con el tipo `MultiLangString` puesto que este último es su padre. Si se quiere acceder a los métodos de `MyMultiLangString` desde objetos `MultiLangString` es posible usar una expresión de conversión de tipo o *typecast*:

```
MultiLangString mlstr = new MyMultiLangString (cstr);
CString           cstr2 = ((MyMultiLangString) mlstr).ascString();
```

Pero la extensión propuesta no funciona tan bien cuando se pretende convertir el texto devuelto por el método `textAtLine` de `EditorBuffer`:

```
EditorBuffer editor = new EditorBuffer();
// ...
MultiLangString mlstr = editor.textAtLine (5);
CString           cstr = ((MyMultiLangString) mlstr).ascString(); // error!
```

El texto devuelto por el editor es un `MultiLangString`. Al no poseer `MultiLangString` el nuevo método de conversión `ascString`, la conversión de tipos de `mlstr` de la última línea fallará. El editor no puede devolver objetos cuyo tipo sea `MyMultiLangString` porque no existía cuando se implementó el editor. Hace falta especializar el editor para que devuelva un texto cuyo tipo sea `MyMultiLangString`.

```
class MyEditorBuffer extends EditorBuffer
{
    // ...
    MultiLangString textAtLine (Unsigned line)
    {
        MultiLangString originalStr = super.textAtLine (line);
        return new MyMultiLangString (originalStr);
    }
}
```

Para que la extensión funcione correctamente, por *cada clase* del *framework* original que devuelva objetos de tipo `MultiLangString` hay que construir una clase especializada que retorne objetos `MyMultiLangString`. El coste de la especialización aumenta proporcionalmente con el tamaño del *framework*. Recordemos que únicamente se está añadiendo una funcionalidad muy simple. ¿Qué ocurre si ese *framework* tiene métodos privados que devuelven objetos `MultiLangString` y no se pueden especializar? Simplemente la extensión no es factible. ¿Qué sucede si se desea añadir la funcionalidad de conversión a `CString` a la clase `java.lang.String`? Incluso suponiendo que no fuese una clase `final`, es inviable especializar todas las clases que devuelven un `java.lang.String`. La especialización del ejemplo deja de ser aceptable en la práctica. La conversión debe realizarse manualmente allí donde sea necesario.

En resumen, existe un importante número de extensiones razonables a grupos de clases que no pueden ser resueltas satisfactoriamente con especialización mediante herencia. Son aquellas que pretenden factorizar nueva funcionalidad común sobre un *framework* existente de terceros, *precisamente aquellas para las que la herencia teóricamente está ideada*. Las extensiones no son factibles en la práctica si necesitan especializar más de un número pequeño de clases. Como resultado, los *frameworks* actuales tienden a ser entes autocontenidos con especializaciones limitadas. Es interesante tener en cuenta que el *framework* potencialmente sí es extensible correctamente si se dispone de su código fuente. En este último caso, el riesgo de modificar por cuenta propia el código fuente de un *framework* de un proveedor puede hacer también inviable en la práctica las modificaciones, si se esperan nuevas versiones del mismo.

Para aumentar las posibilidades de reuso de un conjunto de clases relacionadas, debe definirse alguna técnica que realice extensiones a un *framework* existente *como si* se tuviese el código fuente, pero evidentemente sin tenerlo. En XC se utilizan categorías para esa labor que, al disponer de un sistema de tipos separado de implementación, se aplican por separado a protocolos y a prototipos.

5.3.3.1 Extensión de protocolos

Un protocolo se extiende con la declaración de una nueva categoría para ese protocolo. Todos los mensajes definidos en una categoría de protocolo deben ser distintos de los mensajes definidos en el protocolo principal. Siguiendo con el ejemplo anterior de `MultiLangString`, en XC se puede escribir la extensión como:

```
protocol category Conversions extends: MultiLangString
{
    CString ascCString;
}
```

Cuando se extiende un protocolo con más de una categoría, puede surgir una ambigüedad si dos categorías intentan extender el protocolo original con el mismo mensaje. En ese caso, tiene preferencia la categoría cuya declaración se haya encontrado antes. La segunda declaración generará un error. No se usa en este caso el criterio de unificación de declaración de mensajes porque se espera que dos categorías distintas extiendan aspectos diferentes de un tipo de datos. En cambio, cuando se hereda de otros protocolos se espera que el nuevo tipo de datos dé una implementación consistente de los mensajes heredados. Por ello sí se permite la mezcla de mensajes. En el próximo ejemplo, el mensaje `printA` heredado y declarado en el protocolo `C` corresponde con un único mensaje en `C`. Las declaraciones de `ccat2`, `ccat3` y `ccat4` son erróneas porque intentan declarar mensajes ya heredados, disponibles en el protocolo principal, o disponibles en categorías previamente declaradas.

```
protocol A
{ void printA; }

protocol B
{ void printA; void printB; }

protocol C extends: A, B
{ void printA; void printC; }

protocol category CCat1 extends: C
```

```

{ void printCCat1; }

protocol category CCat2 extends: C
{ void printA; } // error! printA definido en C

protocol category CCat3 extends: C
{ void printB; } // error! printB heredado de B

protocol category CCat4 extends: C
{ void printCCat1; } // error! printC definido en CCat1

```

5.3.3.2 Extensión de prototipos

Un prototipo se extiende con una nueva definición de categoría para ese prototipo. Si existe una categoría de protocolo con igual nombre en el mismo módulo, la categoría de prototipo la adopta implícitamente. Las ambigüedades relacionadas con la declaración de mensajes se resuelven en los protocolos. Los prototipos no se ven afectados.

```

prototype category Conversions extends: MultiLangString
{
    CString ascString
    { // ... }
}

```

El ejemplo anterior de la clase `MultiLangString` en Java se resuelve con sencillez usando categorías en XC. La categoría `Conversions` añade el mensaje y el método de conversión `ascString` directamente al protocolo y al prototipo `MultiLangString`. No hace falta extender con herencia el protocolo y prototipo `MultiLangString` y crear `MyMultiLangString`, porque se trata de una extensión a un tipo y una implementación existente. Tampoco es necesario extender con herencia `EditorBuffer` o cualquier otro protocolo o prototipo que devuelva un `MultiLangString`, como ocurriría con Java y cualquier otro lenguaje orientado a objetos que no soporte extensiones con categorías. En XC no se producen errores al llamar a los métodos de conversión, ya que una vez añadida la categoría `Conversions`, es posible enviar el mensaje `ascString` a cualquier objeto cuyo tipo sea `MultiLangString`.

```

EditorBuffer editor = EditorBuffer.clone;
// ...
MultiLangString mlstr = editor.textAtLine: 5;
CString      cstr  = mlstr.ascString; // ok

```

5.3.4 Extensiones ortogonales

Las extensiones ortogonales definen métodos anteriores y posteriores a cualquier mensaje declarado en un protocolo. Los métodos anteriores y posteriores son métodos estrictos de implementación, no existiendo una declaración asociada a los mismos. Su declaración debe ser igual a la del mensaje base asociado. Por ello, no existe una declaración de extensiones ortogonales de protocolos.

Un aspecto importante de las extensiones ortogonales es el orden en que se combinan y afectan al comportamiento semántico de los objetos. Por ejemplo, no es lo mismo un prototipo que sea primero *thread-safe* y luego persistente, que primero persistente y luego *thread-safe*. En el primer caso se está bloqueando el objeto y luego haciéndolo persistente. En el segundo caso es al revés, pudiendo ser fuente de errores si otros hilos de ejecución intentan acceder al objeto en mitad de la operación de persistencia.

5.3.4.1 Extensión ortogonal de prototipos

Una extensión ortogonal se efectúa dentro de una categoría de prototipo, siendo siempre comprobada en tiempo de compilación. Al realizar la extensión ortogonal usando una categoría se

obtienen dos resultados importantes. Por un lado, la extensión ortogonal mantiene la modularidad del prototipo original, no existiendo acceso privilegiado al mismo. Por otro lado, la extensión ortogonal no requiere construir nuevos metaobjetos ni un árbol de herencia paralelo. Los métodos se añaden directamente al objeto prototipo en cuestión. En el siguiente ejemplo, la categoría `CounterThreadSafe` hace una extensión ortogonal que da un soporte básico para código multitarea.

```
prototype Counter extends: Object
{
    property Unsigned counter;
    // ...
}

prototype category CounterThreadsafe extends: Counter
{
    property System.Threading.Lock objLock;

    after Self init
    {
        self.objLock = System.Threading.Lock.clone; }

    before void setCounter: Unsigned value
    {
        self.objLock.lock; }

    after void setCounter: Unsigned value
    {
        self.objLock.unlock; }
}
```

La propiedad `counter` en el prototipo `Counter` define automáticamente los métodos de acceso `counter` y `setCounter:..`. La categoría `CounterThreadsafe` extiende el objeto prototipo `Counter` con un nuevo campo `objLock`, para bloquear el objeto antes y después de que se acceda al contador. El primer método que intercepta la categoría es `init`, que es parte del protocolo de clonación de objetos heredado de `Object` (véase el apartado 5.2.1.4). El protocolo de clonación que define `Object` usa el mensaje `clone`, que a su vez envía los mensajes `alloc` de petición de memoria e `init` de inicialización por defecto. Así, cuando se clona un contador:

```
Counter c = Counter.clone;
```

se están enviando los mensajes `alloc` e `init` heredados de `Object`. Justo después de inicializar cada objeto, el método posterior `init` asigna el campo `objLock` a una copia del objeto prototipo `Lock`. El método `setCounter:..` es interceptado antes y después por la categoría `CounterThreadsafe` para impedir la modificación concurrente del contador por varios hilos de ejecución. Incluye el código que define una sección crítica antes y después de la actualización del contador usando el objeto de biblioteca `Lock`.

Supongamos que posteriormente se pretende añadir una alarma al contador. Cuando se supere determinado valor, la alarma deberá saltar. La funcionalidad de alarma es ortogonal al propósito del contador original, por lo que se puede añadir como una nueva categoría al prototipo `Counter`.

```
prototype category AlarmCounter extends: Counter after: CounterThreadsafe
{
    void activateAlarm
    {
        // ...
    }

    void deactivateAlarm
    {
        // ...
    }

    after void setCounter: Unsigned value
    {
        if ( self.counter >= ALARM_VALUE )
            self.activateAlarm;
        else
            self.deactivateAlarm;
    }
}
```

La categoría `AlarmCounter` añade los nuevos métodos `activateAlarm` y `deactivateAlarm` para controlar la alarma, así como un método posterior a la modificación del contador para comprobar si debe ésta activarse o no. El elemento más importante de la definición de la categoría es la cláusula

after: que indica el orden de inserción de los métodos anteriores y posteriores en ejecución. La comprobación del valor del contador para activar o desactivar la alarma debe hacerse *después* de que el contador haya sido bloqueado por la categoría `CounterThreadSafe`, para evitar que otro hilo de ejecución modifique el contador tras hacer la comparación con `ALARM_VALUE`.

El lenguaje establece una semántica fija para la ordenación de métodos `before` y `after` provenientes de varias categorías para simplificar al máximo el razonamiento con ellas. Una categoría más prioritaria debe ejecutar sus métodos `before` *antes* que una menos prioritaria y sus métodos `after` *después*. Se trata de un equilibrio entre flexibilidad y simplicidad. Otros lenguajes como CLOS son demasiado flexibles. Pueden definir una ordenación potencialmente diferente por cada clase, haciendo mucho más complejo el razonamiento acerca del comportamiento de los métodos anteriores y posteriores.

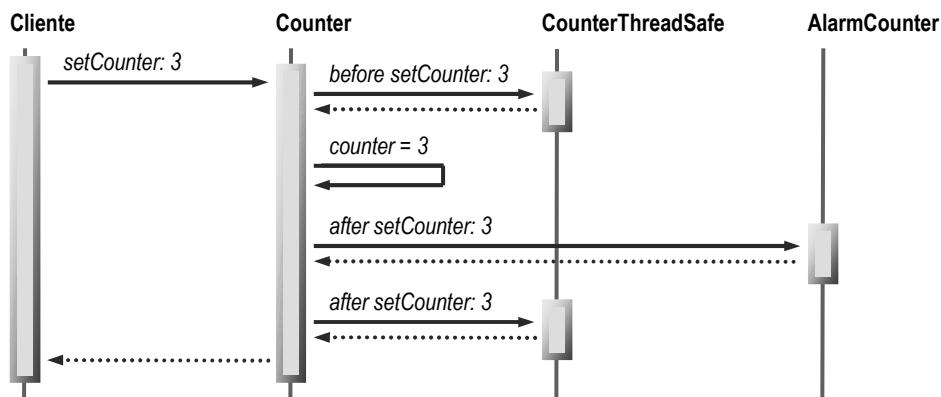


Figura 5-1 Llamadas a métodos anteriores y posteriores en extensiones ortogonales.

AlarmCounter debe ejecutarse después de CounterThreadSafe para asegurar que ningún otro hilo de ejecución está modificando el contador interno.

Volviendo al ejemplo en curso, el significado de `after` en la categoría `AlarmCounter` indica que la categoría siendo definida tiene menor prioridad y ha de ejecutarse *después* de la categoría especificada por la palabra clave. Con esta configuración, el envío del mensaje `setCounter:` es el mostrado en la Figura 5-1.

Ahora supongamos que se desea eliminar la funcionalidad ortogonal `CounterThreadSafe`. La categoría `AlarmCounter` depende explícitamente de la existencia de `CounterThreadSafe`, si bien nada más tienen que ver entre sí. Eliminar la categoría `CounterThreadSafe` genera un error en tiempo de compilación en `AlarmCounter`. O, si no se recompila, un error en tiempo de carga del módulo donde se define `AlarmCounter`, debido a que depende de una categoría no existente. La evaluación del error permite al usuario identificar si la categoría eliminada afecta o no a `AlarmCounter`. En este caso no afecta, por lo que se puede quitar la dependencia con `CounterThreadSafe`.

```
prototype category AlarmCounter extends: Counter
{ // ... }
```

Para añadir y eliminar categorías con naturalidad, es fundamental que implementen comportamientos ortogonales independientes entre sí. El lenguaje garantiza que no se tiene acceso privilegiado a otras categorías para fomentar la ortogonalidad y la modularidad. Pero si una categoría envía mensajes a otras categorías del mismo prototipo, deben añadirse y eliminarse en grupo o recompilar las categorías dependientes para evitar la dependencia. En cualquier caso, el compilador o la carga detecta las inconsistencias. La eliminación de una categoría es un comportamiento no monótono, porque elimina información usada por otras categorías, haciendo necesaria la recompilación.

Consideremos por último que se desea añadir tiempo después la categoría `CounterThreadSafe` otra vez. La declaración original es la correcta si no se conoce la existencia de `AlarmCounter`. En este caso

es responsabilidad del creador de `AlarmCounter` volver a tener en cuenta a `CounterThreadSafe`. Pero si se conoce la existencia de `AlarmCounter`, es importante evaluar si `CounterThreadSafe` debe ejecutarse antes o después de aquella. En este caso debe ejecutarse *antes*. La dependencia se añade modularmente sin modificar `AlarmCounter` usando:

```
prototype category CounterThreadSafe extends: Counter before: AlarmCounter
{ // ... }
```

Análogamente al modificador `after`, el significado de `before` indica que la categoría siendo definida es más prioritaria y debe ejecutarse *antes* que la categoría especificada por la palabra clave.

El ejemplo anterior, si bien relativamente sencillo, sirve para poner de manifiesto que no es difícil que existan dependencias semánticas entre las extensiones ortogonales, dependencias que deben tenerse en cuenta y que corresponden con aspectos de implementación de las extensiones. Es la dependencia de implementación la que obliga a evaluar de nuevo si nuevas extensiones deben situarse antes o después que otras, determinando un orden lineal en las extensiones ortogonales.

- ❖ **Def. 5-16.** Un *orden de linearización de categorías* es una ordenación total obtenida a partir de una ordenación parcial relativa entre varias categorías que realizan extensiones ortogonales²², y el orden de declaración.

El ejemplo previo muestra cómo es viable añadir y eliminar extensiones ortogonales con sencillez y modularidad, sin obligar a la edición del código de otras extensiones. Se ha visto también que no se puede eliminar por completo la dependencia semántica entre extensiones ortogonales, pero sí es minimizable a dos conceptos. (1) El nombre de la extensión ortogonal que codifica su semántica y (2) el orden relativo de linearización de categorías durante ejecución, ambas dependencias explícitas.

Las dependencias adquiridas entre unas extensiones y otras provienen de cuáles son las extensiones anteriores conocidas. Por ejemplo, si `AlarmCounter` se crea primero, `CounterThreadSafe` puede indicar su orden relativo modularmente sin necesidad de modificar ni el prototipo `Counter` ni la categoría `AlarmCounter`. Si por el contrario es `CounterThreadSafe` la que se crea primero, `AlarmCounter` también es capaz de indicar su orden relativo modularmente. La especificación del orden precisa únicamente identificar una categoría `before` o `after`, para que el compilador sepa dónde efectuar la inserción. El orden relativo es la mínima información necesaria para permitir que entre dos categorías cualesquiera pueda añadirse una tercera, ya que se trata de extensiones no planificadas y no puede preverse cuándo se van a agregar.

Por ejemplo, es factible insertar una nueva categoría `CounterTransaction` entre `CounterThreadSafe` y `AlarmCounter` usando:

```
prototype category CounterTransaction extends: Counter after: CounterThreadSafe
{
    before void setCounter: Unsigned value
        { // guardar el estado del contador }

    after void setCounter: Unsigned value
        { // comprobar el nuevo estado, si es inválido restaurar el estado anterior }
}
```

Esta vez, los métodos anterior y posterior sirven para guardar el estado del contador y validar si el nuevo valor es correcto. La comprobación debe hacerse dentro de la sección crítica que activa `CounterThreadSafe`. En caso de que `AlarmCounter` hubiese indicado que se sitúa detrás de `CounterThreadSafe`, la categoría `CounterTransaction` podría insertarse correctamente antes que `AlarmCounter` si se declara después que ésta, ya que la ordenación final de categorías es total y resuelta a partir del orden final de declaración.

²² Matemáticamente, una ordenación parcial de un conjunto de elementos no garantiza un orden estricto entre todos sus elementos con las relaciones $< y >$, sino con las relaciones $\leq y \geq$. La ordenación es relativa porque la relación se indica respecto a otro elemento del conjunto. La ordenación total es obtenida por el orden de declaración de la ordenación relativa, teniendo preferencia una declaración posterior sobre una declaración anterior.

Téngase en cuenta que una categoría que no añada métodos anteriores o posteriores no necesita especificar su orden, puesto que su posición en la linearización de categorías es irrelevante. Sólo es relevante para la ordenación de métodos anteriores y posteriores alrededor de cada método base. Por ejemplo, una categoría que añade un método de conversión del valor del contador a una cadena de texto con formato XML no depende de ninguna otra categoría ya que no define métodos anteriores o posteriores.

```
prototype category XMLConversions extends: Counter
{
    String asXMLString
    { return "<Counter>" + self.counter.asString + "</Counter>"; }
}
```

5.4 Protocolos, prototipos y sus variantes

Los protocolos y los objetos prototípicos describen por defecto el comportamiento de objetos generales que pueden ser extendidos de diferentes formas.

- ❖ **Def. 5-17.** Los *grados de libertad* de un objeto indican sus posibilidades de extensión. Un protocolo tiene dos: herencia y categorías. Un prototipo tiene cuatro: herencia, categorías, variables de instancia, y metamétodos anteriores y posteriores.

El modificador **final** limita que un protocolo sea extendido usando herencia de tipos. El modificador **confined** aplicado a un protocolo indica que un protocolo no sea extendido usando categorías. Ambos se pueden combinar. Por ejemplo:

```
final confined protocol A
{ // ... }
```

representa un protocolo que no es posible extender con categorías, ni heredar de él. Únicamente puede ser adoptado por un prototipo de igual nombre, como se muestra a continuación:

```
protocol B extends: A
{ // ... } // error!

prototype C implements: A
{ // ... } // error!

prototype A
{ // ... } // ok. A implementa el protocolo A por defecto.
```

La Tabla 5-1 resume la aplicación de los modificadores de protocolos. La estructura de la Tabla 5-1 es análoga a la Tabla 5-2—que se verá un poco más adelante—para resaltar sus similitudes y diferencias. Los protocolos sólo tienen un nivel base, no un nivel meta, ya que los metamétodos se asocian directamente a los mensajes del nivel base. En los protocolos únicamente es posible definir más mensajes usando herencia o categorías, que son los grados de libertad controlados por las palabras clave. Por defecto la extensión de protocolos está activa para herencia y categorías. La palabra clave **confined** limita la extensión por categorías y no afecta a la extensión con herencia. La palabra clave **final** limita la extensión por herencia pero no afecta a la extensión por categorías.

Palabra clave	Nivel base	Nivel base
	mensajes	mensajes
por defecto:	Sí	Sí
confined	--	No
final	No	--
	Herencia	Categorías

Tabla 5-1 Variedades de protocolos.

Un esquema parecido se aplica a los prototipos. La flexibilidad que dan los grados de libertad posibles en los prototipos tiene su precio: la pérdida de eficiencia. Para recuperarla el modelo de objetos de XC debe ser más elaborado, ya que, en ciertas condiciones, la sobrecarga de los objetos prototipo generales puede ser un impedimento serio. Para hacer viable la programación de sistemas, es necesario facilitar la construcción de objetos especializados, que obtienen rapidez a costa de ceder en generalidad. Habitualmente esos objetos son sencillos y muchas veces también privados. La generalidad es útil para aquellos objetos prototipo públicos, puesto que ayuda a que los clientes puedan aprovecharlos mejor. Pero, a la vez, son un obstáculo para conseguir implementaciones eficientes.

Los objetos prototipo por defecto son los más generales, teniendo cuatro grados de libertad posibles. Usando prefijos modificadores es posible limitar la generalidad de los prototipos. La palabra clave **fixed** fija el tamaño de los objetos creados, impidiendo la definición en la implementación de nuevas variables de instancia durante la herencia o con categorías. La palabra clave **baselevel** deshabilita la posibilidad de crear métodos anteriores y posteriores ya sea al heredar o extender con categorías. La palabra clave **final** impide la extensión por herencia. Y la palabra clave **confined** prohíbe la extensión con categorías de un objeto prototipo.

Los modificadores **fixed** y **baselevel** son heredados por prototipos derivados²³. Los modificadores **final** y **confined** se aplican a cada prototipo por separado. La Tabla 5-2 resume las opciones posibles. En un objeto prototipo se pueden definir métodos y variables —parte del nivel base— y metamétodos anteriores y posteriores —en el nivel meta—. El mismo esquema se aplica a una extensión por herencia o por categorías, que son las mostradas en la Tabla 5-2. Las entradas vacías indican que el modificador no afecta al valor por defecto.

Palabra clave	Nivel base		Nivel meta	Nivel base		Nivel meta
	métodos	variables	metamétodos	métodos	variables	metamétodos
por defecto: fixed baselevel confined final	Sí	Sí	Sí	Sí	Sí	Sí
	--	No	--	--	No	--
	--	--	No	--	--	No
	--	--	--	No	No	No
	No	No	No	--	--	--
<i>Herencia</i>				<i>Categorías</i>		

Tabla 5-2 Variedades de objetos prototipo.

Por ejemplo, el prototipo siguiente:

```
final confined baselevel prototype A extends: Object
{ // ... }
```

es un objeto que deriva de **Object**, que no se puede extender con categorías o herencia, y que no permite definir metamétodos. Algunos de los objetos prototipo más importantes de XC están definidos en **XC.Lang**, **XC.Runtime** y **XC.Number**. Se muestran en la Figura 5-2.

²³ Las razones se explican en el apartado 8.7.1 *Modificadores de optimización de objetos*.

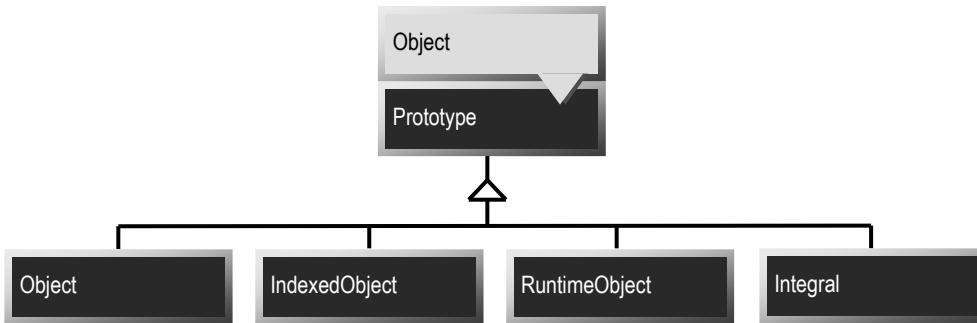


Figura 5-2 Jerarquía básica de protocolos y prototipos de XC.

Todos los prototipos básicos adoptan el protocolo object, ofreciendo distintas implementaciones básicas.

En general, todos los objetos extensibles derivan de object, aquellos en los que se quiere controlar su extensión por motivos de implementación heredan del prototipo básico Prototype, y los objetos indexados lo hacen de IndexedObject. Las características principales son:

- **Prototype.** Es el origen de la jerarquía de objetos prototípico. Implementa el protocolo básico de objetos. Se publica en el protocolo del módulo XC.Lang con el modificador `confined` para evitar que sea extensible usando categorías y bloquear un posible aumento de tamaño, pero permitir que se pueda heredar de él. El tamaño de Prototype es suficiente para contener bien un puntero bien un registro de máquina. El lenguaje garantiza que Prototype no implementa nunca metamétodos.
- **Object.** Representa el prototipo de un objeto genérico, extensible mediante categorías, variables de instancia, herencia y metamétodos. No define ningún modificador.
- **IndexedObject.** Es el objeto prototípico de los objetos indexados, usados para implementar toda clase de arrays. Tampoco define ningún modificador.
- **RuntimeObject.** Es el objeto básico para la implementación del soporte en tiempo de ejecución. Se declara con el modificador `confined` para que el soporte en tiempo de ejecución garantice que los tamaños de los objetos que lo implementan son los esperados por el compilador.
- **Integral.** Es el prototipo usado para implementar los tipos de datos predefinidos. Usa los modificadores heredables `fixed` y `baseLevel`, para tener tamaño fijo a un registro de máquina y optimizar las llamadas a métodos al prohibirse los metamétodos.

Los objetos básicos sirven como ejemplo de uso de los modificadores. Un aspecto importante de ellos es su relación con la herencia. Los objetos en XC son extensibles usando categorías y herencia. Bloquear únicamente futuras extensiones de herencia o categorías con `final` y `confined` en un prototipo dado no es suficiente, porque es necesario también bloquear las extensiones por categorías de sus antecesores. Prototype usa el modificador `confined` para asegurar que no puede ser extendido. Por tanto, es idóneo para nuevos objetos cuyo tamaño se quiera mantener fijo, y no se pretenda heredar o extender mediante categorías, como el prototipo B siguiente:

```
final confined prototype B extends: Prototype
{ // ... }
```

Si se pretende fijar el tamaño del objeto a un valor constante, pero habilitar la herencia y extensiones con mensajes y métodos, debe bloquearse la creación de nuevas variables en las categorías del nuevo objeto con el modificador `fixed` y en objetos ascendientes con el modificador `confined`. El prototipo C es de ese estilo:

```
fixed prototype C extends: Prototype
{ // ... }
```

Los objetos que hereden de `C` tienen también tamaño fijo y siempre igual a `C`. Nótese que si `C` hubiese heredado de `Object`, los objetos que heredasen de `C` tendrían también tamaño fijo e igual a `C`, pero no se podría garantizar un tamaño siempre constante, porque `Object` puede ser extendido con nuevas variables de instancia. El prototipo `Integral` es similar a `C`, pues admite la extensión por herencia, fija el tamaño de sus descendientes con `fixed` y hereda de `Prototype`, aunque en este caso particular incluye, además, el modificador `baselevel` para impedir el uso de metamétodos en los tipos integrales y hacer optimizaciones.

Comportamientos más complejos son posibles. Por ejemplo, `RuntimeObject` es la raíz de herencia de los objetos prototipo del soporte en tiempo de ejecución del lenguaje. Se declara `confined` y hereda de `Prototype`, también `confined`. Externamente, `RuntimeObject` no puede ser extendido con categorías, por lo que se garantiza que el tamaño de `RuntimeObject` no varía, a no ser que su implementación dentro del *runtime* decida que debe incluir variables adicionales. `RuntimeObject` no usa el modificador `fixed` porque en ese caso, todos los prototipos derivados tendrían que tener el mismo tamaño que él y no podrían definir nuevas variables de instancia. `RuntimeObject` da flexibilidad al implementador, puesto que internamente sí puede ser extendido mediante categorías, pero es publicado con restricciones para evitar que terceros modifiquen el comportamiento del objeto básico del soporte en tiempo de ejecución. Los modos de publicación se ven en el apartado 5.5.4.1 *Declaraciones de módulos*.

5.5 Módulos

Los módulos son una estructura sintáctica introducida en los apartados 3.2 *Conceptos básicos* y 3.4.2 *Módulos*. El objetivo principal de los módulos es particionar un programa para distribuir y esconder la complejidad total del mismo, haciéndolo más manejable. Los módulos levantan barreras de abstracción que limitan la propagación de interdependencias entre partes de un programa, siguiendo el principio de encapsulación y las premisas de diseño del capítulo 3. Para limitar y controlar las interdependencias, la comunicación entre módulos debe realizarse con protocolos separados de la implementación. La definición de módulo (Def. 3-16) introduce —pero no explica— los aspectos más relevantes de los módulos, que son los siguientes. Un módulo es:

- *Una partición sintáctica de un programa.* Define una división de un programa en distintas partes identificables únicamente. Reduce el tamaño de los bloques de código que son foco de atención durante el desarrollo, y es donde se asienta el resto de las propiedades de los módulos: abstracción, espacio de nombres, agrupación de elementos relacionados, reglas de ámbito y compilación separada.
- *Un control de abstracción.* Limita la visibilidad de cada una de las partes de un programa a otras, reduciendo las interdependencias entre ellas. La abstracción separa interfaz e implementación. La interfaz del módulo es el protocolo visible a los clientes, la implementación es la que efectúa los cálculos particulares asociados con la abstracción.
- *Un espacio de nombres.* La gestión del espacio de nombres de un programa cobra mayor importancia a medida que los programas se hacen más grandes. La colisión de nombres en un programa es un serio impedimento para su integración en un programa mayor. El espacio de nombres garantiza que los nombres definidos en un módulo son independientes de aquellos definidos en otros módulos.
- *Unas reglas de ámbito para la resolución de referencias a otras partes del programa.* Las reglas de ámbito seleccionan cuál es el elemento del lenguaje asociado a un nombre dado. Existen reglas de ámbito locales para bloques de código, funciones, procedimientos y métodos. Los módulos definen con naturalidad las reglas de ámbito globales, aquellas relacionadas con la estructura general del programa.

- *Un lugar donde declarar elementos relacionados de un programa.* Funciones, procedimientos, constantes, enumerados y declaraciones de clases o prototipos suelen cooperar —a veces estrechamente— para resolver un problema común. El módulo es una unidad sintáctica que hace explícita esa relación de estrecha colaboración.
- *Una unidad de compilación separada.* La compilación separada ayuda a reducir los tiempos de compilación cuando se producen modificaciones parciales en el código, haciendo más escalable el proceso de compilación. También asegura que el código fuente de implementación de otros módulos no tiene que estar disponible en el momento de la compilación. Esta propiedad facilita la construcción de bibliotecas de código precompiladas y fomenta la escalabilidad del proceso de compilación, que ayuda a construir programas más grandes y complejos en tiempos razonables.
- *Una unidad de carga.* Los módulos compilados por separado pueden cargarse en memoria independientemente. La carga dinámica reduce los tamaños de los programas en memoria evitando cargar módulos no usados, compartiendo el código de un módulo entre múltiples programas, y seleccionando el módulo más apropiado para cargar en un momento dado.

5.5.1 Módulos y clases

Los lenguajes modulares basados en tipos de datos abstractos o ADT (*Abstract Data Types*), como Modula-2 o Ada-83, utilizan módulos para implementar tipos de datos, agrupando operaciones asociadas a varias definiciones de datos en un mismo módulo. Usando este esquema, varios lenguajes orientados a objetos extienden el concepto de clase para incluir el concepto de módulo. Eiffel, Smalltalk-80 y parcialmente Java son algunos ejemplos. A fin de cuentas, una clase es una abstracción más fuerte que un ADT, puesto que engloba en una única construcción del lenguaje tres elementos antes independientes: definición de un tipo de datos, sus operaciones asociadas y la agrupación de ambas en una abstracción —la clase— que hace precisamente el papel de un módulo. La reducción del número de abstracciones en el lenguaje es la razón para mezclar clases y módulos. La clase es entonces la encargada de efectuar la importación de los símbolos que necesite [Meyer, 1997; pp. 210-211].

[Szyperski, 1992; pp. 22-23] destaca que disponer de una sola abstracción para múltiples propósitos también tiene sus inconvenientes. Por ejemplo, en un lenguaje sin módulos una biblioteca de funciones estadísticas debe definirse en una clase, independientemente de que la clase no tenga ningún otro significado práctico salvo agrupar un conjunto de métodos. Java usa la construcción `static final` para identificar métodos de clase que no son parte de los objetos. El acceso a las operaciones matemáticas se puede conseguir por dos vías. (1) Heredando de la clase que define las funciones estadísticas, que es denominado en Eiffel *facility inheritance* [Meyer, 1997; pp. 832-833]. O (2) creando un objeto de esa clase y llamando a la operación adecuada, que no está relacionada con el objeto creado. Szyperski llama a ambas opciones pseudo-herencia y pseudo-reenvío, porque son empleadas para simular módulos y nada tienen que ver con el comportamiento usual de las clases. Una abstracción independiente de módulos sirve mejor para esos propósitos.

[Szyperski, 1992; p. 24] resalta también cómo, para hacer frente a la falta de módulos, Eiffel realiza la importación automática de todos los nombres de clases del sistema. Es posible tener acceso a cualquier otra clase en el sistema, bien usando herencia bien declarando una variable cuyo tipo sea la clase deseada [Meyer, 1997; p. 210]. Lo mismo ocurre con Smalltalk-80. En C o C++ no existe el concepto de módulo, usando el preprocesador, el sistema de ficheros y la compilación separada para simular los módulos.

Otro problema identificado en [Szyperski, 1992; p. 25] es el relacionado con el alto acoplamiento entre varias clases que resuelven una abstracción común. Un ejemplo típico es la implementación de una lista enlazada y la implementación de cada nodo de la lista. Las funciones asociadas a la lista deben tener conocimiento de la implementación del nodo para producir una implementación eficiente. C++ usa la cláusula `friend` para esta labor. En Eiffel se usa el mecanismo de exportaciones selectivas en clases [Meyer, 1997; pp. 605-607]. En ambos casos es posible que cualquier clase obtenga acceso

privilegiado potencialmente de cualquier otra, sin que exista ningún tipo de estructura. Szyperski lo denomina *spaghetti-scoping*. Para resolver estos inconvenientes introduce la *regla de eliminación de paranoia* para módulos [Szyperski, 1992; p. 20], en el que se da acceso privilegiado a aquellas definiciones que formen parte de una misma implementación de módulo. Así, el ámbito que definen los módulos sirve también para agrupar implementaciones altamente acopladas. Estas razones hacen conveniente introducir el concepto de módulo separado y ortogonal a la orientación a objetos del lenguaje.

5.5.2 Otros lenguajes modulares

El concepto de módulos aparece por primera vez en [Parnas, 1972; pp. 1056-1057]. Los lenguajes con módulos más conocidos son Modula-2 [Wirth, 1982] y sus descendientes Oberon [Wirth, 1988], Oberon-2 [Moessenseeck & Wirth, 1991], Modula-3 [Cardelli *et alii*, 1992] y Component Pascal [Oberon, 1997]. Ada-83 y Ada-95 [Barnes, 1994] son otros dos ejemplos relevantes, donde reciben el nombre de paquetes o *packages*. Una introducción histórica se puede encontrar en [Szyperski, 1992; pp. 26-27]. La siguiente tabla resume características relevantes de algunos lenguajes conocidos con soporte de módulos y el nuevo lenguaje propuesto.

	Modula-2	Oberon	Modula-3	Ada 95	Component Pascal	Java	C#	XC
Interfaz e implementación	Separado	Unido	Separado	Separado	Unido	Unido	Unido	Separado
Módulos jerárquicos	Sí	No	No	Sí	No	Sí	Sí	Sí
Módulos internos	Sí	No	No	Sí	No	No	No (namespaces)	Sí
Módulos cerrados	Sí	Sí	Sí	No (hijos)	Sí	No (paquete)	No (namespaces)	No (hijos)
Funciones y procedimientos	Sí	Sí	Sí	Sí	Sí	No (static final, static)	No (const, static)	Sí
Ciclos de importación	Sí	No	No	Sí	No	Sí	Sí	Sí
Módulos genéricos	No	No	Sí	Sí	No	No	No	No (prototipos)
Módulos por partes	No	No	No	Sí	No	Sí	Sí	No

Tabla 5-3 Características modulares de XC junto a otros lenguajes con módulos conocidos.

A continuación se da una breve explicación de las características:

- *Interfaz e implementación*. Indica si existe una construcción separada en el lenguaje para identificar interfaces e implementación de módulos. La separación hace viable la compilación de módulos clientes con una interfaz de un proveedor sin que exista una implementación del proveedor.
- *Módulos jerárquicos*. Los módulos se organizan en una jerarquía, al igual que el espacio de nombres. Facilita la organización de grandes subsistemas en módulos relacionados por la jerarquía.
- *Módulos internos*. Son módulos definidos dentro de otros módulos. Ayuda en el ajuste del espacio de nombres en módulos complejos y en la agrupación de funcionalidad relacionada. En C# la cláusula *namespace* tiene una función similar, aunque no define específicamente módulos.
- *Módulos cerrados*. En los módulos cerrados no es posible añadir información adicional al espacio de nombres del módulo. En los módulos abiertos sí. Los módulos abiertos son más

flexibles, porque los módulos siguen siendo ampliables en el futuro sin necesidad de recompilación, aunque corren el riesgo de potenciales colisiones de nombres. Los paquetes de Java son abiertos porque nuevas clases pueden añadirse a cada paquete. Los espacios de nombres de C# son abiertos porque son válidas múltiples declaraciones del mismo espacio de nombres. En ambos casos puede ocurrir la colisión. La variante elegida por Ada 95 no corre el riesgo de colisión porque usa un paquete hijo separado, donde su implementación tiene acceso a la parte privada de la implementación de su padre. La opción elegida por XC es ligeramente diferente a la de Ada 95: un módulo puede definir un protocolo protegido con más información únicamente para sus módulos hijos.

- *Funciones y procedimientos.* Indica si un módulo puede tener definiciones de funciones y procedimientos para crear bibliotecas procedurales como bibliotecas matemáticas o de entrada/salida. Java y C# obtienen el mismo efecto usando los modificadores de miembros de clase `static final` y `const` respectivamente para constantes, y `static` para métodos en ambos lenguajes.
- *Ciclos de importación.* Un ciclo de importación ocurre si un módulo A importa otro B y éste a su vez importa de nuevo el módulo A. Los ciclos ayudan a expresar declaraciones dependientes entre sí distribuidas entre varios módulos, también llamadas declaraciones recursivas. Si los ciclos no son posibles, entonces la única manera para crear declaraciones recursivas es agrupándolas en un módulo común, que impide una separación modular adecuada, mezclando conceptos distintos únicamente porque tienen declaraciones relacionadas.
- *Módulos genéricos.* Los módulos genéricos parametrizan los tipos de datos asociados a un módulo. Los módulos pueden luego ser instanciados, indicando en ese momento qué tipos exactos son utilizados. Muchos módulos genéricos son usados para crear tipos de datos abstractos parametrizados. XC obtiene muchas veces un resultado similar con tipos genéricos aplicados a protocolos y prototipos.
- *Módulos por partes.* Es la posibilidad de definir un mismo módulo en diferentes partes. En Ada se usa la palabra clave `separate` para indicar qué partes del módulo se encuentran descritas en otros ficheros fuente. En Java, la declaración del paquete al que pertenece cada clase es en realidad una definición parcial del paquete. En C# la declaración de un espacio de nombres es también una definición parcial del módulo.

5.5.3 La estructura jerárquica

En XC el nombre completo de un símbolo o *nombre calificado* incluye el nombre del módulo en el que se ha declarado o definido el símbolo. Se pueden declarar y definir módulos dentro de otros módulos. En estos casos, el nombre calificado del símbolo incluye el nombre de todos los módulos en los que se encuentra incluido. Por ejemplo:

```
xc.collections.List
```

representa el símbolo `List`, definido en el módulo `Lists` que se localiza dentro del módulo `Collections`, que a su vez es parte del módulo `xc`.

5.5.3.1 Tipos de módulos

Existen tres tipos de módulos: *módulos principales*, *módulos secundarios* y *módulos internos*. La localización de un módulo en la estructura jerárquica sintáctica define una clasificación relativa entre ellos.

- ❖ **Def. 5-18.** Un *submódulo* o *módulo hijo* es un módulo secundario o un módulo interno.
- ❖ **Def. 5-19.** Un *supermódulo* o *módulo padre* es un módulo principal, secundario o interno que tiene submódulos.
- ❖ **Def. 5-20.** Dos módulos secundarios o internos son *hermanos* si tienen el mismo supermódulo.

La relación jerárquica entre módulos es importante no sólo por la organización estructural, sino porque los submódulos pueden tener acceso privilegiado a declaraciones protegidas de su supermódulo. La jerarquía forma un árbol con múltiples nodos raíz. Cada nodo raíz es un módulo principal.

- ❖ **Def. 5-21.** Un *módulo principal* describe un espacio de nombres completamente nuevo dentro del que se pueden declarar y definir nuevos módulos y símbolos.

Los módulos principales corresponden siempre con el primer nombre de módulo del nombre calificado de un símbolo. El siguiente código muestra una definición del módulo principal xc.

```
module xc
{
    // Definiciones del módulo principal
}
```

- ❖ **Def. 5-22.** Un *módulo secundario* describe un espacio de nombres dentro de un módulo principal u otro submódulo para declarar y definir nuevos módulos y símbolos.

El siguiente código representa una definición del módulo secundario xc.collections.

```
module xc.collections
{
    // Definiciones del módulo secundario
}
```

- ❖ **Def. 5-23.** Un *módulo interno* es un módulo secundario que no tiene una declaración y una definición independiente del módulo que lo contiene y, por tanto, no puede accederse directamente, sino siempre a través del módulo principal o secundario contenedor.

Como los módulos secundarios, estos módulos también describen un espacio de nombres dentro de un módulo principal u otro submódulo. Por ejemplo, el siguiente código muestra una definición del módulo interno xc.System.Memory dentro del módulo secundario xc.System.

```
module xc.system
{
    module Memory
    {
        // Definiciones del módulo interno
    }

    // Otras definiciones del módulo secundario
}
```

Es opcional publicar los módulos internos en una declaración. Aportan los beneficios del uso de módulos a bloques de código dentro de un módulo.

5.5.4 Estructura de un módulo

Los módulos separan y agrupan con claridad las declaraciones y definiciones del lenguaje. Las declaraciones describen los símbolos exportados hacia otros módulos y las definiciones cómo se implementan los símbolos declarados. Se pueden distinguir tres tipos de elementos declarables o definibles:

- Elementos que se consideran exportables y deben ser accesibles a otros módulos.
- Elementos que deben exportarse selectivamente sólo a algunos módulos.
- Elementos estrictamente privados.

5.5.4.1 Declaraciones de módulos

Las declaraciones de módulos revelan el protocolo de un módulo hacia otros módulos. Sólo aquellos símbolos visibles en un protocolo de módulo son accesibles por otros módulos. Dentro de una

declaración de módulo se pueden definir protocolos de objetos, y declarar prototipos, funciones, constantes y enumerados.

El nivel de acceso a los símbolos exportados por un módulo se organiza en tres grupos principales de protocolos de módulo: *públicos*, *protegidos* y *privados*. Todos los símbolos definidos dentro de un mismo nivel de acceso tienen privilegios de acceso de igual a igual entre ellos, cumpliendo la regla de la eliminación de paranoia de [Szyperski, 1992; p. 20].

- ❖ **Def. 5-24.** La *regla de eliminación de paranoia* da acceso privilegiado a los elementos que forman parte de la implementación de un mismo módulo.

Esta regla es importante para facilitar la interacción eficiente entre prototipos relacionados. El módulo sirve para limitar con naturalidad el ámbito de las definiciones privadas, y evitar la asignación *ad-hoc* de privilegios de acceso entre diferentes prototipos. Es lo que ocurre con la cláusula `friend` de C++ o las exportaciones selectivas de Eiffel, que da lugar al *spaghetti-scoping* y hace más confuso el razonamiento acerca del ámbito de una declaración o definición.

- ❖ **Def. 5-25.** Una *declaración pública* de módulo describe los símbolos exportados hacia otros módulos.

La declaración pública precisa siempre el modificador `public`.

```
public module protocol xc.collections
{
    // Declaraciones públicas
}
```

Todas las declaraciones de un protocolo público son públicas, no pudiéndose declarar dentro de su ámbito declaraciones protegidas.

- ❖ **Def. 5-26.** Una *declaración protegida* de módulo describe los símbolos exportados hacia módulos secundarios y módulos internos.

Todas las declaraciones de un protocolo de módulo protegido son protegidas. Por conveniencia, el protocolo público del módulo es importado automáticamente por el protocolo protegido, teniendo preferencia la declaración protegida sobre una declaración pública. Una declaración protegida usa el modificador `protected`.

```
protected module protocol xc.collections
{
    // Declaraciones protegidas
}
```

Los protocolos protegidos sirven para facilitar la extensión posterior controlada de un módulo usando módulos secundarios. La extensión de módulos es una situación habitual en el desarrollo. Por ejemplo, un módulo de colecciones puede querer extenderse con nuevos protocolos de colecciones especializadas. Si los módulos son cerrados y no existe ninguna vía para extender un módulo, la extensión debe hacerse modificando el módulo original, que obliga a recomilar a todos los clientes, no importando si las modificaciones son exclusivamente aditivas.

En Ada 95, los paquetes hijos sirven para este propósito [Barnes, 1994; pp. 432-439]. Las declaraciones y definiciones de un paquete hijo son una extensión de las existentes en su paquete padre. La idea intuitiva es que son una concatenación al final de las declaraciones y definiciones del padre. Por motivos de eficiencia, la implementación del paquete hijo que hace la extensión obtiene acceso a la parte privada del paquete padre. El riesgo de esta opción es que la implementación del hijo adquiere dependencias de aspectos privados del módulo o paquete padre, que puede ser un problema serio si otros desarrolladores pueden hacer una extensión arbitraria a un módulo de un proveedor sin conocimiento de éste. En este caso, las dependencias obtenidas son implícitas, y el módulo padre pierde libertad para cambiar su implementación. En Java ocurre algo similar: al ser posible añadir nuevas clases de terceros a un mismo paquete, se adquiere automáticamente acceso privilegiado a la información con ámbito de paquete de las clases del mismo, dando lugar a interdependencias

implícitas no controladas entre el nuevo código y el código original del paquete. Para evitar esos extremos, XC opta en cambio por requerir un protocolo protegido, accesible únicamente desde submódulos de un módulo dado, que hace explícitas las dependencias entre un módulo y sus submódulos. Este protocolo puede incluir declaraciones adicionales para facilitar implementaciones más eficientes o con mayores privilegios, pero no lo suficiente como para comprometer la implementación privada del módulo original. Una redeclaración protegida de una declaración pública usa el modificador `redefine`, para asegurar que la modificación de la declaración es intencional. Por ejemplo, el protocolo público de `xc.Runtime` declara el prototipo `RuntimeObject` como `final` y `confined`, para que otros módulos no puedan heredar de él o añadirle categorías. Tampoco publica información de herencia.

```
public module protocol XC.Runtime
{
    // ...
    final confined prototype RuntimeObject;
}

protected module protocol XC.Runtime
{
    // ...
    redefine confined prototype RuntimeObject extends: Prototype;
}
```

El protocolo protegido de `xc.Runtime` redefine el mismo prototipo habilitando la herencia e indicando de qué otro objeto hereda la implementación, marcando con `redefine` que la modificación es premeditada.

- ❖ **Def. 5-27.** Una *declaración privada* de módulo es implícita a la definición de la implementación de un módulo.

Una declaración privada de módulo describe los símbolos accesibles únicamente hacia módulos internos.

```
module xc.collections
{
    // Declaraciones y definiciones privadas
}
```

Los protocolos públicos y protegidos de módulo son importados automáticamente por la implementación de un módulo.

- ❖ **Def. 5-28.** Una *definición de módulo* describe la implementación del módulo. Es estrictamente privada y sus símbolos son accesibles únicamente a través de sus declaraciones.

Pueden existir varias declaraciones de módulos, pero sólo existe una definición o implementación de módulo. Un ejemplo de definición de módulo se ha visto durante la declaración de un protocolo privado. En la medida que una declaración de módulo no varíe, su implementación puede variar y compilarse de forma independiente sin que precise la compilación de otros módulos clientes.

5.5.5 Importación de módulos

Tanto las declaraciones como las definiciones de módulos pueden importar otras declaraciones de módulos. Un submódulo no importa automáticamente ninguna declaración de sus supermódulos. Deben especificarse explícitamente con la palabra clave `import`. El nombre del módulo que referencian debe ser un módulo principal o un módulo secundario. Las sentencias de importación aparecen al principio del cuerpo de un protocolo de módulo o de una implementación de módulo, antes que cualquier otra declaración o definición. Por ejemplo, la declaración:

```
public module protocol xc.collections
{
    import xc;
```

```
// Declaraciones públicas del módulo secundario
}
```

importa el protocolo público del módulo principal `xc`. Como el protocolo público de `xc.collections` importa el protocolo público de `xc`, cualquier otro módulo que importe el protocolo público de `xc.collections` importará automáticamente el protocolo público de `xc`. Esta propiedad se llama *transitividad*, que se revisa en el apartado 5.5.5.1²⁴.

Una implementación quizás tenga una lista de importación diferente de su declaración. En el siguiente código, la implementación del módulo `xc.collections` accede, además, al protocolo protegido de su supermódulo `xc` y al protocolo público de su módulo hermano `XC.System`:

```
module xc.collections
{
    protected import xc;
    import XC.System;

    protocol ListInfo
    { // ... }

    // otras declaraciones y definiciones del módulo secundario
}
```

La importación protegida del módulo `xc` es posible porque `xc.collections` es un submódulo de él. En cambio, `xc.Collections` sólo es capaz de acceder al protocolo público de `xc.System`, porque no es un submódulo de él. Un módulo siempre importa automáticamente la declaración pública y protegida de sí mismo implícitamente. En el ejemplo anterior `xc.collections` importa, además, sus propios protocolos público y protegido, si existen. En este caso, no se ha definido protocolo protegido.

Un módulo interno no se puede importar explícitamente, debe importarse su módulo padre. En el ejemplo siguiente se declara un protocolo público del módulo interno `xc.System.Console`:

```
public module protocol xc.system
{
    // otras declaraciones del módulo secundario

    public module protocol Console
    {
        void print: String text;

        // Otras declaraciones del módulo interno
    }
}
```

El módulo interno incluye la declaración del procedimiento `print:`. Un código cliente no puede importar el módulo `xc.System.Console` directamente, porque es un módulo interno. Únicamente puede importar el módulo `xc.system`. Eso es debido a que probablemente el módulo `xc.System.Console` dependa de información declarada en `xc.system`. Las reglas de calificación de símbolos se aplican por igual. A continuación se muestran unos breves ejemplos:

```
import xc.System;           // ok. Módulo secundario importado.
import xc.System.Console;  // error! El módulo interno no se puede importar.

// ...

xc.System.Console.print: "Texto"; // ok. Llamada usando calificador global del módulo
                                // importado.
Console.print: "Texto";        // ok. Llamada usando calificador local al módulo
                                // importado.
print: "Texto";              // error! Símbolo no encontrado.
```

Por último, se deben observar las siguientes restricciones durante la importación:

- Una declaración pública sólo puede importar declaraciones públicas.

²⁴ El protocolo del módulo `xc` importa a su vez el protocolo del módulo `xc.Lang`, por lo que los protocolos y prototipos básicos `xc.Lang.Prototype`, `xc.Lang.Object` y sus variantes están siempre disponibles sin necesidad de usar el calificador de módulo.

- Una declaración protegida es capaz de importar declaraciones públicas, y declaraciones protegidas de sus supermódulos.
- Una definición puede importar declaraciones públicas, y declaraciones protegidas de sus supermódulos.

5.5.5.1 Transitividad

La importación de módulos es *transitiva*: si un módulo C importa un módulo B y éste importa un módulo A, entonces el módulo C importa el módulo A automáticamente. La importación transitiva es una propiedad natural, sólo es necesario importar el módulo principal del que se depende. La transitividad puede dar lugar a intentos de importar un módulo más de una vez. El compilador detecta esos casos y solamente importa los módulos una vez.

La transitividad de la importación de módulos puede producir un problema sutil cuando se realizan cambios en los módulos importados. Si un módulo importado deja de importar otros módulos —por ejemplo si el módulo B deja de importar A— entonces C podría dejar de ser compilable si C usa en su declaración pública elementos declarados en A que no eran usados en B. Es más, puesto que el módulo es la unidad de carga, es posible que se intente cargar un módulo B nuevo sobre un módulo C antiguo. El soporte en tiempo de ejecución almacena las *dependencias transitivas* de módulos importados —los módulos directa e indirectamente importados— y comprueba que todos los módulos importados se encuentran cargados antes de activar el nuevo módulo. En este caso, el soporte en tiempo de ejecución detectará que C ya no se puede cargar. Los módulos prevén también un número de versión de módulo comprobado por el soporte en tiempo de ejecución. La forma exacta en que se debe suministrar un número de versión a un módulo durante el proceso de compilación está todavía en estudio.

5.5.6 Las reglas de ámbito

Las reglas de ámbito gobiernan la resolución de nombres que realiza el compilador para descubrir si un símbolo está definido o no. Estas reglas son también usadas para resolver símbolos que no están completamente calificados.

Las siguientes construcciones generan nuevos ámbitos para la declaración y definición de nombres. En todos los casos, un nuevo ámbito se encuentra identificado con una apertura y cierre de llaves '{' y '}'. Los ámbitos implican precedencia a la hora de seleccionar un símbolo. Por orden de prioridad de resolución, de mayor a menor prioridad:

- (1) *Definición de un bloque local*. Son declaraciones y definiciones privadas de variables y constantes dentro del cuerpo de un bloque local, interno a definición de una función, procedimiento o método.
- (2) *Definición de funciones y procedimientos*. Son declaraciones y definiciones privadas de variables y constantes en el cuerpo de una función o procedimiento.
- (2) *Definición de métodos*. Son declaraciones y definiciones privadas de variables y constantes en el cuerpo de un método.
- (3) *Definición de prototipos*. Son definiciones privadas de métodos en un prototipo dentro de la implementación de un módulo.
- (3) *Declaración y definición privada de protocolos, constantes y enumerados*. Son declaraciones y definiciones privadas en una implementación de un módulo.
- (4) *Definición protegida de protocolos, constantes y enumerados*. Son definiciones en el protocolo protegido de módulo asociado con una implementación de módulo.
- (5) *Declaración protegida de funciones, procedimientos, protocolos, prototipos, constantes y enumerados*. Son declaraciones en el protocolo protegido de módulo asociado con una implementación de módulo.

