

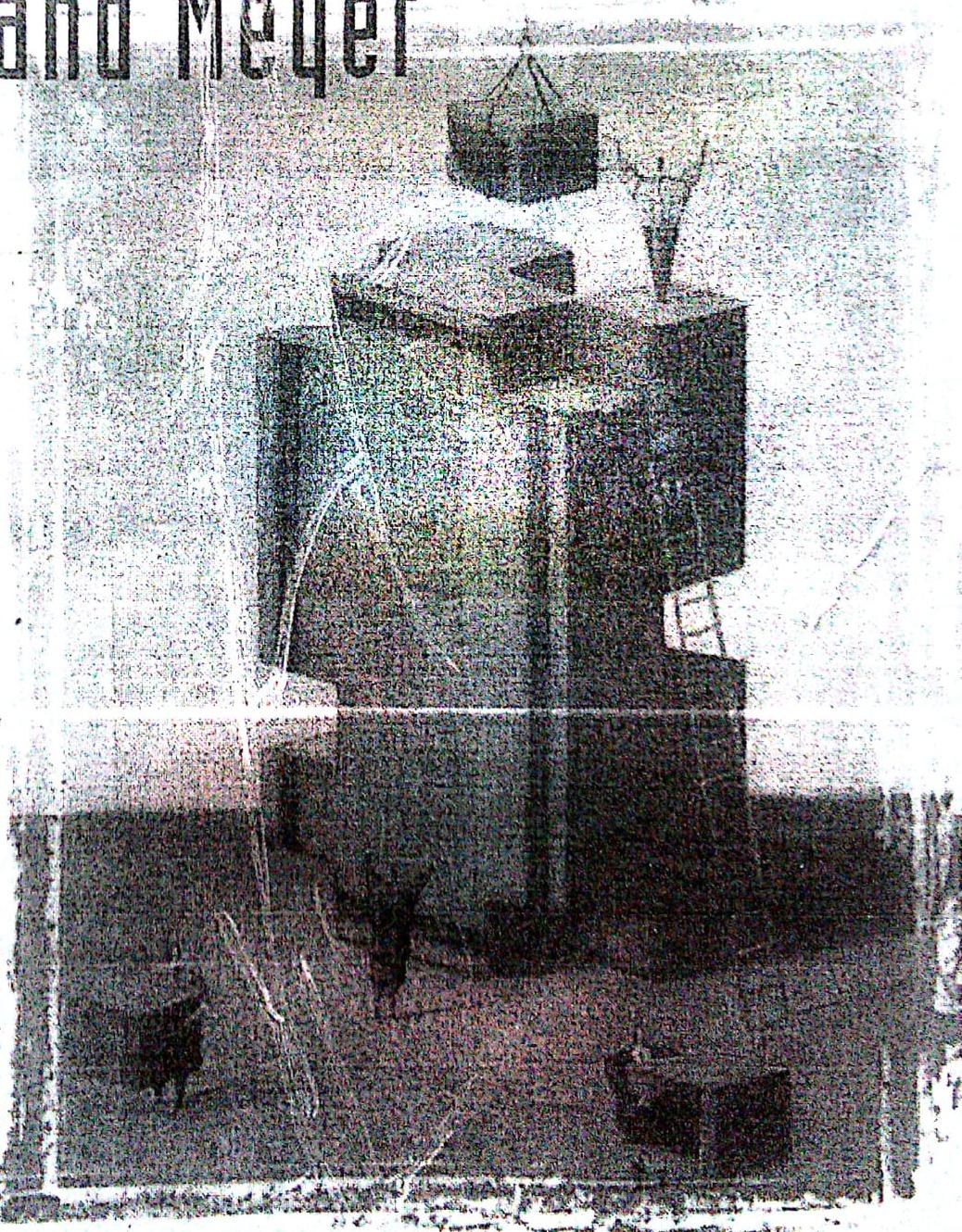
Construcción de Software Orientado a Objetos

Bertrand Meyer

El libro
de consulta
más completo
y definitivo
sobre O-O

Un hito en la O-O
creado por uno
de sus pioneros

El CD-ROM incluye
la edición inglesa
del libro
en hipertexto
y un entorno
completo
de desarrollo
orientado
a objetos



Segunda edición

CONSTRUCCIÓN DE SOFTWARE ORIENTADO A OBJETOS

CONSTRUCCIÓN DE SOFTWARE ORIENTADO A OBJETOS

Segunda edición

Bertrand MEYER

ISE Inc.

Santa Barbara (California)

Traducción:

Miguel Katrib Mora

Universidad de La Habana

Rafael García Bermejo

Universidad de Salamanca

Salvador Sánchez

Universidad Pontificia de Salamanca, campus de Madrid



también ha colaborado:

Juan Manuel Cueva Lovelle

Universidad de Oviedo

Revisión técnica:

Jesús García Molina

Universidad de Murcia

Carmelo R. Fernández Rupérez

European Software Institute

Coordinación de la traducción y revisión técnica:

Luis Joyanes Aguilar

Universidad Pontificia de Salamanca, campus de Madrid



PRENTICE HALL

Madrid • Upper Saddle River • Londres • México • Nueva Delhi • Río de Janeiro • Santafé de Bogotá •
Singapur • Sydney • Tokio • Toronto

Datos de catalogación bibliográfica

Bertrand MEYER
CONSTRUCCIÓN DE SOFTWARE ORIENTADO A OBJETOS.
Segunda edición
PRENTICE HALL, Madrid, 1999

ISBN: 84-8322-040-7
Materia: Informática. 681.3

Formato 195 x 265

Páginas: 1.248

Bertrand MEYER
CONSTRUCCIÓN DE SOFTWARE ORIENTADO A OBJETOS.
Segunda edición

No está permitida la reproducción total o parcial de esta obra
ni su tratamiento o transmisión por cualquier medio o método
sin autorización escrita de la Editorial.



DERECHOS RESERVADOS
© 1999 respecto a la primera edición en español por:
PRENTICE HALL Iberia, S.R.L.
Téllez, 54
28007 Madrid
Simon & Schuster International Group

ISBN: 84-8322-040-7

Depósito Legal: M-34.367-1998

Traducido de:
OBJECT-ORIENTED SOFTWARE CONSTRUCTION, second edition
Prentice Hall PTR
A Simon & Schuster Company
© MCMXCVII
ISBN: 0-13-629155-4

Edición en español:

Editor: Andrés Otero

Diseño de cubierta: DIGRAF

Composición: COPYBOOK, S. L.

Impreso por: COFÁS

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos

CONTENIDO BREVE

(La tabla completa de contenidos comienza en la página vii)

Contenido	vii	21. Herencia, un caso de estudio: "undo" en un sistema interactivo	659
Prólogo	xix	22. Cómo encontrar las clases	681
Prólogo a la segunda edición	xxv	23. Principios de diseño de clases	707
Sobre bibliografía, las fuentes de Internet y los ejercicios	xxvii	24. Utilizando bien la herencia	765
Prólogo a la edición española	xxix	25. Técnicas útiles	823
Intentamos mejorar este libro	xxxvii	26. Un sentido del estilo	827
Parte A: Los problemas	1	27. Análisis orientado a objetos	855
1. Calidad del software	3	28. El proceso de construcción del software	875
2. Criterios de la orientación a objetos	21	29. Enseñanza del método	887
Parte B: El camino a la orientación a objetos	37	Parte E: Temas avanzados	901
3. Modularidad	39	30. Una concurrencia, distribución, cliente-servidor e Internet	903
4. Enfoques para la reutilización	65	31. Persistencia de objetos y bases de datos	983
5. Hacia la tecnología de Objetos	97	32. Técnicas O-O para aplicaciones gráficas interactivas	1009
6. Tipos abstractos de datos	115		
Parte C: Técnicas orientadas a objetos	155	Parte F: Aplicación del método a diferentes lenguajes y entornos	1023
7. La estructura estática: las clases	157	33. Ada y la programación orientada a objetos	1025
8. La estructura de ejecución: los objetos	205	34. Emulación de la tecnología de objetos en entornos no O-O	1045
9. Gestión de la memoria	263	35. De Simula a Java y más allá: los principales entornos y lenguajes O-O	1059
10. Genericidad	299		
11. Diseño por Contrato: construcción de software fiable	313	Parte G: Hacerlo bien	1085
12. Cuando se rompe el contrato: tratamiento de excepciones	387	36. Un entorno orientado a objetos	1087
13. Mecanismos de soporte	413	Epflogo, exponiendo con total franqueza el lenguaje	1105
14. Introducción a la herencia	433		
15. Herencia múltiple	489	Parte H: Apéndices	1107
16. Técnicas de herencia	537	Apéndice A. Extractos de las bibliotecas base	1109
17. Comprobación estricta de tipos	577	Apéndice B. Genericidad frente a herencia	1111
18. Objetos globales y constantes	609	Apéndice C. Principios, reglas, preceptos y definiciones	1133
Parte D: Metodología orientada a objetos: aplicar bien el método	627	Apéndice D. Glosario de tecnología de objetos	1137
19. Sobre la metodología	629	Apéndice E. Bibliografía	1165
20. Patrón de diseño: Sistemas interactivos Multi-Panel	641	Índice	1171