- (6) *Definición pública de protocolos, constantes y enumerados*. Son definiciones en el protocolo público de módulo asociado con una implementación de módulo.
- (7) *Declaración pública de funciones, procedimientos, protocolos, prototipos, constantes y enumerados*. Son declaraciones en el protocolo público de módulo asociado con una implementación de módulo.
- (8) *Definición protegida de protocolos, constantes y enumerados en un módulo importado*. Son definiciones en el protocolo protegido de un módulo importado por un protocolo protegido de módulo o una implementación de módulo.
- (9) *Declaración protegida de funciones, procedimientos, protocolos, prototipos, constantes y enumerados en un módulo importado*. Son declaraciones en el protocolo protegido de un módulo importado por un protocolo público o protegido de módulo, o una implementación de módulo.
- (10) *Definición pública de protocolos, constantes y enumerados en un módulo importado*. Son definiciones en el protocolo público de un módulo importado por un protocolo protegido de módulo o una implementación de módulo.
- (11) *Declaración pública de funciones, procedimientos, protocolos, prototipos, constantes y enumerados en un módulo importado*. Son declaraciones en el protocolo público de un módulo importado por un protocolo público o protegido de módulo, o una implementación de módulo.

El orden de preferencia de declaraciones entre varios módulos importados, es el orden en el código fuente de las sentencias de importación. Es usado por el compilador para identificar cualquier tipo de error en las declaraciones.

5.5.7 Módulos como unidad de carga

El módulo, al ser la unidad de compilación, es también la unidad apropiada de carga. Si la carga se hiciese por cada definición de prototipo, función o procedimiento, sería necesario gestionar con detalle cualquier dependencia de un elemento respecto de cualquier otro. Además, puesto que las constantes y los enumerados pueden ser definidos fuera de prototipos, funciones y procedimientos, tendrían que tenerse en cuenta también. Puesto que el módulo ha de resolver todas las dependencias con otros módulos durante la compilación, es posible aprovechar la información que ha de comprobar el compilador para calcular las dependencias de la carga de los módulos. Esas dependencias son las listas de importación de otros módulos. El compilador especifica el conjunto de dependencias de carga de un módulo usando el cierre transitivo de las listas de importación. Esta lista es registrada durante la compilación y almacenada para ser comprobada en tiempo de carga del módulo. El módulo se convierte entonces en una opción intuitiva para obviar las relaciones con alto acoplamiento entre los elementos declarados o definidos en los módulos. La regla de eliminación de paranoíta fomenta que los elementos acoplados se incluyan en un mismo módulo, garantizando que se cargarán siempre a la vez. También simplifica la gestión de nuevas versiones del código, pues los elementos fuertemente acoplados se actualizan todos a la vez en la siguiente versión del módulo. No es necesario llevar trazas de la versión de cada uno de los elementos particulares, ni éstos pueden encontrarse en distinta versión. El grado de simplificación es importante, porque son los elementos más acoplados los que probablemente sean más sensibles a cambios de versión.

La carga de un módulo puede producirse estáticamente al arrancar el programa o dinámicamente mediante petición cuando el programa se encuentra ya en ejecución. La carga estática y dinámica es similar, en ambos casos debe ejecutarse el código de inicialización del módulo. Cada módulo puede incluir una función de inicialización, llamada `main`, que es ejecutada cuando el módulo se carga.

```
module XC.System
{
    // Declaraciones privadas y definiciones de módulo.
    Integer main: String args[]
        { // ... }
```

{}

5.5.8 Modularidad monótona

Una característica importante del sistema de módulos se expresa en la quinta premisa (Def. 4-9) introducida en el apartado 4.3.4 *Monotonía*, donde se indica que una modificación monótona no debe obligar a recompilar módulos clientes. Siguiendo la cuarta premisa (Def. 3-15), si el código de un programa completo no está disponible, entonces cambios unilaterales monótonos en un módulo proveedor deben de funcionar igualmente con clientes compilados con una versión anterior del módulo proveedor. Cambios monótonos incluyen el añadido de nuevas funciones o procedimientos, nuevos mensajes a protocolos, nuevos métodos a prototipos, creación y adopción de nuevos protocolos, y nuevas variables de instancia de objetos. La estructura de los objetos creados en XC, que se estudia en el apartado 8.3 *La estructura de los objetos*, se encarga de preservar invariantes que mantienen la modularidad y admiten los cambios monótonos anteriores.

- ❖ **Def. 5-29.** La *modularidad monótona* es una propiedad que preserva la modularidad cuando se producen cambios monótonos en un módulo.

Preservar la modularidad monótona evita la recompilación de clientes en un subconjunto importante de modificaciones: aquellas que preservan el protocolo original del módulo proveedor. Por ejemplo, un cambio monótono visible mediante herencia es añadir una nueva variable, un nuevo mensaje, o un protocolo. Esos cambios no necesitan la recompilación de los prototipos derivados. Cambios no monótonos son el añadido o eliminación de parámetros a un mensaje, eliminación de mensajes, métodos, protocolos o prototipos. Estos cambios no preservan el protocolo original del módulo proveedor y, por tanto, requieren la recompilación de los clientes.

5.6 Conveniencias sintácticas

Para mejorar la consistencia del lenguaje, su sintaxis debe ayudar a expresar con naturalidad la intención de los algoritmos y el diseño. La audiencia esperada del lenguaje son desarrolladores con experiencia, probablemente versados en algún lenguaje de la familia de C: C o C++, y quizás Java, Objective-C o C#. No está diseñado para hacer programación exploratoria. Para resultar más natural a esa audiencia, la base de la sintaxis en XC proviene de C, incluyendo un amplio repertorio de operadores.

Como en C, las expresiones son un elemento esencial de XC. Sin embargo, se ha intentado reducir el número de expresiones que dan lugar a confusión. El operador ‘,’ se ha eliminado por tener una semántica más endeble que varias sentencias separadas. Las asignaciones son sentencias, en vez de expresiones, puesto que estas últimas son fuente de errores involuntarios y el código queda menos legible. Los operadores de manipulación de punteros ‘*’ y ‘&’ no existen. El núcleo del lenguaje XC usa objetos y expresiones de mensaje. El compilador identifica sintácticamente expresiones habituales de C y realiza la traducción correspondiente a objetos y mensajes. Los mensajes usan sintaxis inspirada en Smalltalk-80. La sobrecarga de operadores de C++ se ha limitado. En cambio, se han fomentado las expresiones estructuradas, obtenidas esta vez de Javascript. El objetivo es describir concisamente los comportamientos comunes y con semántica conocida, y, a la vez, describir con más claridad los comportamientos particulares que ayuden a comprender con menor esfuerzo el código existente.

5.6.1 Expresiones y protocolos de operador

Los operadores son interesantes porque facilitan la descripción concisa de muchas operaciones comunes, pero siempre que su semántica esté definida. Una fuente de confusión en C++ es el uso indiscriminado de la sobrecarga de operador para significados muy particulares. El resultado es que

los operadores, en vez de facilitar la comprensión de código, la dificultan. Para reducir el problema semántico, en XC los operadores habituales de C se implementan usando mensajes agrupados en protocolos. Los tipos básicos como los tipos integrales adoptan los protocolos asociados a los operadores.

- ❖ **Def. 5-30.** Un *protocolo de operador* identifica un conjunto de mensajes asociados con operadores con una semántica conocida.

Por ejemplo, el protocolo `Identity` es un protocolo de operadores que identifica igualdad e identidad entre objetos²⁵.

```
protocol Identity
{
    Boolean isEqualTo: self value;
    Boolean isNotEqualTo: self value;
    Boolean isExactlyEqual: self value;
    Boolean isNotExactlyEqual: self value;
    Unsigned hash;
}
```

Se obtiene un resultado similar a la sobrecarga de operador identificando operadores en expresiones con los mensajes que forman parte de los protocolos de operadores. Por ejemplo, en la expresión que involucra el operador ‘==’:

```
Unsigned a = 2, b = 3;
Boolean result = (a == b);
```

el compilador identifica que el protocolo `Unsigned` adopta el protocolo `Identity`, accediendo a traducir el operador ‘==’ por el mensaje `isEqual:to:`, que es enviado a la variable `a` con el argumento `b`. La intención es que los operadores se implementen por grupos, para que la semántica asociada a los operadores sea más consistente, al pertenecer siempre cada operador a un mismo protocolo. Los protocolos de operadores se listan en la Tabla 5-4.

Protocolos				
Arithmethic	Identity	Relational	Logical	Binary
+ add:	== isEqualTo:	> isGreaterThan:	! not	~ bitNeg:
- sub:	!= isNotEqualTo:	< isLessThan:	&& and:	^ bitXor:
++ inc:	==> isExactlyEqual:	>= isGreaterThanOrEqual:	or:	& bitAnd:
-- dec:	!=> isNotExactlyEqual:	<= isLessThanOrEqual:		bitOr:
* mul:	Concatenation	DictionaryAccess	IndexedAccess	>> rightshift:
/ div:	+ add:	{ } atKey:	[] at:	<< leftshift:
% mod:		{ } atKey:put:	[] at:put:	

Tabla 5-4 Protocolos y mensajes asociados a operadores.

5.6.2 Expresiones estructuradas

Una expresión estructurada simplifica la construcción y manipulación de colecciones de datos comunes usando protocolos definidos y expresiones. Se inspiran en las expresiones estructuradas de Javascript.

5.6.2.1 Arrays

El lenguaje puede aplicar una expresión estructurada de `array` si un objeto adopta e implementa el protocolo `XC.Collections.IndexedCollection`, que adopta los protocolos de operadores `Identity`, `Concatenation` e `IndexedAccess`. Cuando se crean objetos con expresiones de `array`, el

²⁵ Nótese que el tipo de los argumentos de los mensajes binarios es `Self`, por lo que se especializa correctamente para cada prototipo que adopte `Identity` (véase el apartado 7.4.3 *El tipo covariante Self*).

lenguaje usa el prototipo `xc.collections.Arrays.Array`, o alguna de las especializaciones predefinidas para tipos integrales si puede identificarlas por el tipo del contenido del *array*. Los *arrays* pueden ser de tamaño estático si se acompaña con un índice, o dinámicos, si no se especifica.

```
Object array[]; // Array dinámico. El prototipo es XC.Collections.Arrays.Array
Unsigned uarray[2]; // Array estático de longitud 2. El prototipo es
// XC.Collections.Arrays.UnsignedArray
```

Como la implementación de *arrays* está realizada completamente dentro del lenguaje, la sintaxis anterior existe únicamente por conveniencia. Para cada declaración de *array*, el compilador literalmente compila el equivalente a las siguientes sentencias cuando crea los *arrays* anteriores.

```
XC.Collections.Arrays.Array array = XC.Collections.Arrays.Array.clone;
XC.Collections.Arrays.UnsignedArray =
(XC.Collections.Arrays.UnsignedArray.allocsize: 2).initWith: 0;
```

Una expresión estructurada de *array* define un *array* explícitamente usando corchetes y valores separados por comas. Cada elemento del *array* se inicializa enviando el mensaje `at:put:` al *array* creado de forma equivalente al ejemplo anterior. En este caso se trata de un *array* polimórfico.

```
Object array[] = [ "text", 2, file.name ];
```

Se admite el uso de los operadores de identidad y concatenación con *arrays*. Por consistencia, el resto de los mensajes asociados a los *arrays* también son accesibles desde expresiones normales y estructuradas. El compilador traduce la expresión estructurada a los mensajes apropiados subyacentes. En el ejemplo siguiente, la variable *array* es un *array* polimórfico con cinco elementos, cuatro números y un *array* de cadenas de dos elementos. El resultado almacenado en `length` es ocho. El último elemento de la segunda sentencia es un *array* vacío, de longitud cero.

```
Object array[] = [ 1, 2, 3 ] + [ 4, ["a", "b"] ];
Unsigned length = array.length + [ 6, 7, 8 ].length + [].length;
```

Al traducir el compilador las expresiones estructuradas a mensajes, la misma sintaxis puede ser usada para objetos definidos por el usuario, siempre y cuando adopten el protocolo `IndexedCollection`. Por ejemplo, si `MyArray` implementa `IndexedCollection`, la sentencia siguiente:

```
MyArray myArray = [ "1", "2" ];
```

es compilada al siguiente código equivalente:

```
MyArray myArray = (MyArray.clone.at: 0 put: "1").at: 1 put: "2";
```

5.6.2.2 Diccionarios

Los diccionarios son un tipo particular de tabla *hash*, muy común para organizar información estructurada. La idea es la misma que con *arrays*. Una expresión estructurada de diccionario es aplicable si un objeto implementa el protocolo `xc.collections.HashTables.Dictionary` que adopta los protocolos de operadores `Identity`, `Concatenation` y `DictionaryAccess`. Un diccionario tiene como clave una cadena de texto y como valor cualquier objeto que adopte el protocolo `Object`. La implementación de diccionarios también está realizada completamente dentro del lenguaje, y se muestra como ejemplo en el Apéndice B. La sintaxis es una conveniencia para manejar este tipo de estructuras. Una expresión estructurada de diccionario define explícitamente claves y valores asociados separados por comas y entre llaves. Algunas expresiones estructuradas de diccionario son:

```
Dictionary dict = {} // Diccionario vacío
Dictionary dict2 = { key1 = "a", key2 = 7 }; // Diccionario con 2 elementos
```

El compilador compila el equivalente a las siguientes sentencias cuando crea los diccionarios anteriores:

```

XC.Collections.HashTables.Dictionary dict = XC.Collections.HashTables.Dictionary.clone;

XC.Collections.HashTables.Dictionary dict2 =
    (XC.Collections.HashTables.Dictionary.clone.atKey: "key1" put: "a").atKey: "key2"
        put: 7;

```

El acceso y escritura de entradas en un diccionario también tiene una sintaxis de ayuda. En el código siguiente, la variable `value` se asigna al valor “`a`” del diccionario `dict2` del ejemplo anterior, y se asigna una nueva entrada al diccionario con clave “`a1`” y valor un *array*:

```

String value = dict2{"key1"};
dict2{value + "1"} = [ 1, 2, 3 ];

```

Las expresiones estructuradas sirven para obtener estructuras complejas, dinámicas, legibles y compactas. Por ejemplo, el siguiente fragmento describe un hipotético descriptor de pruebas:

```

Dictionary testDescriptor = { name      = "Test 1",
                             version   = 3,
                             options   = { type     = "full",
                                         targets  = [ "filename1", "filename2" ],
                                         },
                             date      = XC.System.Clock.currentTimeMillis,
                           };

```

Al igual que con los *arrays*, al compilarse las expresiones estructuradas a mensajes, la misma sintaxis puede usarse para objetos definidos por el usuario, en este caso siempre y cuando adopten el protocolo `Dictionary`.

5.6.3 Constantes: objetos constantes y protocolos constantes

El significado de las constantes no es igual en todos los lenguajes. La siguiente definición aclara la interpretación que se les da en XC.

- ❖ **Def. 5-31.** Una *constante* es un valor de un tipo determinado que no cambia después de su inicialización.

En un programa existen dos variantes de constantes:

- *Valores constantes conocidos en tiempo de compilación.* Un valor así corresponde con un objeto constante, cuya inicialización es estática, y cuyo tamaño puede ser inferido en tiempo de compilación. Por ejemplo, las cadenas, los números enteros o los números en coma flotante. Un valor constante de este tipo puede ser publicado en un protocolo de módulo o puede permanecer su valor privado y ser publicado únicamente su símbolo.
- *Valores constantes parcialmente conocidos o no conocidos.* Estos valores corresponden con objetos constantes, quizás con inicialización estática, pero cuyo tamaño no puede ser inferido en tiempo de compilación. Cualquier objeto que derive de `Object` no tiene un tamaño conocido, puesto que `Object` puede ser extendido con nuevas variables de instancia y nuevas categorías. Algunos ejemplos son los diccionarios, los *arrays*, o los valores constantes conocidos en tiempo de compilación pero no publicados.

Si el valor es conocido durante la compilación, entonces es posible realizar simplificaciones en el cálculo de expresiones constantes. Si el valor es parcialmente conocido, no conocido, o no publicado y perteneciente a otro módulo, entonces el cálculo debe realizarse en tiempo de ejecución. En muchos casos, estos cálculos se realizan una sola vez, por lo que la penalización es despreciable. A cambio se obtiene ganancia en modularidad, porque el valor de la constante no se codifica en el código cliente y su valor puede variar en el futuro independientemente de sus clientes. En Java, por ejemplo, las constantes (campos `static final`) definidas en una clase o interfaz se copian en todas las clases e interfaces derivadas y sus clientes [Lindholm & Yellin, 1999; p. 52]. El cambio de una constante en la

clase o interfaz que la publica, obliga a recompilar todas las clases o interfaces que hereden de ella o que la usen.

Lo que debe determinar en última instancia si un objeto es constante es su protocolo. Si un protocolo permite la modificación de un tipo de datos, entonces su protocolo no es constante. El problema, desde el punto de vista del compilador, es que resulta tedioso para el programador asegurar si directa o indirectamente un objeto es modificado dentro del código de un método.

Un ejemplo de este comportamiento se encuentra en C++, donde la distinción entre objetos constantes y no constantes se asemeja más a la protección *hardware* de acceso de lectura y escritura [Stroustrup, 1994; p. 22]. Un objeto constante no puede ser modificado directa o indirectamente en el código de una función o método. Para asegurar este invariante, es necesario que todos los accesos, asignaciones y llamadas a función o métodos sean también constantes. El problema de esta aproximación es que fomenta la proliferación de código duplicado: si una rutina permite recibir 2 parámetros constantes o variables, es necesario incluir una rutina por cada combinación de ellos, es decir, cuatro. Además, si una rutina es llamada desde otra cuyo modificador es `const`, tal rutina ha que tener el modificador `const` también. Si esta rutina permite en otros casos el uso de parámetros no constantes, entonces debe duplicarse: una con el modificador `const` y otra no. En la práctica, escribir código C++ que siga estrictamente estas reglas de valores constantes es tedioso y es la razón por la que es a menudo obviado. El mantenimiento se multiplica por el número de versiones duplicadas de las funciones `const`, no `const`, con parámetros `const` y sin ellos.

Los problemas anteriores de proliferación de código duplicado hacen dudar de la conveniencia de una regla tan estricta sobre los valores constantes. Ahora bien, es posible también una solución intermedia. En vez de dejar al programador toda la responsabilidad sobre el manejo de objetos constantes, y que éste identifique objetos constantes o no implícitamente mediante identificadores —digamos `String` y `constString`— sí resulta interesante poder informar al compilador de que un protocolo es considerado constante. En este caso, el compilador puede hacer comprobaciones adicionales cuando se usan variables de protocolos que se han anotado como tales.

En XC, un protocolo es considerado constante por el compilador si se añade el modificador `const` delante del nombre del protocolo. La variante no constante del mismo debe heredar del protocolo constante, añadiendo los métodos necesarios para realizar modificaciones al objeto. La declaración de variables que sigan estos protocolos es intuitiva²⁶.

```
protocol const String
{
    Unsigned length;
    Char at: Unsigned index;
}

protocol String extends: const String
{
    Selfchain at: Unsigned index put: Char avalue;
}

// ...

const String cstr = "Constant string";
String      str = "Mutable string";
```

Con esa información, el compilador es capaz de comprobar que una variable constante no pueda aparecer en el lado izquierdo de una asignación, y que el protocolo usado por un parámetro `in` debe ser el protocolo constante.

Téngase en cuenta que nada impide al programador incorporar el mensaje `at:put:` en el protocolo constante, de la misma forma que nada impide que la implementación del método `at:put:` sea realmente la del método `at:` y no inserte ningún elemento. Es responsabilidad del programador

²⁶ La palabra clave `selfchain` se explica en el apartado 6.2.5 *Interoperabilidad*. Su tipo es el mismo que `self`, que corresponde con el tipo del protocolo que contiene el mensaje.

mantener la semántica de la declaración de protocolo que realiza. En la práctica, el programador debe hacer exactamente lo mismo si declara los protocolos `String` y `ConstString`, pero con la desventaja de que, en este último caso, el compilador no tiene información adicional que pueda ayudar a encontrar errores.

5.6.4 El nexo de unión con el soporte en tiempo de ejecución.

El soporte en tiempo de ejecución o *runtime* se estudia con más profundidad en el capítulo 8. En el aspecto sintáctico el acceso se hace principalmente con las construcciones usuales del lenguaje. El módulo `XC.Runtime`, también revisado en el capítulo 8, proporciona una interfaz orientada a objetos accesible desde XC. Sólo unas pocas expresiones tienen acceso directo al *runtime*.

5.6.4.1 Expresiones de módulo

Una expresión de módulo es un identificador de módulo escrito en una expresión. El identificador debe estar completamente calificado. Una expresión de módulo devuelve el tipo de datos `XC.Runtime.Module` y a partir de él es posible obtener toda la información de un módulo. Los únicos módulos accesibles en tiempo de compilación son aquellos que se hayan importado. El siguiente ejemplo devuelve el módulo donde está definido el protocolo `Module`.

```
XC.Runtime.Module module = XC.Runtime;
```

5.6.4.2 Expresiones de prototipo

Las expresiones de prototipo hacen referencia al ejemplar creado por defecto para cada prototipo. El ejemplar es accedido usando el nombre del prototipo, que puede estar completa o parcialmente especificado. El tipo de datos de la expresión de prototipo es el tipo de datos del ejemplar. Es utilizado habitualmente durante la creación de nuevos objetos. El resultado es muy similar a si el prototipo actuase como una clase en un lenguaje con clases. Como el nombre del prototipo hace referencia a un ejemplar, se puede clonar directamente.

```
XC.String string = XC.String.clone;
```

El ejemplar como tal no es accesible directamente mediante una expresión de prototipo, para evitar que inadvertidamente se asigne el ejemplar por defecto a una variable y modificarlo. El ejemplar por defecto representa la información de creación e inicialización conocida en tiempo de compilación para una familia dada.

```
XC.String stringExemplar = XC.String; // error!
```

Si se desea otro ejemplar por defecto, es suficiente con clonarlo y asignarlo a alguna variable conocida por otros módulos.

5.6.4.3 Expresiones de selectores

Un selector codifica la información del mensaje asociado a un método. Está disponible en todos los métodos automáticamente. El protocolo de un selector es `XC.Runtime.Selector`. Es accesible desde dentro de un método usando la pseudo-variable `selector`:

```
Console.println: "Message name = " + selector.messageName + ", id = " + selector.id;
```

5.7 Recapitulación

Se ha presentado en este capítulo una introducción a la sintaxis de XC, deteniéndose en aquellos aspectos más interesantes. En primer lugar se ha repasado el conjunto de abstracciones principales del lenguaje, novedoso en su conjunto, que permite describir programas con mayor expresividad, modularidad y flexibilidad que los lenguajes existentes que mantienen control estricto de tipos. El modelo de objetos, la separación entre interfaz e implementación, las extensiones por herencia y los módulos se asientan en conocimientos previos de este campo. Las extensiones ortogonales mediante categorías con comprobación estricta de tipos y linearización explícita, es una técnica original de este trabajo. El tratamiento de metamétodos sin nuevos árboles de herencia paralelos para metaobjetos es otra propuesta nueva, lo mismo que las palabras clave que califican las variantes de objetos prototipo. La propuesta de protocolos protegidos de módulo, usados para facilitar la extensión de módulos sin comprometer la modularidad, junto con los mecanismos de extensión modular con herencia y categorías y constantes, son un paso adelante en la descripción de sistemas más modulares pero a la vez más flexibles y extensibles. La aplicación de operadores y expresiones estructuradas utilizando protocolos es novedosa en lenguajes dedicados a la programación de sistemas. Combinan la conveniencia y expresividad de los operadores y expresiones estructuradas con una semántica definida, manteniendo intactas las abstracciones básicas del lenguaje.

6

El modelo de componentes

6.1 Introducción

Una vez presentado el modelo de objetos en el capítulo 4 y la estructura sintáctica en el capítulo 5, es momento de definir el modelo de componentes. El modelo está basado en gran parte en el modelo de objetos. Se ha puesto un énfasis especial en las características modulares a la hora de establecer el modelo de objetos, para simplificar en gran medida la definición del modelo de componentes. En primer lugar se revisan las características que comprenden un modelo de componentes, y se comparan con las características que el modelo de componentes de XC proporciona, resaltando la necesidad de interoperabilidad. A continuación, se comentan algunos ejemplos de creación de componentes usando las capacidades del lenguaje, que servirán para tener una imagen clara de cómo construir componentes con XC: la estructura de declaración e implementación, definición de contratos, y cómo realizar control de calidad.

6.2 Características del modelo de componentes

El apartado 2.2 *Definiciones preliminares* introdujo términos precisos para caracterizar en este trabajo el muchas veces vago concepto de componente (Def. 2-11). En este capítulo es posible dar una respuesta concreta a la implementación del modelo de componentes (Def. 2-14) de XC y, por extensión, a los componentes que maneja. Debe tenerse muy presente que XC no pretende construir un modelo de componentes que entre en conflicto con otros modelos de componentes existentes como COM, CORBA o EJB. Al contrario, el objetivo es que resulte más sencillo manipular componentes e interoperar con otros modelos de componentes directamente desde el lenguaje. La capacidad de interoperar es fundamental para que la programación basada en componentes tenga éxito, puesto que debe apoyarse en estándares independientes de los lenguajes [Baker, 2000; p. 611-612]²⁷.

Para facilitar esa labor, XC describe abstracciones para soportar modelos de componentes. No debe extrañar que el proceso de definición de los capítulos anteriores haya creado unas abstracciones que se puedan interpretar como un modelo de componentes por sí mismas. Las abstracciones modelan propiedades parecidas a las que se esperan en los modelos de componentes. Fomentar la separación crucial entre interfaz e implementación, envío de mensajes modular, metadatos en el soporte en

²⁷ No obstante, hay que hacer notar que no es objetivo del presente trabajo presentar una implementación de una integración del modelo de componentes de XC con modelos de componentes externos. Su objetivo se limita a proponer un diseño suficientemente flexible, adaptable a ellos en un futuro.

tiempo de ejecución, o módulos como unidad de entrega son las más relevantes. Pero el lenguaje no define qué protocolo exacto de comunicación debe usarse, cuál es la implementación de componentes distribuidos, cómo se describen recursos de componentes o sus formatos de configuración. Esas características deben ser parte de algún estándar, no de un lenguaje particular. El lenguaje es, por tanto, una herramienta para soportar y manipular modelos de componentes, que intenta preservar y fomentar su semántica en el desarrollo de componentes e incluso en los objetos básicos, realzando la visión de componentes por encima de la tradicional orientada a objetos.

Las definiciones dadas en este capítulo son compatibles con las encontradas en [Szyperski, 1998; pp. 30-31]. La Tabla 6-1 resume características importantes de un modelo de componentes y cómo XC aborda su definición y uso. Algunos de ellos no son propósito de la definición del lenguaje, o necesitan bibliotecas adicionales que no forman parte del lenguaje base objetivo de este trabajo. Posteriormente se evalúan las características con más detenimiento, siguiendo una organización inspirada en [Weinreich & Sametinger, 2001; pp. 38-45].

Elemento	Descripción	Abstracciones de XC usadas
Reglas de Composición	Interfaces y reglas para combinar componentes, para crear estructuras mayores, y para sustituir o añadir componentes a estructuras existentes.	Reglas de composición, protocolos, categorías de protocolos, prototipos, categorías de prototipos, herencia, módulos.
Interfaces y Contratos	Especificación del comportamiento y propiedades del componente, definición de la descripción de la interfaz o protocolo .	Protocolos, categorías de protocolos, categorías de prototipos.
Nombrado	Nombres únicos para interfaces y componentes.	Módulos jerárquicos, identificadores de protocolos.
Metadatos	Información acerca de componentes, interfaces y sus relaciones.	Información en tiempo de ejecución o <i>runtime</i> .
Interoperabilidad	Comunicación e intercambio de datos entre componentes de distintos proveedores, implementados en diferentes lenguajes.	Requiere bibliotecas adicionales de acceso a protocolos estándar. (<i>No es parte de la definición del lenguaje</i>).
Empaquetado y Despliegue	Empaquetada implementación y recursos para instalar y configurar un componente.	Módulos, carga dinámica de módulos. Precisa bibliotecas adicionales de gestión de recursos.
Configuración	Interfaces para configurar componentes.	Protocolos o categorías de protocolos proporcionados por el componente. (<i>No es parte de la definición del lenguaje</i>).
Evolución	Reglas y servicios para reemplazar componentes o interfaces por versiones más modernas.	Soporte de información de versión en prototipos, categorías y módulos. (<i>No es parte de la definición del lenguaje</i>).

Tabla 6-1 Elementos básicos de un modelo de componentes.

Adaptado y extendido de [Weinreich & Sametinger, 2001; p. 38].

6.2.1 Reglas de composición

Las reglas de composición del modelo de componentes de XC no indican una implementación determinada del modelo, pudiéndose aplicar también a otros modelos de componentes. Es decir, otros modelos son posibles siguiendo las mismas reglas de composición. Por ejemplo, CORBA (véase el apartado 2.5.3) no depende del lenguaje de programación, así que describe correspondencias de los objetos CORBA con diferentes lenguajes. CORBA establece también el formato de un mensaje y cómo se envía, para poder implementarlo en varios lenguajes. Además, define un IDL para indicar los protocolos de comunicación entre componentes y sus traducciones a diferentes lenguajes. Por último, el acceso a los objetos CORBA precisa la definición de esqueletos que adapten los accesos de cliente y referencias a objetos CORBA a la sintaxis particular de los lenguajes de programación soportados.

Es de esperar que las reglas de composición ayuden a hacer más sencillo una correspondencia de CORBA a XC y a la inversa, porque ambos soportan abstracciones similares. Lo mismo se puede presumir de COM+ (véase el apartado 2.5.1) por ejemplo. Contrastá esta situación con C++, donde

sólo las clases y objetos son usables directamente. El mecanismo de envío de mensajes, la implementación de las interfaces, la descripción de metadatos, o los esqueletos deben ser recreados desde cero.

Las reglas de composición del modelo de componentes son las siguientes:

- ❖ **Def. 6-1.** *Regla 1.* La imagen de un componente en tiempo de ejecución se describe mediante uno o varios objetos, que encapsulan el estado en tiempo de ejecución de la funcionalidad que lleva a cabo el componente.
- ❖ **Def. 6-2.** *Regla 2.* La interacción entre las imágenes en tiempo de ejecución de componentes se efectúa mediante el envío de mensajes.
- ❖ **Def. 6-3.** *Regla 3.* Los protocolos contienen conjuntos de mensajes que representan el comportamiento de los componentes, pero no su implementación.
- ❖ **Def. 6-4.** *Regla 4.* El acceso a la imagen en tiempo de ejecución de un componente se realiza siempre a través de protocolos.

El modelo de componentes de XC indica cuáles son los intervenientes en las reglas de composición y sus características. Para evitar duplicidad se hace referencia a los lugares donde se han introducido las mismas:

- Los objetos de la definición (Def. 6-1) que representan la imagen de un componente son los objetos descritos por el modelo de objetos del capítulo 4.
- El envío de mensajes de la definición (Def. 6-2) para la interacción entre imágenes de componentes en tiempo de ejecución usa la semántica de la técnica de selectores detallada en el apartado 4.4.3 *Selectores*.
- Los protocolos y los mensajes que representan el comportamiento de componentes de la definición (Def. 6-3) se especifican con la sintaxis de protocolos de objetos precisada en los apartados 5.2.1.2 *Protocolos*, 5.4 *Protocolos, prototipos y sus variantes* y 6.2.5 *Interoperabilidad*.
- Las reglas del sistema de tipos de XC que se estudian en el capítulo 7 garantizan que se cumple la definición (Def. 6-4).

El modelo de componentes de XC establece cuáles son las características y semántica del modelo siguiendo las reglas de composición. No impone una implementación fija en todos sus aspectos, asentándose en la especificación de un lenguaje para simplificar su definición. Obviamente el trabajo de especificación hay que llevarlo a cabo de todas formas. Al hacerlo durante la definición de un lenguaje se obtienen importantes beneficios adicionales. Se eliminan redundancias, se logra una sintaxis más clara y cercana a la definición de componentes, y se aplica el sistema de tipos del lenguaje para garantizar aspectos sintácticos, semánticos y de modularidad relacionados con los modelos de componentes. Aspectos éstos que habitualmente no son abordables con facilidad en lenguajes tradicionales por limitaciones inherentes a su diseño.

6.2.2 Interfaces o protocolos y contratos

La separación estricta que hace el lenguaje entre tipos e implementación y el envío de mensajes con selectores fomenta el reuso de caja negra, más modular. En cambio, la herencia fomenta el reuso de caja blanca, en el que detalles de implementación pueden ser apreciables en descendientes y causa de incompatibilidades futuras. Por esa razón, es probable que los protocolos de muchos componentes se decanten por dar únicamente un protocolo de cliente y no de especialización, aun cuando la implementación del componente sí puede estar usando herencia internamente para factorizar código. Un ejemplo de esta aproximación se estudia en el apartado 6.3.1 *Estructura de un componente*. El protocolo del componente, que es un contrato sintáctico mínimo entre el componente y sus clientes,

puede ser extendido con comprobaciones semánticas adicionales, como se ve en el apartado 6.3.2 *Control de calidad*.

6.2.3 Nombrado

La identificación de componentes con nombre único es importante para discriminar entre posibles componentes de distintos proveedores. La capacidad de definir jerarquías de módulos en XC permite usar un esquema de espacio de nombres similar a los paquetes de Java, usando las direcciones de los dominios de Internet. En caso de usar componentes externos, obtenidos de otros modelos de componentes como CORBA o COM+, la opción más plausible es usar directamente su propio sistema de nombrado, implementándolo en una biblioteca adicional.

6.2.4 Metadatos

Los metadatos son información acerca de los componentes, sus protocolos y sus relaciones con otros componentes. Esta información es recolectada por el compilador y almacenada dentro de cada módulo. El apartado 8.2 *El soporte en tiempo de ejecución* explica con más profundidad la información disponible. Los metadatos son útiles por diferentes razones. Herramientas de desarrollo son capaces de analizar más fácilmente la información disponible. Mediante metaprogramación es posible identificar qué elementos de un programa están cargados, y también hace viable realizar envíos de mensajes dinámicamente. Eiffel soporta el añadido de metadatos arbitrarios asociados a las clases con la cláusula `indexing`. C# usa la cláusula `attribute` para la misma función. El soporte de información adicional en XC con metadatos similares a los de Eiffel o C# se encuentra actualmente en estudio.

6.2.5 Interoperabilidad

Existen varios modelos de componentes, cada uno con sus ventajas e inconvenientes. Independientemente de su número, la necesidad de comunicar unos con otros es real. Incluso dentro de un mismo modelo de componentes, diferentes implementaciones deben poder comunicarse entre sí. Pero no siempre ha sido así. Las implementaciones de CORBA, por ejemplo, no fueron interoperables hasta la aparición del protocolo IIOP [Baker, 2000; p. 610]. Es decir, no sólo puede variar la implementación entre dos componentes de un mismo modelo, puede variar también la implementación del modelo de componentes. Facilitar la comunicación efectiva no ya entre componentes sino entre distintos modelos de componentes, requiere que el lenguaje no tome decisiones demasiado rígidas a la hora de definir su propio modelo. Es más, debe ser al contrario. El lenguaje debe facilitar la comunicación entre componentes de distintos modelos. Si las abstracciones que provee son adecuadas, entonces será posible implementar con menor esfuerzo los conectores entre distintos modelos de componentes. XC implementa de forma eficiente el envío de mensajes entre objetos de un mismo proceso. La técnica de envío de mensajes por construcción ya tiene la semántica apropiada en este caso, evitando a la vez tener que crear dos tipos de código de recubrimiento: (1) aquél que implemente una semántica más general de envío de mensajes basada en selectores como DII de CORBA, y (2) aquél que realice la correspondencia entre la semántica general en la semántica particular del modelo de objetos que use cada lenguaje, como los esqueletos de acceso desde cliente a objetos CORBA.

Entre componentes de distintos procesos la situación es diferente, pues harán falta varias implementaciones, que dependen del modelo de componentes con el que se quiera comunicar. Por esa razón, la implementación de componentes distribuidos no forma parte del lenguaje, sino que se delega en bibliotecas externas. Para facilitar la creación de componentes distribuidos—envíos de mensajes entre procesos y entre máquinas—XC provee facilidades para la codificación de los protocolos de envío. Las facilidades son generales, para que se pueda implementar sobre ellas los protocolos que

sean necesarios y no se dependa de una implementación particular. La conveniencia más importante son los modificadores de argumentos de mensajes, que sirven para anotar aspectos del comportamiento del protocolo. La Tabla 6-2 muestra las palabras clave usadas como modificadores.

Palabra clave	Descripción
in	El argumento del mensaje es sólo de entrada.
inout	El argumento del mensaje es de entrada y de salida.
out	El argumento del mensaje es sólo de salida.
oneway	El mensaje retorna inmediatamente. Usado para identificar mensajes asíncronos.
byval	El argumento del mensaje se pasa por valor.
byref	El argumento del mensaje se pasa por referencia.
selfchain	El mensaje devuelve el objeto receptor para facilitar el encadenado de mensajes.

Tabla 6-2 Palabras clave de soporte para llamadas entre procesos.

Un protocolo puede opcionalmente anotarse con los modificadores de la Tabla 6-2. Por ejemplo, un hipotético protocolo de control de un temporizador se define como sigue:

```
protocol Timer
{
    selfchain resetTimer;
    selfchain setTimer: byval Time time;
    byval Time watchTimer;
}
```

Deben observarse ciertos valores por defecto que simplifican las anotaciones²⁸:

- El modificador por defecto de los tipos predefinidos integrales como `Boolean`, `Unsigned`, `Integer` o `Byte` y el tipo opaco `Pointer` es `byval`.
- Los modificadores por defecto de los argumentos de un mensaje son `in` y `byref`, a no ser que se trate de un tipo integral o el tipo opaco `Pointer`, en cuyo caso se usa el modificador `byval`.
- El modificador por defecto de los valores de retorno de mensaje es `out` y `byref`, a no ser que se trate de un tipo integral o el tipo opaco `Pointer`, en cuyo caso se usa el modificador `byval`.

Los modificadores de argumentos de los mensajes del protocolo `Timer` son informativos, para no imponer dependencias con una implementación dada²⁹. El modificador `selfchain` no es forzado por el compilador, para permitir que una implementación distinta pueda devolver otro objeto en lugar del receptor original. El modificador `byval` no indica que la variable `time` se pase por valor en la pila, únicamente representa que se desea pasar la variable `time` por valor en una llamada remota. La posibilidad de anotar los argumentos ayuda a detectar incongruencias en el proceso de envío de mensajes remotos y facilita separar interfaz de implementación. Es responsabilidad de la implementación hacer honor al protocolo anotado. De igual modo que también ha de hacer honor a la semántica de los nombres de los mensajes de los protocolos. El complemento de la declaración de un protocolo anotado es el soporte en tiempo de ejecución, usado para inspeccionar los modificadores de los argumentos y efectuar las acciones correspondientes a cada protocolo.

²⁸ No todos los modificadores se aplican a todos los casos. Por ejemplo, el modificador `in` no tiene sentido en un valor de retorno, el modificador `selfchain` no tiene sentido como modificador de un argumento. El compilador se encarga de discriminar los casos apropiadamente.

²⁹ Como hemos visto, es muy conveniente mantener abiertas las puertas de la implementación a posibles estándares. Por ejemplo, Java definió sus modos de acceso remoto propietarios con RMI, para luego modificar los protocolos de transmisión a IIOP y ser compatible con CORBA. Si los modificadores de argumentos impusiesen una implementación, cambios en estándares serían más costosos de asimilar, pues requerirían revisar la definición del lenguaje.

6.2.6 Empaquetado y despliegue

El empaquetado o *packaging* y el despliegue o *deployment* de componentes usan el sistema de módulos del lenguaje. La razón es sencilla. Tanto entre módulos como entre componentes hay que realizar labores de comprobación de que todos los elementos necesarios para la ejecución se encuentran disponibles en el momento de su registro en el soporte en tiempo de ejecución. Un componente puede estar distribuido en varios módulos y un módulo puede a su vez contener varios componentes relacionados. El módulo es una unidad de empaquetado natural, puesto que tiene compilación separada de declaraciones y definiciones afines. Si la compilación no genera errores, el sistema de tipos garantiza que el módulo es autosuficiente. Es también la unidad conveniente para realizar la carga, porque la compilación del módulo asegura la comprobación de dependencias con módulos importados. Por último, el módulo puede implícitamente cargar varios componentes afines a la vez. Otras razones por las que resulta aconsejable disponer de un sistema de módulos se analizan en el apartado 5.5 *Módulos*.

Al ser el módulo la unidad de despliegue, el mecanismo de módulos de XC debe tener en cuenta las dependencias internas entre módulos, su versión y dependencias con otros módulos importados. Aunque no hay una relación 1:1 entre módulos y componentes, el caso degenerado es aquél en que un componente se implementa con un módulo. Al soportar XC módulos anidados, la disposición más común para un componente —o grupos de componentes afines tratados como un todo— será un módulo principal o secundario y varios submódulos.

6.2.7 Configuración

El despliegue de componentes debe configurarse finalmente en la aplicación de destino. Formatos de configuración son muchos y variados. Últimamente se están poniendo de moda las descripciones en XML, verificables automáticamente mediante DTD o XML Schema, pero no siempre fue así. XC no establece ningún formato de configuración para componentes, delegando la labor en bibliotecas adicionales. En todo caso, debido a que el lenguaje pretende ser usado para manipular distintos modelos de componentes externos, es más aconsejable usar el formato de configuración estándar que establezca cada modelo. La elección de usar una biblioteca está también justificada porque, como apunta [Szyperski, 1998; pp. 32-33], los módulos —al contrario que los componentes— no tienen información de configuración más allá de su propia descripción interna, usada por el soporte en tiempo de ejecución. Para programas escritos únicamente en XC que no accedan a modelos de componentes externos, es trivial implementar un sistema básico de configuración usando las expresiones estructuradas del lenguaje vistas en el apartado 5.6.2, que proveen directamente *arrays* y diccionarios.

6.2.8 Evolución

La evolución de sucesivas versiones de componentes es un tema delicado en proceso de investigación. Cuando un componente se ha de cambiar por otro, es posible que aparezcan problemas relativos a las dependencias implícitas entre el componente sustituido y el nuevo. La aproximación de XC al problema de la evolución toma carácter más básico, intentando comprender las razones de fondo de muchas de esas dependencias en el apartado 2.7 *Interdependencia*. Luego, diseñando las abstracciones del lenguaje para disminuir su número, aumentando la modularidad y poniendo especial énfasis en eliminar las dependencias implícitas. El mecanismo de categorías se ha diseñado expresamente de forma modular para facilitar la evolución de componentes y objetos. La limitación del uso de la herencia de implementación en los componentes exportados es otra estrategia para reducir las dependencias implícitas de reentrada de llamadas de retorno, siendo ésta usada también por COM. Los módulos y los prototipos incluyen información de versión, pero por ahora no se hace uso de esa información, a la espera de tener una primera versión estable del lenguaje.

6.3 Implementación de componentes

Este apartado repasa comportamientos asociados habitualmente con componentes, dando un bosquejo de su implementación en el nuevo lenguaje mediante algunos breves ejemplos. La implementación del modelo de componentes de XC es la que confiere las propiedades finales al modelo y a las reglas de composición, reteniendo cierta flexibilidad a la hora de optar por una implementación particular, como ya se ha visto en el apartado 6.2.

- Los objetos de la definición (Def. 6-1) se implementan modularmente, siendo extensibles mediante categorías también de forma modular.
- La implementación del envío de mensajes de la definición (Def. 6-2) puede variar, siempre que se mantenga la semántica de la técnica de selectores.
- Los protocolos y los mensajes que representan el comportamiento de componentes de la definición (Def. 6-3) se encuentran controlados por el sistema de tipos, que separa declaración e implementación.
- Las reglas del sistema de tipos y la sintaxis de XC garantizan que se cumple la definición (Def. 6-4).

El soporte de componentes en XC debe ser lo suficientemente flexible para implementar distintas semánticas o puntos de variación sobre el modelo de componentes básico, exemplificado por las reglas de composición. La orientación a programación de sistemas de XC no hace aconsejable definir una implementación rígida de un modelo de componentes. Antes al contrario, facilitar la integración modelos de componentes existentes o futuros es más interesante que proponer un modelo estricto propio. Las abstracciones del nuevo lenguaje dan un marco más cómodo para cumplir esta labor que lenguajes de más bajo nivel como C o C++.

6.3.1 Estructura de un componente

Los tres elementos principales de un componente son su protocolo, su implementación y el nexo de unión entre ambos: el modelo de componentes. Los elementos clave de este último se encuentran implementados en el lenguaje, por lo que la organización básica de componentes se simplifica notablemente. XC soporta una estructura jerárquica de módulos. Es aconsejable que un componente o un grupo de componentes afines se describan desde un módulo principal o secundario. No es necesario siquiera que ese módulo tenga implementación, aunque habitualmente es necesario para efectuar las inicializaciones oportunas. En el ejemplo siguiente, se muestra una posible declaración y definición de componente en un módulo. El protocolo del módulo incluye el protocolo del componente, el protocolo del contrato, dos implementaciones distintas del componente, y dos implementaciones adicionales con información del contrato.

La interfaz o protocolo de los componentes se describen con un protocolo de módulo. Un componente debe incluir al menos los protocolos de entrada y de salida. La información de contrato del protocolo se puede especificar mediante una categoría, que permite a otros componentes aplicarlas o no. El protocolo de módulo exporta también varias declaraciones de las implementaciones del componente de forma opaca, impidiendo heredar de ellas.

```
public module protocol ComponentModule
{
    protocol In
    { // ... }

    protocol Out
    { // ... }

    protocol Component extends: In, Out {}

    protocol category InContract extends: In
    { // ... }
```

```

protocol category OutContract extends: Out
    { // ... }

final prototype ComponentImpl1 implements: Component;
final prototype ComponentImpl2 implements: Component;
final prototype ComponentImpl1WithContract implements: Component;
final prototype ComponentImpl2WithContract implements: Component;
}

```

Los prototipos `ComponentImpl1` y `ComponentImpl2` contienen dos implementaciones sin la información de contrato. Los otros dos prototipos incluyen la información del contrato, no necesitando mayor definición. En el ejemplo, la implementación de los contratos es distinta para ambas implementaciones. Este detalle no es visible en el protocolo del módulo.

```

module ComponentModule
{
    prototype ComponentImpl1 extends: Object implements: Component
        { // ... }

    prototype ComponentImpl2 extends: Object implements: Component
        { // ... }

    prototype ComponentImpl1WithContract extends: ComponentImpl1 {}
    prototype ComponentImpl2WithContract extends: ComponentImpl2 {}

    prototype category InContractImpl1 extends: ComponentImpl1WithContract
        implements: InContract
        { // ... }

    prototype category InContractImpl2 extends: ComponentImpl2WithContract
        implements: InContract
        { // ... }

    prototype category OutContractImpl1 extends: ComponentImpl1WithContract
        implements: OutContract
        { // ... }

    prototype category OutContractImpl2 extends: ComponentImpl2WithContract
        implements: OutContract
        { // ... }
}

```

El código cliente puede elegir entre componentes con contrato o sin contrato usando el código siguiente dinámicamente, suponiendo una condición booleana `checkImp` que discrimina cuál usar. Recordemos que la representación o imagen dinámica del estado del componente consta de objetos:

```
Component component = (checkImp) ? ComponentImpl1.clone : ComponentImpl2.clone;
```

También puede elegirse entre los componentes con ayuda de instrucciones de compilación condicional similares a las definidas en C#, que *no* incluyen capacidades de macros, sino únicamente definición de existencia y combinación lógica de símbolos, externos a los identificadores del lenguaje³⁰. Esta última opción es más apropiada para compilar aplicaciones con características adicionales de depuración o de comprobación.

```

#ifdef TEST_CONTRACT
    protocol alias ComponentImpl1X = ComponentModule.ComponentImpl1WithContract;
#else
    protocol alias ComponentImpl1X = ComponentModule.ComponentImpl1;
#endif
// ...

Component component = ComponentImpl1X.clone;

```

³⁰ Las instrucciones de preprocessamiento `#define`, `#undef`, `#ifdef`, `#ifndef`, `#elsif`, `#else`, `#endif` y los operadores '`(`', '`)`', '`&&`', '`||`' y '`!`' no se implementan con un preprocesador externo, sino directamente en el analizador léxico, que opera con ellas de forma similar a como procesa los comentarios.

6.3.2 Control de calidad

Una técnica para fomentar la calidad de los componentes consiste en incorporar código adicional que realice comprobaciones adicionales sobre condiciones del componente. Es de esperar que este código pueda llegar a ser significativo y, por tanto, tener un coste apreciable. Facilitar la elección entre distintos “niveles de servicio” o “calidades de contrato” que comprueben aspectos de la especificación del componente, agiliza el acuerdo entre clientes y proveedores de componentes. Los clientes pueden seleccionar contratos más exhaustivos —y probablemente más caros— o menos rígidos y más baratos, dependiendo de sus necesidades. Elementos identificables en contratos son, por ejemplo, tiempos de respuesta, invariantes, recursos usados o monitorización de estado.

Una vía para hacer explícitos esos contratos es mediante lenguajes formales, como se comenta en el apartado 7.3.4 *Comprobación semántica de tipos*. Sin embargo, los desarrolladores son menos proclives al uso de lenguajes formales para especificar las condiciones de los contratos. Quizá, una aproximación imperativa sea más práctica, porque habilita la aplicación de expresiones y sentencias de control arbitrarias del lenguaje, aunque tiene la desventaja de no poder efectuarse sobre ellas verificaciones formales.

XC permite la definición más cómoda de contratos usando las facilidades de metaprogramación de categorías de protocolos y prototipos, debido a que la comprobación de contratos es una actividad ortogonal a la funcionalidad del componente. Es posible interceptar con métodos anteriores y posteriores los mensajes relevantes enviados a las implementaciones, y separar en distintas categorías contratos diferentes. El ejemplo siguiente esboza una organización de dos contratos, el primero con precondiciones, invariantes y poscondiciones, y el segundo con requisitos de tiempos de respuesta. La categoría de los requisitos de tiempo se incluye después de la categoría de aserciones en la linearización de categorías. Al ser menos prioritaria, sus métodos *before* se ejecutan tras los de la categoría más prioritaria y los métodos *after* justo antes.

```

prototype category ComponentImplAssertionContract extends: ComponentImplWithContract
           implements: InContract
{
    before void method1
    { // comprobar precondiciones e invariantes. }

    after void method1
    { // comprobar poscondiciones e invariantes. }

    // ...
}

prototype category ComponentImplTimingContract extends: ComponentImplWithContract
           implements: InContract
           after: ComponentImplAssertionContract
{
    before void method1
    { // medir tiempos despues de comprobar precondiciones e invariantes. }

    after void method1
    { // medir tiempos antes de comprobar de poscondiciones e invariantes }

    // ...
}

```

Por supuesto, otras opciones son viables, ya que es posible usar toda la potencia del lenguaje para organizar los contratos. Por ejemplo, el proveedor puede ofrecer a los clientes en un módulo aparte las categorías de los contratos acordados, dejando que sea el cliente quien los añada o elimine a su propia discreción en los componentes principales. Este camino se parece funcionalmente a las clases resumidas o *short* de Eiffel [Meyer, 1997; pp. 389-392], que incluyen la información de las invariantes, precondiciones y poscondiciones, pero no la implementación de los métodos. La opción elegida por XC es más informal. En cierto sentido es también más flexible, porque no impone ninguna restricción al tipo de contrato ni a la organización que éste debe tener. La estructura de los contratos es

actualmente un tema de investigación, particularmente en los sistemas de tipos que realizan comprobaciones semánticas, y no existe un consenso claro al respecto. La prudencia aconseja por ahora no definir estructuras de comprobación de contratos particulares al lenguaje.

Otra clase de comprobaciones de control de calidad son las realizadas durante depuración o mejora de prestaciones, para investigar el estado de ejecución de un programa. Las facilidades de metaprogramación son también útiles en estos casos. Para ilustrarlas con más detalle, consideremos un objeto delegado o *proxy* [Gamma, Helm, Johnson & Vlissides, 1995; pp. 207-217] que se hace pasar por otro objeto para habilitar una llamada remota. El objeto delegado se puede construir a partir de una declaración anotada con modificadores de argumentos de mensajes. El siguiente fragmento de código es el extracto de una posible implementación para el temporizador `Timer` del apartado 6.2.5 *Interoperabilidad*. Por brevedad se suponen predefinidos los tipos `Host` y `Socket`³¹.

```

protocol TimerProxy extends: Timer
{
    property Host remoteHost;
    Boolean      openConnection;
}

prototype TimerProxy extends: Object
{
    property Host   remoteHost;
    property Socket remoteConnection;

    Boolean openConnection
    {
        if (remoteConnection == null)
        {
            self.remoteConnection = remoteHost.openConnection;
            return (self.remoteConnection != null) ? true : false;
        }
        else return true;
    }

    selfchain setTimer: byval Time time
    {
        if (self.openConnection == true)
        {
            // Codifica time en un buffer, enviándolo a remoteHost por remoteConnection
            return self;      // Devuelve el objeto proxy local.
        }
    }
}
// ...
}

```

El protocolo `TimerProxy` hereda los mensajes definidos en el protocolo `Timer` y define la propiedad `remoteHost` para inicializar correctamente el objeto *proxy*. Los métodos del prototipo `TimerProxy` comprueban la conexión remota y, si está abierta, codifican los argumentos y los envían por la conexión. Un código cliente podría acceder al temporizador como sigue. Por brevedad supondremos que una máquina remota se encuentra ya en la variable `host` y un tiempo dado se encuentra en la variable `time`.

```

TimerProxy timerProxy = TimerProxy.clone();
timerProxy.remoteHost = host;

Timer timer = timerProxy;
// ...
timer.setTimer: time;

```

En primer lugar el cliente debe usar el protocolo de `TimerProxy` para inicializar el objeto *proxy* `timerProxy` con la máquina remota `host`. Luego debe asignar el objeto *proxy* a un protocolo `Timer`, para limitar la interfaz conocida por el cliente. Posteriormente, otras partes del código cliente no necesitan conocer que `timer` es realmente un objeto *proxy*. El código anterior no es especialmente

³¹ Actualmente XC no soporta excepciones, por lo que se usa la convención de devolver el objeto `null` cuando el objeto es inválido.

novedoso. Simplemente es un ejemplo de cómo las abstracciones del lenguaje sirven para manipular objetos e implementar conductas asociadas a componentes con comodidad.

Ahora bien, cuando se realizan accesos remotos es conveniente comprobar que no se están abriendo demasiadas conexiones remotas, porque es posible degradar las prestaciones o consumir demasiados recursos. Al enmascarar objetos `TimerProxy` como objetos `Timer`, el código cliente puede inadvertidamente efectuar costosas llamadas remotas en vez de llamadas locales mucho más eficientes. En XC el cliente es capaz de efectuar esas comprobaciones ortogonales de control de calidad o depuración a la funcionalidad de `TimerProxy` convenientemente usando una categoría, sin necesidad de tener acceso al código original de `TimerProxy`.

```
protocol category TimerProxyConnectionwatcher extends: TimerProxy
{
    Unsigned numberOfConnections;
}

prototype category TimerProxyConnectionwatcher extends: TimerProxy
{
    shared property numberOfConnections;
    after Boolean openConnection
    {
        if (return == true)
            self.numberOfConnections++;
    }
}
```

La categoría `TimerProxyConnectionwatcher` extiende el protocolo `TimerProxy` con un protocolo para leer la información de conexiones. La implementación usa la variable compartida `numberOfConnections` para contar las conexiones realizadas por todos los objetos de la familia `TimerProxy`. La variable `numberOfConnections` es inicializada a cero por el compilador. El método posterior a `openConnection` comprueba el valor de retorno del método base para saber si se están abriendo nuevas conexiones con la pseudo-variable `return` —disponible únicamente en los métodos posteriores— y que devuelve el valor de retorno del método base. Luego, código cliente de control de calidad puede acceder a la variable compartida a través del objeto prototípo `TimerProxy` o de cualquier otro objeto de la misma familia, usando el mensaje definido en el protocolo de la categoría.

```
Unsigned connectionTest = TimerProxy.numberOfConnections;
```

6.4 Recapitulación

Este capítulo explica el modelo de componentes de XC apoyándose en el bagaje acumulado de los capítulos anteriores. El modelo de componentes de XC es un modelo nuevo, pensado expresamente para manipular componentes creados dentro del lenguaje e interoperar con otros modelos de componentes usando las mismas abstracciones. Por esta razón evita definir protocolos de bajo nivel que usualmente cambiarán entre diferentes modelos de componentes, relegando a bibliotecas la implementación de los estándares apropiados. La construcción del modelo se simplifica en gran medida al reaprovechar las abstracciones del lenguaje. El resultado es un lenguaje cuya semántica está más cercana a la semántica de componentes. Se incide en aquellos aspectos más característicos del modelo: composición y reglas de composición, interfaces, contratos, nombrado de componentes, metadatos, interoperabilidad, empaquetado, despliegue, configuración y evolución.

La segunda parte del capítulo expone algunos ejemplos de implementación de componentes usando las abstracciones del lenguaje. Se presenta el esquema de un componente con dos implementaciones diferentes y contratos semánticos asociados, incluyendo formas de seleccionar modularmente la implementación de los componentes siguiendo un mismo protocolo y diferente implementación de contratos. Por último, se estudian aplicaciones de metaprogramación para el control de calidad de los componentes y cómo implementarlas directamente en XC de forma sencilla. El resultado es código

más limpio y factorizado. La creación de contratos múltiples, un objeto delegado o *proxy* y comprobaciones de depuración añadidas en categorías ilustran la flexibilidad del modelo de componentes y objetos de XC, así como la versatilidad de su mecanismo de expansión modular con categorías.

7

El sistema de tipos

7.1 Introducción

El sistema de tipos es el soporte fundamental de un lenguaje compilado. Identifica programas correctos con respecto a unas reglas definidas por cada lenguaje. El sistema de tipos implementa buena parte de esas reglas, aplicándolas automáticamente a los programas y validando que, al menos, éstas se cumplen. Los sistemas de tipos no son perfectos, y no es fácil diseñar un sistema de tipos que sea suficientemente expresivo para aceptar múltiples variedades de programas. Una parte importante de este capítulo se dedica a evaluar deficiencias de los sistemas de tipos y cómo solventarlas. Cuando la expresividad no es la adecuada, los programas deben apartarse de las reglas del sistema de tipos para describir su funcionalidad. Si el sistema de tipos es demasiado estricto, es posible que ciertas clases de programas no puedan ser expresados en el lenguaje, y deba elegirse otro lenguaje para realizar la labor. Un buen sistema de tipos reduce esa necesidad, a la vez que mantiene y fomenta las reglas del lenguaje.

El capítulo en primer lugar estudia las variantes principales de sistemas de tipos en lenguajes de programación, introduciendo también los conceptos más básicos. A continuación, se estudia cómo la herencia y el polimorfismo afectan a los sistemas de tipos, y las limitaciones de éstos para expresar relaciones entre tipos y subtipos. Se estudia también la separación entre tipos e implementación y las posibilidades para realizar comprobaciones semánticas de tipos, más allá de las usuales comprobaciones sintácticas. Seguidamente se analiza con más detenimiento aspectos más avanzados del sistema de tipos —los tipos covariantes y el encaje o *matching*— como solución a dificultades de expresividad de muchos lenguajes orientados a objetos más tradicionales. Luego se estudia la implementación de tipos genéricos en el lenguaje y las implicaciones de diseño que conlleva. Por último, se revisa cómo el sistema de tipos realiza comprobaciones modulares de tipos entre módulos con declaraciones interdependientes.

7.2 Variantes de sistemas de tipos

Una de las características fundamentales de los lenguajes de programación es su *sistema de tipos*. Por sistema de tipos se entiende el conjunto de reglas que permiten distinguir la representación binaria de los datos en memoria, para no confundir en una ristra de *bits*, por ejemplo, un valor entero con un carácter alfanumérico. Todos los lenguajes de programación soportan algún sistema de tipos, aunque existen notables diferencias en cómo éstos afectan al lenguaje. Atendiendo a esa característica, se pueden distinguir las siguientes variantes principales de lenguajes:

- *Lenguajes sin tipo.* Sólo manejan tipos de datos implícitos predefinidos, como cadenas o listas. Ejemplos son los lenguajes de *script*, tal como TCL o los lenguajes de comandos o *shell* de UNIX. Estos lenguajes son convenientes para pequeños desarrollos en los que las relaciones y operaciones entre los tipos de datos son conocidas o se pueden inferir con facilidad. En general, operar sobre tipos de datos no válidos da lugar a comportamientos indefinidos.
- *Lenguajes con tipos dinámicos.* En un lenguaje de esta clase se admite la definición de nuevas estructuras de datos en memoria, pero el lenguaje sólo comprueba su validez durante la ejecución. LISP y Smalltalk-80 son dos ejemplos. Su principal ventaja radica en la flexibilidad para crear nuevas estructuras de datos complejas, que facilita la programación exploratoria. Operar sobre tipos de datos no válidos da lugar a excepciones en tiempo de ejecución, que habitualmente pueden ser capturadas y manejadas.
- *Lenguajes con tipos estáticos.* Estos lenguajes definen nuevas estructuras de datos en memoria sólo después de haberlas declarado previamente. La declaración es usada durante el análisis o compilación del código fuente para detectar posibles errores al operar sobre los tipos de datos. Este análisis es usado para encontrar numerosos casos de operaciones no válidas y resolverlos antes de la ejecución de los programas. Existen, sin embargo, conjuntos de operaciones que no pueden comprobarse en tiempo de compilación y, dependiendo del lenguaje, pueden ser reconocidas o no.

El lenguaje XC se clasifica dentro de los lenguajes con tipos estáticos, cuya ventaja principal es que permite la comprobación automática —antes de la ejecución de un programa— de familias de errores de operación sobre tipos de datos muy habituales. Ejemplos de estos errores son, entre otros, el envío de mensajes incorrectos, el envío de un número incorrecto de parámetros, el envío de parámetros de tipos de datos inadecuados, la asignación incorrecta de valores a variables, o la herencia inadecuada. La declaración de tipos es, además, una forma de documentación y comunicación indirecta entre quien ha desarrollado el código originalmente y quien debe usarlo o modificarlo después. Ambas propiedades son muy importantes para facilitar los desarrollos a gran escala, donde la comunicación entre quien desarrolla el código y quien lo usa o lo modifica puede que no sea posible. Debido a la poca idoneidad y falta de robustez de los lenguajes sin tipos o con tipos dinámicos para estas tareas, no nos extenderemos más en su descripción.

Durante la definición de un nuevo lenguaje surge sin duda la pregunta acerca de la capacidad de los sistemas de tipos existentes para ser aplicados al nuevo lenguaje. El estudio de sistemas de tipos en lenguajes orientados a objetos es actualmente motivo de continuo e interesante análisis. En síntesis, se puede decir que las propiedades idóneas del sistema de tipos de un lenguaje orientado a objetos no se conocen con exactitud, si bien se entienden muchas propiedades que facilitan ciertas operaciones con tipos de datos, y que son fuente de importantes complicaciones si no se encuentran presentes.

A la hora de dotar a XC de un conjunto de reglas que describa su sistema de tipos, se han tenido en cuenta los problemas descritos en la literatura, tratando de incorporar un sistema de tipos que recoja los avances en este campo, evitando en lo posible los errores cometidos en el pasado. La combinación de estas reglas puede dar lugar, como veremos, a lenguajes con propiedades muy diferentes, que los hacen más aptos para unas tareas y menos para otras. El sistema de tipos de XC es una nueva aportación al conocimiento en este campo.

7.2.1 Aspectos básicos de un sistema de tipos

Como se ha visto antes, el sistema de tipos es una técnica que permite identificar la estructura de la información almacenada en memoria. El significado de los *bits* almacenados en memoria puede variar dependiendo del programa, existiendo muchas interpretaciones posibles. Un conjunto de 32 *bits* puede considerarse un número entero con signo, sin signo, una dirección de memoria, o una máscara para una imagen gráfica, por poner sólo unos pocos ejemplos. Para evitar errores sencillos pero tediosos en la manipulación de la información, ésta se puede estructurar. Se da nombre a cada una de

sus partes para aclarar su significado, e igualmente a conjuntos de las mismas para describir estructuras de datos más complejas. Es decir, se usan nombres para identificar variedades o *tipos* de datos. Las siguientes definiciones caracterizan sin ambigüedad un sistema de tipos, incluyendo términos básicos que se usarán a lo largo del capítulo:

- ❖ **Def. 7-1.** Un *bit* es la unidad básica de información, que corresponde con el valor discreto uno o cero.
- ❖ **Def. 7-2.** La *información* o *datos* se compone de conjuntos de *bits* de longitud variable y quizás dispersos.
- ❖ **Def. 7-3.** Un *tipo de datos* identifica una variedad de información que tiene una interpretación o significado dado.
- ❖ **Def. 7-4.** Un *sistema de tipos* se encarga de reconocer distintas variedades de información o tipos de datos, para dar una interpretación o significado a los conjuntos de *bits* que manipula un programa.
- ❖ **Def. 7-5.** Una *firma* es un nombre que se asigna a un tipo de datos para identificarlo únicamente.
- ❖ **Def. 7-6.** Un tipo de datos puede ser *compuesto*, es decir, contener a su vez un conjunto de tipos de datos, cada uno identificado con su propia firma.
- ❖ **Def. 7-7.** Un sistema de tipos es *nombrado* si identifica los tipos de datos por su firma. Dos tipos de datos se consideran iguales si y sólo si sus firmas son exactamente iguales.
- ❖ **Def. 7-8.** Un sistema de tipos es *estructural* si identifica los tipos de datos por el conjunto de firmas que contienen. Dos tipos de datos se consideran iguales si contienen el mismo conjunto de firmas.

Introduciremos paulatinamente una notación compacta, inspirada en [Bruce, 1996], para describir más concisamente ciertas relaciones importantes entre los tipos de datos. Una firma de mensaje está compuesta por un nombre m , un conjunto de k argumentos a_i cuyo tipo es T_i , y un valor de retorno T_R . Se representa como:

$$(7-1) \quad m = \text{Msg}(a_i : T_i)_{1 \leq i \leq k} : T_R$$

Una notación compacta para (7-1) que hace énfasis en el valor de retorno es:

$$(7-2) \quad m : T_R$$

Un tipo de datos T asociado a un objeto se define como un conjunto de n firmas de mensajes m_j cuyo valor de retorno es T_j :

$$(7-3) \quad T = \text{Obj}\{m_j : T_j\}_{1 \leq j \leq n}$$

En un lenguaje con tipos nombrados (Def. 7-7), los tipos T_1 y T_2 son iguales si sus firmas coinciden, es decir, si $T_1 = T_2$. En cambio, en un lenguaje con tipos estructurales (Def. 7-8), los tipos T_1 y T_2 son iguales si los conjuntos de firmas que contienen concuerdan:

$$(7-4) \quad \text{Obj}\{m_i : T_i\}_{1 \leq i \leq n} = \text{Obj}\{m_j : T_j\}_{1 \leq j \leq n}$$

XC usa tipos de datos nombrados, así que nos centraremos en ellos. En XC una definición de tipo corresponde con una definición de protocolo, hasta tal punto que se usarán como sinónimos. Por esa razón, la ecuación (7-3) sólo identifica firmas de mensajes, y no variables de instancia. Existen tres definiciones principales de tipo:

- ❖ **Def. 7-9.** Una definición *no recursiva* es aquella en la que el nombre del tipo no aparece en su definición:

```
protocol Point
{
    property Integer x;
    property Integer y;
}
```

Nótese que en XC una propiedad de un protocolo define siempre dos mensajes (véase el apartado 5.2.1.2.1 *Mensajes y propiedades de protocolos*). Más concisamente, una definición no recursiva se expresa como:

$$(7-5) \quad T = Obj\{m_j : T_j\}_{1 \leq j \leq n} \text{ con } m_j = Msg(a_i : T_i)_{1 \leq i \leq k} \text{ si } \exists T_j = T \wedge \exists T_i = T$$

Es decir, T no aparece dando tipo a valores de retorno o argumentos de mensajes.

- ❖ **Def. 7-10.** Una definición es *recursiva* si el nombre del tipo aparece dentro de su definición:

```
protocol List
{
    property Object value;
    property List next;
}
```

Y también más concisamente:

$$(7-6) \quad T = Obj\{m_j : T_j\}_{1 \leq j \leq n} \text{ con } m_j = Msg(a_i : T_i)_{1 \leq i \leq k} \text{ si } \exists T_j = T \vee \exists T_i = T$$

Es decir, T aparece dando tipo a valores de retorno o argumentos de mensajes.

- ❖ **Def. 7-11.** Una definición de dos tipos es *mutuamente recursiva* si el nombre del primer tipo aparece en la definición del segundo y el nombre del segundo aparece en la definición del primero:

```
protocol Object
{
    property ObjectRecord record;
}

protocol ObjectRecord
{
    property Object field1;
    property Object field2;
}
```

La formulación en notación compacta se deduce de forma trivial a partir de (7-6). En la práctica, en XC la mayor parte de las definiciones de tipo son mutuamente recursivas. De ahí la importancia de la existencia de declaraciones de módulos interdependientes en el nuevo lenguaje.

Los tipos de datos son utilizados por el compilador para validar el uso de operaciones sobre los datos. Un tipo de datos describe qué operaciones son válidas con los datos que él representa. Los datos se manipulan mediante variables, propiedades, parámetros y valores de retorno de métodos o funciones. Los datos siempre se implementan en XC mediante objetos. Los tipos de datos predefinidos, como los tipos booleano, entero o *array*, son considerados objetos. Los nuevos tipos de datos definidos por el usuario se tratan siempre como objetos, aunque técnicamente algunos no lo sean. Este extremo se estudia en el apartado 8.7.4.2 *Objetos nativos*.

- ❖ **Def. 7-12.** Un sistema de tipos está *unificado* si manipula a todos los tipos de datos de la misma manera.

Smalltalk-80, SELF, Eiffel, C# y XC tienen un sistema de tipos unificado, que opera siempre con objetos. Java, C++, Objective-C y Ada son ejemplos de lenguajes con tipos de datos no unificados. Los sistemas de tipos unificados son más consistentes que aquellos que separan los tipos de datos. Un ejemplo de su conveniencia se encuentra en el apartado 3.6.1 *Consistencia*. La unificación del sistema de tipos no va en detrimento de la eficiencia en tiempo de ejecución. Los aspectos de eficiencia se ven en el apartado 8.7.2 *Optimización de tipos básicos*.

7.3 Tipos y subtipos

Los tipos de datos de lenguajes procedurales son relativamente sencillos, porque su estructura no cambia en general. Los registros variantes de Pascal, los discriminantes de Ada o las uniones de C admiten ciertos cambios en la estructura de los tipos dependiendo de algunas características particulares de ellos. Estos últimos tipos, que llamaremos genéricamente *registros variantes*, son necesarios para modelar distintos casos especiales dentro de un tipo de datos más general. Por ejemplo, un tipo de datos que represente una trama de información que se transmite por red puede tener campos diferentes cuando ha de notificar un error, campos que dependen del valor particular del error. Tipos de datos de esta clase son muy comunes en la modelización de problemas informáticos complejos. En cada momento, las operaciones que manejan el tipo de datos han de discriminar explícitamente en el código los diferentes casos posibles. El código extra necesario para gestionar estos registros variantes es significativo y puede llegar a ser arduo, sobre todo en presencia de múltiples condiciones de discriminación encadenadas. En caso de no disponer de facilidades de creación de registros variantes, es necesario replicar los registros en declaraciones distintas y aumentar notablemente la base de código. Uno de los objetivos iniciales de SIMULA-67 era manipular este tipo de registros con más sencillez [Nygaard & Dahl, 1978; p. 258] e intentar reaprovechar el código común.

La orientación a objetos y la herencia son un sustituto para la manipulación de registros variantes. En ese sentido, simplifican notablemente un programa al reducir considerablemente la cantidad de código que ha de desarrollarse. Sin embargo, la interacción de la orientación a objetos, la herencia y los sistemas de tipos se complica. En la práctica, la herencia en programación orientada a objetos se usa para las siguientes tareas:

1. Heredar una especificación, interfaz o protocolo.
2. Heredar una implementación.
3. Manipular distintos registros variantes como un grupo.

Originalmente, SIMULA-67 realizaba las tres funciones anteriores a la vez durante la herencia, de la misma forma que Smalltalk-80 y muchos lenguajes posteriores conocidos como C++, Eiffel o Java. La combinación simultánea de estos tres aspectos ha sido fuente de notorios debates acerca de la herencia, de lo que debe hacer y de lo que no. En lenguajes con tipos estrictos, la respuesta a qué debe gestionar un sistema de tipos que incorpore herencia produce diferentes interpretaciones. La interpretación más sencilla agrupa en un tipo de datos las tres tareas, como ya se ha visto. Pero otras opciones son también posibles. Java, por ejemplo, identifica los tipos de datos generados por las clases con las tres tareas, y a tipos de datos generados por las interfaces con las tareas 1 y 3. En C++ es posible heredar únicamente implementación —es decir, la segunda tarea— usando la herencia por defecto, que siempre es privada.

Con el tiempo, se ha ido viendo que las tres tareas originales deben ser más independientes, y que desde ese punto de vista pueden resolverse algunos problemas endémicos a los lenguajes orientados a objetos. Es interesante notar que el antecesor directo de C++, C con Clases, ya distinguía expresamente la herencia de implementación [Stroustrup, 1994; p. 54]. La primera tarea identifica qué especificación debe ser heredable: se pueden heredar todos los mensajes, o solo algunos, de una o varias fuentes. La segunda tarea hace lo mismo con la implementación: heredar todos los métodos o sólo algunos, de uno o varios padres. La tercera tarea corresponde con el polimorfismo y la relación tipo-subtipo. El polimorfismo lo trataremos en el apartado siguiente, centrándonos ahora en la relación entre tipos y subtipos.

- ❖ **Def. 7-13.** Un tipo *S* es un *subtipo* de un tipo *T* si en cualquier contexto de programa que espera una expresión del tipo *T* puede usarse una expresión del tipo *S* sin introducir un error de tipo.

A la relación subtipo la identificaremos con el símbolo $<:$, y lo representaremos como $S <: T$. Para que S sea sustituible en una expresión que involucre a T , S debe tener al menos todos los mensajes que define T .

$$(7-7) \quad S = Obj\{m_i : T_i\}_{1 \leq i \leq k} <: T = Obj\{m_j : T_j\}_{1 \leq j \leq n} \quad \text{sii } n \leq k \wedge T \subset S$$

Para los primeros n mensajes de T se cumple que $T_i = T_j$. Véase que la relación $<:$ puede intuitivamente generalizarse si tenemos en cuenta que para cualquier valor de retorno T_i se pueda usar un S_i en su lugar, donde se cumpla que $S_i <: T_i$.

$$(7-8) \quad S = Obj\{m_i : S_i\}_{1 \leq i \leq k} <: T = Obj\{m_j : T_j\}_{1 \leq j \leq n} \quad \text{sii } n \leq k \wedge T \subset S / \forall i \leq n \ S_i <: T_j$$

En este caso, T está incluido en S realizando la sustitución de T_i por S_i , aprovechando que en principio S_i se puede reemplazar en cualquier expresión que tenga T_i , y que $T_i = T_j$ para los n mensajes de T . La ecuación (7-8) es la interpretación que se asocia habitualmente con la relación subtipo.

Generalmente se da a un subtipo el significado de ser más específico, concreto, refinado o explícito que un tipo, y a este último ser más general, menos concreto, menos refinado, menos explícito. La interpretación de qué es un subtipo de datos es así un poco ambigua, existiendo al menos dos sentidos:

- Un *subtipo* es un *superconjunto* o *extensión* de un tipo. Por ejemplo, en muchos lenguajes orientados a objetos un subtipo puede tener más métodos que un tipo. En este caso se dice también que el subtipo es más concreto o refinado, porque define más información que el tipo. Un ejemplo es la extensión de un tipo punto a un tipo punto con color. [Halbert & O'Brien, 1987; pp. 28-29] lo denominan subtipos por generalización. Otra variante de extensión ocurre al usar herencia múltiple. A partir de una ventana de un entorno gráfico y un controlador de teclado es posible definir un subtipo que represente una ventana gráfica que acepte texto introducido por un teclado. [Halbert & O'Brien, 1987; pp. 27-28] llaman a este caso subtipos por combinación.
- Un *subtipo* es un *subconjunto* o *especialización* de los valores de un tipo [Barnes, 1994; p. 48], [Halbert & O'Brien, 1987; pp. 26-27]. Por ejemplo, en Ada, un subtipo más concreto de los números enteros son los días del mes, cuyo rango es de 1 a 31. Es la interpretación tradicional de la programación estructurada. En lenguajes orientados a objetos se puede especializar un tipo general que represente un dispositivo de entrada con subtipos que identifiquen casos particulares como un ratón, un teclado o una tableta gráfica. Otra variedad es la especialización de la implementación, es decir, subtipos usados para cambiar la implementación de un tipo, como los subtipos del tipo lista que implementan ésta como un *array* o estructuras encadenadas. Son llamados subtipos por implementación en [Halbert & O'Brien, 1987; p. 27].

Las dos interpretaciones son, hasta cierto punto, contradictorias y fuente de no pocas confusiones. Pero quizás lo más sorprendente es que ambas son muy aceptadas, usadas, y a veces mezcladas porque las dos son útiles. Se trata de la *paradoja de la extensión-especialización* [Meyer, 1997; pp. 499-500]. La herencia es identificada como extensión o como especialización dependiendo del punto de vista: si se ve como un módulo, o como un tipo o variedad. Por ejemplo, en el primer caso un tipo considerado como un módulo provee unos servicios. Un subtipo implementa los servicios del tipo más los suyos propios, es decir, extiende el tipo original. En el segundo caso, un subtipo —digamos, *gato*— es una variante especial, más concreta, del tipo *animal*. En ambos casos se permite asignar una variable del subtipo al tipo sin problemas, aplicando la definición (Def. 7-13). Pero el camino inverso no siempre se garantiza. Por ejemplo, en Ada es posible asignar un subtipo a un tipo sin riesgo, si bien una asignación de un tipo a un subtipo puede dar lugar a una excepción:

```
subtype Day_Number is Integer range 1..31;
D : Day_Number;

I : Integer;
D := I; -- válido, pero puede lanzar Constraint_Error.
```

```
I := D; -- ok.
```

La excepción es posible porque el subtipo restringe el conjunto de valores válidos del tipo original, es una especialización. Las mismas operaciones son aplicables al tipo y al subtipo, y ciertamente cumplen con las ecuaciones (7-7) y (7-8), pero el subtipo tiene un dominio de valores *menor* que el tipo. Cuando se produce una extensión, se espera que el subtipo tenga un dominio de valores *mayor* que aquél representado por el tipo y, por tanto, impide que se produzcan excepciones. Hay algo que no funciona bien en estos comportamientos, porque tanto la especialización como la extensión son útiles. Es más, la especialización es generalmente vista como más útil que la extensión y, sin embargo, es esta última la que plantea menos problemas, puesto que no genera excepciones. Para complicar las cosas más aún, es común que especialización y extensión ocurran simultáneamente, como en el caso de ser necesario un nuevo método en una especialización.

El origen de las dificultades anteriores está en la unión del sistema de tipos con la herencia. La mayoría de los lenguajes orientados a objetos asocian la declaración de una clase con un tipo de datos. Tendencias a unir el sistema de tipos con la estructura de la herencia ya se encuentran en [Nygaard & Dahl, 1978; p. 261]. Las clases parecían no sólo dar la estructura organizativa para modelar programas sino también una implementación. Eliminar la dualidad en el sistema de tipos se pensaba que simplificaría los conceptos en juego. Sin embargo, como veremos, este no es el caso. En [Johnson, 1986; pp. 316-317] vuelven a aparecer los síntomas del problema de fondo. Se resalta cómo un *array* de enteros no puede ser un subtipo de un *array* de objetos. Si un entero hereda de un objeto, al ser el *array* de enteros más restrictivo que el *array* de objetos, no puede usarse en una expresión que involucre directamente al objeto, pues podría dar lugar a alguna excepción. Johnson deduce que las clases no constituyen buenos tipos de datos. Se trata del mismo problema que en los subtipos de Ada, si bien en Ada no se considera ese comportamiento un problema en absoluto por razones históricas, ya que ocurre igualmente con los subtipos definidos en Pascal. El tratamiento de tipos y subtipos desde expresiones en un lenguaje de programación —como la asignación de un subtipo a un tipo— implica el uso de polimorfismo y la habilidad del compilador para lidiar con ellas. Aclararemos primero estos aspectos antes de continuar con la nebulosa relación entre tipos y subtipos.

7.3.1 Polimorfismo y comprobación estricta de tipos

Bajo el nombre *polimorfismo* se engloban diversas características de los lenguajes orientados a objetos: tipos genéricos, sobrecarga de métodos y herencia. En este apartado, y para centrar la discusión, nos vamos a referir al polimorfismo que proporciona la herencia. Una introducción al polimorfismo se presenta en el apartado 4.3.3 *Polimorfismo*, durante la discusión de las propiedades de la herencia. Los aspectos relacionados con el sistema de tipos se posponen hasta este capítulo para separar dos discusiones relacionadas pero relativamente independientes.

Con el fin de hacer *en principio* más reusable el código, en los lenguajes orientados a objetos la manipulación de datos se puede realizar *polimórficamente*, es decir, una misma variable o parámetro puede referenciar a más de un tipo de datos simultáneamente. Esta es la noción usual de *conformidad de tipos* o *conformance* [Cook, 1992; p. 1] y es resultado de aplicar la definición (Def. 7-13) de subtipo. Decimos “en principio” porque, como veremos, no tiene porqué ser así. Para refrescar las ideas, si un tipo *String* hereda del tipo *Object*, entonces una variable de tipo *Object* puede hacer referencia tanto a un objeto *Object* como a un objeto *String*. Es decir, usando la ecuación (7-8), se cumple que *String* $<:$ *Object*. El tipo *String* se dice entonces que es *polimórfico* porque tiene dos tipos: el suyo propio *String* y el heredado *Object*. El siguiente ejemplo muestra una manipulación polimórfica de datos en XC:

```
protocol Object
{
    void print;
```

```

        // ...
}

protocol String extends: Object
{
    unsigned length;

    // ...
}

// ...

Object obj;
String str = "text";

obj = str;      // Ok. Asignación polimórfica.
obj.print();   // Ok.
obj.length;    // Error de compilación.

```

En un lenguaje orientado a objetos se puede asignar a la variable `obj` la variable `str`, puesto que esta última tiene también el tipo `Object` implícitamente, ya que `String <: Object`. La variable `obj` se dice que es una variable polimórfica (Def. 4-7). La intención es poder manejar muchos tipos de objetos distintos reutilizando el código que manipula objetos de tipo `Object`. Esta manipulación conlleva una *pérdida del tipo de datos* por parte de un compilador con tipos estrictos. El envío del mensaje `print` es reconocido por el compilador porque forma parte del protocolo de `Object`. Sin embargo, el mensaje `length` genera un error de compilación, porque `Object` no declara un mensaje con ese nombre. Nótese que `String` sí posee esa definición. El objeto manipulado es realmente de tipo `String`, por lo que éste sí sería capaz de entender el mensaje `length`, aunque el compilador no pueda asegurarlo³².

El fenómeno de la pérdida del tipo de datos afecta, por tanto, a la capacidad del compilador para comprobar la corrección de los mensajes enviados a una variable polimórfica. En muchos casos, el código ha de discriminar entre los posibles tipos de datos reales almacenados en una variable polimórfica y, para ello, ha de redescubrir en tiempo de ejecución el tipo de datos perdido. En el pasado, en C++ el redescubrimiento se efectuaba con un operador de conversor de tipo o *type cast*. Desafortunadamente, el compilador no realizaba ninguna comprobación sobre la correcta aplicación del conversor, siendo una fuente importante de errores de programación. C++ posteriormente, y reconociendo esa limitación, introdujo el operador de conversión de tipos con comprobación `dynamic_cast` [Stroustrup, 1994; pp. 308-315]. En Java el operador de conversión de tipos siempre incluye una comprobación en tiempo de ejecución. En XC se realiza la comprobación en tiempo de ejecución con el operador de tentativa de asignación ‘?=’, similar al encontrado en Eiffel. El operador devuelve `true` si la asignación es posible y `false` si no lo es. Siguiendo el ejemplo anterior, para recuperar el tipo `String` perdido al asignar la variable `str` a `obj` se usa un código similar al siguiente:

```

String str2;
if (str2 ?= obj)
    // obj es un tipo String y str2 es correctamente asignado
else
    // obj no es un tipo String y str2 no es asignado.

```

Existen situaciones en las que únicamente es posible reconocer el tipo real asociado a una variable polimórfica usando una comprobación en tiempo de ejecución como la anterior. Sin embargo, es deseable que el número de éstas sea el menor posible, puesto que si el compilador es capaz de asegurar el tipo de datos de un objeto, puede comprobar mejor los posibles errores y generar código más eficiente, ahorrándose la comprobación en tiempo de ejecución. El número de operadores de conversión visibles en el código es indicativo de la capacidad del sistema de tipos para expresar la corrección del código creado. Pero hay que tener cuidado. Por ejemplo, C++ puede limitar el número de operadores de conversión visibles realizando conversiones implícitas si existe un operador de conversión. El resultado de aplicar esta técnica puede ser confuso, ya que debe ser evidente cuál es el

³² En un lenguaje orientado a objetos con tipos dinámicos como Smalltalk-80 el mensaje se enviaría correctamente al objeto `str`, que lo reconocería y ejecutaría sin generar error alguno. Por supuesto, en estos últimos lenguajes si se enviara el mensaje mal deletreado `length`, se produciría un error en tiempo de ejecución, y sólo se podría detectar en tiempo de ejecución ese error.

propósito de la conversión. Resulta más claro llamar a un método de conversión con un nombre adecuado para no esconder los cálculos realizados.

Por esa razón, y al igual que en Java, se considera desaconsejable la definición por parte del usuario de operadores de conversión implícitos, manteniendo únicamente las conversiones predefinidas, de semántica conocida. De hecho, los operadores de conversión de tipos de C++ `dynamic_cast`, `static_cast`, `reinterpret_cast` y `const_cast` se han pensado para hacer más explícitas las conversiones en C++ [Stroustrup, 1994; pp. 327-329]. C++ puede reducir además el código polimórfico y los conversores de tipo usando *templates*. En Java, en cambio, el número de operadores de conversión explícitos en código polimórfico para redescubrir tipos perdidos es llamativamente alto y en general no puede evitarse, incurriendo en costes extra en tiempo de ejecución debidos a las comprobaciones asociadas a las conversiones. Como veremos, XC limita significativamente su uso.

Conviene notar también otro escenario donde el compilador pierde la traza del tipo de datos utilizado, ya que un tipo estático se puede convertir en dinámico —y por tanto polimórfico— si se permite realizar alguna acción sobre un mensaje enviado que no haya sido reconocido. Un compilador puede verificar que todos los mensajes estáticamente determinados en el código fuente sean enviados correctamente a objetos almacenados en variables polimórficas. Los mensajes construidos dinámicamente no pueden ser identificados por el compilador, por lo que si el lenguaje proporciona algún mecanismo de envío de mensajes dinámico existe el riesgo de enviar mensajes que no se reconozcan. Si el lenguaje admite la definición de manejadores de mensajes no reconocidos —bien mediante excepciones o con métodos normales— es posible reenviar esos mensajes a otros objetos que sí respondan a ellos. De forma efectiva el objeto original estará respondiendo a mensajes de otro tipo de datos y, por tanto, se comportará como un tipo dinámico.

Para terminar, obsérvese que el polimorfismo únicamente facilita la manipulación de tipos y subtipos usando una misma expresión del lenguaje. La relación existente entre el tipo y sus subtipos es ortogonal a la manipulación de variables polimórficas. Es decir, los subtipos pueden ser especializaciones o extensiones de los tipos independientemente de que sean manipulables polimórficamente. Peor aún, el compilador, al perder la traza del subtipo en una asignación a un tipo, es incapaz siquiera de ayudar a distinguir esa relación. Lo mismo ocurre si se pueden enviar mensajes dinámicamente. Como resalta [Snyder, 1986a; p. 41] las reglas de un sistema de tipos estrictos que validan la existencia de tipos y subtipos son de importancia crítica, porque determinan la corrección de los programas escritos en un lenguaje.

7.3.2 Separación entre tipos e implementación

Hace ya bastantes años que se identificó con precisión la diferencia entre herencia de tipos y herencia de implementación, considerando las diferentes interpretaciones y mecanismos de herencia propuestos un síntoma del problema entre tipos y herencia:

“El origen de muchos de estos problemas reside en que la estructura de la herencia, usada para compartir código, y la jerarquía de subtipos conceptual, que se origina de la especialización, *no* son lo mismo. Ambas se sitúan en diferentes niveles de abstracción en un sistema: la herencia se preocupa de la implementación de las clases, mientras que la jerarquía de subtipos se encarga del comportamiento de las instancias (cómo son vistas desde fuera, desde otros objetos)” [America, 1987; p. 238].

Si bien America habla únicamente de la especialización, la caracterización del problema es aplicable igualmente a la extensión, que de hecho implícitamente usa en la argumentación siguiente:

“Es posible que en muchos casos la relación jerárquica inducida por la herencia y por los subtipos coincida, pero ciertamente no siempre es el caso. Por un lado, en muchos casos es muy posible definir una clase que realmente especializa el comportamiento de otra clase, pero usa una estructura completamente distinta de variables y diferente código incluso para métodos con el mismo nombre (obteniéndose subtipos sin herencia). Por otro lado, es también posible que, simplemente añadiendo algunos métodos a una clase existente, incluso el comportamiento de los antiguos métodos sea esencialmente modificado (los nuevos métodos pueden asignar variables de un objeto en modos tales que un invariante del que dependen

antiguos métodos sea violado). En el último caso, la nueva clase no puede decirse que dé lugar a un subtipo de la antigua, obteniéndose herencia sin subtipos". [America, 1987; p. 238].

La misma idea aparece en [Snyder, 1986a; p. 41], aunque en este caso por una razón distinta: para esconder la dependencia de un descendiente sobre sus ascendientes indirectos por herencia. [Johnson, 1986; p. 317] también identifica que la herencia de implementación nada tiene que ver con la definición de tipos.

Teniendo en cuenta que la programación con componentes tiende de forma natural a separar interfaz e implementación, la solución más sencilla para el diseño del sistema de tipos en XC es identificar la definición de tipos de datos con la definición de interfaces de objetos y componentes. Con ello se consigue evitar la dualidad entre tipos de datos e interfaces que es habitual en muchos lenguajes tradicionales cuando intentan implementar componentes. La Figura 3-7 del apartado 3.4.4 *Envío de mensajes* muestra intuitivamente esta idea. La separación estricta entre tipos e implementación da mayor flexibilidad al lenguaje y, a la vez, trata con más naturalidad el concepto de interfaz asociada a los componentes.

7.3.3 Jerarquías inversas

Las jerarquías inversas son un ejemplo interesante y práctico, que subraya las ideas que se acaban de introducir sobre la conveniencia de separar tipos e implementación. Consideremos un tipo `MutableString`, que incluye un mensaje `setvalue:` para cambiar el contenido de la cadena de texto. Si queremos construir a partir del tipo anterior un nuevo tipo `String` inmutable, debe ser posible eliminar el mensaje `setvalue:` usado para cambiar el contenido. En muchos lenguajes orientados a objetos como C++ o Java esto no es posible. Definir un cuerpo vacío al método asociado a `setvalue:` en `String` sería propenso a confusión. Eliminar el mensaje `setvalue:` en el subtipo `String` impediría que una asignación polimórfica de una variable del tipo `String` al tipo `MutableString` fuese correcta sintácticamente. En lenguajes que mezclan herencia de tipos y herencia de implementación no es posible encontrar una solución satisfactoria. Si se separan ambos conceptos, entonces surge una solución natural, mostrada en la Figura 7-1.

Desde el punto de vista de los tipos, `MutableString` debe ser un subtipo de `String`, porque el primero es una extensión del segundo. El tipo `MutableString` extiende el tipo `String` con el mensaje `setvalue:`. En cambio, desde el punto de vista de la implementación, la jerarquía de implementación puede ser justamente la inversa: donde `String` hereda de `MutableString`.

La organización mediante jerarquías inversas muestra que la implementación `MutableString` satisface tanto el tipo `String` como `MutableString`, y que la implementación de `String` únicamente satisface el tipo `String`, aunque herede de `MutableString` e incluya el método heredado `setvalue:`. Por tanto, para un sistema de tipos que compruebe únicamente los tipos y no las implementaciones, no es posible enviar el mensaje `setvalue:` a un objeto de tipo `String`, pues éste no contiene tal método, aún cuando la implementación puede que sí lo contenga. Esta última información no concierne al sistema de tipos.

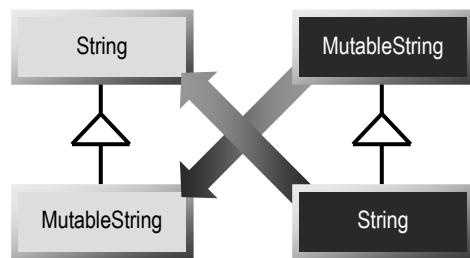


Figura 7-1 Jerarquías inversas.

La separación entre tipos e implementación facilita la construcción de jerarquías inversas con las propiedades adecuadas. El tipo `MutableString` es una extensión del tipo `String`. Por el contrario, la implementación puede ser precisamente la inversa: la implementación de `String` hereda de `MutableString`. En ambos casos cada implementación sigue el tipo o protocolo identificado con su propio nombre.

En XC es posible representar jerarquías inversas con comodidad. El sistema de tipos saca así más provecho de las propiedades de extensión y especialización de la herencia. La separación entre tipos e implementación confiere una semántica de extensión a la herencia de tipos —que no da lugar a excepciones sintácticas— con una implementación de conveniencia para `String` que hereda de `MutableString` —y que es una especialización—. El resultado final obtiene lo mejor de ambas propiedades, es decir, produce herencia de implementación mediante especialización que no genera excepciones sintácticas. Suponiendo que el valor de un `String` se almacena como una cadena C con tipo `CString`, en XC se obtiene:

```
protocol String
{
    CString value;
    Unsigned length;
}

protocol Mutablestring extends: String
{
    void setvalue: CString str;
}

prototype MutableString implements: MutableString
{ // ... }

prototype String extends: MutableString implements: String
{ // ... }
```

XC mantiene dos jerarquías de herencia completamente separadas. La herencia múltiple de tipos aplicada a la declaración y definición de protocolos y la herencia simple de implementación usada en los prototipos. El prototipo `MutableString` no tiene tipo definido por sí mismo. Toma el tipo de la cláusula `implements` que, por claridad, se ha mostrado explícitamente en el ejemplo. El tipo es nombrado, es decir, el compilador busca explícitamente un protocolo que indique a qué firmas de mensaje asociar los métodos definidos en el prototipo. El compilador entonces valida los métodos definidos con las firmas de los mensajes de los protocolos que adopta. A partir de entonces la implementación deja de tener significación para el compilador, que se centra únicamente en las definiciones de tipo dadas por los protocolos. El prototipo `MutableString` tiene asociados dos protocolos `String` y `MutableString`, es decir, puede ser asignado tanto a variables cuyo tipo sea `String` o `MutableString`. Nótese que la regla es independiente de si el protocolo `MutableString` hereda o no de `String`. Si no fuera el caso, la única manera que tendría el prototipo `MutableString` de adoptar los tipos `String` y `MutableString` sería listando ambos tras la cláusula `implements`. Es decir, la herencia de tipos puede verse como una forma compacta de indicar conjuntos de tipos que implementa un prototipo dado.

7.3.4 Comprobación semántica de tipos

En general, los lenguajes y los estudios teóricos sobre sistemas de tipos ignoran la comprobación de los aspectos semánticos de los tipos de datos, debido a que ya plantean suficientes dificultades únicamente los aspectos sintácticos. Con la tecnología actual, la comprobación semántica de tipos se encuentra en su infancia. Eiffel es uno de los pocos lenguajes que tiene soporte directo en el lenguaje para facilitar la comprobación semántica. Meyer propone el *diseño por contrato* [Meyer, 1997; pp. 331-410] para identificar aspectos semánticos en la definición de clases. La relación entre una clase y cualquiera de sus clientes debe verse como un contrato que ambas partes han de cumplir. El contrato se especifica usando código en Eiffel mediante precondiciones, poscondiciones e invariantes. Las precondiciones se asocian a cada método y se ejecutan antes del cuerpo de éste. Las poscondiciones también se asocian a cada método y se ejecutan después del cuerpo. Las invariantes se asocian a cada clase y se comprueban en momentos estables de la vida de los objetos: después de la creación de éstos, antes de ejecutar un método y después de ejecutar un método.

Existen también varios estudios sobre la validez semántica de los subtipos asociados a un tipo de datos. En particular, es interesante el mostrado en [Lamping, 1993]. Propone algunas extensiones sintácticas para comprobar algunos aspectos semánticos de los tipos heredados usando el sistema de tipos. Identifica la existencia de dos interfaces distintas: el *protocolo del cliente* y el *protocolo de especialización* usado por descendientes. Los descendientes deben tener un conocimiento más extenso de sus ancestros para asegurar compatibilidad semántica [Lamping, 1993; pp. 201-202]. Desde un punto de vista diferente, [Liskov & Wing, 1994] proponen describir la semántica de los tipos y subtipos (*behaviors*) mediante descripciones formales que puedan ser verificables con precondiciones, poscondiciones e invariantes. Se advierte por ejemplo que semánticamente un entero de 32 bits no es un subtipo de un entero de 64 bits, debido a diferencias observables de comportamiento. Un entero de 32 bits puede dar lugar a un desbordamiento donde un entero de 64 bits funciona correctamente [Liskov & Wing, 1994; pp. 1835-1836]. [Stata & Guttag, 1995] extienden el trabajo de Liskov y Wing identificando dos aspectos importantes de la herencia semántica. Por un lado, notan que las invariantes aplicables a un cliente no tienen porqué coincidir con las invariantes de un descendiente [Stata & Guttag, 1995; pp. 210-211], debido a que el protocolo de especialización de un descendiente puede requerir romper invariantes válidos para los clientes. Por otro lado, proponen un razonamiento más modular acerca de las invariantes que deben cumplirse durante una especialización. Subdividiendo las variables de instancia y métodos heredados en grupos y creando para ellos reglas de verificación formales más sencillas usando precondiciones, poscondiciones e invariantes.

[Szyperski, 1998; pp. 111-114] identifica una analogía interesante, que surge de la evaluación de las ideas de Stata y Guttag sobre la comprobación semántica parcial de tipos. La analogía es como sigue: para hacer más modular y sencilla la verificación semántica, Stata y Guttag proponen particionar estado y operaciones relacionadas de una clase y verificar los subconjuntos por separado. Szyperski evalúa que, desde el punto de vista de un sistema de componentes, es posible separar físicamente los grupos de Stata-Guttag y crear diferentes clases que se comuniquen mediante composición. La forma de resolver la aparición de múltiples `self` como resultado de la partición de una clase en varias, es resuelta añadiendo variables de instancia que apunten a las distintas clases³³. El aspecto interesante de la analogía es que transforma herencia en composición, más apropiada para diseños con componentes. Como también reconoce [Szyperski, 1998; p. 113], su punto débil es el notable aumento de la percepción de complejidad de los diseños.

La analogía cobra sentido cuando se observa desde el punto de vista de las categorías, tal y como están implementadas en XC. El modelo de objetos con extensiones mediante categorías es capaz de tratar a cada una de ellas como una parte de un objeto. Facilita seguir describiendo un objeto único sin incurrir en el coste de la separación explícita en múltiples objetos, ni en la subdivisión informal usando grupos.

³³ Obviamente, como ya identificaba [Bardou & Dony, 1996], esta estrategia no es buena porque genera dependencias exponenciales con el número de partes de objetos en juego.

Las categorías tienen otras ventajas, como por ejemplo permitir extensiones adicionales sin complicar en exceso el modelo de objetos percibido. La Figura 7-2 muestra las ideas anteriores.

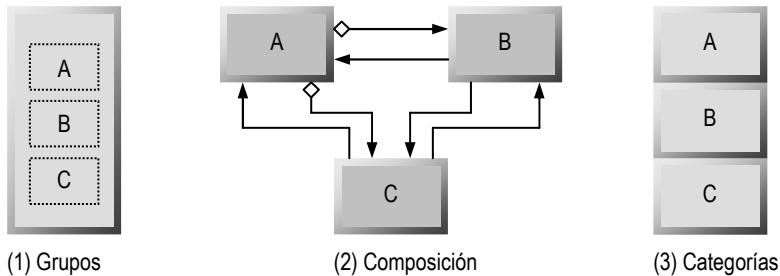


Figura 7-2 Grupos, Composición y Categorías.

(1) Los grupos corresponden con la subdivisión de grupos de métodos de Stata-Guttag. (2) La composición sugerida por Szyperski para evitar herencia. (3) La solución mediante categorías de XC. Adaptado y extendido de [Szyperski, 1998; p. 113].

El soporte para comprobación semántica en XC se realiza con la ayuda de varias funcionalidades básicas del lenguaje. El apartado 6.3.2 *Control de calidad* presenta varios ejemplos de comprobación semántica. Las precondiciones y poscondiciones se construyen con ayuda de métodos anteriores y posteriores. Algunos de los estudios sobre comprobación semántica de tipos usan lenguajes formales para describir las propiedades semánticas de cada implementación. Los lenguajes formales son más difíciles de asimilar por los programadores. Por esa razón, y al igual que en Eiffel, se opta por indicar la información de precondiciones y poscondiciones en el mismo lenguaje fuente, imperativamente. Las comprobaciones semánticas sirven para mejorar la calidad final de la implementación entregada a clientes, y para estar de acuerdo —al menos— en el comportamiento de aquellos aspectos controlados por la comprobación. Pero hoy por hoy no existen técnicas que garanticen que una comprobación semántica sea completa y, por tanto, que pueda garantizar que un subtipo es válido. Por esa razón, y dado que ese código puede llegar a ser bastante más complejo que la implementación original que comprueba, debe tomarse a la comprobación semántica como una técnica que aporta calidad y precisión, pero no asegura ausencia de errores. Es por ello que no hay que subestimar el coste de realizar una buena comprobación.

Eiffel incluye sintaxis específica para indicar las precondiciones y poscondiciones. A falta de mayor investigación en este terreno, XC opta por no introducir sintaxis que quizás haya que cambiar a posteriori, optando por simularlas usando metamétodos, que es una construcción básica más general y aplicable igualmente a otros propósitos. La distribución del código de un prototipo en varias categorías admite aplicar si se desea las ideas de Stata-Guttag sobre la comprobación semántica parcial. En este caso, otra vez XC usa una característica básica del lenguaje más general para dar soporte a esa funcionalidad sin necesidad de hacer más complejo el lenguaje. En este sentido, y puesto que XC no establece una política determinada para la comprobación semántica, puede ser un lenguaje destino apropiado para ensayar distintas opciones de descripción de comprobaciones de semántica de tipos. La aproximación de XC a la descripción de información semántica es a la vez novel y flexible, tanto más cuando lo consigue sin incorporar ningún nuevo concepto al lenguaje. Por ejemplo, encapsulando en categorías el código de comprobación semántica, es posible entregar su código fuente a los posibles clientes, sin necesidad de entregar el código fuente de la implementación objeto de la comprobación. Los clientes pueden decidir incorporar o no la comprobación semántica incluyendo o excluyendo la declaración de la categoría.

7.4 Tipos avanzados

Este apartado explica conceptos más avanzados del sistema de tipos de XC. Un sistema de tipos moderno debe hacer frente a problemas conocidos en los sistemas de tipos tradicionales, que impiden la comprobación estricta de tipos de importantes familias de programas comunes. Existen sólo unos pocos lenguajes experimentales que incorporen simultáneamente separación entre tipos e implementación, los tipos covariantes que se ven en este apartado, y los tipos genéricos que se verán posteriormente. La aplicación del conjunto de estas técnicas a un lenguaje para programación de sistemas es novedosa. El resultado es un sistema de tipos más flexible, capaz de comprobar estáticamente familias más amplias de programas usuales manteniendo la eficiencia en su implementación.

7.4.1 El problema de los métodos binarios

La comprobación semántica no asegura la generación de subtipos porque no es completa. Desafortunadamente, ni siquiera sintácticamente se puede afirmar que se mantiene la relación entre tipos y subtipos, relación necesaria para asegurar una sustitución válida en una variable polimórfica siguiendo la definición de subtipo (Def. 7-13).

El problema de los métodos binarios surge de la observación de una anomalía que aparece en la definición de métodos binarios, como los operadores igual, mayor o menor, y que involucran a dos objetos. Fue identificado inicialmente en [Cook, 1989] junto a otras anomalías del sistema de tipos de Eiffel. Una descripción más extensa se encuentra en [Bruce *et alii*, 1996] y [Bruce, 1996]. Consideremos el tipo `Point` mostrado a continuación en Java:

```
class Point
{
    int x, y;

    boolean isEqualTo (Point p)
    {
        return (this.x == p.x) && (this.y == p.y);
    }

    // ...
}
```

Supongamos ahora que queremos extender `Point` con colores, creando un nuevo prototipo `ColorPoint` que herede de `Point`:

```
class ColorPoint extends Point
{
    Color color;
    boolean isEqualTo (Point p)
    {
        ColorPoint cp = (ColorPoint) p;
        return (this.x == cp.x) && (this.y == cp.y) && (this.color == cp.color);
    }
    // ...
}
```

Como `ColorPoint` hereda de `Point`, es posible realizar una manipulación polimórfica de puntos, de manera que `p2` haga referencia a `cp`. Sin embargo, el siguiente código no funciona:

La variable `p2` contiene un objeto de tipo `ColorPoint`. El método que se ejecuta al enviar el mensaje `isEqualTo` es el definido en el prototipo `ColorPoint`, pero el objeto pasado como parámetro en `p` es únicamente un `Point`, y fallará en tiempo de ejecución el conversor de tipo.

Lo que está ocurriendo es que el método `isEqualTo` no está definido para objetos cuyo tipo sea `Point`, sino sólo para objetos con tipo `ColorPoint`. Para que el cuerpo del método estuviese definido siempre, el método debería rescribirse como:

```
boolean isEqualTo (ColorPoint cp)
{
    return (this.x == cp.x) && (this.y == cp.y) && (this.color == cp.color);
```

Nótese el cambio en el tipo del argumento del método `isEqualTo` de `Point` a `ColorPoint` sobre el *mismo* método heredado. En muchos lenguajes —de los que Java y C++ son dos ejemplos— este cambio está prohibido. En Java y C++ es posible definir un método sobrecargado con el argumento de tipo `ColorPoint`, la diferencia es que *no* sustituye al método con argumento de tipo `Point` heredado.

7.4.2 Covarianza y contravarianza

El problema anterior se caracteriza mejor si tenemos en cuenta el comportamiento del tipo de los argumentos a medida que se definen nuevos tipos de datos heredados.

- ❖ **Def. 7-14.** Un cambio *covariante* de un tipo de datos es un cambio de un tipo por un subtipo de él.
- ❖ **Def. 7-15.** Un cambio *contravariante* de un tipo de datos es un cambio de un tipo por un supertipo de él.

Un ejemplo inspirado en [Bruce, 1996; pp. 8-9] sirve para aclarar estos conceptos. Supongamos una función `length` que, a partir de una cadena de texto de entrada, devuelve su número de caracteres. Siguiendo la misma nomenclatura que en la ecuación (7-1), podemos representar el tipo de esta función como:

$$(7-9) \quad \text{length} = \text{Msg}(a_1 : \text{String}) : \text{Integer}$$

ya que un mensaje y una función definen la misma información de tipo. Generalizando, una función f con argumento de tipo R y valor de retorno de tipo U , se representa como:

$$(7-10) \quad f = \text{Msg}(R) : U$$

Siguiendo la definición de subtipo (Def. 7-13) podemos evaluar qué pasaría si usamos una función f' en lugar de f en una expresión. R se puede sustituir por S si $S <: R$, y U se puede sustituir por T si $T <: U$. Intuitivamente debería poder mantenerse la siguiente relación:

$$(7-11) \quad f' = \text{Msg}(S) : T <: f = \text{Msg}(R) : U \quad \text{si } S <: R \wedge T <: U$$

Con el valor de retorno de f' en T el resultado es el esperado. Si f' se hace pasar por f , no importa que devuelva un valor cuyo tipo tiene más mensajes que el valor esperado, ya que al cumplirse $T <: U$ todos los mensajes de U que espera el código original que llamaba a f están disponibles en T . Se dice entonces que T varía de forma *covariante* respecto a la relación $<:$.

Pero con el argumento *no* ocurre lo mismo. Para que f' se pueda usar en lugar de f en una expresión arbitraria, f' debe aceptar los *mismos* parámetros que f . Es decir, f' debe admitir un parámetro de tipo R . Entonces, f' no puede requerir un argumento de tipo S , porque S define más mensajes que R y R *no tiene*. Es decir, enviar un parámetro de tipo R cuando f' espera un parámetro de tipo S es un error y debemos reformular (7-11) como:

$$(7-12) \quad f' = \text{Msg}(R) : T <: f = \text{Msg}(R) : U \quad \text{si } T <: U$$

La ecuación (7-11) es teóricamente válida si *invertimos* la relación de subtipo de $S <: R$ a $R <: S$. Para evitar confusión, notemos otra vez que el objetivo es sustituir la función f por f' , no sustituir sus argumentos. Si $R <: S$, entonces R define más mensajes que los usados por S . Por tanto, es posible enviar un parámetro R con más mensajes que los requeridos por S , ya que nunca se va a producir un error de envío de mensaje inválido. Por esa razón se dice que los argumentos varían de forma *contravariante* respecto a la relación $<:$. La formulación correcta de (7-11), que expresa la relación covariante de los valores de retorno y la relación contravariante de los argumentos es:

$$(7-13) \quad f' = \text{Msg}(S) : T <: f = \text{Msg}(R) : U \quad \text{si } R <: S \wedge T <: U$$

En los lenguajes orientados a objetos generalmente se encuentra implementada la ecuación (7-12), ya que no resulta útil usar un supertipo como argumento. (7-12) y (7-13) son equivalentes si para la relación $<:$ se considera que un tipo es siempre subtipo de sí mismo, es decir, que $R <: R$. Por esa razón, la ecuación (7-8) que define un subtipo usa la notación compacta, mostrando la posible relación covariante de los valores de retorno, pero no entrando en detalles acerca de los tipos de los argumentos, que no varían.