Dirección del autor:

Bertrand Meyer
Interactive Software Engineering Inc. (ISE)
270 Storke Road, Suite 7
Santa Barbara, CA 93117
USA
805-685-1006, fax 805-685-6869
<meyer@tools.com>, http://www.tools.com

Contenido

Contenido	vii
Prólogo	xix
Prólogo a la 2. ^a edición	xxv
Sobre Bibliografía, las fuentes de Internet y los ejercicios	xxvii
Prólogo a la edición española	xxix
Intentamos mejorar este libro	xxxvii
PARTA A: LOS PROBLEMAS	1
Capítulo 1. Calidad del software	3
1.1. FACTORES EXTERNOS E INTERNOS	1
1.2. UNA REVISIÓN DE LOS FACTORES EXTERNOS	4
1.3. SOBRE EL MANTENIMIENTO DEL SOFTWARE	16
1.4. CONCEPTOS CLAVE INTRODUCIDOS EN ESTE CAPÍTULO	18
1.5. NOTAS BIBLIOGRÁFICAS	19
Capítulo 2. Criterios de la orientación a objetos	21
2.1. SOBRE LOS CRITERIOS	21
2.2. MÉTODO Y LENGUAJE	22
2.3. IMPLEMENTACIÓN Y ENTORNO	31
2.4. BIBLIOTECAS	33
2.5. PARA UNA VISIÓN PREVIA MÁS EXTENSA	34
2.6. NOTAS BIBLIOGRÁFICAS Y RECURSOS DE OBJETOS	34
PARTA B: EL CAMINO A LA ORIENTACIÓN A OBJETOS	37
Capítulo 3. Modularidad	39
3.1. CINCO CRITERIOS	40
3.2. CINCO REGLAS	46
3.3. CINCO PRINCIPIOS	51
3.4. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	61
3.5. NOTAS BIBLIOGRÁFICAS	61
EJERCICIOS	62