Java y C++ definen simultáneamente mensajes y métodos. Sus argumentos se comportan de manera contravariante. Véase que esta restricción impone la pérdida del tipo de los argumentos y obliga a una conversión de tipos adicional dentro de los métodos. Sería deseable que el sistema de tipos pudiese aceptar cambios covariantes en los argumentos, para limitar las operaciones de conversión y soportar métodos binarios. También son útiles los cambios covariantes en los valores de retorno. Como hemos visto, estos últimos surgen con naturalidad. Originalmente en C++ los valores de retorno eran contravariantes. Así, una hipotética operación de suma de coordenadas para objetos de tipo `ColorPoint` debía devolver un objeto `Point`, obligando a operaciones de conversión de tipo adicionales al usar tipos derivados de `Point`. Por ejemplo, usando C++:

```
class Point
{
    virtual Point * add (Point *p)
    { // ... }

    // ...

class ColorPoint : public Point
{
    Point * add (Point *p)
    {
        ColorPoint *cp = (ColorPoint *) p;
        x += cp->x; y += cp->y;
        return cp;
    }

    // ...
}
```

Versiones más recientes de C++ tienen un sistema de tipos un poco más flexible. Aceptan cambios covariantes del valor de retorno para expresar operaciones que de forma natural devuelven el mismo tipo de datos siendo definido. Otra extensión, esta vez a los operadores de conversión, ayuda a comprobar dinámicamente si la transformación es correcta o no con `dynamic_cast`.

```
ColorPoint * add (Point *p)
{
    ColorPoint *cp = dynamic_cast<ColorPoint *>(p);

    if (cp != NULL)
    {
        x += cp->x; y += cp->y;
        return cp;
    }
    else
        // error
}
```

Eiffel es un lenguaje más flexible todavía. Además, admite el cambio covariante de los argumentos y elimina, por tanto, el conversor de tipo `Point` a `ColorPoint`. También considera correctos los cambios covariantes de los tipos de los valores de retorno y variables de instancia.

El problema de los métodos binarios afecta esencialmente a las definiciones polimórficas recursivas (Def. 7-10) en los argumentos de los métodos, es decir, a cambios covariantes en los argumentos. Genera errores no esperados en tiempo de ejecución, que no son detectados por las reglas de tipos de datos de los compiladores. Pero el problema de los métodos binarios es más fundamental. En presencia de definiciones de argumentos de métodos recursivas, una implementación que herede de otra *no crea un subtipo válido*. Ocurre independientemente de si se trata de un método binario o con más argumentos, aunque por razones históricas a estos métodos se les sigue denominando ‘binarios’.

Volviendo al ejemplo inicial del apartado 7.4.1, el tipo `ColorPoint` no es un subtipo válido de `Point` porque pueden existir expresiones en las que se generen errores en tiempo de ejecución. No es un subtipo incluso si —como hace Eiffel— el sistema de tipos supone que se generan subtipos al realizar cambios covariantes y admite llamadas polimórficas³⁴. Si se cambia la firma del método binario para que incluya un cambio covariante en el tipo de sus argumentos, *tampoco* se garantiza que se obtiene un subtipo válido. Porque al aceptar como válida la asignación de `cp` a `p2` —cuyas firmas para el método `isEqualTo` difieren en el tipo del argumento— simplemente se está dejando que la sentencia condicional siguiente encuentre otra vez el error en tiempo de ejecución. Usando una hipotética variante de Java con tipos covariantes aún se produciría un error en el mismo ejemplo:

```
Point      p = new Point();
ColorPoint cp = new ColorPoint();           // Asignación polimórfica
Point      p2 = cp;                         // Error en tiempo de ejecución!
if (p2.isEqualTo (p))                      // p no tiene el tipo ColorPoint
{ // ... }
```

El error, claro está, se encuentra en suponer que la asignación de `cp` a `p2` es válida. No lo es porque `ColorPoint` no es un subtipo de `Point`.

Para garantizar que no se va a producir un error en tiempo de ejecución, debe limitarse el envío del mensaje `isEqualTo` a un objeto cuyo tipo sea *exactamente* `ColorPoint`, usando únicamente como parámetro objetos de tipo `ColorPoint`. Igualmente, restringir a argumentos de tipo `Point` el mensaje `isEqualTo` enviado a objetos de tipo `Point`. El coste de la solución es la pérdida de la relación tipo-subtipo y la invalidez de la asignación polimórfica.

La anomalía de los métodos binarios es común en muchos lenguajes orientados a objetos. Los desarrolladores han tenido que dar soluciones parciales en el ámbito de cada lenguaje al implementar bibliotecas de clases. Por ejemplo, en [Taligent, 1994; p. 16] se describe informalmente el mismo problema, encontrado en la implementación de las bibliotecas de colecciones de Taligent. Su solución propuesta para C++ define dos métodos de comparación, usando sobrecarga en vez de herencia para asegurar que únicamente los métodos de comparación definidos son los apropiados. La clase `Point` declara el operador como sigue:

```
class Point
{
    // ...
    Boolean operator == (const Point &, const Point &);
```

La clase `ColorPoint` hereda de `Point` y declara los operadores ‘==’ como sigue:

```
class ColorPoint : public Point
```

³⁴ Meyer ha propuesto para Eiffel un mecanismo para detectar las llamadas que potencialmente pueden dar lugar a errores en tiempo de ejecución. Las denomina *catcalls* [Meyer, 1997; pp. 636-638].

```
// ...
Boolean operator == (const Point &, const Point &);
Boolean operator == (const ColorPoint &, const ColorPoint &);
}
```

Realizar la comparación entre objetos `Point` y `ColorPoint` en tiempo de compilación es errónea, pues no se define un operador ‘`==`’ para ello, siendo éste el comportamiento deseado. Pero si una variable de tipo `Point` contiene en tiempo de ejecución un objeto de tipo `ColorPoint`, se seleccionará la comparación entre objetos `Point` definida en `ColorPoint`. Esta selección puede ser problemática en otros métodos binarios, como veremos en el apartado 7.4.4 *Variables de instancia covariantes*, porque existe el riesgo de ruptura de invariantes en los tipos derivados.

7.4.3 El tipo covariante `Self`

Supongamos que se pretende tener un nuevo tipo de datos `ColorPoint3D` que herede de `ColorPoint` y que incluya una nueva coordenada `z`. En ese momento vuelve a surgir la necesidad de variar covariantemente el argumento del método `isEqualTo`, porque es necesario añadir la comprobación de la coordenada `z` en la igualdad. El sistema de tipos gana en flexibilidad si soporta la descripción de estos cambios covariantes, que son en la práctica bastante comunes. Usando XC es posible escribir la siguiente definición de protocolo para el tipo `Point`:

```
protocol Point extends: Object
{
    property Integer x;
    property Integer y;
    Boolean isEqualTo: Self p;
}
```

El tipo `Self` hace referencia al tipo de la variable `self`. En el protocolo `Point`, `Self` representa al tipo `Point`. Cuando se hereda el tipo `ColorPoint` de `Point`, la firma del mensaje `isEqualTo:` no cambia en el código fuente:

```
protocol ColorPoint extends: Point
{
    property Color color;
    Boolean isEqualTo: Self p;
}
```

Pero sí cambia el significado de `Self`. En el protocolo `ColorPoint`, `Self` se refiere al tipo `ColorPoint`, no al tipo `Point` original. Como la firma del mensaje se hereda de `Point`, el tipo `ColorPoint` se puede definir como:

```
protocol ColorPoint extends: Point
{
    property Color color;
}
```

Por supuesto, una implementación de `ColorPoint` debe redefinir el método `isEqualTo:`. El tipo `ColorPoint3D` se define análogamente. El tipo `Self` varía covariantemente con la herencia, y es también un tipo recursivo (Def. 7-10). `Self` se aplica a argumentos de mensajes, valores de retorno de mensajes, variables de instancia, y, en general, a cualquier expresión que requiera un tipo de datos. La implementación del método `isEqualTo:` de `ColorPoint` queda entonces como:

```
Boolean isEqualTo: Self cp
{
    return (self.x == cp.x) && (self.y == cp.y) && (self.color == cp.color);
}
```

`Self` expresa el cambio covariante de forma más precisa, clara y concisa, sin necesidad de usar conversores de tipo. Fue propuesto originalmente en Trellis/Owl [Schaffert *et alii*, 1986; p. 10] con el nombre `mytype`, apareciendo luego también en POOL-I [America & van der Linden, 1990; pp. 166-167], donde se llama `MYTYPE`; en BETA [Lehrmann Madsen, Magnusson & Møller-Pedersen, 1990; p.

148] con el nombre `ThisClass`; y en Eiffel con la construcción `like Current` [Eiffel, 1997; pp. 601-604]. [Bruce *et alii*, 1995; pp. 228-231] y [Bruce, 1996; pp. 24-26] lo denominan `MyType` y presentan una discusión más amplia sobre las implicaciones de su uso. En [Bruce 1997; pp. 4-7] y [Bruce, Odersky & Wadler, 1998; pp. 535-538] se usa el término `ThisType`, al aplicar el mismo concepto a una propuesta de extensión del lenguaje Java. En XC se ha optado por el término `self` al representar el tipo de la pseudo-variable `self` más intuitivamente.

La flexibilidad obtenida permite manejar en XC métodos binarios y detectar en tiempo de compilación el error en el siguiente código, que es válido en C++ o Java:

```
Point p = Point.clone;
ColorPoint cp = ColorPoint.clone;

cp.isEqualTo: p; // error en compilación! p no tiene el tipo ColorPoint
```

Otras ventajas se estudian en próximos apartados. Desafortunadamente, el problema de la asignación polimórfica es independiente del uso del tipo `self`. Por esa razón, el fragmento siguiente sigue dando un error en tiempo de ejecución, no siendo detectado por el compilador:

```
Point p2 = cp;
p2.isEqualTo: p; // error en ejecución! p no entiende el mensaje color
```

Recordemos que el error se produce porque `ColorPoint` no es un subtipo de `Point` y la relación `ColorPoint <: Point` no es correcta. Es importante comprender que el problema surge porque `ColorPoint` puede heredar de `Point`, y se admite una asignación polimórfica entre variables de tipo `Point` y `ColorPoint` que en realidad no es correcta. Si `Point` es `final`, entonces no se produce esta anomalía, porque en este último caso nunca se puede asignar polimórficamente una variable de tipo `Point` a algún subtipo que redefina o cree nuevos mensajes, ya que se prohíbe la herencia a partir de `Point`.

7.4.4 Variables de instancia covariantes y ruptura de invariantes

Introduciremos las variables de instancia covariantes con un ejemplo. Supongamos un tipo para listas enlazadas y doblemente enlazadas que tenga un mensaje `concat:` para unir dos listas. Usando XC:

```
protocol List extends: Object
{
    property Object value;
    property List next;

    List concat: List l;

    protocol DoubleList extends: List
    {
        property List prev;
    }
}
```

El mensaje `concat:` es binario. La propiedad `next` define dos mensajes de acceso a un valor de tipo `List`. Expandiendo esa propiedad a su conjunto de mensajes equivalentes, se puede ver que el mensaje `setNext:` es también un mensaje binario.

```
List next;
void setNext: List next;
```

Lo mismo ocurre con la propiedad `prev` de `DoubleList`. Interesa que `DoubleList <: List` para poder manipular listas polimórficamente sin pensar en su implementación particular. El código de una implementación eficiente del método `concat:` de `DoubleList` en un lenguaje sin tipos covariantes —que supondremos similar a XC en su sintaxis— contiene algunas sorpresas:

```

prototype List extends: Object
{
    property Object value;
    property List next;

    List concat: List l
    {
        List aux = self;

        while (aux.next != null) aux = aux.next;
        aux.next = l;
        return self;
    }
}

prototype DoubleList extends: List
{
    property List prev;

    List concat: List l
    {
        DoubleList aux, aux2;

        aux = self;
        while (aux.next != null) aux = aux.next;
        if (aux2 ?= l)
        {
            aux2.prev = aux;
            return self;
        }
        else
            // error!
    }
}

```

Resulta sencillo ahora identificar el problema de los mensajes binarios en la implementación del método `concat:` en `DoubleList`. Si el parámetro `l` es de tipo `List` se produce un error en tiempo de ejecución. El parámetro `l` debe cambiar covariantemente, y ser de tipo `DoubleList` para tener acceso al mensaje `prev`.

```

List concat: DoubleList l
{ // ... }

```

Además, la variable `aux` varía covariantemente dentro del algoritmo del método `concat:`, que permanece esencialmente inalterado. Véase también que en una lista doblemente enlazada no tiene mucho sentido usar el mensaje `concat:` que se hereda de `List`—cuyo argumento es de tipo `List`—porque en ese caso se estaría enlazando una lista enlazada a una lista doblemente enlazada, sin actualizar el campo `prev` del nodo insertado, que no existe. Esta circunstancia se transforma en un error latente en la estructura de datos doblemente enlazada, que ya no puede ser recorrida desde el final hasta el principio por la falta de nodos `prev` intermedios.

Aunque no hay un error explícito inmediato, se produce la *ruptura de un invariante* en la implementación del prototipo `DoubleList`: la actualización correcta de los campos de los nodos de la lista³⁵. La estrategia de definir métodos en los subtipos con argumentos de sus supertipos vista al final del apartado 7.4.1 con el operador ‘`==`’ no es aplicable a esta situación. Son frecuentes las estructuras de datos complejas donde la información de tipo heredada es un inconveniente —más que una ayuda— para definir nuevos tipos de datos correctamente. En definitiva, no sólo por la presencia de mensajes binarios, sino por la propia naturaleza de la implementación del tipo de datos —que en este caso se pretende eficiente— el tipo `DoubleList` no es un subtipo de `List`. Interesa entonces que el sistema de tipos nos ayude a evitar estos problemas, a la vez que nos facilite la escritura de código de una manera más natural.

³⁵ En lenguajes que consideran válida la sobrecarga de métodos con argumentos de distinto tipo, y si se definen dos métodos `concat` sobrecargados con argumentos `List` y `DoubleList`, entonces `DoubleList` hereda la versión de `concat` con argumento de tipo `List`. Al heredarse, inadvertidamente se corre el riesgo de ser llamado este último método para objetos de tipo `DoubleList` guardados en variables polimórficas de tipo `List` y romper la estructura interna de las listas doblemente enlazadas.

Usando `self` para dar tipo covariante al argumento de `concat:` y a la variable interna `aux` se obtiene un código más limpio y expresivo. Aplicándolo a los tipos `List` y `DoubleList` se tiene en XC:

```
protocol List extends: Object
{
    property Object value;
    property self next;

    self concat: self l;
}

protocol DoubleList extends: List
{
    property self prev;
}
```

La definición de los protocolos es igual de compacta que en la versión anterior, pero a su vez es más rica. Gracias a `self` en `List` la propiedad `next` tiene tipo `List`. En `DoubleList` las propiedades `next` y `prev` tienen igual tipo `DoubleList`. La firma del mensaje `concat:` también varía indicando cuáles son los tipos válidos para su argumento en cada caso. Se describen dos protocolos relacionados por herencia, pero que no forman subtipos. Aunque el sistema de tipos pretendiese lo contrario, en realidad `List` y `DoubleList` no son subtipos. Es una vana determinación tratar de imponerlos. La implementación de `concat:` usa la variable `aux` con una definición de tipo covariante:

```
prototype List extends: Object
{
    property Object value;
    property self next;

    self concat: self l
    {
        self aux = self;
        while (aux.next != null) aux = aux.next;
        aux.next = l;
        return self;
    }
}

prototype DoubleList extends: List
{
    property self prev;

    self concat: self l
    {
        self aux = self;
        while (aux.next != null) aux = aux.next;
        aux.next = l; l.prev = aux;
        return self;
    }
}
```

Usando variables y argumentos covariantes, el método `concat:` recobra su simplicidad, porque no es necesario lidiar con un sistema de tipos demasiado restrictivo. También evita el problema de heredar inadvertidamente un método `concat:` con tipo `List` potencialmente dañino. Todo ello sin renunciar a una comprobación estricta de tipos más completa.

En resumen, la dificultad semántica con las definiciones polimórficas tiene su origen en la falsa suposición de que el mecanismo de herencia genera subtipos de datos sustituibles en expresiones polimórficas, tal y como la ecuación (7-12) da a entender. Lo cierto es que no siempre se generan subtipos y es conveniente cambiar las reglas de verificación de tipos para la herencia, si se pretende escribir código flexible que pueda ser comprobado en tiempo de compilación con mayor éxito.

7.4.5 Encaje o matching

Para poder comprobar en tiempo de compilación los ejemplos del apartado anterior, debe dejarse a un lado la creación de subtipos, porque es demasiado restrictiva. Una relación más débil entre un ascendiente y un descendiente por herencia es la relación de *encaje* o *matching* [Bruce, 1996;

pp. 43-44]. La relación es más débil que los subtipos porque considera correcta la relación entre un ascendiente y un descendiente una vez traducido el tipo covariante `self`.

- ❖ **Def. 7-16.** La *traducción* de un tipo T respecto de una variable de tipo V es una sustitución de todas las apariciones libres de la variable V por un tipo S , dentro de la definición del tipo T .

$$(7-14) \quad T[S/V]$$

que se lee: en el tipo T traducir por S la variable V . En particular, para el tipo `List` del apartado anterior, sustituir por `List` la variable `Self`³⁶.

$$(7-15) \quad List[List / Self]$$

Por ejemplo, para el mensaje `concat`: del mismo ejemplo anterior, que es parte del tipo `List`, y cuyo tipo es:

$$(7-16) \quad concat = Msg(l : Self) : Self$$

la traducción de `Self` por `List` se convierte en:

$$(7-17) \quad concat[List / Self] = Msg(l : List) : List$$

Como `Self` aparece en posiciones contravariantes, el tipo `List` no genera subtipos. Por tanto, no diremos que la relación de herencia genera subtipos en este sistema de tipos, sino que realiza *encajes* o *matching* entre tipos de datos [Bruce, 1996; p. 27]:

- ❖ **Def. 7-17.** Un tipo de datos T_1 *encaja* en otro tipo de datos T_2 si T_2 define un subconjunto de los mensajes definidos en T_1 . Se identifica con el símbolo $<\#$:

$$(7-18) \quad T_1 = Obj\{m_i : T_i\}_{1 \leq i \leq k} <\# T_2 = Obj\{m_j : T_j\}_{1 \leq j \leq n} \quad \text{sii } n \leq k \wedge T_2 \subset T_1 \wedge T_j <: T_i$$

Nótese la similitud entre las ecuaciones (7-7) y (7-18). Intuitivamente, la relación $<\#$ realiza la misma labor que la relación $<$: con la salvedad que, en general, no genera subtipos. Los subtipos se generan sólo para un subconjunto de los casos que acepta la relación de encaje: aquellos en los que la sustitución de `Self` se produce únicamente en posiciones covariantes, es decir, como valor de retorno de mensajes. Podemos resumir entonces que dos tipos de datos S y T cumplen:

$$(7-19) \quad S <: T \Rightarrow S <\# T$$

$$(7-20) \quad S <\# T \not\Rightarrow S <: T$$

En la práctica, un sistema de tipos que use la relación de encaje realiza similares comprobaciones de tipo que un lenguaje orientado a objetos tradicional, pero esta vez sin suponer que se generan subtipos. En particular, debe tener especial cuidado en las asignaciones a variables polimórficas, que ocurren por defecto en la mayoría de los lenguajes orientados a objetos. Incluso si un tipo de datos T es correcto ahora, no está garantizado que en un futuro lo siga siendo si es extensible mediante herencia, puesto que un futuro tipo derivado S siempre puede potencialmente añadir un mensaje binario al tipo T . Eliminar asignaciones polimórficas implica fomentar asignaciones monomórficas, donde el tipo de datos de una expresión se considera exacto, independientemente de que el tipo se cree incrementalmente usando herencia.

En XC la situación se agrava, porque es posible extender un tipo de datos con una categoría, y no se prohíbe que ésta defina mensajes binarios. No por ello el lenguaje se vuelve más problemático que un lenguaje orientado a objetos tradicional. De hecho, en los lenguajes tradicionales *tampoco* se pueden definir esos subtipos de todas formas, aunque ellos supongan que sí. La flexibilidad adicional del

³⁶ Esta definición sugiere que el tipo `Self` es un caso particular de variables de tipo, pero con unas propiedades muy definidas.

sistema de tipos simplifica el código de herencia, al ser liberado de restricciones artificiales en el sistema de tipos impuestas para salvaguardar la relación $<:$ que, simplemente, no se preserva.

7.4.6 Comprobación de tipos covariantes y tipos exactos

Los tipos covariantes y los inconvenientes que aparecen al usar métodos binarios, requieren cierta atención por parte del sistema de tipos del compilador. El aspecto más importante es la exactitud de los tipos [Lehrmann Madsen, Magnusson & Møller-Pedersen, 1990; p. 147]. El mismo concepto se estudia en [Palsberg & Schwartzbach, 1990; pp. 153-155] implícitamente en su técnica de sustitución de tipos o *type substitution* para tipos genéricos, que se verá en el apartado 7.5.6.

- ❖ **Def. 7-18.** Un tipo es *exacto* si referencia únicamente a un tipo T , no a ningún otro tipo S donde se cumpla que $S <: T$ o $S < \# T$.

Interpretar los tipos de las expresiones como exactos, en vez de como tipos con potenciales subtipos, evita parte de los problemas asociados con los métodos binarios, que ocurren precisamente si se aceptan subtipos en las expresiones. En este último caso, las expresiones manejan variables polimórficas que, potencialmente, quizás contengan en un futuro subtipos con métodos binarios que generen errores en tiempo de ejecución. Si se limita el número de asignaciones polimórficas, se está asegurando que un número mayor de programas no contendrá esos potenciales errores de tipo.

Es posible interpretar la traducción de `self` en una expresión como un tipo con posibles subtipos, es decir, polimórficamente. Los tipos con potenciales subtipos a veces reciben el nombre de tipos *hash* o *hash types* [Bruce, 1996; pp.44-45]. La interpretación es en principio natural si por defecto las asignaciones del lenguaje son polimórficas. Pero, si se usa `self` en argumentos de mensajes, entonces no se generan subtipos válidos. Este es precisamente el núcleo del problema del sistema de tipos de Eiffel identificado en [Cook, 1989], que tiene una construcción más general que `self`, llamada `like`, usada para realizar cambios covariantes en los argumentos [Meyer, 1997; pp. 601-604]. Un prometedor camino para ir erradicando paulatinamente los errores de tipos como los anteriores pasa por interpretar las ocurrencias de `self` como tipos exactos.

Desde el punto de vista de la implementación del sistema de tipos en XC, resulta de especial importancia conocer el momento de aplicación de la operación de traducción de `self` y el momento de la resolución final de los tipos. Introduzcamos brevemente una notación simple para identificar la resolución de expresiones de tipo usadas en esta discusión. Si S y T son dos tipos, la expresión de resolución de tipo:

$$(7-21) \quad Exp(S, T) \mapsto U$$

indica que la expresión *Exp* con argumentos de tipo S y T se resuelve a un tipo U . Por ejemplo, el tipo de un mensaje con dos argumentos a_1 y a_2 y tipos respectivos T_1 y T_2 se resuelve a su tipo de retorno T_R :

$$(7-22) \quad Msg(a_1 : T_1, a_2 : T_2) : T_R \mapsto T_R$$

También aceptaremos que una expresión se resuelva a otra expresión diferente, manteniendo los tipos de los argumentos. Un ejemplo lo veremos un poco más adelante, al seleccionar distinta semántica de asignación dependiendo de los tipos pasados como argumentos.

$$(7-23) \quad Exp_1(S, T) \mapsto Exp_2(S, T)$$

La diferencia de interpretación en la comprobación de tipos covariantes se hace patente con el método `clone`: heredado de `Object`. Su declaración es la siguiente:

```
self clone;
```

En un lenguaje que interprete `self` como un tipo polimórfico —como hace por ejemplo Eiffel— es posible realizar la asignación siguiente entre los tipos `Point` y `ColorPoint` de ejemplo:

```
Point p = ColorPoint.clone;
```

El operador de asignación tiene precedencia a derechas. Primero se evalúa la expresión resultante de la llamada a `clone` —que tiene tipo `self`— y se traduce por `ColorPoint`. Este es intuitivamente el comportamiento esperado: el resultado de un mensaje cuyo tipo es `self` es el tipo del receptor del mensaje, en este caso el ejemplar por defecto de la familia `ColorPoint`. Luego se pasa a la asignación, donde la variable `p` acepta la asignación polimórfica a `ColorPoint`. Un sistema de tipos que interprete `self` como un tipo polimórfico realiza las siguientes operaciones principales con expresiones de tipo: primero traduce `self` y luego resuelve la expresión de asignación, cuyo tipo final es `Point`.

(7-24)

$$\begin{aligned} \text{clone} &= \text{Msg}() : \text{Self} \\ \text{clone}[\text{ColorPoint}/ \text{Self}] &= \text{Msg}() : \text{ColorPoint} \\ \text{Msg}() : \text{ColorPoint} &\mapsto \text{ColorPoint} \\ p &= \text{Point} \\ \text{Assign}(\text{Point}, \text{ColorPoint}) &\mapsto \text{PolyAssign}(\text{Point}, \text{ColorPoint}) \\ \text{PolyAssign}(\text{Point}, \text{ColorPoint}) &\mapsto \text{Point} \end{aligned}$$

En cambio, si se interpreta `self` como un tipo exacto, la sentencia anterior es inválida. XC implementa esta última semántica. En vez de resolver primero `self` en la expresión de mensaje, la evaluación de la expresión es ligeramente distinta ahora. La evaluación de `self` se retrasa hasta el último momento, al realizar la asignación, porque es necesario comprobar si `self` es compatible con `Point`. Si uno de los parámetros de tipo es `self`, se selecciona la asignación exacta en vez de la asignación polimórfica, dando finalmente un error de tipo, porque `Point` y `ColorPoint` son dos tipos distintos.

(7-25)

$$\begin{aligned} \text{clone} &= \text{Msg}() : \text{Self} \\ \text{Msg}() : \text{Self} &\mapsto \text{Self} \\ p &= \text{Point} \\ \text{Assign}(\text{Point}, \text{Self}) &\mapsto \text{ExactAssign}(\text{Point}, \text{Self}) \\ \text{ExactAssign}[\text{ColorPoint}/ \text{Self}] &= \text{ExactAssign}(\text{Point}, \text{ColorPoint}) \\ \text{ExactAssign}(\text{Point}, \text{ColorPoint}) &\mapsto \text{Error}!! \end{aligned}$$

En este caso se está usando implícitamente el tipo `self` para seleccionar la semántica de la asignación. Para realizar una asignación polimórfica si se interpreta `self` como un tipo exacto —como es el caso de XC— se han de realizar dos pasos. Nótese que, eligiendo esta opción, la asignación polimórfica podría potencialmente dar lugar a un error de tipos en tiempo de ejecución más adelante, pues `ColorPoint` define mensajes binarios.

```
ColorPoint cp = ColorPoint.clone;
Point      p = cp;
```

Ahora, la evaluación de `self` se atrasa hasta el momento de realizar la primera asignación, que resuelve la asignación exacta correctamente. Su resultado es `ColorPoint`. En la segunda sentencia, la asignación involucra a tipos no covariantes, por lo que se selecciona la asignación polimórfica.

(7-26)

```

clone = Msg() : Self
Msg() : Self ↪ Self
cp = ColorPoint
Assign(ColorPoint, Self) ↪ ExactAssign(ColorPoint, Self)
ExactAssign[ColorPoint/ Self] = ExactAssign(ColorPoint, ColorPoint)
ExactAssign(ColorPoint, ColorPoint) ↪ ColorPoint
p = Point
Assign(Point, ColorPoint) ↪ PolyAssign(Point, ColorPoint)
PolyAssign(Point, ColorPoint) ↪ Point

```

Con las mismas reglas, también se puede prohibir la asignación polimórfica, que evite los problemas con los métodos binarios:

```
Self cp = ColorPoint.clone;
Point p = cp;           // error!
```

Otra vez, la evaluación de **self** se demora siempre que sea posible. Si en el momento de realizar la asignación todos los parámetros de tipo son **Self**, no se resuelve **self**, porque el tipo resultante de la asignación se puede inferir y es precisamente **Self**. Es decir, **self** se resuelve en demanda si se necesita más información de tipo, pero sólo cuando es absolutamente necesario.

(7-27)

```

clone = Msg() : Self
Msg() : Self ↪ Self
cp = Self
Assign(Self, Self) ↪ ExactAssign(Self, Self)
ExactAssign(Self, Self) ↪ Self
p = Point
Assign(Point, Self) ↪ ExactAssign(Point, Self)
ExactAssign[ColorPoint/ Self] = ExactAssign(Point, ColorPoint)
ExactAssign(Point, ColorPoint) ↪ Error!!

```

La misma regla se aplica a otras construcciones del lenguaje. Por ejemplo, la declaración del mensaje **concat:** del tipo **DoubleList** del ejemplo anterior es:

```
Self concat: Self l;
```

Al aparecer **self** en una posición contravariante, existe el riesgo de usar un argumento de un tipo no apropiado con el mensaje **concat:** si **self** se interpreta como tipo polimórfico, ya que se trata de un mensaje binario. El ejemplo muestra un envío de mensaje que puede dar un error en tiempo de ejecución usando semántica polimórfica, pues se usa un argumento de tipo **List** cuando el receptor es de tipo **DoubleList**.

```
List      l = List.clone;
DoubleList d1 = DoubleList.clone;
d1.concat: l;
```

En cambio, si se interpreta **self** como un tipo exacto, el envío del mensaje **concat:** a **d1** es erróneo, porque interpretando **self** como un tipo exacto, **concat:** espera como argumento un tipo **DoubleList**, no un tipo **List**. La evaluación con detalle de las dos primeras sentencias ya se ha visto anteriormente y no nos detendremos en ellas. Como se espera, **l** toma el tipo **List** y **d1** el tipo **DoubleList**. Las expresiones de tipo que vienen a continuación analizan la viabilidad del envío del mensaje **concat:**. En este caso no se trata de una expresión de asignación, sino de una expresión de envío de mensaje. Antes de enviar un mensaje, se han de evaluar los tipos de los argumentos para ver si coinciden con aquellos declarados en el mensaje. La evaluación de cada argumento es similar a la asignación vista antes,

aplicándose la misma regla que demora la resolución de `Self` hasta que no quede otra opción. Igualmente, en caso de que aparezca `Self` en una expresión de evaluación de argumentos, se usará semántica exacta y, si no, se usará semántica polimórfica.

(7-28)

$$\begin{aligned}
 l &= List \\
 dl &= DoubleList \\
 concat &= \text{Msg}(a_1 : Self) : Self \\
 \text{Arg}(List, Self) &\mapsto \text{ExactArg}(List, Self) \\
 \text{ExactArg}[DoubleList / Self] &= \text{ExactArg}(List, DoubleList) \\
 \text{ExactArg}(List, DoubleList) &\mapsto \text{Error}!!
 \end{aligned}$$

Interpretar el tipo `Self` como exacto implícitamente tiene la ventaja de no requerir dos tipos explícitamente distintos en el lenguaje: los tipos exactos y los tipos que pueden tener subtipos o *hash types*. Propuestas anteriores realizan una distinción sintáctica entre ambos, como por ejemplo en [Bruce, 1996], [Bruce, 1997], [Bruce, Odersky & Wadler, 1998] y [Bruce & Vanderwaart, 1999]. XC interpreta `Self` siempre como un tipo exacto.

En las experiencias que se han tenido desarrollando en XC, la tendencia natural es a que las expresiones que involucren `Self` tengan siempre tipos exactos. La presencia de la palabra clave identifica con claridad el propósito del código, que pretende ser covariante y, a la vez, recuerda que con variables de ese tipo no es posible obtener subtipos válidos. Por esa razón, XC opta por dar una novedosa interpretación implícita a los tipos exactos, que es obtenida por el compilador fácilmente examinando los tipos de una expresión. Durante el desarrollo, el tipo de las expresiones está siempre en la mente del desarrollador. Cuando éste pretende usar `Self` es porque desea tener tipos covariantes que, por supuesto, no generan subtipos, y son más restrictivos al ser aplicados a ciertas expresiones. La restricción es entonces esperada y no genera sorpresas.

En definitiva, en XC todavía no es posible asegurar la ausencia completa de errores en tiempo de ejecución pero, aún así, el lenguaje es más flexible y capaz de verificar estáticamente más clases de programas que un lenguaje con un sistema de tipos orientado a objetos más tradicional. Si bien los resultados presentes son prometedores, el sistema de tipos de XC todavía no está completo y es necesario obtener más experiencia con programas codificados en XC para validar el sistema de tipos final. Una cuestión abierta es la conveniencia de proporcionar la asignación polimórfica por defecto o no, y si la declaración de tipos exactos se produce también por defecto. En la actualidad, la asignación polimórfica se aplica por defecto, usando el tipo `Self` para identificar implícitamente una asignación exacta. Si no fuese así, todavía parece necesario que la asignación polimórfica debiera existir al menos como opción. Otra opción abierta es el futuro soporte de tipos covariantes mutuamente recursivos, que quedan más allá de los objetivos del diseño del sistema de tipos inicial del lenguaje.

7.5 Tipos genéricos

Los tipos genéricos engloban varias técnicas usadas para proveer variables de tipo en los lenguajes. La razón para considerar un tipo variable es reducir la duplicidad de código, porque es bastante común encontrar código muy similar cuya única diferencia son los tipos de datos que manipulan. El mantenimiento de ese código es costoso y puede llegar a ser muy significativo. Los tipos genéricos son una funcionalidad ortogonal a la orientación a objetos o a la existencia de tipos covariantes, y aparecen en lenguajes funcionales, procedurales y orientados a objetos. Existen distintas técnicas para implementar tipos genéricos, cada una con puntos fuertes y débiles. Es conveniente revisarlas con detenimiento.

7.5.1 El problema de las colecciones

Existen determinados problemas recurrentes que tienen soluciones parecidas y son lo suficientemente importantes como para prestarles especial atención. Un caso notable son las colecciones. Las colecciones son candidatas para ser descritas con herencia y limitar la duplicidad de código. Lugar donde, como puede ya intuirse, nos volveremos a encontrar con mensajes y métodos binarios. Por ejemplo, una lista de enteros y una lista de cadenas de caracteres tienen un protocolo básicamente igual. Utilizando XC:

```
protocol UnsignedList extends: Object
{
    property Unsigned value;
    property UnsignedList next;

    void addLastItem: Unsigned value;
    Boolean containsItem: Unsigned value;
    Unsigned length;
}

protocol StringList extends: Object
{
    property String value;
    property StringList next;

    void addLastItem: String value;
    Boolean containsItem: String value;
    Unsigned length;
}
```

No sólo la declaración de estas listas es parecida, su implementación es esencialmente la misma también. Por ejemplo, el método `containsItem`: podría implementarse de la siguiente forma:

```
prototype UnsignedList extends: Object
{
    Boolean containsItem: Unsigned value
    {
        UnsignedList list = self;

        while (list.next != null)
        {
            if (list.value == value) return true;
            list = list.next;
        }
        return false;
    }
    // ...
}

prototype StringList extends: Object
{
    Boolean containsItem: String value
    {
        StringList list = self;
        while (list.next != null)
        {
            if (list.value == value) return true;
            list = list.next;
        }
        return false;
    }
    // ...
}
```

Este escenario es muy similar en muchas colecciones, como *arrays*, conjuntos, tablas *hash*, listas o árboles. Las colecciones forman parte de cualquier programa no trivial, siendo una ayuda fundamental a la hora de representar y manipular estructuras de datos complejas. Por ello, es conveniente revisar si pueden expresarse con mayor naturalidad y menor duplicidad dentro del lenguaje. Resulta tedioso tener que redactar una y otra vez tipos de datos especializados que son fundamentalmente iguales, como es el caso de las listas anteriores. Tanto más cuanto más comunes son.

Para centrar la discusión, es interesante repasar cómo se resuelven las colecciones en un lenguaje procedural común, que sea compilado y no tenga soporte de tipos genéricos. El patrón habitual en C para describir colecciones como la anterior consiste en usar el puntero a un tipo indefinido `void *`.

```
struct List
{
    void        *value;
    struct List *next;
};

void    List_addLastItem (struct List *self, void *value);
int     List_containsItem (struct List *self, void *value);
unsigned List_length      (struct List *self);

int List_containsItem (struct List *self, void *value)
{
    UnsignedList list = self;
    while (list->next != NULL)
    {
        if (list->value == value) return TRUE;
        list = list->next;
    }
    return FALSE;
}
```

Véase el uso del primer parámetro, que contiene la estructura sobre la que se aplican las funciones de lista, y las funciones de lista declaradas por separado, típicas de la emulación de programación orientada a objetos usando programación estructurada. Llamadas a la función `List_containsItems` pueden ser:

```
List_containsItem (list, (void *)10);
List_containsItem (list, (void *)"string value");
```

En ellas, el tipo original se pierde al realizar la conversión a `void *`. El punto más conflictivo es la selección de la sentencia de comparación de elementos en la función `List_containsItem`. El ejemplo realiza una comparación de punteros válida para un valor entero camuflado de `void *` (en caso de que la arquitectura de la máquina lo permita) pero probablemente inválida para cadenas de caracteres.

El código anterior está plagado de problemas que requieren considerable esfuerzo adicional para sortearlos. No existe manera de asegurar que no se pueden insertar elementos heterogéneos dentro de la lista. La conversión directa a `void *` es fuente de errores involuntarios si no se es muy cuidadoso, puesto que trata igualmente a valores numéricos y a punteros a cadenas de caracteres. Para evitar cometer errores conviene anotar el tipo del valor entero y la cadena antes de realizar la conversión, mediante el uso de estructuras con tipo y valor similares a:

```
struct Object { int type; void    *value; };
struct Unsigned { int type; unsigned value; };
struct String  { int type; char   *value; };
```

Debe también evitarse pasar el valor en crudo como parámetro y enviar siempre una de las estructuras. Si se pasa por referencia, es posible que sea necesario también gestionar la memoria dinámica de estas estructuras. Luego, es ineludible comprobar el tipo de las estructuras en tiempo de ejecución para efectuar una conversión al tipo correcto del valor, y usar una función de comparación distinta para enteros y cadenas. Sin entrar en detalles, algo similar al código siguiente:

```
struct List
{
    struct Object *value;
    struct List   *next;
};

int List_containsItem (struct List *self, struct Object *value)
{
    struct List *list = self;
    while (list->next != NULL)
    {
        switch (value->type)
        {
```

```

        case UNSIGNED:
            if (unsignedIsEqual (list->value, (struct Unsigned *)value))
                return TRUE;
            break;
        case STRING:
            if (stringIsEqual (list->value, (struct String *)value))
                return TRUE;
            break;
        default:
            /* error */
    }
    list = list->next;
}
return FALSE;
}

```

Si no se comprueba el tipo, probablemente el programa abortará al intentar comparar erróneamente una cadena con un número. Aun así—y aunque pueda parecer un poco sorprendente visto desde este punto de vista—la mayoría de las colecciones en C se implementan de esta forma, puesto que aún con el código y complejidad adicional, resultan más sencillas de mantener que multitud de colecciones similares replicadas. En C con Clases—antecesor de C++—se daban guías para conseguir colecciones con distintos tipos, manteniendo la comprobación de tipos usando el preprocesador de C [Stroustrup, 1994; pp. 51-52]. Sin embargo, y como reconoce su autor, la solución era simple pero a la vez tosca.

7.5.2 Tipos polimórficos

Una solución al problema de las colecciones puede formularse en lenguajes que soportan herencia de tipos, estén éstos unidos a una implementación o no. En la mayoría de ellos se pueden definir listas polimórficas. Por ejemplo, en XC puede escribirse el siguiente fragmento de código:

```

@protocol Object extends: Comparable;
@protocol Unsigned extends: Object;
@protocol String extends: Object;

@protocol List extends: Object
{
    property Object value;
    property List next;

    void addLastItem: Object value;
    Boolean containsItem: Object value;
    Unsigned length;
}

prototype List extends: Object
{
    Boolean containsItem: Object value
    {
        List list = self;

        while (list.next != null)
        {
            if (list.value == value) return true;
            list = list.next;
        }
        return false;
    }
    // ...
}

```

El tipo `Object` adopta el protocolo `Comparable`, y los tipos `Unsigned` y `String` heredan de `Object`, por lo que también adoptan el protocolo `Comparable`. Estas condiciones garantizan que no se producen errores en tiempo de compilación relacionados con la existencia de los operadores de comparación, y se evita la duplicidad de código. Podemos construir variables que contienen listas de enteros y cadenas de caracteres mediante las siguientes declaraciones de variable:

```

List UnsignedList;
List StringList;

```

El resultado no es del todo satisfactorio. Para empezar, no hay diferencia entre una lista de enteros y una lista de cadenas. Además, el compilador pierde la información de tipo asociada a los elementos asignados a la lista. A partir de entonces, todos los elementos recuperados de la lista son de tipo `Object`, cuando en realidad son tipos enteros o cadenas de caracteres. La recuperación del tipo debe producirse en tiempo de ejecución, comprobando cuál es el tipo real de cada elemento insertado en la colección. En colecciones homogéneas de datos esta circunstancia es una desventaja en un lenguaje compilado con tipos estrictos, puesto que se retrasa innecesariamente al tiempo de ejecución comprobaciones que intuitivamente deberían realizarse en tiempo de compilación. Para colecciones heterogéneas, el compilador no puede inferir correctamente el tipo de los elementos de la colección, por lo que la comprobación en tiempo de ejecución anterior se hace necesaria. Por último, en los tipos polimórficos surge también el problema de los métodos binarios y la probabilidad de errores de tipos en tiempo de ejecución.

7.5.3 Tipos paramétricos

Una solución diferente al problema de las colecciones surge de la observación que, entre ambas declaraciones e implementaciones, es variable únicamente el tipo de los elementos almacenados dentro de las listas. Si consideramos que puedan existir *variables de tipo*, entonces podremos expresar ambas listas —y muchas variantes más— como una única familia. Es decir, podemos imaginar que existe un tipo `T`, variable, que sustituido adecuadamente en un fragmento de código apropiado, represente el código de las dos listas anteriores evitando la duplicidad. C++ y Ada poseen estos tipos. Usando C++:

```
template <class T>
class List
{
    T      value;
    List *next;

public:
    Boolean containsItem (T value)
    {
        List *list = this;
        while (list->next != NULL)
        {
            if (list->value == value) return TRUE;
            list = list->next;
        }
        return FALSE;
    }

    void addLastItem (T value)
    { // ... }

    // ...
};
```

El tipo `T` indicado en el *template* es un *tipo paramétrico*, una variante de tipo genérico. Se dice que presentan *polimorfismo paramétrico*, en honor a las variables de tipo pasadas como parámetros. Con ellas es fácil construir dos tipos distintos de lista comprobables en tiempo de compilación, simplemente rellenando el parámetro con un tipo conocido al declarar una variable:

```
List<unsigned> unsignedList;
List<String>   StringList;
```

Si no se indica ninguna restricción en los parámetros, entonces el tipo paramétrico es *no restringido o libre*. Éste es el caso de los *templates* en C++. Si se indica alguna restricción, entonces el tipo es *restringido*, como ocurre en Ada. Los parámetros libres son más generales que los parámetros restringidos, y pueden parecer una mejor opción. Sin embargo, la flexibilidad adicional tiene su precio: es posible que existan tipos no restringidos que den errores de compilación o de ejecución. En el ejemplo anterior, si el tipo `T` no soporta el operador '`==`', entonces se producirá un error de

compilación. Estos errores no son predecibles a priori, puesto que hasta que no se declara una variable y se termina la definición del tipo, no se puede asegurar la ausencia de errores.

Los tipos paramétricos restringidos, si bien no tan generales, pueden resolver la inconveniencia anterior. Supongamos por un momento que el protocolo `Comparable` define el operador ‘`==`’. Entonces, las declaraciones modificadas siguientes servirían para indicar a un compilador que rechazase los tipos que no adopten el protocolo `Comparable`. Es más, puede verificarse en tiempo de compilación que sólo mensajes del protocolo `Comparable` se envían dentro de la definición del tipo paramétrico. El código usa una hipotética extensión a C++ para indicar las restricciones, donde la restricción se expresa como una clase abstracta.

```
template <class T>
class Comparable
{
    Boolean operator == (T &value) = 0;
    Boolean operator != (T &value) = 0;
};

template <class T : Comparable<T> >           // Tipo restringido
class List
{ // ... }
```

Este ejemplo sugiere que el tipo paramétrico no restringido es, de hecho, *demasiado* general, y que ésta es la causa de que puedan producirse errores de tipo durante su uso. C++ adolece de este problema en su implementación de tipos genéricos. En cambio, el tipo paramétrico restringido—menos general—representa mejor las características de la implementación de listas, que demandan que los tipos de los elementos de las listas puedan ser comparables entre ellos.

Usando tipos paramétricos es posible encontrarse con el problema de los métodos binarios si se efectúan asignaciones polimórficas y las clases u objetos definen métodos binarios. Sin embargo, conviene resaltar que si no existen asignaciones polimórficas, entonces no se producen errores de tipos.

7.5.4 Tipos Virtuales

Los *tipos virtuales* —cuya referencia principal es el lenguaje BETA— plantean una solución distinta al problema de las colecciones, identificando tipos de datos que pueden refinarse mediante herencia. Estos tipos reciben en BETA el nombre de *clases virtuales*, pero quizás sea más acertado el término *tipo virtual* [Thorup, 1997; pp. 445-446]. El calificativo *virtual* es una analogía con el comportamiento de un procedimiento virtual en SIMULA-67 o una función virtual en C++, donde la palabra *virtual* se asocia con la capacidad de ser redefinido por subclases. En los tipos virtuales, el refinamiento está restringido a subtipos del tipo virtual. En el próximo ejemplo se usa la construcción hipotética `typedef as` para indicar un tipo virtual en código XC. El lenguaje en realidad no dispone de esa construcción.

```
protocol Object extends: Comparable;
protocol Unsigned extends: Object;
protocol String   extends: Object;

prototype List extends: Object
{
    typedef Element as Object;           // Tipo virtual

    property Element value;
    property List next;

    Boolean containsItem: Element value
    {
        List list = self;

        while (list.next != null)
        {
            if (list.value == value) return true;
            list = list.next;
        }
        return false;
    }
}
```

Los tipos virtuales introducen variables de tipo —en este caso `Element`— que deben ser resueltas al hacer referencia a la colección `List` o cualquiera de sus subtipos. Si se desea incorporar protocolos, entonces la variable de tipo debe estar presente también en el protocolo, para poder realizar correctamente la comprobación de tipos, y asignar algún tipo concreto al identificador `Element`.

```

protocol List extends: Object
{
    typedef Element as Object; // Tipo virtual

    property Element value;
    property List next;

    void addLastItem: Element value;
    Boolean containsItem: Element value;
    Unsigned length;
}

protocol StringList extends: List
{
    typedef Element as String; // Tipo virtual
}

protocol UnsignedList extends: List
{
    typedef Element as Unsigned; // Tipo virtual
}

```

Los tipos virtuales tienen la virtud de permitir comprobaciones en tiempo de compilación de los tipos asociados a las colecciones, sin necesidad de esperar a que se declare una variable en el código cliente que complete la declaración de tipo. La idea de fondo es simple e interesante: ya que de todas formas se pretende usar un subtipo en lugar del tipo original `Object` como valor de la variable `Element`, ¿por qué no especificar el valor directamente en el tipo heredado, de manera que pueda comprobarse sin más demora en tiempo de compilación? En el ejemplo, el protocolo `stringList` especializa a `List` y resuelve en ese momento la variable de tipo `Element`, quedando el tipo `StringList` completamente especificado. El tipo virtual sólo acepta especializar la variable de tipo `Element` con un subtipo del tipo originalmente indicado para esa variable en `List`, que es `Object`.

Por tanto, los tipos virtuales implícitamente incluyen una restricción en la variable de tipo que evita los inconvenientes de los parámetros de tipo no restringidos. También completa de forma natural las declaraciones de tipo, y no las retrasa hasta que un cliente decida crear variables que instancien el tipo, como ocurre con los tipos paramétricos. Una tercera ventaja es que con tipos virtuales es posible realizar cambios covariantes en variables de instancia y argumentos de métodos similares a `self`, pero manualmente. En el ejemplo siguiente se usa la variable de tipo `Myself` para simular el tipo covariante `self`, que se inicializa al nombre del tipo siendo definido, en este caso `List`. En el tipo derivado `StringList`, se inicializa la variable `Myself` al tipo derivado para simular el cambio covariante.

```
protocol List extends: object
{
    typedef Element as Object; // Tipo virtual
    typedef MySelf as List; // Tipo virtual
```

```

property Element value;
property MySelf next;

void addLastItem: Element value;
Boolean containsItem: Element value;
Unsigned length;
}

protocol StringList extends: List
{
    typedef Element as String;           // Tipo virtual
    typedef MySelf as StringList;       // Tipo virtual
}

```

Sin embargo, y al igual que con los tipos paramétricos, en caso de usar tipos que permitan subtipos, las asignaciones polimórficas asociadas con tipos que definan mensajes y métodos binarios pueden dar lugar a errores de tipos en tiempo de ejecución. Es decir, la simulación de `Self` con la variable de tipo `MySelf` es interpretada como un tipo polimórfico por los tipos virtuales, y no como un tipo exacto.

7.5.5 Tipos homogéneos y heterogéneos

Al manejar colecciones y, en general, cualquier variante de tipos genéricos, un aspecto importante del sistema de tipos está relacionado con el tamaño de implementación de los tipos de datos. El ejemplo de la lista es indicativo:

```

prototype List extends: Object
{
    property Object value;
    property List next;
    // ...
}

```

El valor contenido en `List` es de tipo `Object`, que es una referencia a cualquier tipo de objeto. Las referencias tienen siempre el mismo tamaño, y en ciertas ocasiones es posible aprovechar este invariante.

- ❖ **Def. 7-19.** Los *tipos homogéneos* describen todos los tipos mediante referencias y, por tanto, todos los parámetros de tipo tienen igual tamaño.

Esta propiedad simplifica algunas posibles inconveniencias. Por ejemplo, usando referencias es posible generar un único código que, al menos, asigne correctamente el tamaño de cualquier referencia. Si, además, las variables de tipo se pueden restringir, se asegura que existe un conjunto mínimo de mensajes que entienden los objetos, pudiendo construirse correctamente los envíos de mensajes, y evitando también errores en tiempo de compilación³⁷.

En C++ o Eiffel, en cambio, se pone especial énfasis en la eficiencia, no considerándose apropiado usar siempre referencias a objetos. Por ejemplo, usando C++:

```

template <class T>
class List
{
    T value;
    List *next;
    // ...
};

```

³⁷ Estrictamente hablando, todavía es posible obtener errores de tiempo de ejecución, debido a la posible presencia de métodos binarios.

El tipo τ puede instanciarse con un valor, como `float`, `struct X`, o con un puntero. En cada caso, el tamaño final de la implementación de objetos de tipo `List` varía según el tamaño del tipo pasado como parámetro.

- ❖ **Def. 7-20.** Los *tipos heterogéneos* describen los tipos con o sin referencias. Cada parámetro puede tener distinto tamaño.

La razón principal para tener varios tamaños posibles es la optimización de la velocidad del código generado. Los tipos homogéneos requieren el uso de tipos de datos por referencia, que son una sobrecarga cuando se trabaja con tipos sencillos. Los tipos heterogéneos evitan indirecciones innecesarias al aplicarlos con tipos sencillos como `arrays` de `int`, `unsigned` o `float`. Igualmente ocurre con estructuras de datos que puedan expandirse.

Esta ventaja tiene su coste: ya no es posible tener una única versión del código del tipo heterogéneo, puesto que el código generado ha de tener en cuenta los aspectos de gestión de los tamaños de los tipos, siendo necesarias múltiples versiones de los mismos. El tamaño del código adicional generado por los tipos heterogéneos no es nada despreciable, lo que dio lugar a buscar trucos que limiten la duplicación [Stroustrup, 1994; pp. 346-348]. Además, los tipos heterogéneos tienen una importante desventaja intrínseca: la falta de modularidad, porque su descripción en código fuente se ha de compilar al menos para cada tipo heterogéneo de tamaño distinto.

7.5.6 Sustitución de tipos implícita

La *sustitución de tipos implícita*³⁸ o *type substitution* es una técnica poco conocida que consiste en aplicar una semántica de tipos exactos a reemplazos de tipos arbitrarios en una clase dada para obtener una clase nueva, con el objetivo de aumentar la reusabilidad del código. Se trata de un mecanismo ortogonal y complementario a la herencia. La sustitución arbitraria es usada para evitar la introducción de variables de tipo como las encontradas en los tipos paramétricos y los tipos virtuales, pudiéndose sustituir cualquier tipo siempre que se preserve la ausencia de errores de tipo [Palsberg & Schwartzbach, 1990; pp. 153]. Si tenemos dos tipos T y S , y se pretende que S se reutilice en la definición de T , entonces han de cumplirse las siguientes dos condiciones: (1) *monotonía* entre T y S , y (2) *estabilidad*. La monotonía se ha estudiado en el apartado 4.3.4, la segunda se define como:

- ❖ **Def. 7-21.** La *estabilidad* entre tipos existe si se cumple que, si se preserva la igualdad entre elementos de T , también se preserva entre elementos de S .

La sustitución de tipos implícita se puede producir directamente sobre una clase para crear una clase nueva con el tipo sustituido. Es equivalente a una instanciación de un tipo paramétrico, con la salvedad que la clase original no declara ninguna variable de tipo y cualquiera de los tipos que forman parte de la declaración puede ser sustituido. La propuesta usa lenguajes con clases, aunque su aplicación es general.

Por ejemplo, usando una hipotética extensión a la sintaxis C++, la clase `StringStack` se instancia a partir de `Stack`, sustituyendo el tipo `Object` de `Stack` por `String` en el momento de la instanciación. Nótese cómo `Stack` originalmente no declara variables de tipo, permitiéndose sustituciones no planeadas a priori.

```
class Stack : public Object
{
    void push (Object obj);
    Object pop;
    // ...
};

Stack<Object = String> StringStack;           // Sustitución
```

³⁸ Aunque el adjetivo ‘implícito’ no aparece en la propuesta original, se incluye aquí para expresar con mayor precisión la técnica usada y diferenciarla de otra variante que se verá posteriormente.

El único problema de una sustitución de esta índole es la necesidad de tener acceso al código fuente de la clase `Stack` para efectuar la sustitución, que es un comportamiento poco modular. La sustitución también se puede realizar durante la herencia. Tiene cierto parecido con los tipos virtuales de BETA, pero ahora el tipo virtual es implícito. Al contrario que en BETA, la sustitución de tipos implícitamente también hace cambios covariantes similares a `Self`, si se está creando una clase derivada durante la sustitución. El código de ejemplo, otra vez en una hipotética extensión a C++, realiza, además, una sustitución de `Object` por `String`.

```
class List : public Object
{
    Object value;
    List next;

    void concat (List l);
    // ...
};

class StringList : public List<Object = String> // Sustitución y herencia covariante
{ // ... };
```

La clase `List` es un tipo recursivo. La sustitución de tipos automáticamente actualiza covariantemente las declaraciones recursivas en `StringList` al tipo `StringList`, de manera que, por ejemplo, el método `concat` en `StringList` tiene un argumento de tipo `StringList`, en vez de mantener el tipo `List` original. Su comportamiento es similar a un `Self` implícito.

El aspecto más interesante de las sustituciones es la semántica que se usa en las sustituciones de tipos y durante la herencia. La semántica es siempre *exacta* para preservar la propiedad de estabilidad [Palsberg & Schwartzbach, 1990; p. 159]. La condición de estabilidad requiere mantener la igualdad entre elementos de tipo `List` y, como resultado de una sustitución de tipos en `List`, también mantener la igualdad entre elementos de tipo `StringList`. La estabilidad es sólo posible si (1) la sustitución del tipo recursivo `List` por `StringList` afecta a todas las referencias a `List` por igual —es decir, se produce un cambio covariante— y (2) si las sustituciones tienen semántica exacta. El resultado es doble: se mantiene la corrección en las asignaciones entre variables del mismo tipo al crear subclases, y las subclases implícitamente no generan subtipos. Se trata, por tanto, de otra perspectiva sobre el mismo problema de los métodos binarios. La propuesta incluye también la definición explícita de variables polimórficas, llamadas *heterogéneas* [Palsberg & Schwartzbach, 1990; pp. 157-158], similares a los tipos `hash` [Bruce, 1996; pp. 44-45].

7.5.7 Tipos genéricos en XC: Sustitución de tipos explícita

XC usa una variante de tipos genéricos, más modular, en donde existen menos potenciales errores de tipos debidos a los métodos binarios. La variante se inspira en cuatro de las técnicas estudiadas: los tipos covariantes recursivos exactos, los tipos polimórficos homogéneos, los tipos virtuales, y la semántica de sustitución de tipos exactos de [Palsberg & Schwartzbach, 1990] para tipos genéricos. Esas técnicas han sido estudiadas en los apartados 7.4.6 *Comprobación de tipos covariantes y tipos exactos*, 7.5.3 *Tipos paramétricos*, 7.5.4 *Tipos Virtuales* y 7.5.6 *Sustitución de tipos implícita*. Por parecerse más en su conjunto a la sustitución implícita de tipos, esta nueva técnica recibe el nombre de *sustitución de tipos explícita*.

La elección adecuada de la implementación de los tipos genéricos para XC requiere sopesar con cuidado las ventajas e inconvenientes de cada una de las opciones anteriores. Además, es deseable que la implementación mantenga la modularidad, tenga en cuenta los problemas asociados con los métodos binarios, y facilite la creación de código correcto —comprobable en tiempo de compilación— que no genere errores en tiempo de ejecución. El objetivo es reducir aquellos fragmentos de código que necesiten comprobaciones en tiempo de ejecución indispensables para capturar la potencial existencia de esos errores.

Para los objetivos de XC, los tipos paramétricos heterogéneos constituyen un problema serio, pues carecen de la modularidad deseada para el lenguaje. La sustitución de una variable de tipo paramétrico heterogéneo por un tipo cuyo tamaño de almacenamiento sea mayor que el original puede invalidar el código compilado para el tipo genérico, que quizás espere y reserve tamaños distintos para las variables de tipos. Este es el caso de los *templates* de C++, y una de las razones para disponer del código fuente durante la instanciación de los *templates*. Al completarse la definición de tipos durante la declaración de variables en el código cliente, es en ese momento cuando se ha de proceder a la compilación del código asociado al tipo paramétrico heterogéneo. Puesto que la cuarta premisa del lenguaje (Def. 3-15) impide suponer que se tenga conocimiento del código fuente de otros módulos, las posibilidades de uso de tipos paramétricos heterogéneos sin exponer código fuente a otros módulos se restringen al interior de la implementación de un módulo, opción que se considera demasiado restrictiva.

En cambio, una implementación de tipos genéricos homogéneos que use referencias a objetos tiene la virtud de preservar el tamaño de los objetos referenciados a través de variables de tipos y, por ello, no necesita acceder al código fuente. Pero no todo son ventajas. El uso de tipos homogéneos obliga también a dar siempre una semántica de referencia para los objetos. Es decir, no es posible definir estructuras expandidas que se almacenen en pila, se pasen por valor a métodos o funciones, o se devuelvan como valor de retorno. Si se aceptan estas estructuras, entonces los tipos genéricos dejan de ser modulares, a no ser que se prohíba utilizar estructuras expandidas o por valor como valores válidos en las variables de tipos. Esta última opción no es deseable, porque daría lugar a un sistema de tipos no unificado y menos consistente, donde las variables de tipos son aplicables únicamente a un subconjunto de todos los tipos disponibles en el lenguaje.

Los tipos virtuales de BETA son mucho menos conocidos que los tipos paramétricos, existiendo muy pocos lenguajes que los implementan. No obstante, los tipos virtuales tienen la ventaja de especificar en el momento de crear un tipo derivado la especialización asociada a las variables de tipos y nuevos mensajes si es necesario. Como se vio en los apartados 7.5.3 *Tipos paramétricos* y 7.5.4 *Tipos Virtuales*, interesa que las variables de tipos sean también restringidas.

La especialización de tipos genéricos de XC con herencia se parece a los tipos virtuales de BETA. También a la sustitución de tipos implícitos con herencia si se indican explícitamente las variables de tipo. El código siguiente de `List` y `StringList` en XC usa estos tipos genéricos. La sintaxis sólo cambia ligeramente con respecto a la hipotética de tipos virtuales mostrada en el apartado 7.5.4.

```
protocol List extends: Object <Element = Object>
{
    property Element value;
    property Self next;

    Boolean containsItem: Element value;
    // ...
}

protocol StringList extends: List <Element = String>
{
    self valuesWithSubstring: Element substring;
}
```

Los caracteres ‘<’ y ‘>’ indican la presencia de una variable de tipo —en este caso `Element`— que debe ser siempre restringida a algún tipo válido, para hacer posible la comprobación estática de los mensajes enviados a variables cuyo tipo está indicado por `Element`. Como en los tipos virtuales de BETA y en la sustitución de tipos implícita, una vía para especializar una variable de tipo es con herencia. En el ejemplo la variable `Element` toma el valor `String`. El tipo `String` es válido porque hereda de `Object`, lo que asegura la monotonía y, por tanto, que los mensajes compilados originalmente para `Object` siguen siendo válidos con el nuevo tipo `String`. Al usar herencia durante la especificación del tipo genérico, es posible también especializar métodos existentes o añadir mensajes y métodos relacionados con el nuevo prototipo `StringList`, como ocurre con el mensaje

`valuesWithSubstring`: Nótese que el nuevo mensaje no es aplicable semánticamente al tipo original `List`. De ahí la conveniencia de la especialización del tipo genérico durante el proceso de herencia. Estos detalles se ven más claramente en la implementación:

```

prototype List extends: Object <Element = Object>
{
    property Element value;
    property Self next;

    Boolean containsItem: Element value
    {
        self list = self;
        while (list.next != null)
        {
            if (list.value == value) return true;
            list = list.next;
        }
        return false;
    }
    // ...
}

prototype StringList extends: List <Element = String>
{
    self valuesWithSubstring: Element substring
    {
        self list = self, retList = null;
        while (list.next != null)
        {
            if (list.value.containsSubstring: substring)
                retList.addLast: list.value;
            list = list.next;
        }
        return retList;
    }
}

```

La implementación de `StringList` define el nuevo método `valuesWithSubstring` y hereda el resto de los métodos de `List`. La implementación ha de especificar otra vez las variables de tipos usadas en los protocolos por claridad. El prototipo `StringList` especializa `List` modularmente. Con la información disponible en su protocolo es posible conocer a qué mensajes responde `StringList`. La restricción a `Object` en la variable de tipo `Element` y el requisito de que `String` derive de `Object`, son suficientes para garantizar que los mensajes enviados en la implementación original de `List` son igualmente aplicables a `StringList`. Todo ello sin necesidad de acceder al código fuente de `List`.

Al contrario de lo que ocurre con los tipos virtuales de BETA, y al igual que en sustitución de tipos implícitas, en XC la especialización de un tipo genérico no tiene porqué heredar siempre de otro. Su razón de ser es mantener la eficiencia, ya que es posible especializar un tipo genérico más eficiente —con modificadores `final`, `confined` o `baselevel`— que tenga una implementación más optimizada, y que no es viable si la especialización depende obligatoriamente del árbol de herencia. También es posible aprovechar la implementación de un tipo genérico más general y restringir sus grados de libertad para mejorar la eficiencia del nuevo tipo genérico especializado. Por ejemplo, una implementación eficiente del tipo `List` podría usar un prototipo similar al siguiente:

```

protocol List extends: Object <Element = Object>
{ // ... }

prototype List extends: Object <Element = Object>
{ // ... }

final protocol FastList = List <Element = Object>;
final baselevel prototype FastList = List <Element = Object>;
final baselevel prototype FastStringList = FastList <Element = String>;

```

Nótese la diferencia en la sintaxis. En este caso no se usa la cláusula `extends`, que enfatiza que no se pueden añadir nuevos mensajes o métodos, o especializar métodos heredados. La sintaxis de asignación durante la definición del protocolo y el prototipo `FastList` indica que se usa exactamente

la misma definición que el protocolo o prototipo que es asignado. Ni el protocolo ni el prototipo pueden definir un cuerpo. El protocolo `FastList` es un protocolo *distinto* de `List`. Es un protocolo final que no puede extenderse con herencia, aunque sí es posible añadirle categorías. Una categoría añadida a `FastList` no afecta al protocolo `List`. Lo mismo ocurre con los prototipos `FastList` y `FastStringList`, que definen nuevos objetos prototípicos diferentes, aunque con la misma implementación que el prototipo original. En este ejemplo, al ser final y baselevel restringen la herencia y los metamétodos, pero admiten extensiones con categorías, al no utilizar el modificador `confined`. Usando una categoría es posible añadir el mensaje y método `valueswithSubstring`: del ejemplo anterior a `FastStringList`.

```
protocol category StringOperations extends: FastStringList <Element = String>
{
    self valueswithSubstring: Element substring;
}

prototype category StringOperations extends: FastStringList <Element = String>
{
    self valueswithSubstring: Element substring
    { // ... }
}
```

El resultado no es igual al obtenido en el ejemplo previo con especialización de tipos genéricos con herencia, donde es posible sobrescribir métodos existentes. En cambio, al usar sustitución de tipos explícita sin intervención de la herencia se prima la eficiencia de la implementación. La elección entre flexibilidad o eficiencia depende del desarrollador, si bien la interfaz publicada de ambas listas de cadenas de caracteres es similar desde el punto de vista de los clientes. La sustitución de tipos explícita sin herencia de XC tienen un comportamiento parecido a la sustitución de tipos implícita sin herencia y a los tipos paramétricos homogéneos.

La diferencia más importante entre los tipos virtuales y los tipos paramétricos homogéneos con respecto a los tipos genéricos implementados en XC es la semántica de la sustitución. En XC, al igual que en la sustitución de tipos implícita, la semántica de sustitución de una variable de tipo por su valor usa siempre tipos exactos, en vez de usar tipos polimórficos. La diferencia fundamental con la sustitución de tipos implícita es consecuencia de hacer explícita la sustitución: Al especificar explícitamente las variables de tipo a sustituir, incorporar restricciones en el tipo de las variables y usar tipos homogéneos, XC evita la necesidad del acceso al código fuente del tipo genérico, obteniendo una implementación más modular.

Los tipos genéricos de XC usan una generalización de la semántica de sustitución exacta de `self`. La implementación del compilador no es significativamente diferente para `self` que para una variable de tipo. De hecho, `self` puede emularse trivialmente en XC usando estas variables ya que su semántica también es exacta:

```
protocol List extends: Object <Element = Object, MySelf = List>
{
    Element value;
    MySelf next;
    // ...
}
```

El uso de `self` es tan habitual en XC que justifica tener una palabra reservada, resultando mucho más claro que la sustitución covariante realizada durante la herencia por la sustitución de tipos implícita. La misma semántica aplicada a `self`—vista en el apartado 7.4.6 *Comprobación de tipos covariantes y tipos exactos*—se emplea en la sustitución de variables de tipos. Hasta tal punto son equivalentes que el sistema de tipos de XC trata internamente a `self` como una variable de tipo predefinida. Estas variables, al igual que `self`, están claramente especificadas en los protocolos y prototipos, por lo que el compilador tiene información suficiente para aplicar correctamente la semántica de asignación apropiada en cada expresión. La semántica exacta facilita la construcción de implementaciones genéricas que posean menos errores en tiempo de ejecución.

Los tipos genéricos de XC tienen también cierta similitud con la propuesta de [Bruce & Vanderwaart, 1999] para tipos virtuales, que plantea mezclar las ideas de tipos virtuales de BETA con el tipo recursivo covariante `MyType`, equivalente a `self`. Las diferencias con [Bruce & Vanderwaart, 1999] son varias. (1) El énfasis en las propiedades modulares de los tipos genéricos. (2) El uso de la semántica de tipos exactos—aplicada automáticamente por el compilador durante el análisis de expresiones de tipo—en vez de tipos exactos y tipos *hash* separados. (3) La posibilidad de especializar un tipo genérico sin usar herencia. Y (4) la falta de soporte para tipos covariantes mutuamente recursivos, que se consideran fuera del ámbito de una primera versión del sistema de tipos de XC.

Desafortunadamente, no todo son ventajas. Una dificultad significativa en el sistema de tipos de XC surge con los tipos optimizados, que se estudian en el apartado 8.7.2 *Optimización de tipos básicos*. En un lenguaje dedicado a la programación de sistemas, es imperativo asegurar que los tipos básicos se evalúan a la velocidad esperada. El caso general de tipos genéricos visto en este apartado debe revisarse para dar cabida a estos requisitos. Los tipos genéricos exactos sin especialización con herencia son una opción para salvaguardar la eficiencia, pero no son adecuados siempre. Los tipos optimizados, si bien sólo son unos pocos tipos de datos, son críticos para mantener la eficiencia.

Por ejemplo, en XC el tipo `Unsigned` es optimizado por el compilador para ocupar únicamente una palabra de máquina. El código generado para un tipo optimizado es *diferente* al código habitual. En el ejemplo anterior, la comparación con el operador ‘==’ en el método `containsItem`: se traduce por una comparación entre palabras de máquina que no involucra el envío de ningún mensaje. Es decir, se produce un código distinto para los tipos usuales y para los tipos optimizados, si un tipo optimizado es susceptible de aparecer en una variable de tipo. Se trata de una situación habitual en colecciones básicas como `arrays` o listas, pero menos común con otros tipos de datos más complejos. La elección actual en XC se decanta por una solución intermedia. Se considera que es mejor favorecer la implementación modular de tipos genéricos, aun cuando algunos tipos particulares requieran una solución de compromiso. Por ejemplo, para el caso del tipo `UnsignedList`, el código asociado a `containsItem`: debe duplicarse si se pretende mantener la eficiencia.

```
prototype UnsignedList extends: List <Element = Unsigned>
{
    Boolean containsItem: Element value
    {
        List list = self;
        while (list.next != null)
        {
            if (list.value == value) return true;
            list = list.next;
        }
        return false;
    }
    // ...
}
```

Si la implementación de `UnsignedList` se produce en el mismo módulo que la implementación de `List`, no se compromete la modularidad del código, siguiendo la regla de eliminación de paranoíta (Def. 5-24). Pero en cambio sí se incurre en el coste extra de duplicar código que no debería ser necesario duplicar. La solución actual no es del todo satisfactoria, y se están estudiando posibles alternativas, siempre que mantengan la preferencia por favorecer la implementación modular de tipos genéricos. Una posible variante consiste en aprovechar la idea de la eliminación de paranoíta anterior. Dentro de un mismo módulo, el compilador puede ser lo suficientemente perspicaz para duplicar automáticamente el código asociado a un tipo genérico que especifique alguna variable de tipos cuya especialización sea un tipo optimizado. Es probable que el compilador pueda llegar a reconocer esos escenarios porque: (1) un tipo optimizado debe cumplir ciertos requisitos comprobables en tiempo de compilación (véase el apartado 8.7.1 *Modificadores de optimización de objetos*) y (2) la

implementación de un tipo genérico debe especificar siempre los tipos asociados a las variables de tipos.

7.6 Tipos opacos y modularidad

Un último aspecto relacionado con el sistema de tipos tiene que ver con la resolución de las dependencias mutuamente recursivas entre módulos. La declaración de módulos con referencias interdependientes requiere cuidado por parte del compilador a la hora de decidir si un símbolo se encuentra suficientemente especificado durante la comprobación de tipos. Por ejemplo, las siguientes declaraciones podrían estar en tres módulos separados. Las declaraciones adelantadas o *forward* del módulo C del próximo ejemplo permiten que los módulos A y B se comprueben correctamente en tiempo de compilación, independientemente del orden de compilación de los módulos A y B:

```
public module protocol C
{
    protocol String;
    protocol Unsigned;
    protocol Boolean;
}

public module protocol A
{
    import C;

    protocol String extends: Object
    {
        Unsigned length;
        self add: String value;
        // ...
    }
}

public module protocol B
{
    import C;
    protocol Unsigned extends: Object
    {
        String asString;
        Boolean isEqual: self value;
        // ...
    }
}
```

El compilador está usando en este caso *tipos opacos*, es decir, tipos no completamente especificados, de los que se conoce únicamente su nombre. Con ellos es viable comprobar tipos que de otra forma precisarían revelar información adicional, para ser verificados. Los tipos opacos no son válidos siempre. Un ejemplo ilustra mejor este punto. En el módulo de implementación A un objeto `String` podría optimizar el método `add`: comprobando si la nueva cadena a añadir tiene longitud cero, y en ese caso, devolver la cadena original en vez de efectuar un proceso superfluo de concatenación, que no modificará la cadena original. La sentencia de comparación de la longitud llama al mensaje `isEqual`: del protocolo `Unsigned` a través del operador ‘`==`’, pero ese mensaje no está declarado en el protocolo del módulo C y, al no ser su declaración accesible, causará un error de compilación.

```
module A
{
    prototype String extends: Object
    {
        self add: String value
```

```

        if (value.length == 0) return self; // error!
    }
    // ...
}
// ...
}
}

```

Los módulos de implementación tienen más necesidades que los módulos de declaración porque pueden usar mensajes definidos en los protocolos de los tipos importados. La implementación del módulo `A` debe, por tanto, importar `B` para acceder a la declaración de `isEqual:` en el protocolo `Unsigned` y evitar el error de compilación. No obstante, la declaración del módulo `A` no necesita describir específicamente esa relación, relegándola a un aspecto de implementación encapsulado, de manera que se disminuya la interdependencia entre módulos. La reducción es notable, pues cualquier módulo que sólo importe la declaración de `A` no adquiere la dependencia implícita transitiva con la declaración de `B`. Se evita así una propagación exponencial de la dependencia implícita de `B` a través de `A`. Los tipos opacos son un aspecto esencial de un sistema de tipos modular.

La comprobación de tipos se complica un poco al añadir `alias`. Un alias es equivalente a un `typedef` en C. Define un nuevo nombre de tipo para otro tipo, sea éste conocido o no. Un tipo no conocido se declarará o definirá posteriormente. Por ejemplo, la siguiente declaración define un alias llamado `String` para el protocolo `xc.String`, definido en el módulo `xc`. El protocolo `String` devuelto por el mensaje `asString` del protocolo `Unsigned` se refiere en realidad al protocolo `xc.String`, pues `String` es un alias.

```

protocol alias String = XC.String;
protocol Unsigned extends: Object
{
    string asString;
    // ...
}

```

El compilador ha de efectuar la traducción del alias y considerar el tipo `xc.String` como opaco hasta que se defina con detalle, manteniendo la modularidad. Un alias o un tipo opaco puede aparecer también en una declaración de variable de tipo. En el siguiente ejemplo el protocolo `Array` define una variable de tipo llamada `Element` restringida a `Object`. `StringArray` hereda del protocolo `Array` especializando la variable de tipo `Element` del protocolo `Array` con el alias `String`, que corresponde con el protocolo `xc.String`:

```

protocol alias String = XC.String;
protocol Array extends: Object <Element = Object>
{
    Element at: Unsigned index;
    // ...
}
protocol StringArray = Array <Element = String>;

```

Es importante que el compilador sea capaz de resolver referencias como la anterior, particularmente entre módulos con declaraciones mutuamente recursivas. El protocolo devuelto por la variable de tipo `Element` del mensaje `at:` heredado de `Array` del protocolo `StringArray` debe ser el tipo opaco `XC.String`. Otra vez, el tipo opaco garantiza un mínimo de interdependencia entre declaraciones mutuamente recursivas con independencia de la flexibilidad del sistema de tipos, y ayuda a definir un sistema de tipos más consistente. Los tipos opacos, ya sean descritos a través de declaraciones adelantadas, alias o variables de tipo deben acomodar la definición final del protocolo al que hacen referencia una vez ésta se encuentre en un módulo posterior. A partir de entonces, el compilador trata al tipo opaco como si fuese un tipo explícito, pues éste ya se ha declarado con exactitud. El uso de tipos opacos, tipos explícitos, tipos heredados, variables de tipo y alias es ortogonal, al poderse combinar entre sí, dando lugar a un sistema de tipos más consistente.

7.7 Recapitulación

El estudio de los sistemas de tipos para lenguajes compilados pone de manifiesto las dificultades para obtener un sistema de tipos expresivo que detecte el mayor número posible de errores en tiempo de compilación, un camino que todavía no se ha terminado de recorrer. Los lenguajes tradicionales orientados a objetos poseen un sistema de tipos demasiado simple e inflexible, que no considera válidas muchas familias de programas. La separación entre tipos e implementación, y la incorporación de tipos covariantes y tipos genéricos, añade una parte importante de la flexibilidad deseada al sistema de tipos, aunque no siempre está exenta de inconvenientes. El sistema de tipos de XC debe cumplir, además, con los requisitos de modularidad, flexibilidad en la comprobación de tipos durante la compilación, y aptitud para la programación de sistemas y programación con componentes. Se estudian diferentes opciones de implementación de partes del sistema de tipos y sus inconvenientes, seleccionando un conjunto de técnicas cuya unión configuran un sistema de tipos expresivo, muy poco común en lenguajes de programación de sistemas. La técnica de sustitución de tipos explícita para los tipos genéricos de XC es novedosa. Al usar variables de tipo homogéneas, restringidas y una interpretación de tipos exactos en sus expresiones, constituye una implementación de tipos genéricos más modular y apropiada para el lenguaje. La técnica trata uniformemente al tipo covariante `self` y al resto de variables de tipos. Facilita la construcción incremental de tipos de datos mediante herencia comprobables en tiempo de compilación, y reduce considerablemente la posibilidad de encontrar errores en tiempo de ejecución. Por último, los alias ayudan a definir tipos opacos, aquellos de los que únicamente se conoce su nombre, no las operaciones que definen. Son usados para aumentar la encapsulación de la información de tipos necesaria durante las declaraciones mutuamente recursivas entre protocolos de módulos.

8

El soporte en tiempo de ejecución y el compilador

8.1 Introducción

El soporte en tiempo de ejecución o *runtime* para abreviar, es el nexo de unión entre el sistema operativo y el lenguaje. Es el encargado de implementar las abstracciones básicas y proporcionar una capa intermedia de acceso a recursos del sistema operativo, manejables a más alto nivel. La estructura de objetos, el mecanismo particular de envío de mensajes, la gestión de memoria, o un acceso conveniente a otros recursos son algunos ejemplos. El soporte en tiempo ejecución está íntimamente relacionado con la implementación del compilador, pues este último se encarga de la generación eficiente de código para ser usada por el primero. XC es un lenguaje lo suficientemente complejo para necesitar una implementación prototípica que permita evaluarlo. El objetivo del compilador es valorar las características de XC, y servir como vehículo en la definición precisa de las propiedades finales del lenguaje, estando algunas todavía en estudio. El prototipo actual compila rutinariamente fragmentos de código de miles de líneas.

Se examina en primer lugar la estructura e implementación de objetos en XC, el procedimiento de creación de objetos y los algoritmos principales usados. A continuación se estudia en detalle el registro de activación de envío de mensajes, usado para el envío eficiente de mensajes mediante cadenas de métodos, y los procedimientos de inicialización. Posteriormente se revisa el diseño general de compilador, pasando luego a los aspectos de implementación más interesantes: el proceso de importación de módulos, los contextos de evaluación, y la estructura del código generado. Luego se revisan aspectos de eficiencia que el compilador ha de tener en cuenta para una implementación viable en la práctica de las abstracciones del lenguaje. Por último, se analizan varias características del lenguaje que lo hacen más idóneo para la programación de sistemas, y que reducen la necesidad de lenguajes de más bajo nivel como C. Se revisan las optimizaciones más importantes realizadas por el compilador, como la optimización de tipos integrales o prototipos `final` y `confined`, el soporte para funcionalidad nativa y acceso a bibliotecas, optimización de `arrays`, el código en línea y el código no seguro.

8.2 El soporte en tiempo de ejecución

La capa del lenguaje que se encuentra inmediatamente encima del sistema operativo es el soporte en tiempo de ejecución. La estructura de objetos, el envío de mensajes y la interrogación de propiedades de los programas en ejecución son las abstracciones fundamentales que implementa.

Usualmente mediante bibliotecas también es responsable de dar acceso a las abstracciones de bajo nivel del sistema operativo: gestión de memoria, gestión de procesos e hilos de ejecución, o soporte básico para distribución. Una parte importante de la funcionalidad del soporte en tiempo de ejecución actual está escrita también en XC, para facilitar la incorporación incremental de un protocolo de reflexión e introspección. Veremos en este capítulo que esta decisión plantea algunos problemas técnicos. Dadas las ventajas de flexibilidad, capacidad de cambio, y la simplicidad final obtenida, vale la pena el esfuerzo adicional. La ventaja principal de esta opción es que limita la funcionalidad *ad-hoc* dada por el lenguaje a la suministrada por la biblioteca básica del *runtime*, escrita en C. De esta manera se reduce también el número de conceptos ofrecidos por el núcleo del lenguaje, y facilita que el lenguaje se pueda extender a sí mismo usando únicamente el soporte dado por el núcleo. Este ha sido también el enfoque de lenguajes clásicos como Smalltalk-80 o CLOS. La estructura interna del soporte en tiempo de ejecución está completamente orientada a objetos. Al contrario que en Java, por ejemplo, donde existen funciones nativas que traducen y exportan las estructuras de su soporte en tiempo de ejecución a un modelo de objetos, el *runtime* de XC contiene directamente objetos del propio lenguaje que son accesibles de forma natural desde él. Este esquema fue desarrollado por primera vez en Smalltalk-80. Se espera que el protocolo de reflexión e introspección de XC cambie incrementalmente con el uso prolongado del lenguaje, y al incorporar facilidades para otros requisitos no funcionales como persistencia o distribución. Este proceso iterativo continuará hasta que se obtenga un soporte en tiempo de ejecución con funcionalidad estable.

El módulo `XC.Runtime` contiene las declaraciones del protocolo de reflexión e introspección, así como la implementación del mismo. Los objetos definidos en el *runtime* son técnicamente metaobjetos, puesto que son objetos que describen aspectos del propio lenguaje. Constituyen una interfaz de acceso a la información recopilada por el compilador sobre módulos, módulos importados, categorías, protocolos, mensajes, métodos y funciones, que es registrada durante la carga de módulos. El módulo `XC.Runtime` se asienta sobre la biblioteca básica del *runtime*. La biblioteca se encarga de la descripción de objetos, los procedimientos de creación de nuevos objetos, el proceso básico de inicialización —donde todavía no se puede siquiera enviar mensajes a objetos— y el registro de información de módulos.

8.3 La estructura de los objetos

La estructura de los objetos de XC es una de sus contribuciones más novedosas. Se propone una estructura de objetos que preserva la modularidad del lenguaje, mantiene simplicidad conceptual, facilita la construcción de componentes a partir de ellos, y posee flexibilidad controlada para adaptarse mejor a los cambios. La representación física de los objetos es mucho más compleja que su representación conceptual. Al igual que en otros lenguajes con prototipos como SELF o Kevo, XC da la ilusión de manipular objetos completos, con su propio conjunto de métodos y variables. Cada objeto puede ser clonado para obtener nuevos objetos de la misma familia. La estructura de los objetos en XC, al contrario de lo que ocurre con SELF o Kevo, no varía en tiempo de ejecución. Es descrita estáticamente y comprobada en tiempo de compilación. Puede también ser extendida como en otros lenguajes basados en prototipos. La extensión se realiza con categorías, igualmente verificables en tiempo de compilación.

8.3.1 El mapa de los objetos

La estructura de los objetos de XC está inspirada por SELF [Chambers, 1992a; p. 47-48] y Kevo [Taivalsaari, 1992; p. 11]. Por motivos de eficiencia, los objetos no contienen toda la información que los constituyen, sino que ésta se encuentra dividida en dos partes: el mapa del objeto y el objeto en sí.

- ❖ **Def. 8-1.** El *mapa del objeto* incluye toda la información común a una familia de objetos: métodos definidos, variables compartidas, nombre de la familia de objetos, e información de la estructura del objeto.
- ❖ **Def. 8-2.** El *objeto* contiene aquella información que no es común, el estado particular de cada objeto.

Los mapas son una técnica de implementación para optimizar la representación interna de los objetos. Se muestran en la Figura 8-1. Duplicar aquellas partes compartidas de un objeto en cada uno de ellos demanda gran cantidad de memoria. Es mucho más eficiente almacenar las partes comunes en un único sitio —el mapa del objeto— si bien incurre en el coste añadido de una referencia en cada objeto a su mapa.

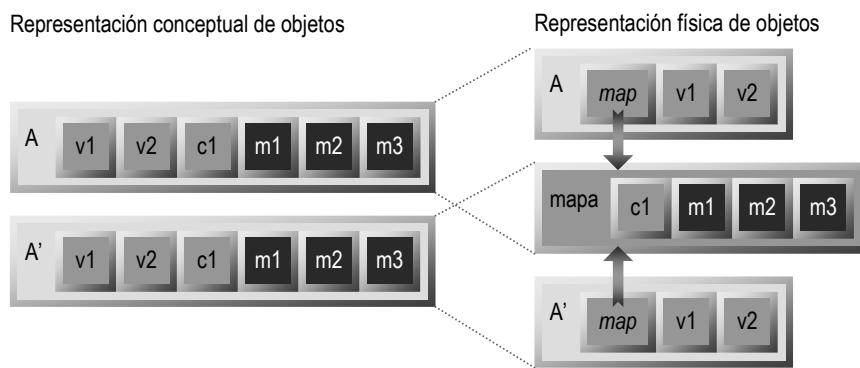


Figura 8-1 *Objetos y mapas de objetos*.

Los objetos son autosuficientes y conceptualmente contienen toda la información que necesitan: datos, datos compartidos y métodos. La implementación internamente separa aquella parte común de una misma familia en un mapa, añadiendo una referencia al mapa en cada objeto.

8.3.2 La semántica de clases

La ventaja de la técnica de los mapas es que obtiene un comportamiento equivalente al de las clases sin necesidad de hacerlas visibles en el lenguaje, es decir, sin incluir un concepto nuevo. Mantener la implementación de los mapas escondida evita problemas de incoherencias que aparecen si se usan técnicas de clonado o delegación [Dony, Malenfant & Cointe, 1992; pp. 209-210]. Sin entrar en demasiados detalles, por un lado la clonación no mantiene la semántica de clases si no garantiza que la modificación de un prototipo modifica todos los objetos clonados ya existentes a partir de él. Es decir, no se garantiza la semántica si los objetos clonados no siguen la descripción del prototipo una vez modificado. Kevo es una excepción, ya que consigue garantizar la semántica porque mantiene explícitamente la lista de objetos que pertenecen a una misma familia clonada. Suministra *operaciones de módulo* que actúan sobre la familia completa durante las modificaciones [Taivalsaari, 1992; pp. 7-8]. SELF garantiza la semántica de clases a través del entorno de desarrollo [Smith & Ungar, 1995 ; p. 310]. Por otro lado, la delegación no mantiene la semántica de clases si al modificar un prototipo éste se vuelve un caso excepcional, dejando de ser un ejemplar prototípico para todos los objetos que delegan en él. Para evitar ambos casos los prototipos podrían ser inmutables, pero en ese caso tales objetos tendrían un comportamiento “especial” que se alejaría de la filosofía de los prototipos. Por ello [Dony, Malenfant & Cointe, 1992; p. 209] concluyen que los objetos prototípico no son una buena estrategia para implementar la semántica de clases.

Curiosamente SELF, que sí ofrece mapas, también proporciona unos objetos especiales o *traits* para las partes compartidas por herencia mediante delegación. Estos objetos guardan métodos compartidos de sus objetos descendientes —no de sí mismos— por lo que si se envía un mensaje compartido al objeto *trait*, éste puede ser incapaz de responder correctamente. Supongamos un objeto *punto*, con sus

coordenadas particulares x e y , y un método `mover` que manipula las coordenadas. El mensaje `mover` se delega en un objeto *trait* para que cada punto no tenga una copia del método. Los puntos acceden al método `mover` del objeto *trait* por delegación cuando se les envía el mensaje `mover`. ¿Pero qué ocurre si se envía el mensaje `mover` directamente al objeto *trait*? El método intentará acceder a las coordenadas x e y , pero éstas no son compartidas, se encuentran en los objetos *punto* y no en el objeto *trait*. El resultado es que el envío del mensaje falla. Es decir, se acaban teniendo objetos que son “diferentes” a los objetos normales [Smith & Ungar, 1995; p. 312]. [Dony, Malenfant & Cointe, 1992; p. 210] deducen que los objetos prototipo especiales como los *traits* tampoco son una buena solución para dar una semántica de clases. Proponen usar únicamente los mapas de SELF.

[Borning, 1986; p. 39] sugirió la misma idea de los mapas: las partes compartidas de los prototipos deben ser relegadas a un detalle de implementación, puesto que es posible obtener partes compartidas con diferentes técnicas. La semántica de la delegación no tiene porqué ser la más apropiada. En particular Borning presenta un modelo de prototipos basado en restricciones sin delegación que emula la herencia: reglas que permiten heredar nombres de campos, heredar comportamiento mediante métodos y heredar protocolos, preservando la monotonía. Si se mantienen esas tres restricciones, se obtienen objetos descendientes con la semántica de la herencia [Borning, 1986; p. 37] y la técnica de delegación no se hace necesaria. Kevo tampoco usa delegación por la misma razón, manteniendo las partes compartidas escondidas en el modelo de objetos.

XC consigue dar una semántica de clases usando mapas de objetos invisibles al modelo conceptual, y directamente prohibiendo modificaciones estructurales en los objetos prototipo una vez creados. Se usa la herencia y extensiones controladas por el compilador para propagar cambios. Así, los prototipos son extensibles, pero siempre se mantiene la semántica de clases asociada a todos los miembros de una misma familia. La semántica de delegación no se utiliza por considerarse poco modular. Los mapas —al igual que en SELF— son inmutables. Con todo existe una diferencia importante: SELF consiente que un objeto cambie dinámicamente. En caso de modificación, SELF genera un nuevo mapa para el nuevo objeto modificado. En cambio, XC no admite modificaciones en una familia de objetos, para asegurar que las características del prototipo comprobadas en tiempo de compilación se mantienen en tiempo de ejecución. El mapa, al ser inmutable, corresponde con el *tipo de implementación* del objeto. Independientemente del protocolo usado para acceder al objeto, éste último pertenece a una familia de objetos fija, cuya definición está almacenada en su mapa.

8.3.3 Extensiones modulares

Los objetos prototipo sin clases dan una imagen de simplicidad al usuario. El modelo de objetos de XC mantiene la modularidad de las extensiones de objetos con categorías y la extensión mediante herencia, sin complicar el esquema conceptual. Una propiedad muy importante de las categorías de XC es que poseen *modularidad monótona*.

- ❖ **Def. 8-3.** La modularidad es *monótona* si cambios monótonos en extensiones con herencia y extensiones con categorías son también modulares.

Los cambios monótonos en herencia se garantizan sintácticamente evitando el borrado de métodos o variables durante una derivación. Sintácticamente también, los cambios monótonos en categorías están garantizados al prohibir la redefinición de métodos o variables. Con categorías de extensión es importante que no se redefinan métodos o variables existentes, aunque aparentemente el cambio parece monótono. No lo es porque no se puede asegurar una semántica que mantenga la modularidad. Si por ejemplo se redefiniese algún método: ¿Qué pasaría con el método original? ¿Cómo se garantizaría que una categoría que invalide un método desde otro módulo no está cometiendo algún error durante el reemplazo? ¿Cómo obtendría acceso privilegiado a los datos de un objeto cuya estructura no es visible desde el otro módulo? Las respuestas a estas preguntas comprometen la modularidad de la extensión con categorías.

El compilador y el *runtime* simplifican la implementación de la modularidad en extensiones de herencia y de categorías observando que, en lo referente a la disposición concatenada en memoria de un objeto, la extensión de un prototipo mediante herencia es equivalente a la extensión del mismo mediante una categoría. Las diferencias semánticas se reducen entonces a distinguir si una categoría es primaria o es una categoría de extensión.

- ❖ **Def. 8-4.** Una *categoría primaria* es implícitamente creada por una definición de prototipo. Puede recibir extensiones con otras categorías o por herencia.
- ❖ **Def. 8-5.** Una *categoría de extensión* se limita a extender una categoría primaria e indicar cómo se inserta dentro de la lista de extensiones de esa categoría.

Una vez se maneja internamente un único concepto —la categoría— es posible implementar sobre él con más sencillez los mecanismos modulares estudiados en el capítulo 5. Las reglas a seguir son:

- Se prohíbe a una categoría conocer el origen del desplazamiento de sus variables —compartidas o no— dentro de un objeto.
- Como consecuencia del punto anterior, se impide el acceso a las variables internas de una categoría, tan sólo a los métodos de acceso. El acceso directo no es posible si no se conoce el desplazamiento de las variables.
- Sólo se acepta hacer referencia a otra categoría o a la categoría primaria por nombre.
- Sólo se permite a una categoría conocer su posición relativa respecto a otra categoría en la lista de extensiones de categorías de un prototipo, no su posición absoluta dentro de la misma.
- Como consecuencia del punto anterior, no se admite a una categoría conocer su posición dentro del objeto.

Cuando se concatenan varias categorías, ninguna de ellas es capaz a priori de acceder a otra categoría, ni conocer en qué orden van a ser añadidas, ni saber cuál es el desplazamiento final de sus variables. Toda esta información no está disponible en tiempo de compilación por motivos de modularidad. En tiempo de ejecución, por el contrario, es necesario dar una respuesta a todas las cuestiones dejadas abiertas durante la compilación.

Las categorías no necesitan replicarse por cada objeto, sólo las nuevas variables de instancia que definen deben hacerlo. Por eso la información de las categorías debe ir en el mapa del objeto. Por tanto, el mapa ha de tener información general del objeto como la tabla de métodos, el nombre de la familia o los protocolos adoptados, e información particular para cada categoría.

- ❖ **Def. 8-6.** El *mapa de la categoría* incluye información de descripción de la categoría, el desplazamiento de la categoría dentro del mapa, el desplazamiento de las variables compartidas y el desplazamiento de las variables de instancia.

La Figura 8-2 muestra un ejemplo esquematizado de la representación conceptual y física de objetos en XC. Los mapas almacenan también información acerca de la *geometría* del objeto que puede ser consultada en tiempo de ejecución:

- El mapa del objeto lleva cuenta del número de *bytes* que ocupa el objeto sin contar el mapa, el número de *bytes* del mapa, el número de mapas de categorías, la lista de categorías de extensión, el desplazamiento de las variables compartidas dentro del mapa y su número.
- El mapa de cada categoría almacena una referencia a la descripción en el *runtime* de la categoría, su desplazamiento dentro del mapa, el desplazamiento dentro del objeto de sus variables de instancia y el desplazamiento de sus variables compartidas.

Al decantarse por la construcción en tiempo de ejecución de la estructura final de los objetos, XC equilibra los requisitos de modularidad con la eficiencia. Cada mapa de objetos ha de ser creado una sola vez. A partir de entonces la clonación de nuevos objetos sólo requiere clonar la parte no compartida. Como todos los objetos en XC son iguales, en la Figura 8-2 da igual mandar un mensaje

clone al objeto A o al objeto A', pues ambos pertenecen a la misma familia. El mapa mantiene también referencias a un ejemplar de su familia y a un ejemplar de su familia padre por conveniencia.

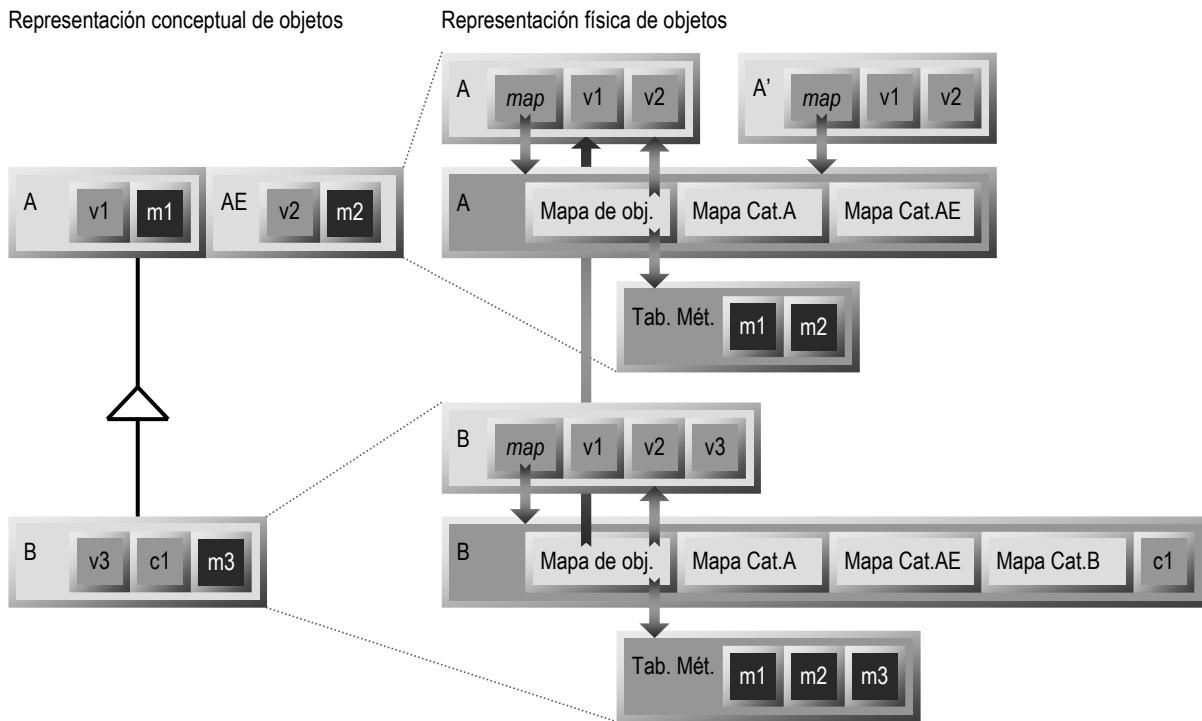


Figura 8-2 El mapa de un objeto.

A la izquierda, el prototipo A define una variable de instancia v1 y un método m1. Tiene una categoría de extensión AE con una segunda variable v2 y un método m2. De A deriva un prototipo B que define una variable v3, una variable compartida c1 y un método m3. A la derecha se presenta un esquema de los objetos A, A' y B en tiempo de ejecución. Los mapas incluyen referencia a un ejemplar de su familia y a un ejemplar de su familia padre, si es que tienen.

Chambers, a propósito de la implementación de mapas en SELF, resume excelentemente las cualidades fundamentales de los mapas, que son igualmente aplicables a XC:

“Desde el punto de vista de la implementación, los mapas se parecen mucho a las clases, y consiguen ahorro parecido de espacio para los datos compartidos. Además, el mapa de un objeto guarda sus propiedades estáticas, tal y como lo hace una clase en un lenguaje basado en clases. No obstante, los mapas son completamente invisibles al programador de SELF. Los programadores todavía operan en un mundo poblado por objetos autosuficientes que en principio podrían ser únicos. La implementación simplemente está optimizando la representación y la ejecución aplicando patrones de uso existentes, es decir, la presencia de familias de clonación.” [Chambers, 1992a; p. 62].

8.3.4 El algoritmo de creación de objetos

Hemos visto que los objetos en XC no se pueden construir estáticamente. Por motivos de modularidad, el compilador no asume en tiempo de compilación cuál es la geometría final de un objeto, puesto que ésta puede variar con el añadido de nuevas categorías. La descripción de cada una de las categorías tampoco asume la estructura del objeto final para permitir extensiones modulares. El compilador, en cambio, sí conoce la descripción del prototipo, la descripción de cada una de las categorías y el orden relativo entre ellas. Toda esta información queda disponible para que el *runtime* construya los objetos finales dinámicamente.

Los objetos son creados en dos fases:

1. Se crea el mapa del objeto, si éste no existe.

2. Se crea el objeto propiamente dicho.

Cuando se crea un mapa de objetos, muchas categorías primarias pueden estar involucradas. Cada una de ellas quizá tenga sus propias categorías de extensión. El mapa se ha de crear siempre a partir de una categoría primaria de referencia. La lista de categorías primarias heredadas es conocida revisando el atributo `parent` de cada una de ellas a partir de la categoría de referencia. La estructura de un mapa de objetos incluye la información común asociada al mapa y toda la información de categorías. Son usadas conjuntamente para tener una visión completa de las partes compartidas y no compartidas de cada objeto.

El algoritmo de creación del mapa del objeto a partir de una categoría primaria es el siguiente:

1. Descubrir cuántas categorías participan en la creación del mapa y ordenarlas usando el algoritmo del apartado 8.3.5.
2. Revisar la lista de categorías y acumular los valores de geometría del objeto: El tamaño total del mapa del objeto, el tamaño de la parte no compartida del objeto, el número de categorías, y el número y desplazamiento de las variables compartidas y no compartidas.
3. Pedir memoria para el mapa del objeto y llenar la información de geometría.
4. Resolver una referencia a un ejemplar de la familia padre a partir de la lista de categorías.
5. Resolver la tabla de métodos del objeto según el algoritmo del apartado 8.3.7.

8.3.5 El algoritmo de linearización de categorías

El algoritmo preserva el orden de la lista de categorías para facilitar la existencia de invariantes que puedan usarse durante optimizaciones. El orden comienza con la categoría raíz, luego sus extensiones y así con todas las categorías derivadas.

1. Añadir la categoría primaria a una lista vacía.
2. A partir de la categoría primaria, buscar la categoría primaria padre y añadirla al principio de la lista.
3. Repetir 2. hasta llegar a la categoría raíz.
4. A partir de la categoría raíz, insertar justo después de ella todas sus categorías de extensión en el orden dado por el compilador.
5. Repetir 4. con todas las categorías derivadas.

8.3.6 La estructura de la tabla de métodos

La implementación del soporte para metaprogramación ha de resolver las llamadas a los métodos anteriores y posteriores. En lenguajes dinámicos como SELF o Smalltalk-80, es posible modificar las tablas de métodos de los objetos base directamente, suplantando un método original por otro que realice las llamadas a los metamétodos y al método base. CLOS, otro lenguaje dinámico, directamente soporta metamétodos. En estos lenguajes el paso de argumentos a los metamétodos va en el entorno de referencia o *referencing environment*, y no suele tener demasiadas dificultades.

En lenguajes compilados tradicionales que usan una pila como técnica de paso de argumentos, es más complicado efectuar la llamada a los metamétodos. Simplemente, estos lenguajes no esperan que se realicen múltiples llamadas por cada llamada a método base. Así, extensiones a lenguajes compilados como C++ precisan el uso de preprocesadores que inserten las llamadas adecuadas a los metamétodos.

- ❖ **Def. 8-7.** Una *cadena de métodos* contiene la lista de métodos aplicables a un mensaje. Los métodos en una cadena de métodos son: *anteriores* o *before*, *base*, y *posteriores* o *after*.

Cada entrada de la tabla de métodos almacena una cadena de métodos. El método base es el método habitual de los modelos de objetos. Los métodos anteriores y posteriores permiten interceptar a los métodos base antes o después de que se ejecuten. En vez de definir objetos externos que intercepten los métodos base —que es la técnica usual de lenguajes que soportan metaprogramación— XC simplifica el modelo de objetos final usando las categorías para ese propósito. Los modelos de objetos tradicionales han de incorporar metamétodos en otros objetos porque los objetos originales no son directamente extensibles. Los metaobjetos tienden a formar una jerarquía paralela a la de los objetos que interceptan, complicando el modelo conceptual final. En XC, en cambio, las extensiones forman parte directamente del objeto extendido, obviando la necesidad de jerarquías paralelas y simplificando el esquema conceptual. El razonamiento con metamétodos puede dar lugar a confusión si las reglas de composición de los métodos es demasiado flexible. XC opta por dar una semántica bien definida. La Figura 8-3 muestra dos objetos A y B, donde el segundo hereda del primero, junto con los métodos base, anteriores y posteriores que definen cada una de sus categorías.

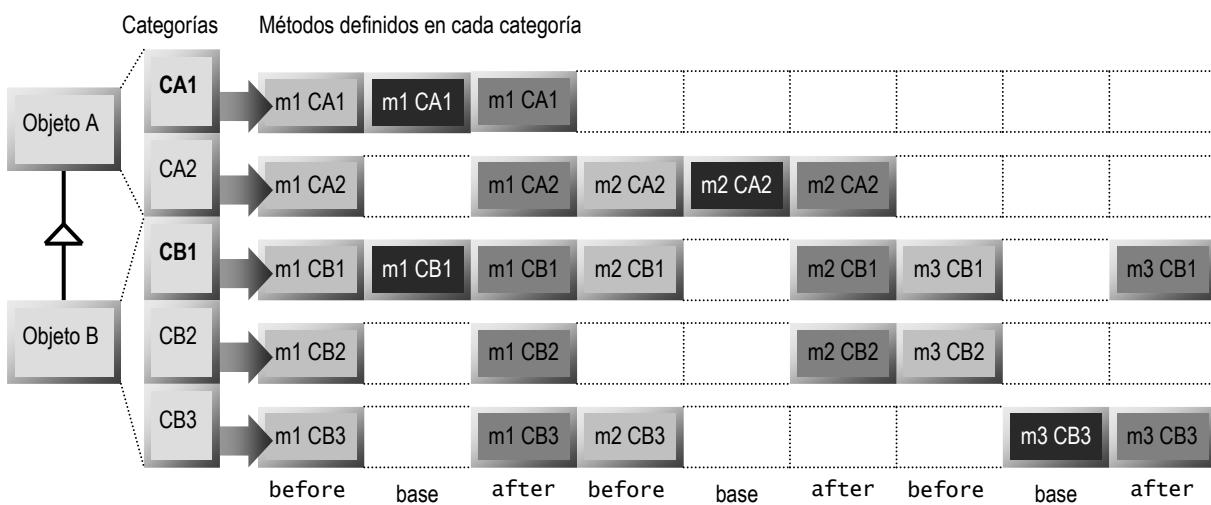


Figura 8-3 Métodos base, anteriores y posteriores un objeto.

El objeto A define dos categorías CA1 y CA2. El objeto B define tres categorías CB1, CB2 y CB3, heredando las categorías de A. Las categorías definen los métodos base m1, m2 y m3 y metamétodos para ellos. Las categorías primarias son CA1 y CB1. La linearización de categorías para A es {CA2, CA1}, para B es: {CB3, CB2, CB1, CA2, CA1}.

La Figura 8-4 presenta el resultado de la construcción de las tablas de métodos finales de los objetos A y B de la Figura 8-3, suponiendo que CA1 y CB1 son las categorías primarias, y que existen las siguientes linearizaciones de categorías, de mayor a menor prioridad:

- Para el objeto A: {CA2, CA1}.
- Para el objeto B: {CB3, CB2, CB1, CA2, CA1}.

La linearización determina la ordenación final de los metamétodos alrededor de los métodos base. Los métodos anteriores se organizan de mayor a menor prioridad, los métodos posteriores a la inversa. Esta organización es apropiada para implementar operaciones ortogonales como bloqueos entre múltiples hilos de ejecución, seguridad o persistencia. Cada entrada en la tabla de métodos es una cadena de métodos, que se ejecuta en sucesión desde el primer eslabón hasta el último.

La Figura 8-3 y la Figura 8-4 muestran un ejemplo general de organización de métodos y metamétodos, cuyo propósito es meramente ilustrativo del proceso de construcción de tablas de métodos. En la práctica, los metamétodos asociados con una funcionalidad ortogonal dada se implementarán todos en una única categoría, para centralizar cada aspecto ortogonal en una categoría distinta que pueda ser añadida o separada individualmente a los métodos base. Las cadenas de métodos son diferentes para cada prototipo. Son calculadas en el momento de crear el primer ejemplar de cada familia. El algoritmo que genera las tablas de métodos se describe a continuación.

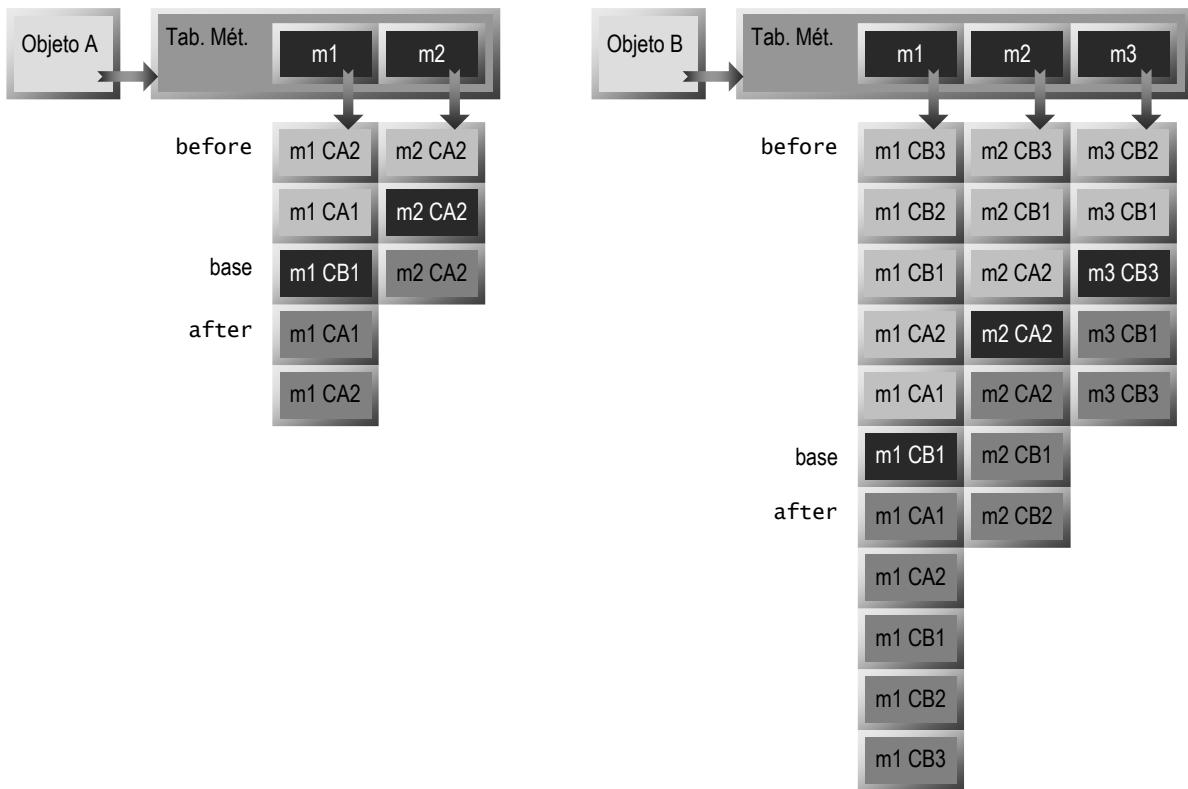


Figura 8-4 Tablas de métodos y cadenas de métodos asociadas.

Cada entrada de la tabla de métodos contiene una cadena de métodos con la ordenación final de los metamétodos y los métodos base. Los metamétodos siempre se ejecutan, los métodos base son sobreescritos según el árbol de herencia.

8.3.7 El algoritmo de construcción de la tabla de métodos

Después del registro de métodos, que ocurre cuando un módulo es registrado, todos los métodos de las categorías tienen un índice de selector único para cada método. Este índice es la entrada a la tabla de métodos, que se implementa como un *array*. El algoritmo recibe el mapa del objeto como parámetro, incluyendo la lista de categorías creadas con el algoritmo del subapartado 8.3.5.

En este punto, el mapa del objeto ya está creado y sus n categorías están inicializadas. La lista de categorías $c_1 \dots c_n$ tiene a c_1 como la raíz de herencia. Cada categoría primaria ha ordenado sus categorías de extensión de tal forma que, para una sublista $c_x \dots c_{x+m}$ de la lista anterior, los métodos *before* son ejecutados de c_x a c_{x+m} y los métodos *after* son ejecutados de c_{x+m} a c_x . Para precalcular la tabla de métodos con una única pasada se utiliza el algoritmo mostrado a continuación. Se usa cursiva para introducir comentarios relevantes en el algoritmo:

Se recorre la lista de categorías desde c_1 a c_n . Añadir cada método dentro de una cadena de métodos, asociada a cada entrada de la tabla de métodos.

Por cada categoría, y por cada método dentro de cada categoría, obtener su índice de selector único, accediendo a la tabla de métodos en la posición indicada por ese índice. Entonces:

Si no hay ningún método existente en la celda de la tabla de métodos

Añadir el nuevo método a la celda creando una nueva cadena de métodos.

En otro caso: comprobar el tipo del método existente

Si el método existente es un método **before**

Si el nuevo método es un método **before**

Añadir el nuevo método al inicio de la cadena.

En otro caso, si el nuevo método es un método base: *Intentar añadirlo justo después de todos los métodos **before** de la cadena. Se comienza comprobando el siguiente método de la cadena:*

Si el siguiente método es **after**

Añadir el nuevo método justo antes del método **after**.

En otro caso: *el siguiente método es base. Se ha encontrado un método base heredado. Dado que se recorre la lista de categorías desde la raíz de herencia, ahora se trata de algún objeto derivado.*

Sobrescribir el método heredado con el nuevo método.

En otro caso: *el nuevo método es un método after*

Añadir el método al final de la cadena.

En otro caso, si el método existente es un método **after**

Si el nuevo método es **before** o base

Añadir el nuevo método al principio de la cadena.

En otro caso: *el nuevo método es after*

Añadir el nuevo método al final de la cadena.

En otro caso: *el método existente es base*

Si el nuevo método es **before**

Añadir el nuevo método al inicio de la cadena.

En otro caso, si el nuevo método es base. *Se ha encontrado un método heredado. El nuevo método es de algún objeto derivado.*

Sobrescribir el método heredado con el nuevo método.

En otro caso: *el nuevo método es after*

Añadir el nuevo método al final de la cadena.

8.4 Implementación del envío de mensajes

Una introducción inicial al envío de mensajes en el contexto de los modelos de componentes se vio en el apartado 3.4.4. Posteriormente, en el apartado 4.4 se analizaron varias alternativas a la implementación de un mecanismo de envío de mensajes durante el estudio del modelo de objetos de XC. En ese apartado se argumenta que el mecanismo de envío elegido está basado en el uso de selectores, debido a que es un mecanismo modular y permite una implementación directa de los protocolos. También se decanta por el uso de envío de mensajes simple en vez de multimétodos.

En este apartado se analiza cómo el soporte en tiempo de ejecución implementa el envío de mensajes mediante selectores, que precisa la colaboración entre el soporte en tiempo de ejecución y el compilador. La implementación del envío de mensajes describe cuál es procedimiento de resolución de un método a partir de un mensaje, si un objeto responde a un mensaje, y cómo se invoca finalmente a un método. El resultado debe ser eficiente, puesto que en programas orientados a objetos y en llamadas a componentes el uso de mensajes es muy extenso.

8.4.1 Envío eficiente de mensajes

Mecanismos eficientes de envío de mensajes mediante selectores para lenguajes compilados sin máquina virtual se encuentran implementados en Objective-C. Son interesantes en este aspecto porque realizan un equilibrio entre la eficiencia de C y la flexibilidad de Smalltalk-80. Otras implementaciones que usan alguna variante de selectores se hallan en lenguajes con máquina virtual o interpretados como Smalltalk-80, SELF, CLOS o Dylan. Requieren distintas técnicas de implementación y por ello no se estudiarán aquí. Otros lenguajes como C++, Eiffel, o Modula-3 se han decantado por la implementación de envío de mensajes de SIMULA-67: la tabla virtual.

Existen dos formas de implementar un mecanismo de selección: (1) usando una función especial o *función mensajera* que abstrae el procedimiento de resolución de mensajes o (2) dejando que el compilador inserte el código de resolución de métodos en cada envío de mensaje. La segunda opción es ligeramente más eficiente, pero ocupa más espacio. En ambos casos es importante separar los dos elementos principales de un envío de mensajes: el algoritmo de resolución de métodos o *method lookup*, y la invocación final del método encontrado.

La primera implementación del mecanismo de selección que no usa una tabla virtual es la de Smalltalk-80 [Goldberg & Robson, 1989; pp. 420-421]. En [Cox & Novobilsky, 1991; pp. 151-156] se describe la implementación original de Objective-C usando una función mensajera. Está inspirada en la implementación de Smalltalk-80. La implementación actual de Objective-C de Apple [Apple, 2002] deriva de la implementada por NeXT [NeXT, 1995]. También usan una función mensajera. La implementación del compilador de Objective-C en GCC (*GNU Compiler Collection*), incluida la versión 3.3 [GNU, 2003], opta en cambio por la variante de inserción de código. Expande en línea el algoritmo de resolución de métodos y efectúa luego una llamada normal a función.

Realizar eficientemente un envío de mensajes consiste en seleccionar dentro de la tabla de métodos el método asociado a un mensaje y ejecutarlo. La tabla de métodos puede ser un diccionario que asocie nombres de mensaje con métodos. Es el caso de CommonObjects [Snyder, 1986b; p. 27], CLOS [Kiczales & Rodríguez, 1990; pp. 99-100] o SOM [Forman & Danforth, 1999; p. 19]. Una forma más eficiente es un *array*, donde el nombre de mensaje tiene asociado un número. Si ese número es conocido, entonces una simple indirección a través del *array* permite acceder al método buscado para ese mensaje. Como se ha visto en el apartado 4.4.3 *Selectores*, si se usan tablas virtuales el *array* es compacto y el consumo de memoria es mínimo. En cambio, si se escoge la técnica de selectores, se producen huecos en el *array* (véase la Figura 4-13). Estos huecos ocupan potencialmente mucho espacio. Resulta entonces prioritario encontrar algún modo de compactar esa información.

8.4.1.1 Técnicas de compactación de la tabla de métodos

En XC la tabla de métodos se asocia a cada prototipo. Es posible también usar una tabla de métodos global donde la clave de selección sea el selector junto con la clase o prototipo [Holst & Szafron, 1997; p. 280]. La ocupación de memoria es en principio similar, puesto que en ambos casos han de mantenerse duplicadas las entradas de un mismo selector para varias clases o prototipos distintos. Las tablas virtuales son una técnica eficiente de compactar las tablas de métodos, a costa de no mantener la relación 1:1 entre índices de selectores y mensajes. Cualquier implementación realista de las tablas de métodos debe reducir los huecos que genera la técnica de los selectores.

La estrategia de compactación usada en XC es la de *arrays dispersos* o *sparse arrays* [Driesen, 1993; pp. 262-263], usada también en la implementación de Objective-C en GCC. Un *array disperso* es una estructura en *array* en dos niveles, como muestra la Figura 8-5. El primer nivel almacena referencias a trozos del *array* o *buckets* que a su vez contienen los datos. Las propiedades de esta estructura son interesantes. Por un lado, un redimensionado del *array* es más rápido, porque sólo el *array* de referencias a trozos debe redimensionarse. La copia de valores al nuevo *array* es también más rápido porque sólo se copian las referencias del primer nivel. El acceso a los elementos del *array* es un poco más lento, ya que precisa una indirección más.

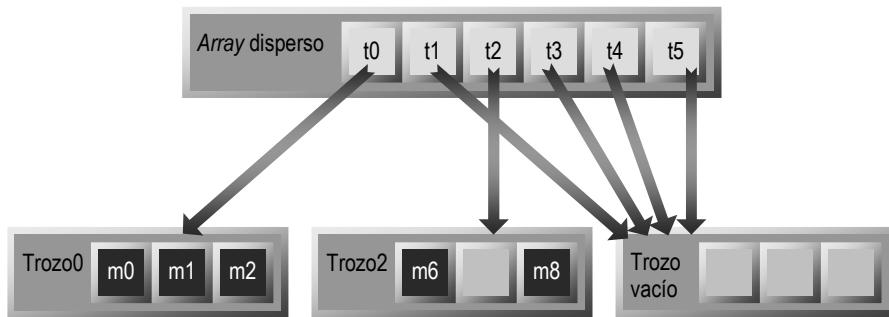


Figura 8-5 *Arrays dispersos*.

La tabla de métodos se organiza como un *array* de dos niveles para reducir la memoria usada por los huecos. La tabla mostrada almacena 18 métodos, que se distribuyen en las posiciones indicadas por los índices de sus selectores.

Con *arrays dispersos*, una ordenación adecuada de los selectores da lugar a reducciones significativas de memoria, porque los bloques de métodos asociados a las clases o prototipos heredados pueden compartirse debido a la doble indirección de los *arrays dispersos*. La implementación actual de XC no realiza esta última optimización, pero puede intuirse con una descripción sencilla. Si en la Figura 8-5 suponemos que los métodos m0, m1 y m2 son métodos heredados de un prototipo A, entonces por la propiedad de unicidad de los selectores siempre se encontrarán en los mismos desplazamientos dentro de la tabla de métodos. El *array* disperso mostrado en la Figura 8-5 contiene, además, los métodos m6 y m8, que podemos suponer de un prototipo B derivado de A. Un nuevo prototipo C que herede de A puede compartir el primer trozo con el prototipo B usando una referencia desde su propio *array* disperso, si B y C no sobreescriben los métodos de A. Igualmente, puede compartir el mismo trozo vacío.

Otras técnicas de compactación son posibles. Las tres revisadas a continuación son más prácticas en sistemas que ya han evolucionado y se consideran razonablemente estables, puesto que el tiempo de cómputo de las tablas es relativamente largo. La última técnica es aplicable también en entornos de desarrollo. Al igual que con los *arrays dispersos*, con estas técnicas el tiempo de resolución de métodos es constante. La selección de índices para selectores deja de tener relación 1:1 con los mensajes, pero aun así los selectores globales son posibles en la práctica gracias al uso de algoritmos más elaborados. Los algoritmos hacen comprobaciones en tiempo de ejecución en el lugar de llamada para poder volver a identificar selectores ambiguos y así recuperar la identificación global de los selectores. Una desventaja de todas estas técnicas es que presuponen que se tiene acceso al código fuente para realizar un análisis global de los mensajes enviados y el árbol de herencia, requisito necesario para seleccionar adecuadamente los índices de los selectores. Esta es la razón principal por la que se ha desestimado su incorporación a XC, ya que la selección de índices no sólo debe ser eficiente en tiempo de ejecución, sino tener también un comportamiento modular y compilación separada, como se ha visto en el apartado 4.4.3 *Selectores*. La selección debe ser también compatible con un conocimiento estrictamente parcial del sistema siguiendo la premisa cuarta de diseño (Def. 3-15), tanto en tiempo de desarrollo como en tiempo de ejecución, porque en presencia de bibliotecas

dinámicas no se puede asegurar que en tiempo de ejecución se tiene acceso a todo el código que forma parte de un sistema.

- En la técnica de *coloración de selectores* o *selector coloring* el color de un selector es un número único para una clase o prototipo dado. Dos selectores pueden compartir un color si nunca aparecen dentro de la misma clase o prototipo. De esta manera se reducen los huecos. La selección de los colores debe resolver un problema equivalente al problema de coloración de un grafo en teoría de grafos, que es NP completo [Driesen, Hözle & Vitek, 1995; p. 261]. Existen aproximaciones al problema de coloración en tiempo polinomial [André & Royer, 1992; p. 116], pero el tiempo de construcción del grafo sigue siendo prohibitivo [André & Royer, 1992; p. 120]. La variante de *coloración incremental de selectores* [André & Royer, 1992] necesita menos tiempo —aunque todavía significativo— volviendo más prácticos a los algoritmos de coloración.
- La técnica de *desplazamiento de filas* o *row displacement* considera a cada tabla de métodos de una clase o prototipo como una fila en una matriz bidimensional. Cada fila contiene los huecos mostrados en la Figura 4-13. La idea consiste en desplazar las filas de tal modo que los selectores de distintas clases o prototipos no se superpongan, aplanándose la matriz bidimensional en un array unidimensional. El problema que resuelve es similar a la minimización de tablas para *parsers* guiados por tablas [Driesen, Hözle & Vitek, 1995; p. 262].
- La técnica de *índices de selectores compactos en tablas de métodos* o *compact selector-indexed dispatch tables* [Vitek & Nigel Horspool, 1994] es la que menos tiempo requiere para computar las tablas de métodos, hasta el punto de ser aplicables a entornos de desarrollo. La técnica separa dos tipos de selectores: los estándar y los conflictivos. Se mantienen dos tablas de métodos, una para los métodos estándar y otra para los conflictivos. Los selectores se compactan en las tablas en varias fases, de forma tal que todavía son discriminables por código en línea que se añade en el lugar de llamada. Los índices de compactación son muy buenos, hasta un 99% para grandes bibliotecas de clases [Vitek & Nigel Horspool, 1994; p. 433].

8.4.1.2 Cachés de envío de mensajes en el lugar de llamada

La inferencia de tipos puede eliminar llamadas polimórficas a costa de necesitar recompilación. Esta técnica se ensayó originalmente en el compilador de SELF [Hözle, Chambers & Ungar, 1991], [Chambers, 1992a], donde recibe el nombre de *cachés polimórficas en línea* o *polymorphic inline caches*. La inferencia tiene más sentido en lenguajes que usan máquinas virtuales como SELF, Smalltalk-80 o Java. La máquina virtual puede monitorizar la ejecución de métodos y reconocer dinámicamente si en un lugar de llamada determinado —es decir, durante el envío de un mensaje a una variable polimórfica— están respondiendo múltiples métodos, sólo unos pocos, o quizás uno sólo. La máquina virtual tiene entonces la capacidad de cambiar el código intermedio compilado para evitar procedimientos de resolución de métodos complejos, traduciéndolos por llamadas más simples.

Por ejemplo, si en un mismo lugar de llamada se producen en tiempo de ejecución dos llamadas a métodos distintos, quiere decir que están en juego sólo dos clases o prototipos. En ese momento la máquina virtual tiene más conocimiento sobre ese lugar de llamada que el compilador. En vez de suponer que se trata de un objeto anónimo, cuyo tipo es el declarado en la variable polimórfica o cualquiera de sus subtipos, la máquina virtual ahora conoce que sólo dos subtipos están siendo utilizados en ese lugar de llamada. Teniendo en cuenta esta nueva información, la máquina virtual puede sustituir el código original que envía un mensaje a un objeto desconocido con dos comparaciones, que comprueban si el objeto pertenece a alguna de las dos clases o prototipos que se están usando realmente. El caso por defecto continúa siendo un procedimiento general de resolución de métodos, más costoso. Pero si una de las dos clases o prototipos identificados entran en juego, entonces el coste se transforma en una o dos comparaciones más una llamada directa a función, mejorando la eficiencia en tiempo de ejecución y actuando como una caché. Las cachés en el lugar de llamada son complementarias a las tablas de métodos. Estas últimas guardan generalmente los resultados del algoritmo de resolución de métodos en las clases o prototipos, las primeras lo hacen

antes de acceder a la tabla de métodos, pero en vez de guardar únicamente datos, almacenan el código optimizado de resolución de métodos para unos pocos casos muy comunes.

Es posible aplicar la misma técnica a lenguajes compilados usando un análisis global, estudiado por ejemplo en [Zendra, Colnet & Collin, 1997] para un compilador de Eiffel. Si se tiene acceso al código fuente y se puede construir el árbol de tipos de datos que identifica exactamente cuántas clases o prototipos potencialmente están involucrados en un lugar de llamada dado, es posible compilar el código de las cachés en el lugar de llamada antes de ejecutar el código. Si se usa compilación a código nativo —reconocible directamente por una máquina— el proceso se complica, porque para habilitar extensiones en el futuro es necesario recompilar el código resultante del análisis global. El análisis global es, en principio, poco modular, pues depende de información en otros módulos para realizar la optimización final. En cambio, las cachés de llamada introducidas por máquinas virtuales pueden ser modulares si construyen las cachés en demanda, a medida que el código se ejecuta. En este sentido, la opción de compilar a código objeto nativo, limita el campo de aplicación de estas optimizaciones, aunque no las elimina por completo. Por ejemplo, si se ha terminado una aplicación y no se espera que terceros la extiendan, es posible realizar un análisis global que optimice el código justo antes de la entrega de la aplicación al cliente.

XC usa una variante de la técnica de cachés polimórficas, cuyo origen está en la implementación de Objective-C [Cox & Novobilsky, 1991; pp. 151-156]. Técnicamente, la implementación usa una caché monomórfica en el lugar de llamada [Driesen, Hölzle & Vitek, 1995; p. 258], donde se almacena el selector asociado a un mensaje, no directamente el método a ejecutar. Es una caché monomórfica porque sólo almacena un valor, no múltiples. Además, no genera código —que requeriría un análisis de tipos no modular— sino que sólo almacena datos, manteniendo (1) modularidad y (2) resolución en tiempo de compilación debido a que XC no usa ninguna máquina virtual.

Existen cachés separadas por módulo, para mantener la modularidad. El compilador puede resolver las referencias a la caché de selectores estáticamente en tiempo de compilación si tiene acceso al código fuente. Al limitar a un módulo la visibilidad de código fuente debido a las premisas de diseño de XC, la caché debe construirse también por módulo. Las cachés son compartidas entre los mensajes enviados en un módulo siempre que tengan el mismo selector. De esta forma, si se envía el mensaje `m1` diez veces en un módulo, sólo una entrada de caché es usada. La Figura 8-6 muestra esta idea. Si bien la referencia a la caché es resuelta durante la compilación, su valor —es decir, el selector finalmente asociado a `m1`— sólo es conocido en tiempo de ejecución. Por esa razón, la resolución de los selectores asociados a los métodos y almacenados en la caché de selectores se produce durante el registro de cada módulo.

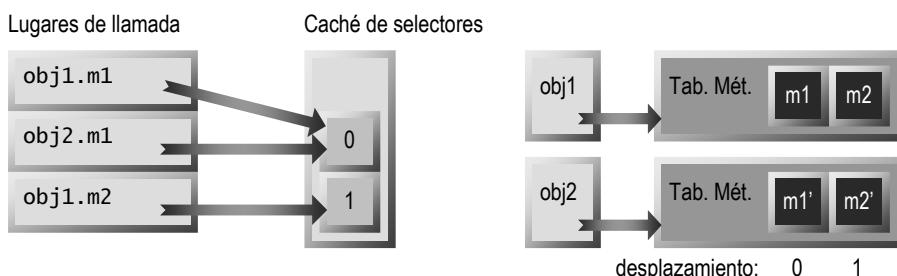


Figura 8-6 Caché de selectores monomórfica en el lugar de llamada.

El selector usado por `obj1` y `obj2` al enviar el mensaje `m1` es el mismo, pero los métodos que responden no tienen porqué serlo. La caché de lugar de llamada para el valor del selector es compartida por ambos lugares de llamada, dentro de un módulo dado. El selector usado por `obj1` para el mensaje `m2` es diferente.

El funcionamiento de la caché de selectores es el siguiente. Durante la compilación, se resuelve dónde se almacenará la caché de selectores en cada lugar de llamada, generando una referencia a la zona de almacenamiento por cada lugar de llamada, y compartiendo aquellas entradas asociadas a mensajes

iguales. Durante el registro, se asigna un selector a cada mensaje. En la Figura 8-6 se asocia el selector 0 al mensaje `m1` y el selector 1 al mensaje `m2`, que son precisamente los desplazamientos en la tabla de métodos de los objetos `obj1` y `obj2`, usados para seleccionar las implementaciones de los mensajes `m1` y `m2`. Cuando se envía el mensaje `m1` al objeto `obj1`, la función mensajera accede a la caché de selectores que indica la posición exacta en la tabla de métodos del método asociado a `m1`, en este caso en el desplazamiento 0. La caché mejora la eficiencia de las llamadas, al no necesitar realizar una búsqueda exhaustiva del método asociado a un mensaje, o una búsqueda por clave en una tabla *hash* tradicional.

8.4.2 Registro de activación de métodos

El envío de mensajes en XC ejecuta la cadena de métodos asociada a cada mensaje. Es de extrema importancia que el envío de mensajes sea eficiente. Al usar protocolos, la implementación de un objeto en general no está asociada únicamente con sus protocolos, por lo que el número de mensajes enviados en un programa es muy significativo. Si el envío del mensaje tiene argumentos, entonces existe el riesgo de que la ejecución de una cadena de métodos sea demasiado lenta, porque cada metamétodo necesita que se le pasen los mismos parámetros que al método base, para que pueda tomar acciones con respecto al contexto de ejecución del método base³⁹. Una cadena de métodos que contenga un método anterior, un método base y otro posterior es equivalente al siguiente código en C:

```
beforeMethod(metaself, self, arg1, arg2, arg3);
baseMethod(self, arg1, arg2, arg3);
afterMethod(metaself, self, arg1, arg2, arg3);
```

donde `metaself` es la referencia al metaobjeto y `self` es la referencia al objeto base⁴⁰. En lenguajes compilados que usen una pila la sobrecarga puede ser notable, empeorando con el número de argumentos y número de metamétodos a ejecutar. Otra variante consiste en empaquetar los argumentos antes de realizar las llamadas:

```
addArg(argData, arg1);
addArg(argData, arg2);
addArg(argData, arg3);
beforeMethod(metaself, self, argData);
baseMethod(self, argData);
afterMethod(metaself, self, argData);
```

Esta técnica reduce el número de datos copiados en la pila, pero introduce coste adicional en el empaquetamiento y acceso a cada argumento. Esta técnica era usada por Open C++ versión 1. En XC, las extensiones con metamétodos se efectúan con categorías que pertenecen al objeto base, no usando metaobjetos externos. Por tanto, el parámetro `metaself` no es necesario y la estructura de la pila durante las llamadas a los metamétodos y el método base permanece invariante. La optimización realizada en XC saca partido de este invariante, aprovechando el marco de pila para todos los métodos que forman parte de una misma cadena de métodos. Los datos son pasados en la pila una sola vez y la ejecución de cada método de la cadena reutiliza el registro de activación original. La función mensajera `xc_sendmsg` se encarga del proceso.

Desgraciadamente, no es posible implementar de manera portable en C el reuso del registro de activación, pero sí es factible minimizar el código no portable y seguir compilando a C el resto del código. Además, el encadenamiento de ejecución de métodos requiere un registro de activación mayor que el estándar de una función C. Por esas razones, XC opta por definir un nuevo registro de activación para el envío optimizado de mensajes y la ejecución de sus métodos asociados, que es encapsulado y

³⁹ No todos los metamétodos necesitan acceder a todos los argumentos. Algunos metamétodos quizás no accedan a ninguno, disminuyendo su coste de llamada. Es posible realizar llamadas a metamétodos que no incluyan los argumentos sobrantes, a costa de realizar llamadas separadas a cada metamétodo. Por razones de optimización, ese no es el camino seguido.

⁴⁰ En esta discusión estamos suponiendo inicialmente el caso general, donde los metaobjetos son objetos distintos a los objetos base y necesitan su propio `self`.

gestionado internamente por la función mensajera en ensamblador. Supongamos el envío del mensaje `m1` al objeto `obj` con dos argumentos, y cuyo selector se encuentra en la variable `sel`. El compilador genera una llamada en C similar a la siguiente:

```
xc_sendmsg (obj, sel, arg1, arg2);
```

La llamada involucra dos registros de activación distintos, mostrados en la Figura 8-7, e implementados para la arquitectura x86 de Intel [Intel, 1999] en el prototipo del compilador. El primer registro de activación es el usual de C, mostrado a la izquierda de la figura. Se crea al llamar a la función mensajera. El segundo registro de activación se muestra a la derecha. Es el registro de activación de métodos y es controlado por la función mensajera⁴¹.

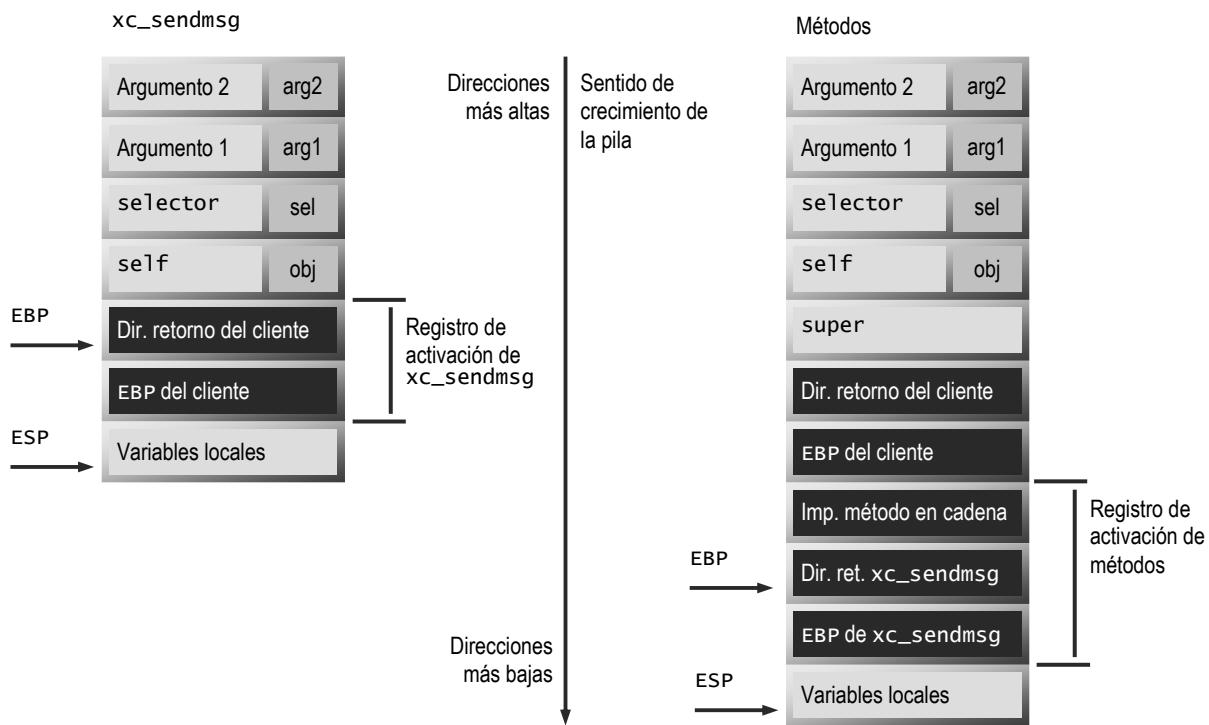


Figura 8-7 Registros de activación.

A la izquierda se muestra el registro de activación de la función mensajera, a la derecha el registro de activación de métodos. EBP apunta al registro de activación actual, ESP es el puntero de pila.

En la arquitectura x86 se usan las convenciones siguientes: El puntero de pila se almacena en `ESP`. El registro de activación de cada función se guarda en `EBP`, y el valor de retorno se pasa en el registro `EAX`. Una llamada a función C desde un código cliente pone los argumentos en la pila en orden inverso a su declaración y luego ejecuta una instrucción `call`, que preserva la dirección de retorno también en la pila. El prólogo de una función C guarda la posición del registro de activación de la función que llama, almacenado en `EBP`, y actualiza `EBP` para que apunte al nuevo registro de activación:

```
push ebp  
mov ebp, esp
```

Los parámetros quedan en desplazamientos positivos de `EBP` y las variables locales en desplazamientos negativos. El epílogo de una función restaura el registro de activación anterior guardado en la pila en `EBP` y retorna.

⁴¹ El parámetro `super` del segundo registro de activación es usado para envíos de mensajes que usan la cláusula `super`. Se explica posteriormente en el apartado 8.4.3.1.

```
pop ebp
ret
```

La implementación de la función mensajera, que se ve en el apartado 8.4.3, realiza el encadenamiento de la ejecución de métodos y al final restaura el registro de activación original. El registro de activación de métodos es mayor que el de funciones C, y mantiene el encadenamiento de direcciones de retorno usadas por los depuradores para localizar las direcciones de funciones en la pila. Para explotar la compilación y optimización final realizada por compiladores existentes de C y C++, y para reaprovechar los parámetros del mensaje que ya se encuentran en la pila, el prototipo del compilador de XC compila los métodos como funciones normales C con un parámetro fantasma, que nunca se usa. Sirve para indicar al compilador de C que genere los desplazamientos correctos de los parámetros en la pila, una vez la función mensajera modifica el registro de activación. Por ejemplo, el método `alloc` del prototipo `Prototype`, que se encuentra en el módulo `xc.Lang`:

```
prototype Prototype
{
    // ...
    self alloc
    {
        return xc.System.ObjectMemory.cloneFrom: self;
    }
}
```

es compilado a la siguiente función C que incluye el parámetro de tipo `struct xc_ghost` no usado. Su tamaño es exactamente el del registro de activación de métodos más el de la función mensajera, permitiendo que el compilador acceda a los desplazamientos correctos en la pila de las variables `super`, `self` y `selector` sin ningún coste extra.

```
struct xc_object *
XC_Lang_Prototype_alloc (struct xc_ghost _,
                         struct xc_object * self, struct xc_selector * selector)
{
    return (struct xc_object *) (XC_System_ObjectMemory_cloneFrom_ (self));
}
```

El compilador también genera números de línea de referencia al fichero fuente original en XC para facilitar la depuración simbólica de código XC directamente, incluyendo ejecución paso a paso y puntos de ruptura. En el caso particular de cumplir ciertas condiciones de optimización —que se estudian en el apartado 8.7.1 *Modificadores de optimización de objetos*— el compilador puede asegurar que una llamada directa al método es equivalente a enviar un mensaje, y generar una función que no incluya el parámetro fantasma para las llamadas directas.

8.4.3 La función mensajera

La función mensajera esconde los aspectos no portables del proceso de envío de mensajes. Es responsable de las siguientes tareas:

- Selección de la cadena de métodos a ejecutar.
- Creación del registro de activación de métodos usado por todos los métodos de una misma cadena.
- Secuenciación de la ejecución de los métodos de la cadena, preservando el valor de retorno del método base.
- Reestablecimiento del registro de activación del cliente al final de la cadena y devolución del valor de retorno al cliente.

La declaración en C de la función mensajera es la siguiente:

```
struct xc_object *
xc_sendmsg (struct xc_object * self, struct xc_selector * sel, ...);
```

La búsqueda de la cadena de métodos a ejecutar como respuesta a un mensaje se hace dentro de la función mensajera usando un *array* disperso, descrito en el apartado 8.4.1.1. En el apéndice C se muestra el código de búsqueda y de la función mensajera. Medidas preliminares de tiempos indican que la implementación actual de la función mensajera es aproximadamente el doble de lenta que una llamada *virtual* en C++ que no tenga cuerpo. En caso de envíos de más de dos parámetros y cadenas de métodos mayores la diferencia disminuye, porque se están reutilizando los parámetros almacenados en la pila, que incurren en coste adicional en C++. Para cuerpos de métodos con operaciones más complejas, el coste del envío del mensaje también disminuye, puesto que el porcentaje dedicado al envío del mensaje del tiempo total de ejecución del método es menor. Téngase en cuenta que la función mensajera en XC envía mensajes preservando la modularidad, cosa que no ocurre en C++.

La implementación actual en el prototipo del compilador, si bien no definitiva, muestra que es factible construir sistemas de envío de mensajes que ejecuten cadenas de métodos eficientemente. Aún existe espacio para nuevas optimizaciones. A continuación se detallan posibles mejoras a la función mensajera actual, pero no implementadas todavía:

- El bucle interno de la función mensajera se mueve a través de la cadena de métodos recogiendo las direcciones de los métodos y realizando las llamadas. La implementación existente de la cadena de métodos usa una lista enlazada terminada en `NULL`. En cada acceso, el recorrido por la lista enlazada produce una indirección más que con un *array*. Es probable también que una implementación con *array* mejore la localidad de referencias.
- La implementación del *array* disperso puede tener una indirección menos en cada acceso.
- Un compilador que generase código nativo podría crear directamente los prólogos y epílogos de métodos sin dificultades, utilizando un segundo registro de activación similar al mostrado en la Figura 8-7, y efectuando la labor que ahora realiza `xc_sendmsg` para aprovechar un compilador de C o C++ existente sin modificaciones. En ese caso, la función mensajera no sería estrictamente necesaria y probablemente mejoraría los tiempos de envío de mensajes.

8.4.3.1 Funciones mensajeras especiales

La función mensajera anterior espera que el objeto `self` sea un objeto normal, no un objeto optimizado. En un objeto normal existe siempre una referencia al mapa de la familia de objetos. En un objeto optimizado no, como se verá en el apartado 8.7.2 *Optimización de tipos básicos*. Si el objeto no tiene mapa, la función mensajera anterior fallará en la búsqueda de la tabla de métodos del objeto optimizado. El sistema de tipos que expone XC a sus usuarios es homogéneo. Eso quiere decir que un objeto optimizado también debe poder heredar métodos definidos en sus antecesores y ser capaz de responder a los mensajes asociados, como si fuese un objeto normal. Para implementar estos comportamientos, el compilador siempre genera un objeto prototipo normal para todas las familias de objetos optimizados. Al usar un sistema estático de control de tipos, el compilador puede identificar que se está enviando un mensaje a un objeto optimizado cuyo método asociado es heredado. En este caso, el compilador genera una llamada a una función mensajera especial que usa el mapa del objeto prototipo, en vez del mapa —inexistente— del objeto optimizado. Por ejemplo, para el tipo básico `Unsigned`, la función mensajera usada es:

```
struct xc_object *
xc_unsigned_sendmsg (xc_raw_unsigned self, struct xc_selector *sel, ...);
```

Los envíos de mensajes usando la cláusula `super` también usan una función mensajera distinta. La cláusula `super` indica que el procedimiento de búsqueda del método asociado a un mensaje ha de comenzar en el prototipo padre. En este último caso, la declaración de la función mensajera varía, incluyendo un ejemplar que referencia a un objeto de la familia de su prototipo padre. El ejemplar es usado por la función mensajera para conocer cuál es su mapa asociado y la tabla de métodos que debe utilizarse para seleccionar la cadena de métodos. La declaración de la función mensajera especial es:

```
struct xc_object *  
xc_super_sendmsg (struct xc_object *super,  
                  struct xc_object *self, struct xc_selector *sel, ...);
```

Cuando el compilador genera una llamada a `super`, pone en la pila a un ejemplar de la familia padre, en la posición indicada como `super` en el registro de activación de la Figura 8-7, actuando como un parámetro más a la función mensajera. Cuando se genera una llamada a la función mensajera normal, el parámetro `super` no se usa, aunque su espacio sí se reserva en el registro de activación.

8.5 Inicialización

La inicialización del soporte en tiempo de ejecución es un proceso que ha de realizarse con sumo cuidado debido a que XC contiene muchas definiciones recursivas que han de compilarse correctamente. Buena parte del propio *runtime* está escrito directamente en XC, por lo que implícitamente existen muchas declaraciones mutuamente recursivas que, de alguna forma, deben ser resueltas. Para tener una idea de este problema veamos la declaración del protocolo de módulos del soporte en tiempo de ejecución realizada en el módulo `XC.Runtime`:

```
final protocol Module extends: RuntimeObject  
{  
    property const String name;  
    property const Unsigned flags;  
    property const Unsigned metaFlags;  
    property const Unsigned languageversion;  
    property const Unsigned version;  
    property const Unsigned state;  
  
    property const ModuleImportArray moduleImports;  
    property const CategoryArray categories;  
}
```

En primer lugar el protocolo `Module` define un conjunto de operaciones de objetos de tipo `Module` implementado en `XC.Runtime`, que responde también al protocolo de objetos básicos heredado de `RuntimeObject`. Las propiedades hacen referencia a protocolos de otras familias de objetos, como `String` y `Unsigned`. `RuntimeObject` a su vez tiene dependencias con `String`, `Unsigned` y otros protocolos.

Algunas de las propiedades de un módulo son *arrays*. Uno de ellos es la lista de módulos importados, otro las categorías y prototipos definidos en el módulo. Veamos uno como ejemplo. La declaración del primero de ellos es:

```
final protocol ModuleImportArray = Array <Element = ModuleImport>;
```

El *array* de módulos importados hereda del protocolo `Array`, que es un tipo genérico cuya variable de tipo definida es `Element`. Se desea que el *array* de módulos importados contenga sólo módulos importados. Para obtener errores en tiempo de compilación se especializa el tipo genérico `Array` al tipo `ModuleImportArray`, donde la variable de tipo está asociada a objetos cuyo protocolo sea `ModuleImport`. La declaración de este último tipo corresponde con una referencia a un módulo importado:

```
final protocol ModuleImport extends: RuntimeObject  
{  
    property const String name;  
    property const Unsigned flags;  
}
```

La declaración de `ModuleImport` no tiene novedades sintácticas. Las declaraciones públicas de la mayor parte de los protocolos de `XC.Runtime` son `final`, para evitar que puedan ser suplantadas por otras definiciones incompatibles con el código generado por el compilador.

Si bien la inicialización correcta de un conjunto extenso de declaraciones mutuamente recursivas tiene sus complicaciones, el beneficio obtenido es una interfaz de reflexión más simple y consistente. Por ejemplo, para listar los módulos importados por otro módulo se usa un fragmento de código similar al siguiente:

```
Module      module = XC.Runtime;
ModuleImport modImp;
Unsigned    length = module.moduleImports.length;

Console.print: "Module name: '" + module.name + "'\n";
for (Unsigned i = 0; i < length; i++)
{
    modImp = module.moduleImports[i];
    Console.print: "Module name: '" + modImp.name + "'\n";
}
```

La primera sentencia asigna el módulo registrado en el *runtime* cuyo identificador es `XC.Runtime` a una variable de tipo `Module`. A partir de ahí, el acceso mediante reflexión es inmediato, de igual eficiencia que el acceso a otros objetos, e intuitivo si se conoce la sintaxis del lenguaje. Todas las estructuras del *runtime* de XC son objetos del lenguaje y se acceden directamente como en cualquier otra expresión. Contrastá esta circunstancia con Java, donde el acceso a los mecanismos de reflexión impone un coste adicional nada despreciable, debido a que ha de hacerse una traducción intermedia entre el modelo de objetos proporcionado por el paquete `java.lang.reflect` y las estructuras internas del *runtime* de Java.

El acceso al *array* `moduleImports` usa exactamente el mismo código que el definido en el *array* estándar del lenguaje `xc.collections.Arrays.Array`. Durante el diseño del lenguaje, el añadido de propiedades y métodos asociados a los objetos del *runtime* también se simplifica notablemente: se puede implementar desde dentro del lenguaje y aparecer disponible automáticamente como parte del protocolo de reflexión al recompilar el módulo `XC.Runtime`.

8.5.1 Registro de módulos

La unidad de carga o registro en el soporte en tiempo de ejecución es el módulo. Cada módulo contiene toda la información necesaria para asegurar que, una vez cargado correctamente, es posible usar los prototipos definidos en el módulo. El registro de módulos comienza con la comprobación de dependencias de los módulos importados. Las dependencias almacenadas en cada módulo son transitivas (véase el apartado 5.5.5.1 *Transitividad*). Un módulo no se activa a no ser que todos sus módulos dependientes hayan sido activados previamente. La activación de módulos se produce en dos fases para permitir interdependencias entre los módulos más básicos del soporte en tiempo de ejecución:

1. Se registran las tablas de los módulos, cuya estructura se resume en el apartado 8.6.3 *Generación de código para módulos*. Se registra la tabla de selectores en primer lugar, que resuelve los índices de los selectores. Luego se comprueban los módulos importados. Si hay dependencias pendientes, se intenta registrar primero el módulo dependiente. Si no se consigue se aborta el registro del módulo. En cambio, si los módulos dependientes se registran correctamente, se procede a registrar las categorías definidas y con ellas sus métodos.
2. Se terminan de registrar las categorías. El registro de categorías en la primera fase es parcial, existiendo referencias al objeto `null` (véase el apartado 8.5.2) que deben ser resueltas antes de activar finalmente el módulo. Por último, se construyen los ejemplares por defecto de cada prototipo enviando el mensaje `clone`. De esta forma, la definición del código fuente de todos los objetos puede realizarse íntegramente dentro de XC.

8.5.2 El objeto `null`

Existe un objeto que requiere especial atención para entender el proceso de inicialización del soporte en tiempo de ejecución: el objeto `null`. Es usado para identificar un objeto nulo, no válido o no inicializado. Para mantener la uniformidad en el sistema de tipos, es imprescindible que el objeto `null` sea igual a un objeto normal en el *runtime*. Es decir, debe existir como tal, debe ser capaz de responder a mensajes, asignarse a propiedades de objetos y seguir el protocolo `Object`. Si no fuese así, las sentencias siguientes no se podrían verificar correctamente en tiempo de compilación y, por tanto, no serían seguras:

```
Object a, b = null;
a = b.clone;
```

¿Cómo descubre el compilador que `b` no puede responder al mensaje `clone`, si `b` es de tipo `Object` y `Object` declara un mensaje `clone`? Si la segunda sentencia se encuentra lejos de la primera, el compilador no puede inferir de ninguna manera que `b` llegó a ser `null` en algún momento, puesto que ese valor puede depender de la historia del programa en tiempo de ejecución.

En el lenguaje Objective-C, por ejemplo, el objeto `nil` (equivalente al objeto `null` en XC) está definido como un objeto cuya referencia es igual a cero. Tal referencia se encuentra fuera del espacio de memoria de cualquier proceso, y es equivalente al valor de macro `NULL` de C. El inconveniente de esta opción es que es posible enviar un mensaje a un objeto cuya dirección es cero, y obtener una excepción del sistema operativo de error de acceso a memoria que comúnmente termina con el proceso en curso.

Como el valor asignado a una variable o a una propiedad varía en tiempo de ejecución, es imposible conocer a priori si alguna de ellas contiene una referencia a cero. Para evitar la excepción del sistema operativo, sería necesario incluir en todos los envíos de mensaje una comprobación de la referencia implícita `self` a `null`, en cuyo caso se podría bien ignorar el mensaje, bien enviarlo manualmente a un objeto `null` predefinido. El mejor lugar para poner esta comprobación es la función mensajera, porque en otro caso se incuraría en añadido de código extra por cada envío de mensaje de cada cliente. Esta verificación es hecha, por ejemplo, por la implementación de Objective-C de Apple [Apple, 2002a].

La comprobación de `null` en la función mensajera puede parecer despreciable, pero su efecto en un programa en ejecución no lo es, ya que el número de envíos de mensaje en un programa de XC es muy alto. La existencia de un sistema de tipos con protocolos independientes de implementación y soporte de componentes obliga a usar envío de mensajes para casi todas las operaciones. Es importante que el envío de mensajes sea tan eficiente como sea posible.

El proceso de creación de objetos en XC es mucho más complejo que una petición de memoria. Sin embargo, para crear objetos es necesario que muchas estructuras del *runtime* estén inicializadas, con lo que nos encontramos otra vez con el problema de recursividad mutua de definiciones. El objeto `null` debe seguir un protocolo que ha de indicar al sistema de tipos que no se puede heredar de `null`, que tiene un tamaño fijo y que deriva del protocolo `Object`. Esta es la definición del protocolo:

```
final protocol Null extends: Object {}
```

El protocolo debe ser explícito en parte para evitar tener que poner parches especiales en el sistema de tipos asociados a la gestión del tipo del objeto `null`. Al heredar de `Object`, el objeto `null` hereda las definiciones de todas las categorías que se puedan añadir a `Object`. Asimismo, la declaración explícita permite implementar el comportamiento especial de `null` desde dentro del propio lenguaje —como obviar el mensaje `clone`— a la vez que hereda otras operaciones del objeto raíz `Prototype`. Sirve, además, para que sea posible añadir a `null` categorías que puedan ayudar a trazar problemas cuando se envían mensajes a `null`, extensión que se hace también directamente desde XC.

La solución para la implementación de `null` consiste en reservar e inicializar parcialmente de forma estática en alguna parte del *runtime* un objeto cuya referencia será asignada al objeto nulo. Durante la

inicialización del *runtime* se terminará la inicialización completa de `null`, cuando se encuentre su definición completa.

La necesidad de tener una inicialización estática de `null` es causa de una limitación inherente en la implementación del *runtime* de XC. El objeto `null` necesita una declaración explícita porque pueden declararse nuevas categorías que extiendan a `null`. Cada categoría ocupa espacio en memoria, pero el tamaño de `null` se ha de reservar estáticamente, por lo que la implementación ha de elegir un número arbitrario máximo de categorías que se puedan añadir.

8.6 Diseño general del compilador

El prototipo del compilador tiene un diseño tradicional. A partir de un fichero fuente en XC se genera un fichero compilado en código C, que posteriormente se compilará y enlazará con un compilador de C y un enlazador o *linker* estándar. Los subsistemas principales son el analizador léxico, el analizador sintáctico, las acciones semánticas y el generador de código. El prototipo del compilador se ha implementado con el lenguaje C y tiene unas 32.000 líneas de código. Usa el generador de analizadores léxicos *flex* y el generador de analizadores sintácticos LR *bison*, ambos de GNU.

El analizador léxico se encarga de la generación de símbolos terminales o *tokens* para ser consumidos por el analizador sintáctico. El analizador léxico gestiona también la lectura de los módulos importados, no precisando herramientas adicionales para ello. Acciones léxicas se encargan de seleccionar el bloque de código adecuado que ha de suministrarse al analizador sintáctico en cada momento.

El analizador sintáctico implementa la gramática del lenguaje XC. Contiene actualmente alrededor de 400 producciones y se encuentra descrita en el Apéndice A. La estructura general del lenguaje que define la gramática se ha analizado en el capítulo 5. El analizador sintáctico se encarga de disparar las acciones semánticas asociadas a cada producción de la gramática.

Las acciones semánticas organizan las declaraciones y definiciones que van siendo reconocidas por el analizador sintáctico en una tabla de símbolos global, efectuando las comprobaciones adicionales necesarias. Las acciones semánticas implementan las reglas de ámbito de XC vistas en el capítulo 5 y el sistema de tipos visto en el capítulo 7. En puntos determinados del análisis semántico, se disparan las reglas de generación de código.

La generación de código se realiza por módulo. El código generado es C debido a que hoy en día se puede considerar un ensamblador de alto nivel independiente de plataforma. Además de la envidiable característica multiplataforma, tiene la triple ventaja de simplificar el proceso de generación de código, facilitar la depuración del mismo, y servir a la vez como documentación de referencia de la semántica final del código generado.

8.6.1 Análisis léxico e importación de módulos

Para limitar complejidad y ambigüedades de la gramática de XC, el analizador léxico genera *tokens* adicionales para ciertos símbolos especialmente ambiguos, como nombres de función o nombres de localizadores de argumentos en mensajes. También limita ambigüedad y fomenta un estilo consistente de codificación mediante reglas léxicas simples que habitualmente forman parte de las convenciones de codificación: los nombres de módulo, protocolo o prototipo, por ejemplo, deben comenzar con mayúsculas; los nombres de variables, métodos y funciones deben en cambio empezar con minúsculas.

Los módulos son la abstracción principal de organización de programas escritos en XC, donde la implementación de un módulo depende de las declaraciones explícitas de otros módulos. La importación de módulos se implementa directamente dentro del analizador léxico, sin necesidad de

ningún preprocesador. Tiene la ventaja de que la semántica de la importación está definida directamente en el lenguaje. No es posible redefinir símbolos mediante macros que puedan confundir a programas complejos con múltiples importaciones.

El proceso de importación se realiza en el analizador léxico porque es éste quien suministra de forma opaca el flujo de *tokens* al analizador sintáctico. Los ficheros importados corresponden siempre con declaraciones de módulo. Cuando se importa un nuevo módulo, el flujo de *tokens* pasa a ser el del nuevo módulo importado y, cuando éste termina, vuelve al flujo de *tokens* original. Este proceso es recursivo, siendo posible importar múltiples módulos que importen a su vez múltiples módulos, antes de volver al módulo inicial que realizó la primera importación. Para mantener la eficiencia, el analizador léxico asegura que las declaraciones de módulo son leídas una sola vez de fichero. La misma técnica sirve para aceptar bucles de importación sin riesgo: un módulo A que importa un módulo B que a su vez importa el módulo A, sólo lee una vez las declaraciones de A y de B. La Figura 8-8 muestra un esquema del análisis léxico y el proceso de importación de módulos.

Permitir bucles de importación ayuda a la consistencia, puesto que un módulo puede importar cualquier otro módulo sin restricciones adicionales. Además, facilita la construcción de código interdependiente. Código de este tipo es muy usual cuando se accede al *runtime* del lenguaje, o cuando se implementa parte de éste dentro del propio lenguaje.

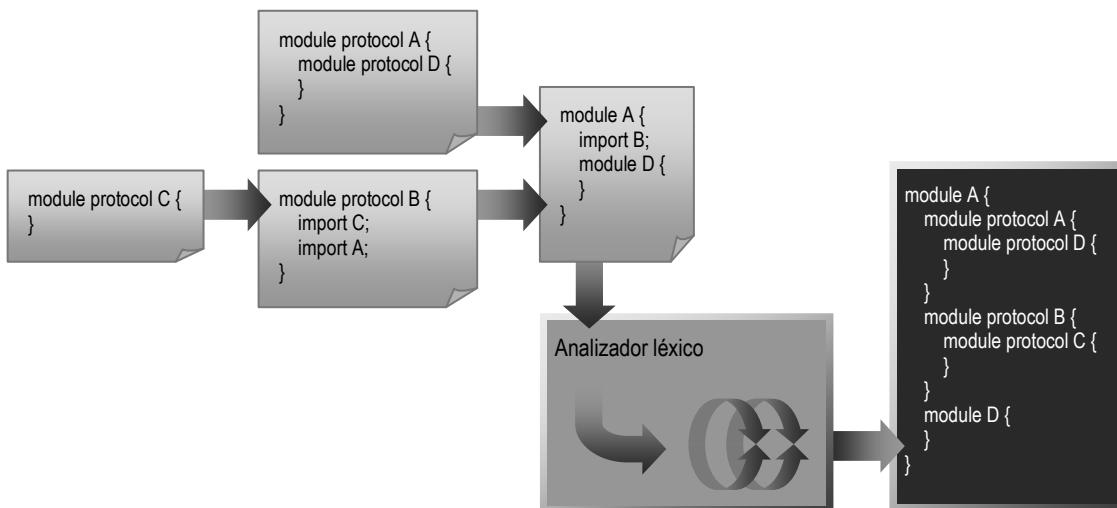


Figura 8-8 Analizador léxico.

Se encarga de leer los ficheros con las declaraciones de módulo y resolver la importación de módulos incluida la lectura de nuevos ficheros internamente, mostrando un único flujo de *tokens* al analizador sintáctico.

Existen también dos modos de importación implícita. El primero es la importación de la declaración de un mismo módulo. En la Figura 8-8 el módulo A importa implícitamente su propio protocolo de módulo. El segundo modo es la importación de módulos externos desde módulos internos. Un módulo interno es un módulo que explícitamente se encuentra dentro de la declaración y definición de otro externo. En la Figura 8-8 el módulo D es un módulo interno al módulo A.

8.6.2 Evaluación sintáctica y semántica de módulos

Como se vio en el apartado 5.5.5.1, la importación de módulos es transitiva. Desde el punto de vista de la compilación, significa también que en un momento dado se pueden estar evaluando distintas declaraciones de módulos a la vez. En la Figura 8-8 el protocolo de módulo A se analiza primero y luego se hace lo mismo con el protocolo de B. Pero antes de que termine su análisis se ha de examinar el protocolo de C. La definición de módulos internos debe evaluar cada módulo interno antes

de terminar de analizar el módulo externo, por lo que también se pueden estar evaluando varias definiciones de módulo a la vez. En la Figura 8-8 durante la evaluación del módulo D todavía se está evaluando el módulo A.

La compilación de cada módulo se realiza por separado. Proporcionar flexibilidad en el sistema de módulos a la hora de escribir el código fuente en XC requiere especial atención por parte del compilador. Se deben mantener contextos de análisis independiente de cada declaración o definición de módulo para (1) indicar correctamente los errores referidos al módulo adecuado y al número de línea correcto, y (2) generar el código de los módulos de definición evaluados. Los contextos de análisis se organizan mediante una pila, generándose un nuevo contexto semántico por cada declaración o definición de código en proceso de evaluación. Se puede generar código de más de un módulo a la vez, si se está evaluando las definiciones de módulos internos. La Figura 8-9 muestra este proceso:

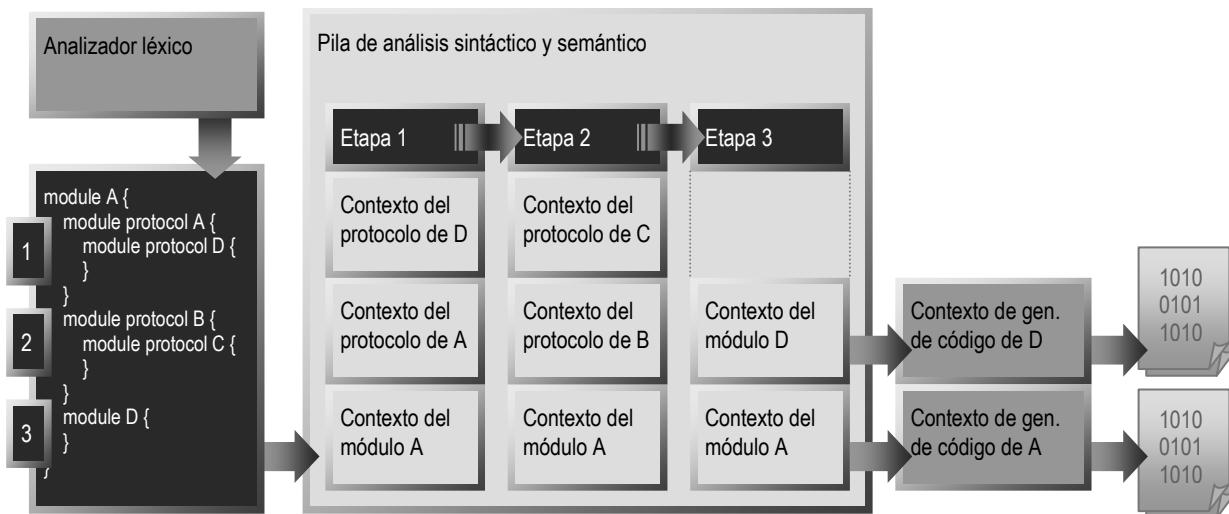


Figura 8-9 Análisis sintáctico, semántico y generación de código.

El analizador sintáctico genera un nuevo contexto de análisis semántico por cada módulo. En la figura se muestran tres momentos de este proceso (señalados con 1, 2 y 3) a medida que se lee el flujo de *tokens* generado por el analizador léxico. Cada contexto de análisis semántico de un módulo de implementación tiene su propio contexto de generación de código.

8.6.3 Generación de código para módulos

La gestión de generación de código se realiza en cada definición de módulo. Debido a que el módulo es una unidad de carga del *runtime*, cada módulo incluye una inicialización interna, llamada durante el registro del módulo dentro del *runtime*. El proceso de registro se estudia con más detalle en el apartado 8.5 *Inicialización*.

Por cada módulo se producen dos ficheros en C: un fichero de cabeceras .h y un fichero de implementación .c. La convención actualmente implementada en el compilador genera un nombre de fichero con el nombre completo de ámbito de un módulo en el directorio local, que no presupone una organización de directorios igual a la de la jerarquía de módulos del programa⁴². La excepción son los módulos internos. El compilador genera un fichero incluyendo el código del módulo externo y todos

⁴² Sería sencillo implementar una opción de compilación que generase una estructura de directorios al estilo de Java si así se desea.

los módulos internos definidos, para enfatizar la propiedad de los módulos internos de no poder ser importados explícitamente, sino sólo a través de un módulo externo⁴³.

El generador de código prepara la información compilada de métodos, estructura de objetos, objetos constantes y soporte de envío de mensaje en tablas. Toda esta información es registrada cuando el módulo se carga en memoria. La organización del código generado de un módulo puede verse en la Figura 8-10.

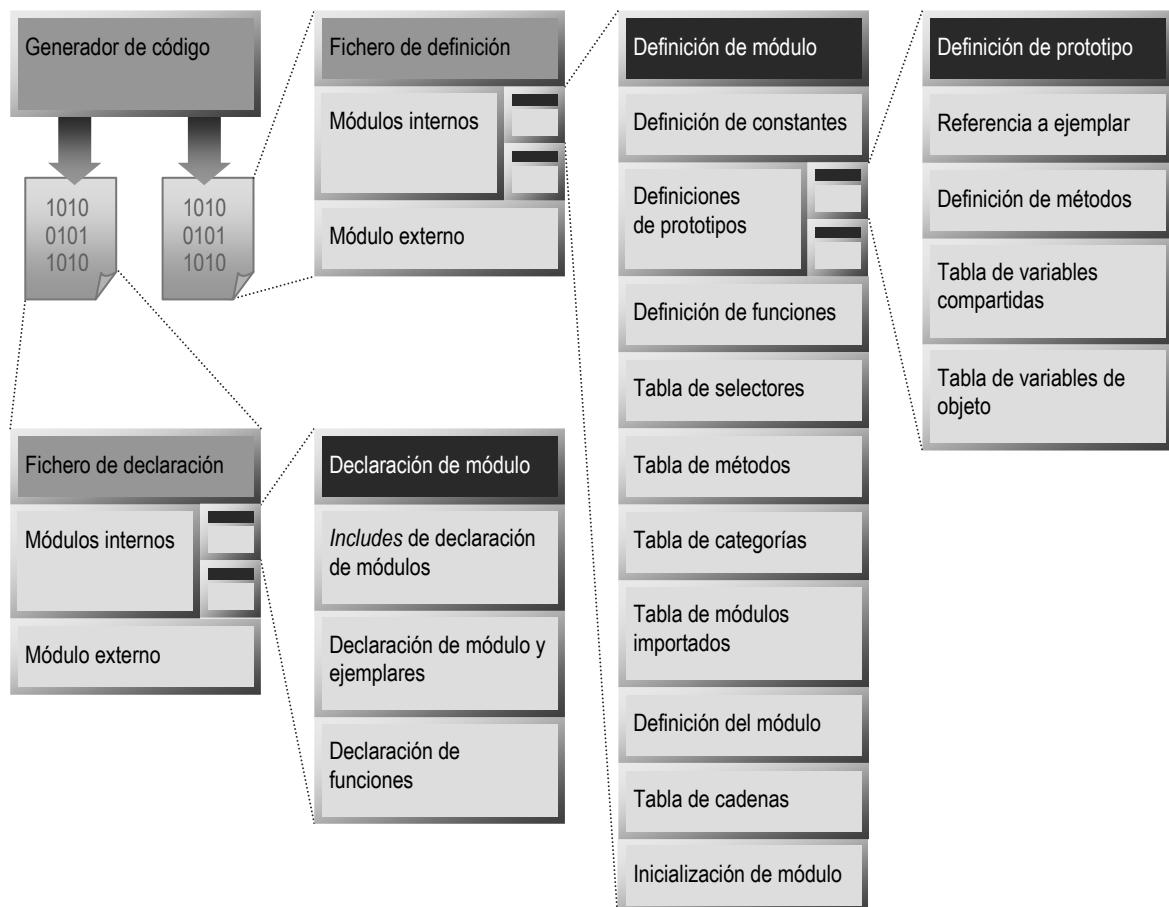


Figura 8-10 Estructura de código generado para cada módulo.

Consta de dos ficheros en lenguaje C: un fichero de declaraciones y un fichero de definiciones. La estructura de un módulo externo y un módulo interno es la misma. En la figura se ha expandido un módulo interno para ver su estructura.

La declaración de módulo publica ejemplares de cada prototipo que, por convención, tienen el mismo nombre que el dado al prototipo en el código fuente. Son inicializados al final de la inicialización del módulo. Conviene destacar también en la declaración de módulo la exportación del objeto que representa precisamente ese módulo dentro del *runtime* (y que sigue el protocolo `XC.Runtime.Module`). Sirve como nexo de unión entre el soporte en tiempo de ejecución y la información generada por el compilador.

Toda la estructura interna de definición de módulos creada por el compilador está orientada a objetos. La información de definición es directamente accesible por el código de reflexión del *runtime*—escrito en XC— y no necesita traductores adicionales entre objetos y las estructuras internas. Todos los

⁴³ Sería también sencillo implementar una opción de compilación que generase ficheros separados para cada módulo interno. El compilador internamente trata la generación de código de cada módulo independientemente. Un módulo interno simplemente comparte los descriptores de ficheros de salida con los de su módulo externo.

protocolos seguidos por los elementos que forman la declaración y la definición de un módulo se acceden desde XC usando el módulo `xc.Runtime`: módulos, módulos importados, categorías, selectores, métodos o argumentos de métodos son algunos de ellos. Un ejemplo servirá para revelar la estrecha relación entre el código generado por el compilador y el *runtime*. Un objeto prototipo para selectores llamado `Selector` se define en la implementación del *runtime* como:

```
final fixed prototype Selector extends: RuntimeObject
{
    property const String messageName;
    property const Unsigned id;
}
```

La tabla de selectores global al módulo generada por el compilador —mostrada en la Figura 8-10— corresponde con la siguiente definición dentro del *runtime*:

```
final fixed prototype SelectorArray = Array <Element = Selector>;
```

La tabla contiene objetos cuyo prototipo es `Selector`. La definición de la tabla de selectores precisa únicamente una línea en el *runtime*, necesaria para realizar correctamente la comprobación de tipos. La implementación se obtiene directamente de `Array`. Puesto que los selectores del *runtime* son de sólo lectura, se exportan únicamente los métodos de acceso pero no los de modificación, implícitos a una declaración realizada con `property`.

```
final protocol Selector extends: RuntimeObject
{
    const String messageName;
    const Unsigned id;
}
```

El nombre del mensaje asociado con el selector es un `String`, que el compilador resuelve durante la compilación a una de las entradas de la tabla de cadenas presente también en el mismo módulo, indicada en la Figura 8-10, y cuya definición de una sola línea es similar a `SelectorArray`. El contenido de la tabla de cadenas son objetos constantes que siguen el protocolo `const string`. El número `Unsigned` asociado al selector es asignado durante la inicialización del módulo.

El selector asociado a un método es directamente accesible desde dentro del código de un método con la palabra clave `selector`, similar sintácticamente a `self`. El selector referenciado es el generado directamente por el compilador, y se encuentra almacenado en la tabla de selectores antes mencionada. Como el nombre del mensaje es también un objeto, es posible obtener una copia de la cadena con el nombre del mensaje asociado al método usando:

```
const String msgName = selector.messageName.clone;
```

Esta sentencia constata la conveniencia de la naturaleza orientada a objetos del código generado por el compilador y del soporte en tiempo de ejecución: simplemente es más consistente, pues mantiene las mismas reglas de manipulación que el resto del lenguaje.

8.7 Optimización

La optimización intenta dar respuesta a la última cualidad importante de un lenguaje de programación: la eficiencia. Existen características de cualquier lenguaje que son contrapuestas, siendo necesario encontrar algún equilibrio intermedio. Por ejemplo, un sistema de tipos homogéneo es más consistente, pero dificulta el manejo eficiente de tipos integrales. Si no se da una solución a este problema, la viabilidad práctica del lenguaje queda en entredicho, puesto que un importante número de cálculos se realizan con esos objetos básicos. Lo mismo se puede decir de otros objetos sencillos sobre los que no se espera que se realicen modificaciones no planificadas: Es simplemente un gasto innecesario de recursos codificar estos objetos como entes más generales —y por tanto menos eficientes— si no se esperan modificaciones, o si se trata de objetos privados. Si XC pretende ser

aplicable a la programación de sistemas, debe aceptar cierta relajación controlada de las reglas del sistema de tipos para permitir optimizaciones. Esa flexibilidad se debe introducir con sumo cuidado para no perjudicar otras abstracciones del lenguaje.

8.7.1 Modificadores de optimización de objetos

Los prefijos modificadores de objetos son usados como indicaciones que limitan la generalidad de los objetos y ayudan al compilador a generar código más eficiente. Cada objeto tiene varios grados de libertad. Cuantos más tenga, más general es y menos posibilidades de optimización existen. Una introducción a los modificadores de los objetos prototipo se ve en el apartado 5.4 *Protocolos, prototipos y sus variantes*.

XC admite que una declaración de prototipo sea más restrictiva en el protocolo público que en el protocolo protegido o la implementación, para limitar el acceso de clientes externos a ciertos prototipos. Por ejemplo, en el código siguiente el prototipo `windowRect` se declara `final confined` públicamente, que impide a módulos que no sean submódulos de `MyModule` heredar de `windowRect` o añadirle categorías. El protocolo de módulo protegido opcionalmente puede redefinir `windowRect` para que sus submódulos sólo puedan añadir categorías.

```
public module protocol MyModule
{
    // ...
    final confined prototype windowRect;
}

protected module protocol MyModule
{
    // ...
    redefine final prototype windowRect extends: Object;
}
```

La posibilidad de restringir los modificadores de declaraciones de prototipos en un protocolo de módulo afecta a la capacidad de optimización. Cuando el compilador compila otros módulos, no puede suponer que un modificador `final` en una declaración de prototipo ayude a realizar una optimización, porque el modificador puede haberse añadido como una restricción de acceso. Para indicar al compilador que una declaración de prototipo se refiere a información de implementación, se usa el modificador `implementation`. En este caso, el compilador sabe que realmente el modificador `final` se aplica a la implementación del prototipo. Los modificadores que acompañan a `implementation` deben coincidir con aquellos que aparecen en la implementación real.

```
protected module protocol MyModule
{
    // ...
    redefine implementation
    final prototype windowRect extends: Object;
}

module MyModule
{
    // ...
    final prototype windowRect extends: Object
        { // ... }
```

Por defecto, el tamaño de un prototipo no es conocido por el compilador, debido a que un objeto es extensible con categorías que incluyan variables de instancia extra. El modificador `fixed` restringe el tamaño máximo de los objetos a aquél definido por el prototipo que incluya ese modificador. Si un objeto prototipo tiene un tamaño fijo, entonces todos sus posibles descendientes también tienen un tamaño fijo. Esta restricción es útil para los tipos de datos básicos predefinidos que se ven en el apartado 8.7.2. También es útil, por ejemplo, para asegurar que existe un espacio de almacenamiento

fijo para ellos en memoria. Invariante que puede ser usado por un recolector de basura, o una política particular de gestión de memoria para optimizar la petición y liberación de memoria de ese tipo de objetos. El modificador `fixed` es transmitido durante la herencia, manteniendo invariante el tamaño de los objetos derivados. Esta regla facilita que las mismas optimizaciones realizadas con un prototipo sean válidas con sus descendientes. Si no fuese así, el compilador tendría pocas posibilidades de optimización, porque la herencia seguiría abierta. Esto es debido a que las operaciones efectuadas con el prototipo `fixed` continuarían siendo potencialmente polimórficas y, por tanto, podrían llegar a manejar objetos descendientes de tamaño distinto. En tal situación, el compilador debería generar código válido para todos los objetos del subárbol de herencia, perdiéndose el efecto del modificador.

Los modificadores `fixed` y `baselevel` son sensibles al árbol de herencia, siendo esa la razón de fondo para que su comportamiento sea heredado por prototipos descendientes. Por ejemplo, supongamos un prototipo `A` que tiene el modificador `fixed`, y que el prototipo `A` hereda del prototipo `Object` que no tiene ningún modificador. Entonces, es posible que prototipo `Object` sea extendido usando una categoría que añada una variable de instancia `y`, y, por tanto, que el prototipo `A` aumente de tamaño. El modificador `fixed` asegura que los prototipos descendientes de `A` mantienen el invariante de igual tamaño que `A`, pero ese tamaño podría variar si se tiene acceso a un prototipo antecesor. Para evitar ese extremo, si se quiere fijar a un tamaño dado el prototipo `A`, éste debe heredar de la raíz de herencia del lenguaje `Prototype`, que se publica con el modificador `confined` y no puede ser extendido con categorías. El ejemplo siguiente muestra esta última idea. El protocolo público del módulo `MyMathModule` declara el prototipo `Complex` como `fixed`, heredando directamente de `Prototype`.

```
public module protocol MyMathModule
{
    // ...
    fixed prototype Complex extends: Prototype;
}

module MyMathModule
{
    // ...
    fixed prototype Complex extends: Prototype
    {
        property Double real;
        property Double imag;
        // ...
    }
}
```

El mismo razonamiento se aplica a `baselevel`. Si un prototipo `A` deriva de `Object`, entonces es posible que se puedan añadir extensiones a métodos anteriores y posteriores usando una categoría de `Object`. Si un prototipo quiere prohibir todos los metamétodos entonces debe evitar que se creen metamétodos en sus ascendientes, ya que los metamétodos son acumulativos. Para ello ha de heredar de `Prototype` e incluir el modificador `baselevel`. La implementación del lenguaje garantiza que `Prototype` no ejecuta nunca metamétodos. Al igual que con el modificador `fixed`, es posible que por razones de implementación el número de ascendientes sea más amplio, y que éstos no incluyan el modificador `baselevel`. En ese caso es responsabilidad del desarrollador que los ascendientes sean privados o públicos con el modificador `confined` y no incluyan metamétodos, para preservar el invariante.

Otra fuente de optimización se produce cuando se garantiza que la relación entre un mensaje y un método es 1:1, es decir, que el envío de un mensaje es *monomórfico*. En ese caso, el procedimiento de resolución del método asociado a un mensaje siempre da el mismo valor y, si el valor es conocido en tiempo de compilación, es posible obviar todo el procedimiento de resolución y realizar directamente la llamada al método usando una llamada a subrutina tradicional. Si una llamada es potencialmente polimórfica, entonces el compilador no es capaz de realizar la optimización. Obtener llamadas monomórficas en XC es más complejo que con un lenguaje orientado a objetos tradicional, porque el

modelo de objetos es más flexible. Hacen falta las siguientes condiciones para asegurar que un prototipo envía mensajes monomórficos:

- *El prototipo es una hoja en el árbol de herencia.* Se identifica mediante `final` —o `final implementation` en una declaración de prototipo—. Esta es usualmente la única condición para obtener llamadas monomórficas en otros modelos de objetos. Todos los métodos asociados a los mensajes de un objeto `final` son encontrados por el compilador, puesto que o pertenecen al objeto prototipo o a alguno de sus ascendientes.
- *El prototipo ha prohibido la ejecución de métodos anteriores y posteriores.* La ejecución de metamétodos invalida la relación monomórfica entre mensajes y métodos, porque cuando existen metamétodos es necesario ejecutar una cadena de métodos, no un método únicamente. Debe usarse el modificador `baselevel`.
- *Ningún prototipo antecesor admite la ejecución de metamétodos.* Los metamétodos no se sobrescriben, sino que se acumulan. Si un antecesor puede potencialmente ejecutar un metamétodo, entonces no se garantiza la relación monomórfica. La excepción es el prototipo raíz `Prototype`, que la implementación del lenguaje garantiza que no ejecuta nunca metamétodos.
- *El protocolo asociado al prototipo es también final.* Si el protocolo asociado a un prototipo no es `final`, entonces es posible que otros prototipos implementen el mismo protocolo, y no habría información suficiente para decidir a qué método corresponde cada mensaje. Si el protocolo es `final`, el único prototipo que puede adoptar el protocolo `final` es aquél que tenga el mismo nombre que el protocolo, desapareciendo la ambigüedad.

Un ejemplo práctico se verá en el apartado 8.7.3 *Optimización del envío de mensajes*. El prototipo del compilador de XC genera código C, que es compilado a código objeto nativo usando las reglas de optimización de un compilador de C. El compilador de XC actual no hace ningún intento por optimizar más allá de las indicaciones dadas con los modificadores. No existe tampoco ningún impedimento en el diseño del lenguaje para generar código que sea interpretable por una máquina virtual. La cuestión de cómo realizar optimizaciones más agresivas es dejada para futuros trabajos de investigación.

8.7.2 Optimización de tipos básicos

Los tipos básicos son aquellos cuyos datos caben en una palabra del procesador, e incluyen a todos los tipos integrales como `Unsigned`, `Integer`, `Byte` o `Short`, al tipo `Boolean` y ciertos tipos predefinidos que son punteros como `Pointer` o `MethodImp`. Dependiendo de la arquitectura de la plataforma de destino, el número de tipos básicos optimizables puede cambiar. En general, cada objeto incluye además de los datos una referencia al mapa del objeto, que contiene información compartida a los objetos de una misma familia, incluyendo su tabla de métodos. Si el lenguaje posee tipos estrictos, es posible identificar las expresiones que involucren a objetos con tipos básicos. La optimización de tipos básicos consiste en eliminar la referencia al mapa dentro del contexto de una expresión con tipos estrictos conocidos. Una optimización similar es realizada por el lenguaje C#. La virtud de esta optimización es que acaba con la necesidad de los tipos primitivos de Java o C++, consiguiendo un sistema de tipos homogéneo. Por ejemplo, las dos expresiones siguientes son optimizables porque el tipo involucrado `unsigned` es un tipo básico. El compilador conoce en todo momento cuál es el tipo de los argumentos y el tipo asociado al resultado.

```
unsigned i = 3, j = 3 + i;
```

El compilador anota el mapa del prototipo `unsigned` aparte, lejos del almacenamiento real del objeto, disminuyendo el tamaño del objeto a una palabra de procesador, que puede entonces evaluarse a la misma velocidad que en C. Mientras el compilador sea capaz de mantener el conocimiento del tipo exacto del objeto en una expresión, es posible obviar la inclusión del mapa del objeto. En caso del envío de mensajes como:

```
i.print;
3.print;
```

el sistema de tipos del compilador puede deducir que `i` es una variable de tipo `unsigned` y generar código que busque el método asociado al mensaje `print` en el mapa de la familia `unsigned`. Por la misma razón es capaz de resolver el mensaje enviado al objeto `3`, mostrando un sistema de tipos completamente unificado. Ambas sentencias imprimen el mismo valor. Una expresión que no mantenga el conocimiento exacto de los tipos de datos obliga al compilador a crear un objeto completo. Por ejemplo:

```
Object obj = i;
```

La variable `i` es assignable a la variable `obj` porque `unsigned` adopta también el protocolo `Object`. El resultado de la expresión es de tipo `Object`. El compilador pierde traza del tipo `unsigned`, y por eso debe crear un objeto `i` no optimizado que incluya su mapa antes de asignarlo a `obj`. Lo mismo ocurre en el caso de un *array* de `Object`.

```
Object array[2] = [ 1, 2 ];
```

La operación inversa también es posible, usando el operador '`?=`' de tentativa de asignación. El compilador genera código que comprueba en tiempo de ejecución si el tipo es integral y en ese caso extrae el tipo optimizado.

```
unsigned u;
if (u ?= obj)
    // obj es un tipo unsigned y u es correctamente asignado
else
    // obj no es un tipo unsigned y u no es asignado
```

En el ejemplo siguiente, en cambio, el compilador no pierde traza del tipo optimizable, aunque el resultado sea un *array*. Eso es posible porque un *array* incluye información del tipo de los objetos que contiene.

```
unsigned array[2] = [ 1, 2 ];
Unsigned x = array[0];
Object y = array[1];
```

El compilador puede almacenar dentro del *array* los valores optimizados para `1` y `2` ahorrándose la creación de los objetos completos. La asignación de la variable `x` únicamente manipula objetos optimizados. La asignación de la variable `y` ha de construir un objeto completo a partir del objeto optimizado `2`. La manipulación de *arrays* optimizados es importante, porque facilita la programación con estructuras de bajo nivel como *buffers* o tablas desde dentro del lenguaje, sin requerir operaciones nativas.

8.7.3 Optimización del envío de mensajes

El apartado 8.4 *Implementación del envío de mensajes* ha analizado la ejecución eficiente del mecanismo general de envío de mensajes. En este apartado se estudian otras optimizaciones posibles. El envío de mensajes usuales a los tipos básicos puede mejorarse aún más, siempre que se garantice el tipo resultante de las expresiones. Si no existe optimización, las expresiones que involucran operadores son traducidas a su mensaje equivalente (véase la Tabla 5-4 en el apartado 5.6.1 *Expresiones y protocolos de operador*). La expresión:

```
unsigned u = 3 * 4;
```

es interpretada por el compilador como:

```
unsigned u = 3.mul: 4;
```

ya que `Unsigned` sigue el protocolo de operador `Arithmetic` y el mensaje `mul`: asociado al operador '*' se encuentra en ese protocolo. En la práctica, los mensajes de operador enviados a objetos cuyo tipo sea básico, como los tipos integrales o booleanos, son reconocidos por el compilador y traducidos a su equivalente en C, evitando un envío de mensaje. Esto es posible porque los tipos integrales y el tipo booleano cumplen todos los requisitos vistos en el apartado 8.7.1 *Modificadores de optimización de objetos* para realizar más eficientemente las llamadas a los métodos. El tipo `Unsigned` deriva de `Integral` que a su vez deriva de `Number`. Se declara como sigue:

```
protocol Number extends: Object, Relational, Arithmetic
{ // ... }

protocol Integral extends: Number, Binary
{ // ... }

final protocol Unsigned extends: Integral
{ // ... }

fixed baselevel prototype Integral extends: Prototype
{ // ... }

final prototype Unsigned extends: Integral
{ // ... }
```

El prototipo `Integral` tiene el tamaño fijo de un registro de máquina y no acepta metamétodos, pero no es optimizable porque no es un prototipo `final`. El tipo `Unsigned` sí es optimizable porque restringe metamétodos correctamente, su protocolo es `final` y el prototipo también es `final`. Los protocolos y prototipos no son `confined` para facilitar el añadido de mensajes y métodos adicionales de utilidad. Al ser sus tamaños fijos no pueden crecer, manteniendo el invariante del tamaño, que a su vez permite optimizar la gestión de la referencia al mapa de memoria vista en el apartado 8.7.2 *Optimización de tipos básicos*. Las técnicas anteriores hacen posible que, para el código orientado a objetos siguiente en XC (completamente arbitrario salvo por sus propósitos ilustrativos):

```
Integer value = 0;
for (Unsigned i = 0, j = 0; i < 10; i++, j++)
{
    value = (i*10 + (3 & i)) >> 1;
    while (j < 5)
        value++;
}
```

el compilador genere el siguiente código en C:

```
xc_raw_integer value = 0;
xc_raw_unsigned i;

for (i = 0, j = 0; i < 10; i++, j++)
{
    value = ((i*10) + (3 & i)) >> 1;
    while (j < 5)
        value++;
}
```

8.7.3.1 Funciones en línea

Es posible usar funciones y métodos en línea para optimizar ciertas operaciones comunes muy sencillas.

- ❖ **Def. 8-8.** Una *función en línea* es una forma optimizada de función, que elimina el coste de llamada a la función expandiendo su código de definición en cada cliente.

Las funciones en línea sustituyen a las macros de C, pero debe tenerse siempre en cuenta que las operaciones en línea *son inherentemente no modulares*. Los siguientes extractos de guías de desarrollo de Taligent para C++ [Taligent; 1994; pp. 62-64] son muy ilustrativos:

"Evite las definiciones en línea porque son compiladas en el código que realiza la llamada, haciéndolas difíciles de revisar. Para que los clientes puedan compilar una clase que contiene una definición en línea,

deben tener el código fuente para esa función. Una vez el código fuente está en circulación, ya no puede ser cambiado sin romper la compatibilidad binaria. Es más, si una definición en línea se refiere a los detalles internos de una clase, esos detalles nunca podrán cambiar.” [Taligent; 1994; p. 62]

“Algunas veces es aceptable usar definiciones en línea si la eficiencia es extremadamente importante. No obstante, si hace esto, *nunca podrá cambiar la definición una vez el código se entregue*, y frecuentemente no se ahorra tanto.” [Taligent; 1994; p. 64]

Se usa el modificador `inline` para identificar las rutinas en línea. Un ejemplo de definiciones en línea aceptable son algunas funciones matemáticas simples, que involucran muy pocos cálculos, como la función que calcula el máximo entre dos números o una función de redondeo. Un ejemplo de esta última es:

```
inline Unsigned round: Unsigned value toupperBound: Unsigned bound
{
    return (value / bound) + (1 - value % bound);
}
```

En particular, y para el ejemplo anterior, el código generado por el compilador es igual de eficiente que usando directamente C.

8.7.4 Funciones y objetos nativos

Los objetos y funciones nativas son el nexo de XC con funcionalidad escrita en otros lenguajes, acceso al sistema operativo u otras bibliotecas de código externas. El lenguaje intermedio de comunicación es C. Si una biblioteca externa es accesible desde C, entonces es posible construir enlaces nativos con XC. La funcionalidad nativa se identifica con la palabra clave `native`. Es una forma especial de código no seguro, que se revisa con mayor detenimiento en el apartado 8.7.5. La palabra clave `native` es un modificador estrictamente de implementación.

8.7.4.1 Funciones nativas

La manera más sencilla de hacer interfaz con código externo es usando una función nativa. La implementación de una función nativa no tiene cuerpo, porque éste se especifica en algún otro lugar externo, fuera del lenguaje. Por ejemplo el módulo `xc.System.Console` define funciones de acceso a las bibliotecas básicas que acceden a la consola. La siguiente función imprime un texto:

```
native void print: String text;
```

La implementación nativa de esa función es la siguiente:

```
void xc_System_Console_print_ (struct xc_object *text)
{
    xc_printf (XC_STRING("%s"), ((struct xc_string *)text)->data);
```

Las estructuras `xc_object` y `xc_string` corresponden con la estructura básica en C de un objeto y una cadena de texto en el *runtime*. El nombre de la función indica el módulo donde se encuentra.

8.7.4.2 Objetos nativos

No todos los objetos en XC tienen que ser objetos del lenguaje. En general, las bibliotecas proporcionarán sus propios objetos o estructuras. Los objetos nativos ayudan a definir una correspondencia entre ellos y XC.

- ❖ **Def. 8-9.** Un objeto es *completamente nativo* si la implementación está escrita exclusivamente de forma nativa.
- ❖ **Def. 8-10.** Un objeto es *parcialmente nativo* si la implementación es híbrida, parte escrita en XC y parte escrita nativamente.

Para construir objetos nativos o parcialmente nativos es posible que sea necesario identificar espacio para datos.

- ❖ **Def. 8-11.** Una *propiedad nativa* representa un dato cuyo acceso está vetado al lenguaje, y sólo puede ser accesible a través de métodos nativos.

Las propiedades nativas no son accesibles directamente desde el lenguaje. Por esa razón nunca devuelven un tipo de datos y el tipo del valor de retorno es `void`.

```
native property void myProperty;
```

Se pueden mezclar propiedades nativas y no nativas en un objeto. La declaración y nombrado de la propiedad nativa en el lenguaje ayuda a identificar en qué posición se encuentra una propiedad nativa en un objeto y qué tamaño tiene. El espacio de almacenamiento de una propiedad nativa puede ser variable. El tamaño por defecto de una propiedad nativa la obtiene el compilador con la expresión C:

```
sizeof(struct xc_object *).
```

Es posible cambiar el tamaño por defecto de una propiedad nativa usando la palabra clave `storage`. Los objetos nativos se utilizan para exponer estructuras en C al lenguaje. Los campos de una estructura en C han de representarse y accederse con propiedades nativas. Por ejemplo, un número complejo de una biblioteca C, definido como:

```
struct Complex
{
    double real;
    double imag;
};
```

puede representarse en XC usando un módulo que esconda la naturaleza nativa del objeto:

```
public module protocol NativeComplexModule
{
    final confined protocol NativeComplex
    {
        property Double real;
        property Double imag;
        Self alloc;
        void dealloc;
        Self init;
        Self initReal: Double real imag: Double imag;
        Self clone;
    }

    implementation
    final confined baselevel prototype NativeComplex;
}

module NativeComplexModule
{
    const Unsigned NATIVE_DOUBLE_SIZEOF = 8;

    final confined baselevel prototype NativeComplex
    {
        native property storage: NATIVE_DOUBLE_SIZEOF void real;
        native property storage: NATIVE_DOUBLE_SIZEOF void imag;
        native Double real;
        native Double imag;
        native void setReal: Double real;
        native void setImag: Double imag;
        native Self alloc;
        native void dealloc;

        Self initReal: Double real imag: Double imag
        { self.real = real; self.imag = imag; return self; }

        Self clone
        { return self.alloc.init; }

        Self init
        { return self.initReal: 0.0 imag: 0.0; }
    }
}
```

Deben observarse algunos aspectos importantes en la implementación. La declaración del módulo publica un protocolo `final confined`, puesto que una estructura C no es extensible de ninguna forma. El protocolo no adopta el protocolo de `Object`, tan solo declara los mensajes de creación necesarios. El resto del protocolo es normal. La implementación usa un objeto parcialmente nativo. Como la estructura interna del objeto nada tiene que ver con un objeto de XC, el prototipo `NativeComplex` no hereda de `Object` o `Prototype`, usando otra vez los modificadores `final` y `confined`. El modificador `baselevel` indica al compilador que prohíba definir métodos anteriores o posteriores en el prototipo.

Al tener el protocolo y el prototipo los modificadores `final` y `confined`, —que implica también tamaño fijo— y el modificador `baselevel` aplicado a toda su jerarquía —que es únicamente el prototipo `NativeComplex`— el compilador puede optimizar todas las llamadas de métodos a funciones. Por tanto, el compilador no necesita usar el puntero al mapa de objetos definido en `Prototype`, característico de todos los objetos de XC, y necesario para adoptar el protocolo `Object`. Los métodos `alloc` y `dealloc` son necesarios para crear objetos nativos de forma independiente al lenguaje. El método `alloc` debe devolver siempre una referencia. Los métodos `clone`, `init` e `initReal:imag:` implementan las convenciones usuales de creación de objetos en XC por comodidad, y no necesitan ser definidos nativos.

Los métodos de acceso nativos deberán convertir la estructura C `xc_double` asociada al tipo predefinido `double` en XC, con el tipo `double` de la biblioteca nativa. En el caso de los tipos predefinidos de C la operación es trivial. Al definirse los métodos de acceso, es posible usar el operador de asignación con el objeto nativo y operar con comodidad hasta cierto punto.

```
NativeComplex nc = NativeComplex.clone;
nc.real = 0.3;
nc.imag = nc.real;
```

Si `NativeComplex` adopta e implementa el protocolo `Object`, entonces se convertirá en un objeto de XC indistinguible y manipulable como cualquier otro objeto. Por ejemplo, podrá asignarse a variables de tipo `Object` y a `arrays`, cuyo contenido es una variable de tipo restringida al menos al tipo `Object`. Si además implementa alguno de los protocolos de operador, entonces también podrá usarse en expresiones con operadores.

8.7.4.3 Geometría de objetos

El tamaño final de un objeto de XC no es conocido por el compilador en general porque depende del número de categorías que tenga un objeto, aspectos vistos en el apartado 8.3 *La estructura de los objetos*. En ciertos casos particulares, como los objetos con el modificador `fixed`, es posible llegar a conocer el tamaño total de un objeto. Sin embargo, no es conveniente depender de esa información, porque genera dependencias implícitas. Lo que el compilador sí conoce es la geometría de cada categoría principal o prototipo y cada categoría de extensión por separado. Para ayudar a conocer la geometría de un objeto XC incluye los operadores `sizeof:` y `offset:`. La utilidad de los operadores aumenta al acceder a funcionalidad nativa.

Si el operador `sizeof:` se aplica a un nombre de prototipo o categoría, devuelve el tamaño en *bytes* de la categoría asociada. Si se aplica a una propiedad, da el tamaño en *bytes* de la propiedad.

```
Unsigned categorySize = sizeof: MyPrototype;
Unsigned propertySize = sizeof: MyPrototype.myProperty;
```

Únicamente en el caso particular de un prototipo `final confined` que no herede de ningún otro objeto o que herede de `Prototype`, se puede asegurar que el operador `sizeof:` da el tamaño completo del objeto prototipo. El operador `offset:` da el desplazamiento en *bytes* de una propiedad dentro de su categoría.

```
Unsigned propertySize = offset: MyPrototype.myProperty;
```

8.7.5 Código no seguro

En todo lenguaje de programación que intente cubrir la programación de sistemas existen expresiones de bajo nivel que no pueden evaluarse correctamente con el sistema de tipos. Por ejemplo, la petición de memoria para objetos devuelve un trozo de memoria que no tiene tipo. De algún modo hay que asignar tipo a ese trozo de memoria. Para facilitar la construcción de módulos de bajo nivel en XC y depender lo menos posible de funcionalidad nativa escrita directamente en C o C++, el lenguaje identifica zonas de código no seguro con la palabra clave `unsafe`. La palabra clave `native` implica *siempre* la palabra clave `unsafe`. Permitir código no seguro puede dar la impresión de abrir una “caja de Pandora” de problemas potenciales de todo tipo al lenguaje. El autor cree firmemente que, sin facilidades controladas de esta índole, simplemente la programación de sistemas no es viable.

Identificar el código no seguro es importante porque obliga al programador a asegurar explícitamente que sabe lo que está haciendo, y que está violando el sistema de tipos. En zonas de código de esta naturaleza tan especial, es responsabilidad del desarrollador asegurar que la semántica de tipos es la apropiada, la misma que a menudo tendría que preservar en C o C++. Sin una vía de escape de esta índole, la elección de XC como lenguaje para muchos desarrollos quizás estuviese seriamente comprometida. Al estar el código no seguro delimitado por una palabra clave, un entorno de desarrollo podría llegar a ofrecer ayudas al análisis de qué partes del código son más sensibles durante cambios de diseño o mantenimiento.

Por suerte, la cantidad de código no seguro necesaria en una aplicación es bastante limitada. XC reduce aún más la necesidad de código no seguro al ofrecer un sistema de tipos rico y expresivo con tipos covariantes, tipos exactos y tipos genéricos, que asignan correctamente tipos estáticos a una amplia variedad de problemas comunes, revisados en el capítulo 7. La identificación de código no seguro como una funcionalidad de un lenguaje de programación es originaria de Cedar. Posteriormente fue usada por Modula-3 y recientemente por C#. El enfoque seguido en XC se asienta en los avances efectuados por esos lenguajes.

El modificador `unsafe` se aplica únicamente a la implementación. Un protocolo o una declaración de función nunca incluye ese modificador en su firma. El modificador `unsafe` ha de añadirse a cualquier método o implementación de función que use el tipo predefinido `Pointer` o el operador de conversión de tipo o `typecast`. El operador de conversión de tipo o `typecast` es equivalente al que existe en C. El compilador no inserta ninguna comprobación en tiempo de ejecución para validar la operación de cambio de tipo —al contrario de lo que ocurre con el operador ‘`?=`’ de tentativa de asignación, que sí es un operador seguro—. La aplicación correcta del operador de conversión es responsabilidad del desarrollador.

8.7.5.1 Memoria de bajo nivel y memoria de objetos

Habilitar programación de bajo nivel en un lenguaje donde todos los tipos son objetos introduce algunas construcciones y ejemplos interesantes del lenguaje. El módulo `xc.system` da acceso a dos de las abstracciones de bajo nivel más importantes. La mayoría de las operaciones que definen son no seguras, y tienen su equivalente en las rutinas de gestión de memoria de C.

- ❖ **Def. 8-12.** La *memoria de bajo nivel* es una abstracción de un *buffer* de memoria cuyo direccionamiento tiene resolución de *bytes* y manipula el tipo `Pointer`.

El módulo `xc.system.Memory` provee el protocolo nativo de más bajo nivel para la memoria. Permite pedir y liberar memoria explícitamente, gestión de *buffers*, o lectura y escritura de tipos integrales básicos. Por ejemplo, las siguientes declaraciones de funciones nativas dan acceso a las funciones de C `malloc()` y `memmove()`, cuya lectura y propósito es más evidente usando la sintaxis de envío de mensajes de XC.

```
native Pointer alloc: Size inBytes fillWith: Byte value;
native Pointer moveTo: Pointer dest from: Pointer source length: Size inBytes;
```

Obviamente, un mal uso de estas funciones de bajo nivel no seguras puede dar lugar a una excepción del sistema operativo que acabe con el proceso en ejecución. Para reducir la necesidad de usar el módulo de más bajo nivel, existe otra abstracción de memoria más adaptada a las capacidades del lenguaje.

- ❖ **Def. 8-13.** La *memoria de objetos* es una abstracción de un *buffer* de memoria que usa un protocolo más seguro, en el que siempre se manipulan objetos y nunca el tipo `Pointer`.

La memoria de objetos crea y clona objetos simples y objetos indexados, usados estos últimos para implementar eficientemente toda clase de *arrays*. Un objeto indexado siempre contiene otros objetos. La memoria de objetos provee funciones para acceder o modificar los objetos en los índices de un objeto indexado. Como los *arrays* pueden tener un tamaño significativo, no es eficiente clonar el *array* para luego cambiar sus valores. Los *arrays* se inicializan con la siguiente función nativa:

```
native Object allocFrom: Object aPrototype indexedSize: unsigned size;
```

Por supuesto, enviando el mensaje `clone` a cualquier *array* se obtiene el efecto deseado de copia. El protocolo `IndexedObject` contiene el mensaje `allocIndexedSize:`, que permite construir objetos indexados en demanda y es un ejemplo de implementación de un método no seguro y operador de conversión de tipos, similar a C. La palabra clave `unsafe` indica al compilador que se pueden relajar las reglas de comprobación de tipos, y dejar aplicar el operador de conversión de tipo bajo la supervisión del desarrollador. El empleo del tipo covariante `self` en el operador de conversión da automáticamente —y sin necesidad de redefinición posterior— el tipo correcto a la memoria pedida por cualquier prototipo actual o futuro que herede de `IndexedObject`.

```
unsafe Self allocIndexedSize: unsigned size
{
    return (Self) ObjectMemory.allocFrom: self indexedSize: size;
```

Los objetos indexados son interesantes porque son ejemplo de una construcción especial pensada para incrementar la eficiencia del código. El código que accede a un elemento de un *array* dentro de la clase `Array` del módulo `XC.Collections.Arrays` es el siguiente⁴⁴:

```
Object at: unsigned index
{
    return (index < self.length) ? ObjectMemory.objectFrom: self at: index : null;
```

La resolución del acceso al *array* se realiza con la función `objectFrom:at:`, que es expandida en línea por el compilador para optimizar el acceso al *array*. El acceso a la propiedad `length` del *array* en el método `at:`, usado para controlar sus límites, precisa un envío de mensaje. Ese mensaje podría ser evaluado en línea para ciertos tipos particulares de *array*, como los *arrays* de tipos integrales, que son optimizables⁴⁵.

Consideremos un último ejemplo de código no seguro. El tipo `Integral` es el padre de todos los tipos integrales que caben en una palabra de máquina como `Integer`, `Unsigned` o `Short`. Si se quiere proporcionar un nuevo descendiente `Byte` al tipo `Integral`, sería interesante que existiese un método de conversión entre el tipo `Integral` y `Byte`, porque de esa manera se podrían manejar *bytes* como objetos `Integral` y pedir la conversión —con o sin perdida de precisión— cuando fuese necesario. El código siguiente realiza esa labor sin modificar el código existente del tipo `Integral`:

```
const Unsigned BYTE_BITS_LENGTH = 8;
const Unsigned BYTE_BITS_SIZE = (1 << BYTE_BITS_LENGTH);
```

⁴⁴ XC no soporta actualmente excepciones, por lo que se usa la convención de devolver el valor de retorno `null` en caso de acceder fuera del *array*.

⁴⁵ XC garantiza que los tipos integrales son optimizables. Con esa información es posible expandir en línea el método en cuestión si se compilan dos métodos distintos: uno expandido para el código en línea, y otro normal para llamadas polimórficas realizadas por antecesores, que se registra de la forma usual en la tabla de métodos.

```

const Unsigned BYTE_MASK      = (BYTE_BITS_SIZE - 1);

final prototype Byte extends: Integral
{ // ... }

prototype category ByteConversions extends: Integral
{
    unsafe Byte asByte
    {
        return ((Byte)self) & BYTE_MASK;
    }

    unsafe Byte asCheckedByte
    {
        if ((self & ~BYTE_MASK) == 0)
            return ((Byte)self) & BYTE_MASK;
        else
            // error
    }
}

```

8.8 Recapitulación

Con este capítulo se completa la descripción general del estado actual del lenguaje XC. En primer lugar se explica la estructura de objetos del lenguaje, que le confiere importantes características de modularidad y flexibilidad. Por ser una de sus aportaciones más novedosas al diseño de lenguajes, se describe funcionalmente su implementación con detalle: el algoritmo de creación de objetos, el algoritmo de linearización de categorías, la estructura de la tabla de métodos y el algoritmo de construcción de la tabla de métodos. La implementación del envío de mensajes modular también se estudia con detalle, incluyendo una versión preliminar optimizada del registro de activación de envío de mensajes y la función mensajera. La aproximación seguida por XC para su implementación en lenguajes de programación de sistemas es también novel. Para finalizar con el soporte en tiempo de ejecución se dan detalles de su proceso de inicialización.

El prototipo del compilador de XC tiene una organización tradicional, separada en análisis léxico, evaluación sintáctica y semántica y generación de código C. Se repasan brevemente sus elementos más relevantes, para dar una visión de alto nivel de cómo se compila en la práctica el lenguaje.

Por último, el capítulo se centra en las características de optimización incorporadas al lenguaje, indicando cómo se implementan. Las optimizaciones estudiadas son muy importantes para hacer factible la programación de sistemas. Entre ellas, el esquema de modificadores de objetos y las condiciones de optimización de los objetos de XC son una aportación novedosa. Otros aspectos relacionados con la eficiencia y manipulación de código de bajo nivel son estudiados para completar las características implementadas en el prototipo: funciones en línea, funciones y objetos nativos y código no seguro.

9

Conclusiones

9.1 Resultados obtenidos

A partir del estudio de cómo mejorar los procesos de construcción de sistemas informáticos mediante el ensamblaje modular de componentes, este trabajo propone guías de diseño para lenguajes de programación que simplifiquen la construcción, ensamblado y modificación modular de componentes. El tipo de lenguajes al que van orientadas las guías de diseño es a lenguajes imperativos compilados con capacidades para la programación de sistemas. Para facilitar la construcción de componentes con las propiedades modulares deseadas, se analizan técnicas de implementación que puedan ser aplicables bien a lenguajes existentes, bien a la creación de nuevos lenguajes. Siguiendo estos criterios, este trabajo expone, razona y realiza una selección novedosa de las técnicas más idóneas, comprobándolas con la definición e implementación de un prototipo de un nuevo lenguaje de programación, llamado XC, y su compilador correspondiente. Se trata de un lenguaje orientado a la programación con componentes y objetos que incluye soporte para la programación de sistemas.

Se estudian las tendencias actuales de ingeniería del *software* para el diseño y construcción de sistemas informáticos con componentes y los modelos de componentes más aceptados, haciendo una revisión de sus cualidades más positivas y de sus carencias. Se propone un novedoso análisis formal cualitativo de sistemas informáticos desde un punto de vista diferente. Surge de la observación de que las carencias de modularidad y adaptabilidad de los sistemas informáticos tienen que ver con la falta de flexibilidad de éstos ante los cambios. Limitación que es debida a que directa o indirectamente asumen un comportamiento determinado de otros fragmentos de código —pequeños como una subrutina o grandes como un subsistema informático— que no los hace adaptables a los cambios.

El análisis presentado estudia las interdependencias entre fragmentos de código usando la teoría de conjuntos. Los conjuntos contienen aserciones locales asociadas a las sentencias de los fragmentos de código, que identifican aquellos aspectos asumidos como verdad por el código, el programador o el diseñador. Los resultados principales de este análisis son dos. En primer lugar, la identificación de la relevancia del conjunto de interdependencias implícitas existente entre esos fragmentos. Su particularidad reside en que, por definición, no son conocidas. En segundo lugar, se constata la importancia de la propagación exponencial de las dependencias implícitas como una de las causas principales de la percepción de complejidad de los programas y su falta de modularidad. El análisis caracteriza formalmente con más precisión y menos ambigüedad el conocido principio de encapsulación, la ruptura de la encapsulación, el reemplazo de componentes, y los serios inconvenientes asociados con el reemplazo, aspectos que en general son únicamente discutidos difusamente. Los resultados obtenidos se usan como base para una revisión crítica de las técnicas básicas de construcción de programas, sugiriendo modificaciones en la construcción y ensamblado de programas para hacerlos más modulares, usando reglas que limiten la proliferación de

interdependencias implícitas. El objetivo es crear programas más correctos, más complejos, y más modulares.

Se definen luego varias premisas de diseño para lenguajes de programación modular y se examina cómo se puede crear el soporte para modelos de componentes usando abstracciones de lenguajes de programación. Las premisas se aplican al diseño del lenguaje XC a lo largo del trabajo. Se razona la conveniencia del soporte de programación de sistemas y la necesidad de dotar a los lenguajes con este tipo de soporte con abstracciones de más alto nivel que no supongan una renuncia a la eficiencia o a la flexibilidad puntual de la programación de bajo nivel. Los resultados obtenidos abren las puertas a desarrollos más sofisticados y complejos de sistemas informáticos modulares y orientados a componentes con menor esfuerzo.

Para conseguir un lenguaje de propósito general práctico, se analizan posteriormente guías de consistencia, robustez, escalabilidad y eficiencia para lenguajes de programación, formulándose un conjunto pequeño de abstracciones que facilitan la construcción modular de componentes y mecanismos de extensión modular para ellos. El resultado obtenido es un modelo de componentes basado en un modelo de objetos subyacente extensible que, además, es conceptualmente sencillo.

También se obtiene un sistema de tipos independiente de la implementación, estático, modular, y orientado a objetos. Se evalúan los problemas de modularidad de la herencia y la conveniencia de limitar su aplicabilidad en beneficio de otras características más modulares.

El diseño de XC propone, describe e implementa un conjunto de técnicas originales que facilitan la evolución modular de sistemas basados en componentes. Son destacables las siguientes:

- Un pequeño conjunto de abstracciones consistentes y una gramática flexible para facilitar el uso del lenguaje.
- Un modelo de objetos novedoso basado en prototipos, que incorpora herencia y extensiones a objetos con mayor modularidad a sistemas construidos con el lenguaje.
- Soporte de extensiones de tipos y extensiones de implementación mediante un nuevo sistema de categorías modulares y un sistema de priorización de éstas, comprobables siempre en tiempo de compilación.
- Un nuevo modelo simplificado de extensiones ortogonales eficiente, igualmente verificables en tiempo de compilación, que facilita la metaprogramación y la programación con aspectos, sin necesidad de incorporar nuevas abstracciones. El modelo extiende directamente los objetos afectados usando categorías, en vez de metaobjetos externos y/o metaclasses, simplificando notablemente los diseños orientados a objetos en el lenguaje y, en particular, las relaciones de herencia de implementación.
- Para obtener una mejor comprobación de tipos en tiempo de compilación que reduzca la posibilidad de encontrar envíos de mensaje inválidos en tiempo de ejecución, se usa un sistema de tipos separado de implementación, con tipos covariantes exactos, y tipos genéricos que usan una nueva técnica llamada sustitución de tipos explícita. La técnica aplica la semántica exacta a la resolución de variables de tipos que, junto a las variables de tipos restringidas y homogéneas, dan como resultado una implementación más modular de los tipos genéricos.

Junto a los aspectos conceptuales, se diseña e implementa una sintaxis capaz de manejar las abstracciones del lenguaje con comodidad. El compilador de XC hace uso de la información sintáctica para realizar comprobaciones adicionales a las usuales del sistema de tipos, generar código más eficiente o simplificar el tratamiento de expresiones complejas.

Una propuesta nueva es el control sintáctico de los modificadores de los grados de libertad de las variantes de objetos prototipo, usados por el compilador para hacer optimizaciones. El desarrollador es capaz de indicar propiedades de la implementación de los objetos y componentes que mejoran su

eficiencia a cambio de limitar su generalidad, sin poner en riesgo las propiedades modulares del lenguaje.

La sintaxis flexible de XC ayuda a definir con naturalidad los protocolos protegidos de módulo, usados para extender módulos sin comprometer la modularidad, y los mecanismos de extensión modular con herencia y categorías, que describen sistemas más modulares y, a la vez, más extensibles. Adicionalmente, y para facilitar el desarrollo de sistemas complejos se incluye un soporte completo de módulos jerárquicos con posibilidad de declaraciones mutuamente recursivas.

Los operadores y expresiones estructuradas combinan conveniencia y expresividad. Usan una técnica novel en programación de sistemas, traduciéndose conceptualmente a protocolos con semántica definida y sin comprometer su eficiencia.

Las técnicas anteriores se aplican a la programación de componentes, que son definidos y extendidos incrementalmente de forma modular, y a los que pueden incorporarse opcionalmente comprobaciones semánticas, de control de calidad o depuración sin comprometer la modularidad. Se propone un nuevo modelo de componentes que, usando las abstracciones básicas del lenguaje, sirva para manipular componentes creados dentro del lenguaje e interoperar con otros modelos de componentes existentes. Para soportar directamente la construcción de componentes se usa un sistema de tipos separado de implementación, un sistema de envío de mensajes modular basado en selectores, y un mecanismo de despliegue final mediante módulos. Para soportar la factorización de código dentro de los componentes, se incluye un modelo de objetos subyacente que soporta herencia múltiple de interfaz y herencia simple de implementación.

Por último, y para comprobar la viabilidad de la propuesta desde el punto de vista de la eficiencia y la programación de sistemas, se implementan las siguientes optimizaciones: Optimización de objetos prototípico, envío eficiente de mensajes y ejecución eficiente de metamétodos, tipos integrales considerados como objetos manteniendo la eficiencia de los tipos primitivos, objetos nativos, funciones nativas, funciones en línea, y código no seguro.

9.2 Futuras líneas de investigación

La definición de un lenguaje de programación es una tarea que necesita tiempo para que las ideas que implementa se asienten, y se demuestre mediante un uso prolongado si dan fe de las expectativas puestas en ellas. Es esperanza del autor que los pensamientos y análisis expuestos en este trabajo inspiren el desarrollo de nuevas ideas en este campo.

La versión actual del lenguaje es la 0.7. Algunos aspectos del lenguaje todavía se encuentran en estudio. En particular es notable la falta de un sistema de excepciones, un diseño final para el recolector de basura y el soporte de múltiples hilos de ejecución. Existe la posibilidad de poder implementar los dos últimos directamente dentro del lenguaje. Particularmente complejo es ofrecer un protocolo más modular para la herencia de implementación, si es que tal protocolo debe existir. Otros aspectos menores también en estudio son pequeñas facilidades sintácticas como la declaración automática de protocolos constantes, o modificadores de argumentos y reenvío de llamadas (*forwarding*) para construir una infraestructura de componentes remotos directamente en el soporte en tiempo de ejecución.

El lenguaje incorpora un conjunto de técnicas de programación muy expresivas que no se encuentran reunidas en otros lenguajes, por lo que los idiomas de diseño en XC todavía se hallan en un estado germinal. Los diseños y hábitos actuales no aprovechan algunas capacidades del lenguaje. Nuevas necesidades y desafíos permitirán refinar las abstracciones existentes, pero no parece prudente aumentar su número en un horizonte cercano. El núcleo del lenguaje probablemente esté bastante completo en la actualidad.

Tampoco existen aún bibliotecas básicas maduras, tan solo algunos ensayos. La viabilidad de la construcción de bibliotecas eficientes que den utilidad al lenguaje sí es patente. Las facilidades nativas junto con los protocolos se han diseñado para simplificar la labor de incorporar código existente al lenguaje y el resultado es bastante satisfactorio. En cualquier caso, el diseño de un buen conjunto de bibliotecas básicas probablemente tenga extensión suficiente para un trabajo equivalente al presentado aquí.

Muy relacionado con la construcción de las bibliotecas básicas está la completitud del sistema de tipos. Más investigación es necesaria para obtener un sistema de tipos que no genere errores en tiempo de ejecución. La tendencia en la construcción del sistema de tipos en XC es hacia favorecer una semántica exacta, en vez de una semántica polimórfica. La semántica exacta tiene la ventaja adicional de adaptarse bastante mejor a los modelos de componentes que la semántica polimórfica. Hoy por hoy, la asignación polimórfica tradicional es el origen de la mayoría de los problemas más sutiles asociados al sistema de tipos de los lenguajes orientados a objetos. Sería deseable poder encontrar un sistema de tipos que exprese aquellas relaciones polimórficas deseadas usando una semántica exacta, sin tener que incluir explícitamente asignaciones polimórficas. La dirección de la investigación actual en el sistema de tipos de XC sigue este preciado objetivo.

La implementación del soporte en tiempo de ejecución completamente dentro del lenguaje, es una meta a medio plazo que requerirá considerable investigación, particularmente mayor hábito con el diseño en XC y mejores bibliotecas de base. Para que estas últimas sean viables, el sistema de tipos de XC debe estar esencialmente terminado primero. El soporte en tiempo de ejecución actual es elegante aunque relativamente básico, si bien ha permitido demostrar que es posible construir la mayor parte del soporte en tiempo de ejecución directamente dentro del lenguaje sin merma de eficiencia, simplificando sustancialmente la gestión del mismo así como posibles extensiones posteriores. Sin embargo, es importante sopesar con sumo cuidado las facilidades de introspección que ofrezca. Cambios en la semántica de la implementación del soporte en tiempo de ejecución —como abogan los partidarios de las implementaciones abiertas— pueden producir variaciones sutiles o no deseadas en la semántica de las abstracciones que provee el lenguaje que lo hagan más inconsistente.

Por último, y también a medio plazo, se encuentra el objetivo de la construcción de un compilador de XC enteramente en XC, que sirva como referencia del lenguaje. La intención es generar un compilador de XC compilado en C, cuya extensión a otras plataformas se pueda realizar mediante retoques de portabilidad en el generador de código.

Apéndice A

Gramática de XC

Este apéndice incluye una referencia de las palabras reservadas y la gramática de XC en notación BNF extendida. La gramática es LR(1) y se ha implementado con el compilador de gramáticas `bison` de GNU. `bison` no maneja bien las gramáticas ambiguas LR(1). Las reglas de producción han de ser más complejas para evitar ambigüedad. La gramática contiene alrededor de 400 reglas de producción. `bison` reduce reglas de producción de la derecha a la izquierda, tratando de localizar —para una lista de símbolos terminales y no terminales descrita a la derecha de la regla— un símbolo no terminal a la izquierda. La meta es ser capaz de reducir símbolos hasta encontrar el axioma de la gramática. Si éste se encuentra, se aceptará la gramática como correcta.

La gramática mostrada no es aún la gramática final de XC, se trata de la versión 0.7. No obstante, es relativamente madura e incluye producciones específicas para conveniencias sintácticas como por ejemplo: múltiples declaraciones y definiciones de módulo por fichero, permitir ‘;’ superfluos en bloques de código y declaraciones, sentencias nulas explícitas, bloques vacíos, modificadores opcionales, o precedencias razonables de operador que eviten paréntesis accesorios. Existen más palabras reservadas que las implementadas actualmente.

Símbolos terminales

Palabras clave						
<code>after</code>	<code>confined</code>	<code>fixed</code>	<code>inout</code>	<code>private</code>	<code>self</code>	<code>while</code>
<code>alias</code>	<code>continue</code>	<code>for</code>	<code>module</code>	<code>property</code>	<code>self</code>	
<code>baselevel</code>	<code>decode</code>	<code>framework</code>	<code>monitor</code>	<code>protected</code>	<code>Selfchain</code>	
<code>before</code>	<code>default</code>	<code>goto</code>	<code>native</code>	<code>protocol</code>	<code>shared</code>	
<code>break</code>	<code>do</code>	<code>if</code>	<code>null</code>	<code>prototype</code>	<code>single</code>	
<code>byref</code>	<code>else</code>	<code>implements</code>	<code>offset</code>	<code>public</code>	<code>sizeof</code>	
<code>byval</code>	<code>encode</code>	<code>implementation</code>	<code>oneway</code>	<code>readonly</code>	<code>storage</code>	
<code>case</code>	<code>enum</code>	<code>import</code>	<code>out</code>	<code>redefine</code>	<code>super</code>	
<code>category</code>	<code>extends</code>	<code>in</code>	<code>override</code>	<code>return</code>	<code>switch</code>	
<code>condition</code>	<code>final</code>	<code>inline</code>	<code>primitive</code>	<code>selector</code>	<code>unsafe</code>	

Tabla 9-1 Identificadores reservados de XC.

Símbolos terminales

FUNCTION_IDENTIFIER

IDENTIFIER

LOWERCASE_IDENTIFIER

MESSAGE_ARG_IDENTIFIER

NUMBER

STRING

UPPERCASE_IDENTIFIER

Tabla 9-2 Símbolos terminales generados por el analizador léxico.

Gramática

Program structure:

```

program ::= 
    program_definition
  | program program_definition

program_definition ::= 
    module_declaration
  | module_definition

module_declaration ::= 
    scope_modifier module protocol qualified_uppercase_identifier
    { module_declaration_body }

module_declaration_body ::= 
    module_declaration_item_list
  | module_import_list module_declaration_item_list

module_declaration_item_list ::= 
    module_declaration_item
  | module_declaration_item_list module_declaration_item

module_declaration_item ::= 
    ;
  | prototype_declaration
  | protocol_declaration
  | protocol_definition
  | constant_declaration
  | constant_definition
  | function_declaration
  | enum_definition
  | module_declaration
  | alias_definition

module_import_list ::= 
    module_import
  | module_import_list module_import

module_import ::= 
    import qualified_uppercase_identifier ;
  | import module_declaration
  
```

Modifiers:

```

scope_modifier ::= 
    private
  | protected
  | public

flag_modifier ::= 
    final
  | unsafe
  | inline
  | static
  | shared
  | primitive
  | before
  | after
  | override
  | fixed
  | confined
  
```

```

| implementation
| baselvel
| scope_modifier

flag_modifier_list ::= 
    flag_modifier
| flag_modifier_list flag_modifier

```

Identifiers:

```

var_or_const_identifier ::= 
    lowercase_identifier
| uppercase_identifier
| identifier

identifier ::= 
    IDENTIFIER

lowercase_identifier ::= 
    LOWERCASE_IDENTIFIER

function_identifier ::= 
    FUNCTION_IDENTIFIER

uppercase_identifier ::= 
    UPPERCASE_IDENTIFIER

message_arg_identifier ::= 
    MESSAGE_ARG_IDENTIFIER

qualified_identifier ::= 
    lowercase_identifier
| identifier
| qualified_uppercase_identifier
| qualified_uppercase_identifier . lowercase_identifier
| qualified_uppercase_identifier : identifier

qualified_uppercase_identifier ::= 
    uppercase_identifier
| qualified_uppercase_identifier . uppercase_identifier

```

Constant declaration:

```

constant_declaration ::= 
    opaque_constant_declaration ;

opaque_constant_declaration ::= 
    const qualified_uppercase_identifier const_declaration_identifier_list

const_declaration_identifier_list ::= 
    const_declaration_identifier
| const_declaration_identifier_list , const_declaration_identifier

const_declaration_identifier ::= 
    var_or_const_identifier optional_array_index_qualifier

```

Protocol declaration:

```

alias_definition ::= 
    protocol alias uppercase_identifier = qualified_uppercase_identifier ;

protocol_declaration ::= 
    basic_protocol_header ;

```

```

protocol_header ::= 
  flag_modifier_list basic_protocol_header
  | basic_protocol_header

basic_protocol_header ::= 
  protocol protocol_uppercase_identifier
  | protocol category protocol_uppercase_identifier

extends_protocol_qualifier ::= 
  extends: protocol_identifier_list

equal_protocol_qualifier ::= 
  = protocol_identifier

typevar_protocol_qualifier ::= 
  < typevar_definition_list >

protocol_definition ::= 
  protocol_header protocol_definition_block
  | protocol_header extends_protocol_qualifier typevar_protocol_qualifier protocol_definition_block
  | protocol_header equal_protocol_qualifier typevar_protocol_qualifier ;

protocol_definition_block ::= 
  {}
  | { protocol_definition_body }

protocol_definition_body ::= 
  protocol_definition_body_item
  | protocol_definition_body protocol_definition_body_item

protocol_definition_body_item ::= 
  ;
  | property_declaration
  | message_declaration

protocol_identifier_list ::= 
  protocol_uppercase_identifier_list

non_void_protocol_identifier ::= 
  protocol_uppercase_identifier
  | protocol_keyword_identifier
  | enum qualified_uppercase_identifier

protocol_identifier ::= 
  non_void_protocol_identifier
  | protocol_void_keyword_identifier

protocol_uppercase_identifier_list ::= 
  protocol_uppercase_identifier
  | protocol_uppercase_identifier_list , protocol_uppercase_identifier

protocol_uppercase_identifier ::= 
  qualified_uppercase_identifier
  | const qualified_uppercase_identifier

protocol_keyword_identifier ::= 
  Self
  | Selfchain

protocol_void_keyword_identifier ::= 
  void

```

Property declaration:

```

property_declaracion ::= 
    flag_modificador_list property_declaracion_header
    | property_declaracion_header

property_declaracion_header ::= 
    property protocolo_identificador lowercase_identificador optional_array_index_qualifier ;

optional_array_index_qualifier ::= 
    λ
    | array_index_qualifier

array_index_qualifier ::= 
    []
    | [ primary_expression ]

```

Prototype declaration:

```

prototype_declaracion ::= 
    prototype_header ;
    | prototype_header extends_prototype_qualifier implements_prototype_qualifier
        typevar_prototype_qualifier before_or_after_prototype_qualifier_list ;

prototype_header ::= 
    flag_modificador_list basic_prototype_header
    | basic_prototype_header

basic_prototype_header ::= 
    prototype uppercase_identificador
    | prototype category uppercase_identificador

extends_prototype_qualifier ::= 
    extends: qualified_uppercase_identificador

equal_prototype_qualifier ::= 
    = qualified_uppercase_identificador

typevar_prototype_qualifier ::= 
    < typevar_definition_list >

implements_prototype_qualifier ::= 
    implements: protocolo_identificador_list

before_or_after_prototype_qualifier ::= 
    before: qualified_uppercase_identificador
    | after: qualified_uppercase_identificador

before_or_after_prototype_qualifier_list ::= 
    before_or_after_prototype_qualifier
    | before_or_after_prototype_qualifier_list before_or_after_prototype_qualifier

```

Function declaration:

```

function_declaracion ::= 
    flag_modificador_list function_declaracion_header
    | function_declaracion_header

function_declaracion_header ::= 
    protocolo_identificador message_identificador_declaracion ;

```

Message declaration:

```

message_declaracion ::= 
    flag_modificador_list message_declaracion_header
    | message_declaracion_header

message_declaracion_header ::= 
    protocolo_identificador message_identificador_declaracion ;

message_identificador_declaracion ::= 
    lowercase_identifier
    | function_identifier
    | message_argument_declaration_list

message_argument_declaration_list ::= 
    message_argument_declaration
    | message_argument_declaration_list message_argument_declaration

message_argument_declaration ::= 
    message_arg_identifier protocolo_identificador var_or_const_identifier

```

Module definition:

```

module_definition ::= 
    modulee qualified_uppercase_identifier { module_definition_body }

module_definition_body ::= 
    module_definition_item_list
    | module_import_list module_definition_item_list

module_definition_item_list ::= 
    module_definition_item
    | module_definition_item_list module_definition_item

module_definition_item ::= 
    ;
    | prototype_declaration
    | prototype_definition
    | protocol_declaration
    | protocol_definition
    | function_definition
    | constant_definition
    | variable_definition
    | enum_definition
    | module_definition
    | alias_definition

```

Constant definition:

```

constant_definition ::= 
    constant_statement ;

```

Variable definition:

```

variable_definition ::= 
    property declaration_statement ;

```

Enumeration definition:

```

enum_definition ::=

  enum uppercase_identifier { enum_definition_block }

enum_definition_block ::=

  enum_item_without_colon
  | enum_item_list
  | enum_item_list enum_item_without_colon

enum_item_list ::=

  enum_item_with_colon
  | enum_item_list , enum_item_with_colon

enum_item_with_colon ::=

  enum_item ,

enum_item_without_colon ::=

  enum_item

enum_item ::=

  var_or_const_identifier
  | var_or_const_identifier = primary_expression

```

Property definition:

```

property_definition ::=

  flag_modifier_list property_definition_header
  | flag_modifier_list native property_definition_header
  | property_definition_header
  | native property_definition_header

property_definition_header ::=

  property storage_qualifier protocol_identifier lowercase_identifier
    optional_array_index_qualifier ;
  | property_declaration_header

storage_qualifier ::=

  storage: primary_expression

```

Type variable definition:

```

typevar_definition_list ::=

  typevar_definition
  | typevar_definition_list , typevar_definition

typevar_definition ::=

  uppercase_identifier = protocol_uppercase_identifier

```

Prototype definition:

```

prototype_definition ::=

  prototype_header
  | prototype_header extends_prototype_qualifier implements_prototype_qualifier
    typevar_prototype_qualifier before_or_after_prototype_qualifier_list
    prototype_definition_block
  | prototype_header equal_prototype_qualifier typevar_prototype_qualifier ;

prototype_definition_block ::=

  {}
  | { prototype_definition_body }

```

```

prototype_definition_body ::=  

    prototype_definition_body_item  

    | prototype_definition_body prototype_definition_body_item

prototype_definition_body_item ::=  

    ;  

    | property_definition  

    | method_definition

```

Function definition:

```

function_definition ::=  

    flag_modifier_list function_definition_header  

    | function_definition_header

function_definition_header ::=  

    protocol_identifier message_identifier_declaration method_definition_body  

    | native protocol_identifier message_identifier_declaration ;

```

Method definition:

```

method_definition ::=  

    flag_modifier_list method_definition_header  

    | method_definition_header

method_definition_header ::=  

    protocol_identifier message_identifier_declaration method_definition_body  

    | native protocol_identifier message_identifier_declaration ;

method_definition_body ::=  

    { }  

    | { statement_list }

```

Statements:

```

statement_list ::=  

    statement  

    | statement_list statement

statement ::=  

    ;  

    | statement_with_semicolon ;  

    | statement_without_semicolon

statement_with_semicolon ::=  

    expression_statement  

    | do_statement  

    | return_statement  

    | declaration_statement  

    | null  

    | break  

    | continue

statement_without_semicolon ::=  

    block_statement  

    | if_statement  

    | while_statement  

    | for_statement  

    | switch_statement

expression_statement ::=  

    message_expression  

    | assignment_expression  

    | postfix_expression

```

```

block_statement ::= 
  { }
  | { statement_list }

if_statement ::= 
  if ( expression ) statement else_statement

else_statement ::= 
  λ
  | else statement

while_statement ::= 
  while ( expression ) statement

do_statement ::= 
  do statement while ( expression )

for_statement ::= 
  for ( for_init ; for_expression ; for_next ) statement

for_init ::= 
  λ
  | for_expression_statement_list
  | declaration_statement

for_expression ::= 
  primary_expression
  | for_expression , primary_expression

for_next ::= 
  λ
  | for_expression_statement_list

for_expression_statement_list ::= 
  expression_statement
  | for_expression_statement_list , expression_statement

switch_statement ::= 
  switch ( expression ) { switch_block_statement_list }

switch_block_statement_list ::= 
  switch_block_statement
  | switch_block_statement_list switch_block_statement

switch_block_statement ::= 
  switch_label block_statement

switch_label ::= 
  default :
  | primary_expression :

return_statement ::= 
  return expression
  | return

constant_statement ::= 
  const qualified_uppercase_identifier declaration_assignment_list

declaration_statement ::= 
  non_void_protocol_identifier declaration_identifier_list

declaration_identifier_list ::= 
  declaration_identifier
  | declaration_identifier_list , declaration_identifier

declaration_identifier ::= 

```

```

var_or_const_identifier optional_array_index_qualifier
| declaration_assignment

declaration_assignment_list ::= 
    declaration_assignment
| declaration_assignment_list , declaration_assignment

declaration_assignment ::= 
    var_or_const_identifier optional_array_index_qualifier = expression

```

General expressions:

```

expression ::= 
    structured_expression
| primary_expression

```

Structured expressions:

```

structured_expression ::= 
    dictionary_value_expression
| array_value_expression
| structured_operator_expression
| structured_message_expression
| ( structured_expression )

structured_operator_expression ::= 
    structured_expression == primary_expression
| primary_expression == structured_expression
| structured_expression != primary_expression
| primary_expression != structured_expression
| structured_expression === primary_expression
| primary_expression === structured_expression
| primary_expression !== structured_expression
| structured_expression !== primary_expression
| structured_expression + primary_expression
| primary_expression + structured_expression
| structured_expression + structured_expression

structured_message_expression ::= 
    dictionary_value_expression . message_send_expression
| array_value_expression . message_send_expression

array_value_expression ::= 
    [ ]
| [ array_value_expression_list ]

array_value_expression_list ::= 
    expression
| array_value_expression_list , expression

dictionary_value_expression ::= 
    { }
| { dictionary_value_expression_list }

dictionary_value_expression_list ::= 
    dictionary_value_expression_item
| dictionary_value_expression_list , dictionary_value_expression_item

dictionary_value_expression_item ::= 
    var_or_const_identifier = expression
| string_expression = expression

```

Primary expressions:

```

primary_expression ::= object_expression
| operator_expression
| cast_expression
| ( primary_expression )

object_expression ::= number_expression
| array_expression
| dictionary_expression
| message_expression
| identifier_expression
| string_expression
| self
| super
| property
| false
| true
| null
| sizeof_expression
| offset_expression
| selector_expression
| return_expression

sizeof_expression ::= sizeof: qualified_uppercase_identifier
| sizeof: qualified_uppercase_identifier . lowercase_identifier

offset_expression ::= offset: qualified_uppercase_identifier
| offset: qualified_uppercase_identifier . lowercase_identifier

selector_expression ::= selector

return_expression ::= return

number_expression ::= unsigned_number_expression

unsigned_number_expression ::= NUMBER

string_expression ::= STRING

operator_expression ::= unary_expression
| binary_expression
| ternary_expression

unary_expression ::= prefix_expression
| postfix_expression

prefix_expression ::= + primary_expression
| - primary_expression
| ~ primary_expression
| ! primary_expression

postfix_expression ::= primary_expression ++
| primary_expression --

```

```

binary_expression ::=

arithmetic_expression ::=

    primary_expression + primary_expression
    primary_expression - primary_expression
    primary_expression * primary_expression
    primary_expression / primary_expression
    primary_expression % primary_expression

relational_expression ::=

    primary_expression == primary_expression
    primary_expression != primary_expression
    primary_expression === primary_expression
    primary_expression !== primary_expression
    primary_expression > primary_expression
    primary_expression < primary_expression
    primary_expression >= primary_expression
    primary_expression <= primary_expression

logical_expression ::=

    primary_expression && primary_expression
    primary_expression || primary_expression

bit_expression ::=

    primary_expression ^ primary_expression
    primary_expression & primary_expression
    primary_expression | primary_expression
    primary_expression << primary_expression
    primary_expression >> primary_expression

assignment_attempt_expression ::=

    primary_expression ?= primary_expression

assignment_expression ::=

    identifier_expression = expression
    array_expression = expression
    dictionary_expression = expression
    assignment_operator_expression
    message_expression = expression

identifier_expression ::=

    qualified_identifier

array_expression ::=

    primary_expression array_index_qualifier

dictionary_expression ::=

    primary_expression { primary_expression }

assignment_operator_expression ::=

    primary_expression += primary_expression
    primary_expression -= primary_expression
    primary_expression *= primary_expression
    primary_expression /= primary_expression
    primary_expression %= primary_expression
    primary_expression ^= primary_expression
    primary_expression &= primary_expression
    primary_expression |= primary_expression
    primary_expression <<= primary_expression
    primary_expression >>= primary_expression

```

ternary_expression ::= primary_expression ? primary_expression : primary_expression

*cast_expression ::= (protocol_keyword_identifier) primary_expression
| (qualified_uppercase_identifier) primary_expression*

Message and function expressions:

*message_expression ::= primary_expression . message_send_expression
| function_expression*

*function_expression ::= function_identifier
| message_argument_expression_list
| qualified_uppercase_identifier . function_identifier
| qualified_uppercase_identifier : message_argument_expression_list*

*message_send_expression ::= lowercase_identifier
| message_argument_expression_list*

*message_argument_expression_list ::= message_argument_expression
| message_argument_expression_list message_argument_expression*

message_argument_expression ::= message_arg_identifier expression

Apéndice B

Ejemplos de código

Es difícil describir un lenguaje de programación sin proporcionar alguna pieza de código que permita visualizar de primera mano cómo se reflejan los conceptos descritos en este trabajo. Este apéndice incluye el código experimental operativo de una declaración e implementación de tablas *hash* y diccionarios, escrito enteramente en XC y soportado directamente por la sintaxis del lenguaje a través de expresiones estructuradas. El código es suficientemente sencillo para ser mostrado en poco espacio y suficientemente complejo para dar una idea de la sintaxis y uso de XC en un ejemplo más real. La tabla *hash* es únicamente un protocolo base. Un diccionario de objetos se crea mediante la siguiente sentencia:

```
import XC.Collections.HashTables;
// ...
Dictionary dict = { key1 = "val1", key2 = 4 };
```

El compilador traduce la sintaxis anterior al siguiente conjunto de mensajes equivalente:

```
dict = (Dictionary.clone.at: "key1" put: "val1").at: "key2" put: 4;
```

Tanto la declaración como la implementación de diccionarios importa otros módulos, no mostrados por brevedad.

Declaración de diccionarios

```
public module protocol XC.Collections.HashTables
{
    import XC;
    import XC.Collections;
    import XC.Collections.Arrays;

    protocol KeyArray = ArrayList <Element = String>;
    protocol ValueArray = ArrayList <Element = Object>;

    protocol CollisionList extends: Object
    {
        property KeyArray keys;
        property ValueArray values;
        Self allocSize: Unsigned size;
        Unsigned length;
        void dbgPrint: String indent;
    }

    protocol Dictionary extends: HashTable <Key = String, Element = Object>
    {
        property Unsigned numElements;
        property CollisionList nodes[];
    }
}
```

```

    } prototype Dictionary extends: Object;
}

```

Definición de diccionarios

```

module XC.Collections.HashTables
{
    import XC.System;
    import XC.Runtime;

    const Unsigned REALLOC_LIMIT      = 10000;
    const Unsigned PTR_SIZE_BITS     = 32;      // Assume a 32-bit processor
    const Unsigned PTR_ALIGN_BITS    = 2;        // Assume a 4 byte alignment
    const Unsigned DEFAULT_DICT_SIZE = 16;
    const Unsigned DEFAULT_COLLIST_SIZE = 4;

    final confined prototype KeyArray = ArrayList <Element = String>;
    final confined prototype ValueArray = ArrayList <Element = Object>;

    final confined prototype CollisionList extends: Prototype
    {
        property KeyArray keys;
        property ValueArray values;

        self alloc
        {
            return self.allocSize: DEFAULT_COLLIST_SIZE;
        }

        self allocSize: Unsigned size
        {
            self newInstance = super.alloc;

            newInstance.keys = KeyArray.allocsize: size;
            newInstance.values = ValueArray.allocSize: size;
            return newInstance;
        }

        self init
        {
            self.keys.init;
            self.values.init;
            return self;
        }

        Unsigned length
        {
            return self.keys.length;
        }

        void dbgPrint: String indent
        {
            Unsigned length = self.length;
            Object obj;
            String str;

            for (Unsigned i = 0; i < length; i++)
            {
                Console.print: indent + "[" + i.asString + "] " +
                    "(key: '" + self.keys[i].asString + "', val: '" +
                    self.values[i].asString + "')\n";
            }
        }

        prototype Dictionary extends: Object implements: HashTable <Key = String, Element = Object>
        {
            property Unsigned numElements;
            property CollisionList nodes[];

            self alloc
            {
                return self.allocSize: DEFAULT_DICT_SIZE;
            }

            before self allocSize: Unsigned size
            {

```

```

// Request a power of two value
Contract.assert: (size & (size - 1)) == 0;
Console.print: "Before method Dictionary.allocSize:\n";
}

self allocSize: Unsigned size
{
    Self newInstance = super.alloc;
    newInstance.nodes = Array.allocsize: size;
    return newInstance;
}

after Self allocsize: Unsigned size
{
    Console.print: "After method " + self.family + "." + selector.messageName + "\n";
}

self init
{
    self.numElements = 0;
    self.nodes.init;
    return self;
}

Element atKey: Key key
{
    CollisionList colList;
    Unsigned i, length;
    colList = self.nodes[key.hash % self.nodes.length];
    Contract.assert: colList != null;
    length = colList.length;
    for (i = 0; i < length; i++)
    {
        if (colList.keys[i] == key)
        {
            return colList.values[i];
        }
    }
    return null;
}

Selfchain atKey: Key key put: Element value
{
    CollisionList colList;
    Unsigned i, length, nodeIndex;
    nodeIndex = key.hash % self.nodes.length;
    colList = self.nodes[nodeIndex];
    if (colList == null)
    {
        colList = CollisionList.clone;
        self.nodes[nodeIndex] = colList;
    }
    length = colList.length;
    colList.keys[length] = key;
    colList.values[length] = value;
    return self;
}

Selfchain reallocsize: Unsigned newsize
{
    CollisionList newNodes[], oldColList, newColList;
    Unsigned i, j, length, oldSize, nodeIndex;
    Key key;
    oldSize = self.nodes.length;
    if (oldSize >= newSize)
        return self;

    // If requested size is too big, it's likely that there is some error
    // hanging around.

    Contract.assert: (newSize - oldSize < REALLOC_LIMIT);
    newNodes = (Array.allocsize: newSize).init;
    // After a reallocation hash values for each entry change, so we have to
    // reinsert items again. Maybe there is a faster way to do this based
}

```

```

// on knowledge about how the hash function behaves.

for (i = 0; i < oldsize; i++)
{
    oldColList = self.nodes[i];
    length = oldColList.length;
    for (j = 0; j < length; j++)
    {
        key = oldColList.keys[j];
        nodeIndex = key.hash % oldsize;
        newColList = newNodes[nodeIndex];
        if (newColList == null)
        {
            newColList = CollisionList.clone();
            newNodes[nodeIndex] = newColList;
        }
        length = newColList.length;
        newColList.keys[length] = key;
        newColList.values[length] = oldColList.values[j];
    }
    self.nodes = newNodes;
    return self;
}

void dbgPrint
{
    String indent = "    ";
    String indent2 = indent + indent;
    CollisionList colList;

    Console.print: "Dictionary dump:\n{\n";
    for (Unsigned i = 0; i < self.nodes.length; i++)
    {
        Console.print: indent + "[" + i.asString + "]\n";
        colList = self.nodes[i];
        if (colList != null)
            colList.dbgPrint: indent2;
    }
    Console.print: "}\n";
}
}

```

Apéndice C

Aspectos de implementación

En el capítulo octavo se han dado nociones de la implementación de algunos aspectos relevantes de XC. En la mayoría de los casos, la descripción facilitada es suficiente para tener una idea general de su funcionamiento interno. Quizás, el aspecto más interesante que no se incluye en ese capítulo y que es de importancia crítica para el lenguaje es la implementación de la función mensajera. La ejecución de cadenas de métodos exige una implementación cuidadosa para no perder eficiencia. Consta de unas pocas instrucciones en ensamblador para la arquitectura x86 de Intel.

La función mensajera

La búsqueda de la cadena de métodos a ejecutar se realiza con la función `xc_method_lookup`, que accede a la tabla de métodos usando el índice del selector como entrada en el *array* disperso `bucket_array`.

```
struct xc_method_imp_list *  
xc_method_lookup (struct xc_sparse_array *self,  
                  unsigned index)  
{  
    return &((self->bucket_array[index >> XC_BUCKET_BITS])->  
              data[index & XC_BUCKET_MASK]);  
}
```

El registro de activación y la función mensajera se describe en los apartados 8.4.2 y 8.4.3. La función `xc_method_lookup` es expandida en línea dentro de la función mensajera. Luego se crea el registro de activación de métodos y se ejecutan los métodos de la cadena en `msg_loop`. En `msg_ret` se comprueba si se ha terminado la cadena —implementada como una lista enlazada terminada en `NULL`— y, si es así, en `msg_end` se restaura el registro de activación anterior y se retorna.

```
struct xc_object *  
xc_sendmsg (struct xc_object *self, struct xc_selector *sel, ...)  
{  
    xc_method_lookup (self->map->mth_dispatch, sel->sel_id);  
    /* method implementation return value is stored in eax register.  
     * edx      : method implementation chain node  
     * [edx]    : method implementation pointer  
     * [edx+4]  : next method implementation chain node  
     * ebp     : sendmsg ebp (caller frame pointer)  
     * ebp+4   : return addr  
     * ebp+8   : method implementation chain node  
     * ebp+12  : original ebp, caller's ebp (caller's frame pointer)  
     * ebp+16  : original ret, caller's return addr. */  
  
    asm  
    {  
        // create new stack frame just below xc_sendmsg stack frame  
        mov edx, eax  
        // edx current imp. chain  
    }
```

```
        mov ecx, dword ptr [edx+4] // Get next mth. implementation
        mov esp, ebp             // original msgsend stack ptr.
        sub esp, 4               // jump the super placeholder

        // method call chain loop

msg_loop: push ecx           // save imp. chain ptr.
          push offset msg_ret // msgsend return addr.
          jmp dword ptr [edx]  // method imp. call
msg_ret: pop edx            // restore imp. chain ptr.
          cmp edx, 0            // if chain is finished return
          je msg_end
          mov ecx, dword ptr [edx+4] // Get next mth. implementation
          jmp msg_loop

        // restore original sendmsg stack frame

msg_end: add esp, 4          // jump back the super placeholder
          pop ebp              // restore original ebp
          ret

}
```

Bibliografía

- Abelson, 1996 *Structure and interpretation of computer programs.* 2nd Edition. H. Abelson, G. Sussman. MIT Press, 1996.
- America, 1987 *Inheritance and Subtyping in a Parallel Object-Oriented Language.* P. America. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag Berlin Heidelberg, 1987. pp. 235-242.
- America & van der Linden, 1990 *A Parallel Object-Oriented Language with Inheritance and Subtyping.* P. America, F. van der Linden. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) and European Conference on Object-Oriented Programming (ECOOP). ACM Press, 1990.
- André & Royer, 1992 *Optimizing Method Search with Lookup Caches and Incremental Coloring.* P. André, J. Royer. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1992. pp. 110-126.
- Andrews, 2000 *Foundations of Multithreaded, Parallel, and Distributed Programming.* G. Andrews. Addison-Wesley, 2000.
- Apple, 1995 *Dylan Reference Manual.* Draft. Apple Computer, 1995.
- Apple, 2002a *Objective-C implementation.* Project obj4 Version 222. www.opensource.apple.com/projects/darwin/6.0/projects.html. Apple Computer, 2002.
- Apple, 2002b *I/O Kit Fundamentals.* Mac OS X 10.2 Developer Documentation. developer.apple.com/techpubs/macosx/Darwin/General/IOKitFundamentals/IOKitFundamentals.pdf. Apple Computer, 2002.
- Apple, 2002c *Carbon Specification.* Mac OS X 10.2 Developer Documentation. developer.apple.com/techpubs/macosx/Carbon. Apple Computer, 2002.
- Baker, 2000 *The Making of Orbix and the iPortal Suite.* S. Baker. Proceeding of International Conference on Software Engineering (ISCE), ACM Press, 2000. pp. 609-616.
- Bardou & Dony, 1995 *Propositions pour un nouveau modèle d'objects dans les langages à prototypes.* D. Bardou, C. Dony. Actes de Langages et Modèles à Objects (LMO), Nancy, France, pp. 93-109. 1995. Ref. en [Bardou & Dony, 1996; p. 133]
- Bardou & Dony, 1996 *Split Objects: a Disciplined Use of Delegation within Objects.* D. Bardou, C. Dony. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1996. pp. 122-137.
- Barnes, 1994 *Programming in Ada.* J. Barnes. Addison-Wesley, 1994.
- Biddle & Tempero, 1996 *Understanding the Impact of Language Features on Reusability.* R. Biddle, E. Tempero. Technical Report CS-TR-95/17. Dept. Computer Science. Victoria University of Wellington. New Zealand, 1996.
- Bobrow & Stefik, 1983 *The Loops Manual.* D. Bobrow, M. Stefik. Intelligent Systems Laboratory, Xerox PARC, 1983. Ref. en [Kiczales, Rivières & Bobrow, 1991].

- Bobrow *et alii*, 1986 *CommonLoops: Merging LISP and Object-Oriented Programming*. D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1986. pp. 17-29.
- Booch, 1994 *Object-Oriented Analysis and Design with Applications - 2nd ed.* Grady Booch. Addison-Wesley, 1994.
- Borning, 1981 *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*. A. Borning. ACM Transactions on Programming Languages and Systems. Vol. 3, No. 4. ACM Press, Oct. 1981. pp. 353-387.
- Borning, 1986 *Classes versus Prototypes in Object-Oriented Languages*. A. Borning. Proceedings of ACM/IEEE Fall Joint Computer Conference. ACM Press. Nov. 1986. pp. 36-40.
- Bouraqadi-Sââdani, Ledoux & Rivard, 1998 *Safe Metaclass Programming*. N. Bouraqadi-Sââdani, T. Ledoux, F. Rivard. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1998. pp. 84-96.
- Bracha & Cook, 1990 *Mixin-Based Inheritance*. G. Bracha, W. Cook. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1990. pp. 303-311.
- Britton & Parnas, 1981 *A-7E Software Module Guide*. K. Britton, D. Parnas. Washington D.C. Naval Research Laboratory, Report 4702, 1981. Ref. en [Booch 1994; p. 56]
- Brooks, 1975 *The Mythical Man-Month*. F. Brooks. Addison-Wesley, 1975. Reimpreso y extendido en: *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, 1995.
- Brooks, 1987 *No Silver Bullet. Essence and Accidents of Software Engineering*. IEEE Computer, Vol. 20, Nº.4, 1987. pp. 10-19.
- Bruce *et alii*, 1995 *On binary methods*. K. Bruce, L. Cardelli, G. Castagna, J. Eifrig, S. Smith, V. Trifonov, G. Leavens, B. Pierce. Theory and Practice of Object Systems (TAPOS). Vol. 1, No. 3, 1995. pp. 221-242.
- Bruce, 1996 *Typing in object-oriented languages: Achieving expressibility and safety*. K. Bruce. Technical report, www.cs.williams.edu/~kim/README.html. Williams College, 1996.
- Bruce, 1997 *Increasing Java's expressiveness with ThisType and match-bounded polymorphism*. K. Bruce. www.cs.williams.edu/~kim/README.html. Williams College, 1997.
- Bruce, Odersky & Wadler, 1998 *A Statically Safe Alternative to Virtual Types*. K. Bruce, M. Odersky, P. Wadler. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag Berlin Heidelberg, 1998. pp. 524-549.
- Bruce & Vanderwaart, 1999 *Semantics-Driven Language Design: Statically Type-Safe Virtual Types in Object-Oriented Languages*. K. Bruce and J. Vanderwaart. Mathematical Foundations of Programming Semantics (MFPS), 1999.
- Cardelli *et alii*, 1992 *Modula-3 language definition*. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson. SIGPLAN Notices, 27(8) : 15-43, ACM Press, August 1992.
- Chambers, 1992a *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Languages*. C. Chambers. Ph.D. Thesis. Standford University. 1992.
- Chambers, 1992b *Object-Oriented Multi-Methods in Cecil*. C. Chambers. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag Berlin Heidelberg, 1992. pp. 33-56.

- Chambers & Leavens, 1994
Typechecking and Modules for Multi-Methods. C. Chambers, G. Leavens. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1994. pp. 1-15.
- Chambers & Chen, 1999
Efficient Multiple and Predicate Dispatching. C. Chambers, W. Chen. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1999. pp. 238-255.
- Chiba, 1993
Open C++ Programmer's Guide. S. Chiba. Technical Report 93-3. Department of Information Science, www.csg.is.titech.ac.jp/~chiba/openc++.html. University of Tokyo, 1993.
- Chiba, 1995
A Metaobject Protocol for C++. S. Chiba. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1995. pp. 285-299.
- Chiba, 1998
Macro Processing in Object-Oriented Languages. S. Chiba. Technology of Object-Oriented Languages and Systems (TOOLS). IEEE Press, November 1998. pp. 113-126.
- Clements & Northrop, 2002
Software Product Lines. Practices and Patterns. P. Clements, L. Northrop. Addison-Wesley, 2002.
- Cohen, 1999
From Product Line Architectures to Products. Position paper for the European Conference on Object-Oriented Programming (ECOOP). Workshop on Object-Technology for Product-Line Architectures, Lisbon, Portugal, 1999. Ref. en [Czarnecki & Eisenecker, 2000; pp. 29-30].
- Cook, 1989
A Proposal for Making Eiffel Type-safe. W. Cook. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag Berlin Heidelberg, 1989. pp. 57-70.
- Cook, 1992
Interfaces and Specifications for the Smalltalk-80 Collection Classes. W. Cook. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1992. pp. 1-15.
- Council & Heineman, 2001
Definition of a Software Component and Its Elements. B. Council, G. Heineman. En [Heineman & Council, 2001], pp. 5-19.
- Cox & Novobilsky, 1991
Object-Oriented Programming. An Evolutionary Approach. 2nd Edition. B. Cox, A. Novobilski. Addison-Wesley, Reading, MA, 1991.
- Cox, 1996
Superdistribution. B. Cox. Addison-Wesley, Reading, MA, 1996.
- Craig, 2000
The Interpretation of Object-Oriented Programming Languages. I. Craig. Springer-Verlag, London, 2000.
- Czarnecki & Eisenecker, 2000
Generative Programming. Methods, Tools and Applications. K. Czarnecki, U. Eisenecker. Addison-Wesley, 2000.
- Dahl, Myrhaug & Nygaard, 1968
SIMULA 67 Common Base Definition. O. Dahl, B. Myrhaug, K. Nygaard. Norwegian Computing Center S-2. Oslo, 1968. Ref. en [Nygaard & Dahl, 1978; p. 263].
- Dahl & Hoare, 1972
Hierarchical Program Construction in Structured Programming. O. Dahl, C. Hoare. Academic Press, NY, 1972. pp. 174-220. Ref. en [Nygaard & Dahl, 1978; p. 260].
- Dahl, Dijkstra & Hoare, 1972
Structured Programming. O. Dahl, E. Dijkstra, C. Hoare. Academic Press, NY, 1972. Ref. en [Booch, 1994; p. 41].
- Dony, Malenfant & Cointe, 1992
Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. C. Dony, J. Malenfant, P. Cointe. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1992. pp. 201-217.

- Driesen, 1993 *Selector Table Indexing and Sparse Arrays*. K. Driesen. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1993. pp. 259-270.
- Driesen, Hölzle & Vitek, 1995 *Message Dispatch on Pipelined Processors*. K. Driesen, U. Hölzle, J. Vitek. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag Berlin Heidelberg, 1995. pp. 253-282.
- ECMA, 1999 *ECMAScript Language Specification*. Standard ECMA-262. 3rd Edition. European Computer Manufacturers Association (ECMA), Geneva, Switzerland. June 1993.
- ECMA, 2002a *C# Language Specification*. Standard ECMA-334. 2nd Edition. European Computer Manufacturers Association (ECMA), Geneva, Switzerland. December 2002.
- ECMA, 2002b *Common Language Infrastructure (CLI)*. Standard ECMA-335. 2nd Edition. European Computer Manufacturers Association (ECMA), Geneva, Switzerland. December 2002.
- Fichman & Kemerer, 1997 *Object Technology and Reuse: Lessons from Early Adopters*. R. Fichman, C. Kemerer. IEEE Computer, Vol. 14, Nº. 10, October 1997. pp. 47-59.
- Forman & Danforth, 1999 *Putting Metaclasses to Work. A new Dimension in Object-Oriented Programming*. I. Forman, S. Danforth. Addison-Wesley, Reading, MA, 1999.
- Gamma, Helm, Johnson & Vlissides, 1995 *Design Patterns. Elements of Reusable Object-Oriented Software*. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison-Wesley, 1995.
- GNU, 2003 *GCC Home Page*. The GNU Project. www.gnu.org/software/gcc.html, 2003
- Goldberg & Robson, 1989 *Smalltalk-80. The language*. A. Goldberg, D. Robson. Addison-Wesley, 1989.
- Gorton & Liu, 2002 *Software Component Quality Assessment in Practice: Sucesses and Practical Impediments*. I. Gorton, A. Liu. Proceeding of International Conference on Software Engineering (ISCE), ACM Press, 2002. pp. 555-558.
- Graube, 1989 *Metaclass Compatibility*. N. Graube. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1989. pp. 305-316.
- Halbert & O'Brien, 1987 *Using Types and Inheritance in Object-Oriented Languages*. D. Halbert, P. O'Brien. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag Berlin Heidelberg, 1987. pp. 20-31.
- Heineman & Councill, 2001 *Component-Based Software Engineering. Putting the Pieces together*. G. Heineman and W. Councill, eds. Addison-Wesley. 2001.
- Heller, 1993 *XView Programming Manual. Version 3.2*. D. Heller, T. Van Raalte. O'Reilly & Associates, 1993.
- Hewitt, 1977 *Viewing Control Structures as Patterns of Passing Messages*. C. Hewitt. Artificial Intelligence Journal, Vol. 8, 1977. pp. 323-364. Ref. en [Craig, 2000; p. 68].
- Holst & Szafron, 1997 *A General Framework for Inheritance Management and Method Dispatch in Object-Oriented Languages*. W. Holst, D. Szafron. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag Berlin Heidelberg, 1997. pp. 276-301.
- Hölzle, Chambers & Ungar, 1991 *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches*. U. Hölzle, C. Chambers, D. Ungar. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag Berlin Heidelberg, 1991. pp. 21-38.

- IBM, 1994 *SOMobjects Developer Toolkit User's Guide*. Version 2.1. IBM, 1994.
- IBM, 1999 *CICS Transaction Server for OS/390: Version 1 Release 3 Implementation Guide*. IBM Red Books, www.redbooks.ibm.com. IBM, 1999.
- Illback, 1999 *Software Reuse: Data Conversions Experiences and Issues*. J. Illback. Simposium on Software Reusability (SSP), ACM Press, 1999. pp. 10-16.
- Ingalls, 1986 *A Simple Technique for Handling Multiple Polymorphism*. D. Ingalls. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1986. pp. 347-349.
- Intel, 1999 *Intel Architecture Optimization Reference Manual*. Intel, 1999.
- ISO, 1991 *ISO 9003. Quality Management and Quality Assurance Standards (Part 3) – Guidelines to Apply ISO 9001 for Development, Supply and Maintenance of Software*. International Organization For Standardization, Genova, 1991.
- Jacobson, Griss & Johnsson, 1997 *Making the Reuse Business Work*. I. Jacobson, M. Griss, P. Jonsson. IEEE Computer, Vol. 30, Nº 10, October 1997. pp. 36-42.
- Jacobson, Booch & Rumbaugh, 1999 *The Unified Software Development Process*. Version 5.0. I. Jacobson, G. Booch, J. Rumbaugh. Addison-Wesley, 1999.
- Johnson, 1986 *Typechecking Smalltalk*. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1986. pp. 315-321.
- Jones, 1995 *Patterns of large software systems: Failure and success*. C. Jones. IEEE Computer, Vol. 28, Nº 3 March 1995. pp. 86-87.
- Joyner, 1996 *C++?? A Critique of C++ and Programming and Language Trends of the 1990s*. 3rd Edition. I. Joiner. www.progsoc.uts.edu.au/~geldridg/cpp/cppcv3, 1996.
- Kaen *et alii*, 1990 *Feature-Oriented Domain Analysis (FODA)*. Feasibility Study. K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.
- Keene, 1989 *Object-Oriented Programming in Common LISP. A Programmer's Guide to CLOS*. S. Keene. Addison-Wesley, 1989.
- Kernighan & Ritchie, 1988 *The C Programming Language*. 2nd Edition. B. Kernighan, D. Ritchie. Prentice Hall, Englewood Cliffs, NJ, 1988.
- Kiczales & Rodriguez, 1990 *Efficient Method Dispatch in PCL*. Proceedings of the ACM Conference on LISP and Functional Programming. ACM Press, 1990. pp. 99-105.
- Kiczales, Rivières & Bobrow, 1991 *The Art of the Metaobject Protocol*. G. Kiczales, J. Rivières, D. Bobrow. MIT Press, 1991.
- Kiczales, 1992 *Towards a New Model of Abstraction in the Engineering of Software*. G. Kiczales. Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures, 1992.
- Kiczales *et alii*, 1997 *Aspect-Oriented Programming*. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira, J. Loingtier, J. Irwin. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, 1997. pp. 220-242.
- Kiczales *et alii*, 2001 *Getting Started with AspectJ*. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Communications of the ACM, Vol 44, No. 10. ACM Press. October 2001, pp. 59-65.
- Kristensen, Madsen & Møller-Pedersen, 1987 *The BETA Programming Language*. B. Kristensen, O. Madsen, B. Møller-Pedersen, in Research Directions in Object-Oriented Programming. B. Shriver, P. Wegner (ed.), MIT Press, 1987.

- LaLonde, Thomas & Pugh, 1986 *An exemplar based Smalltalk.* W. LaLonde, D. Thomas and J. Pugh In Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1986, pp. 322-330.
- Lamping, 1993 *Typing the Specialization Interface.* J. Lamping. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1993. pp. 201-214.
- Leavens & Millstein, 1998 *Multiple Dispatch as Dispatch on Tuples.* G. Leavens, T. Millstein. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1998. pp. 374-387.
- Lieberman, 1981 *A Preview of ACT 1.* H. Lieberman. MIT AI Laboratory Memo No. 625, June 1981.
- Lieberman, 1986 *Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems.* H. Lieberman. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1986. pp. 214-223.
- Lindholm & Yellin, 1999 *The Java™ Virtual Machine Specification.* 2nd Edition. T. Lindholm, F. Yellin. Addison-Wesley, 1999.
- Liskov & Wing, 1994 *A Behavioral Notion of Subtyping.* B. Liskov, J. Wing. ACM Transactions of Programming Languages and Systems (TOPLAS), Vol 16. Issue 6. ACM Press, Nov. 1994. pp. 1811-1841.
- Lehrmann Madsen, Magnusson & Møller-Pedersen, 1990 *Strong Typing of Object-Oriented Languages Revisited.* O. Lehrmann Madsen, B. Magnusson, B. Møller-Pedersen. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, 1990. pp. 140-149.
- Mendhekar, Kiczales & Lamping, 1994 *Compilation Strategies as Objects.* A. Mendhekar, G. Kiczales, J. Lamping. Informal Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Workshop on Object-Oriented Compilation — What are the Objects? 1994.
- Meyer, 1997 *Object-Oriented Software Construction.* 2nd Edition. B. Meyer. Prentice Hall, 1997.
- Microsoft, 2003a *Microsoft Visual Basic .NET Documentation.* Microsoft Developer Network (MSDN) Library. msdn.microsoft.com/vbasic/. Microsoft, 2003.
- Microsoft, 2003b *Microsoft Visual C++ .NET Documentation.* Microsoft Developer Network (MSDN) Library. msdn.microsoft.com/visualc/. Microsoft, 2003.
- Microsoft, 2003c *The Component Object Model Specification.* Microsoft Developer Network (MSDN) Library. Specifications. Microsoft, 2003.
- Microsoft, 2003d *DCOM Architecture.* Microsoft Developer Network (MSDN) Library. Backgrounder. Component Object Model. Microsoft, 2003.
- Microsoft, 2003e *COM+.* Microsoft Developer Network (MSDN) Library. Platform SDK, Component Services: COM+. Microsoft, 2003.
- Microsoft, 2003f *Microsoft Transaction Server.* Microsoft Developer Network (MSDN) Library. Platform SDK, Component Services: Microsoft Transaction Server 2.0. Microsoft, 2003.
- Microsoft, 2003g *Windows API.* Microsoft Developer Network (MSDN) Library. Platform SDK Documentation. Microsoft, 2003.
- Mitchell, 2003 *Concepts in Programming Languages.* J. Mitchell. Cambridge University Press, 2003.
- Moesssenboeck & Wirth, 1991 *The Programming Language Oberon-2.* H. Moesssenboeck, N. Wirth. Structured Programming, Vol. 12. No. 4. April, 1991.

- Moon, 1986 *Object-Oriented Programming with Flavors.* D. Moon. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1986. pp. 1-8.
- Myers, Giuse & Zanden, 1992 *Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writting Methods.* B. Myers, D. Giuse, B. Zanden. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1992. pp. 184-200.
- NeXT, 1995 *Object-Oriented Programming and the Objective-C Language.* NEXTSTEP 3.3 Developer Documentation, NeXT Inc., 1995.
- Nye & O'Reilly, 1993 *X Toolkit Intrinsics Programming Manual.* A. Nye, T. O'Reilly. The Definitive Guide to the X Window System, Volume 4. O'Reilly & Associates, 1993.
- Nygaard & Dahl, 1978 *The Development of the SIMULA Languages.* K. Nygaard, O. Dahl. ACM SIGPLAN Notices, Vol. 13, No. 8. ACM. Press. August 1978. p. 245-272.
- OASIS, 2003 *Universal Description and Discovery Integration of Web Services.* Version 3.0. www.uddi.org/specification.html, OASIS, www.oasis-open.org, 2003
- OMG, 2002a *CORBA Components.* Version 3.0. Object Management Group. www.omg.org/technology/documents/corba_spec_catalog.htm, June 2002.
- OMG, 2002b *Common Object Request Broker Architecture: Core Specification.* Version 3.0. Object Management Group. www.omg.org/technology/documents/corba_spec_catalog.htm, November 2002.
- OSF, 1991 *OSF/Motif Programmer's Guide.* Release 1.1. Open Software Foundation. Prentice Hall, 1991.
- Palsberg & Schwartzbach, 1990 *Type Substitution for Object-Oriented Programming.* J. Palsberg, M. Schwartzbach. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, 1990. pp. 151-160.
- Parnas, 1972 *On the Criteria To Be Used in Decomposing Systems Into Modules.* D. Parnas. Communications of ACM. Vol 15, No 12. ACM Press, 1972. pp. 1053-1058.
- Paulk, 1995 *The Capability Maturity Model.* M. Paulk et alii Addison-Wesley, 1995.
- Pratt & Zelkowitz, 1999 *Programming Languages: Design and Implementation.* 3rd edition. Prentice Hall, 1999.
- Randell, 1979 *Software Engineering in 1968.* B. Randell. Proceedings of the Fourth International Conference on Software Engineering, IEEE Computer Society, p. 1-10.
- Rao, 1991 *Implementational Reflection in Silica.* R. Rao. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag Berlin Heidelberg, 1991. pp. 251-267.
- Rivières & Smith, 1984 *The Implementation of Procedurally Reflective Languages.* J. Rivières, B. Smith. Conference Record of the ACM Symposium on LISP and Functional Programming. ACM Press, 1984. pp. 331-347.
- Schaffert et alii, 1986 *An Introduction to Trellis/Owl.* C. Shaffert, T. Cooper, B. Bullis, M. Kilian, C. Wilpolt. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1986. pp. 9-16,
- Schoenmakers, 2000 *The TOM Tome.* P. Schoenmakers. Revision 1.23. www.gerbil.org/tom/. Eindhoven, the Netherlands, 2000.
- Smith, 1994 *Prototype-Based Languages: Object Lessons from Class-Free Programming (Panel).* R. Smith (Moderator). Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1994. pp. 102-112,

- Smith & Ungar, 1995 *Programming as an Experience: The Inspiration for SELF*. R. Smith, D. Ungar. European Conference on Object-Oriented Programming (ECOOP) Conference Proceedings. Springer-Verlag, 1995. pp. 303-330.
- Snyder, 1986a *Encapsulation and Inheritance in Object-Oriented Languages*. A. Snyder. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1986. pp. 38-45.
- Snyder, 1986b *Common Objects: An Overview*. A. Snyder. ACM Sigplan Notices. Vol. 21. No. 10. ACM Press, October 1986. pp. 19-28.
- Stata & Guttag, 1995 *Modular Reasoning in the Presence of Subclassing*. R. Stata, J. Guttag. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1995. pp. 200-214.
- Steele, 1990 *Common LISP: The language*. 2nd Edition. G. Steele. Digital Press, 1990.
- Stein, Lieberman & Ungar, 1987 *Treaty of Orlando*. L. Stein, H. Lieberman, D. Ungar. Addendum to the Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1987. pp. 43-44. Extended version: *A Shared View of Sharing: The Treaty of Orlando*. L. Stein, H. Lieberman, D. Ungar. Object-Oriented Concepts, Databases, and Applications. ACM Press, 1989. pp. 31-48.
- Stein, 1987 *Delegation Is Inheritance*. L. Stein. Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA). ACM Press, 1987. pp. 138-146.
- Stevens, Myers & Constantine, 1979 *Classics in Software Engineering*. Structured design. W. Stevens, G. Myers, L. Constantine. Yourdon Press, 1979. Ref. en [Booch, 1994; p. 137].
- Steyaert, 1994 *Open Design of Object-Oriented Languages. A Foundation for Specialisable Reflective Language Frameworks*. P. Steyaert. Ph.D. Thesis. Vrije Universiteit Brussel. 1994.
- Stroustrup, 1994 *The Design and Evolution of C++*. Addison-Wesley, 1994.
- Sun, 1989 *OPEN LOOK Graphical User Interface Functional Specification*. Sun Microsystems. Addison-Wesley, 1989.
- Sun, 1997 *JavaBeans™ API Specification*. Version 1.01. Sun Microsystems, 1997.
- Sun, 2001 *Enterprise JavaBeans™ Specification*. Version 2.0. Sun Microsystems, 2001.
- Sun, 2002 *Java™ 2 Platform, Enterprise Edition (J2EE™) version 1.3 Documentation*. java.sun.com/j2ee, Sun Microsystems, 2002.
- Swinehart, Zellweger, Beach & Hagmann, 1986 *A Structural View of the Cedar Programming Environment*. D. Swinehart, P. Zellweger, R. Beach, B. Hagmann. ACM Transactions on Programming Languages and Systems, Vol. 8, No. 4. October 1986. pp. 419-490.
- Szyperski, 1992 *Import Is Not Inheritance. Why We Need Both: Modules and Classes*. C. Szyperski. European Conference on Object-Oriented Programming (ECOOP) Conference Proceedings. Springer-Verlag, 1992. pp. 19-32.
- Szyperski, 1998 *Component Software. Beyond Object-Oriented Programming*. C. Szyperski, ACM Press - Addison-Wesley, 1998.
- Taivalsaari, 1992 *Kevo — A Prototype-based Object-Oriented Language Based on Concatenation and Module operations*. A. Taivalsaari. Technical Report DCS-197-1R, University of Victoria, BC, Canada, 1992.
- Taivalsaari, 1996 *On the Notion of Inheritance*. A. Taivalsaari. ACM Computer Surveys, Vol. 28, No. 3. ACM Press, September 1996.
- Taligent, 1994 *Taligent's Guide to Designing Programs. Well Mannered Object-Oriented Design in C++*. Addison-Wesley, 1994.
- Taligent, 1995 *Inside Taligent Technology*. S. Cotter, M. Potel. Addison-Wesley, 1995.

- Traas & Hillegersberg, 2000 *The Software Component Market On the Internet. Current Status and Conditions for Growth.* Software Engineering Notes (SIGSOFT). Vol. 25, No. 1. ACM Press, January 2000. pp. 114-117.
- Tracz, 2001 *COTS Myths and Other Lessons Learned in Component-Based Software Development.* W. Tracz. En [Heineman & Councill, 2001], pp. 99-111.
- Thorup, 1997 *Genericity in Java with Virtual Types.* K. Thorup. European Conference on Object-Oriented Programming (ECOOP) Conference Proceedings. Springer-Verlag, 1997. pp. 444-471.
- Ungar & Smith, 1991 *SELF: The Power of Simplicity.* D. Ungar, R. Smith. LISP And Symbolic Computation: An international Journal, 4, 3. Kluwer Academic Publishers, 1991.
- Vitek & Nigel Horspool, 1994 *Taming Message Passing: Efficient Method Look-up for Dynamically Typed Languages.* J. Vitek, R. Nigel Horspool. Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, 1994. pp. 432-449.
- W3C, 2000 *Extensible Markup Language (XML).* Version 1.0 (2nd Edition). www.w3.org/TR/REC-xml. World Wide Web Consortium, 2000.
- W3C, 2001a *Web Services Description Language (WSDL).* Version 1.1. www.w3.org/TR/wsdl. World Wide Web Consortium, 2001.
- W3C, 2001b *Simple Object Access Protocol (SOAP).* Version 1.1. www.w3.org/TR/soap. World Wide Web Consortium, 2001.
- W3C, 2003a *The World Wide Web Consortium (W3C).* www.w3.org. World Wide Web Consortium, 2003.
- W3C, 2003b *Simple Object Access Protocol (SOAP).* Version 1.2. www.w3.org/TR/soap12-part1/. World Wide Web Consortium, 2003.
- Wallnau, Hissam & Seacord, 2002 *Building Systems from Commercial Components.* K. Wallnau, S. Hissam, R. Seacord. Addison-Wesley, 2002.
- Wang & King, 2000 *Software Engineering Process. Principles and Applications.* Y. Wang and G. King. CRC Press, 2000.
- Wang, Schmidt & O'Ryan, 2001 *Overview of the CORBA Component Model.* N. Wang, D. Schmidt, C. O'Ryan. En [Heineman & Councill, 2001], pp. 557-571.
- Weinreich & Sametinger, 2001 *Component Models and Component Services: Concepts and Principles.* R. Weinreich, J. Sametinger. En [Heineman & Councill, 2001], pp. 33-48.
- Wirth, 1982 *Programming in Modula-2.* N. Wirth. Springer-Verlag, Berlin, 1982.
- Wirth, 1988 *The Programming Language Oberon.* N. Wirth. Software Practice and Experience, Vol. 18, No. 7, July 1988.
- Yakote, 1992 *The Apertos Reflective Operating System: The Concept and its Implementation.* Y. Yakote. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA). ACM Press, 1992. pp. 414-434.
- Zendra, Colnet & Collin, 1997 *Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler.* O. Zendra, D. Colnet, S. Collin. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA). ACM Press, 1997. pp. 125-141.