Capítulo 4. Enfoques para la reutilización	65
4.1. LAS METAS DE LA REUTILIZACIÓN	65
4.2. ¿QUÉ ES LO QUE SE DEBERÍA REUTILIZAR?	68
4.3. LA REPETICIÓN EN EL DESARROLLO DE SOFTWARE	71
4.4. OBSTÁCULOS NO TÉCNICOS	72
4.5. EL PROBLEMA TÉCNICO	78
4.6. CINCO REQUISITOS RELATIVOS A LAS ESTRUCTURAS DE LOS MÓDULOS	80
4.7. ESTRUCTURAS MODULARES TRADICIONALES	85
4.8. SOBRECARGA Y GENERICIDAD	89
4.9. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	94
4.10. NOTAS BIBLIOGRÁFICAS	95
Capítulo 5. Hacia la tecnología de Objetos	97
5.1. LOS INGREDIENTES DE LA COMPUTACIÓN	97
5.2. DESCOMPOSICIÓN FUNCIONAL	99
5.3. DESCOMPOSICIÓN BASADA EN OBJETOS	109
5.4. CONSTRUCCIÓN DE SOFTWARE ORIENTADO A OBJETOS	110
5.5. CUESTIONES	111
5.6. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	113
5.7. NOTAS BIBLIOGRÁFICAS	113
Capítulo 6. Tipos abstractos de datos	115
6.1. CRITERIOS	116
6.2. VARIACIONES DE IMPLEMENTACIÓN	116
6.3. HACIA UNA VISIÓN ABSTRACTA DE LOS OBJETOS	119
6.4. FORMALIZAR LA ESPECIFICACIÓN	123
6.5. DE LOS TIPOS ABSTRACTOS DE DATOS A LAS CLASES	134
6.6. MÁS ALLÁ DEL SOFTWARE	140
6.7. TEMAS SUPLEMENTARIOS	140
6.8. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	150
6.9. NOTAS BIBLIOGRÁFICAS	151
EJERCICIOS	152
PARTE C: TÉCNICAS ORIENTADAS A OBJETOS	155
Capítulo 7. La estructura estática: las clases	157
7.1. LOS OBJETOS NO SON EL SUJETO	157
7.2. EVITAR LA CONFUSIÓN ESTÁNDAR	158
7.3. EL PAPEL (ROL) DE LAS CLASES	161
7.4. UN SISTEMA DE TIPOS UNIFORME	162
7.5. UNA CLASE SENCILLA	163
7.6. CONVENCIONES BÁSICAS	168
7.7. EL ESTILO ORIENTADO A OBJETOS	171
7.8. EXPORTACIÓN SELECTIVA Y OCULTACIÓN DE INFORMACIÓN	181
7.9. UNIÉNDODO TODO	183

7.10. DISCUSIÓN	192
7.11. CONCEPTOS CLAVE INTRODUCIDOS EN ESTE CAPÍTULO	202
6.12. NOTAS BIBLIOGRÁFICAS	203
EJERCICIOS	204
Capítulo 8. La estructura de ejecución: los objetos	205
8.1. OBJETOS	205
8.2. LOS OBJETOS COMO HERRAMIENTA DE MODELADO	216
8.3. MANIPULACIÓN DE OBJETOS Y REFERENCIAS	218
8.4. PROCEDIMIENTOS DE CREACIÓN	222
8.5. MÁS SOBRE LAS REFERENCIAS	226
8.6. OPERACIONES SOBRE REFERENCIAS	228
8.7. OBJETOS COMPUESTOS Y TIPOS EXPANDIDOS	239
8.8. CONEXIÓN: SEMÁNTICA POR REFERENCIA Y POR VALOR	246
8.9. BENEFICIOS Y PELIGROS DE TRABAJAR CON REFERENCIAS	250
8.10. DISCUSIÓN	255
8.11. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	260
6.12. NOTAS BIBLIOGRÁFICAS	261
EJERCICIOS	261
Capítulo 9. Gestión de la memoria	263
9.1. QUÉ LES PASA A LOS OBJETOS	263
9.2. EL ENFOQUE DESPREOCUPADO	274
9.3. RECUPERACIÓN DE MEMORIA: LAS CUESTIONES	276
9.4. LIBERACIÓN DE MEMORIA CONTROLADA POR EL PROGRAMADOR	277
9.5. EL ENFOQUE DEL NIVEL DE COMPONENTES	279
9.6. GESTIÓN AUTOMÁTICA DE MEMORIA	283
9.7. RECUENTO DE REFERENCIAS	284
9.8. RECOLECCIÓN DE BASURA	286
9.9. ASPECTOS PRÁCTICOS DE LA RECOLECCIÓN DE BASURA	291
9.10. UN ENTORNO CON GESTIÓN DE MEMORIA	293
9.11. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	296
9.12. NOTAS BIBLIOGRÁFICAS	297
EJERCICIOS	297
Capítulo 10. Genericidad	299
10.1. GENERALIZACIÓN DE TIPOS HORIZONTAL Y VERTICAL	299
10.2. NECESIDAD DE LA PARAMETRIZACIÓN DE TIPOS	300
10.3. CLASES GENÉRICAS	302
10.4. ARRAYS	307
10.5. EL COSTE DE LA GENERICIDAD	310
10.6. DISCUSIÓN: NO SE HARÁ TODAVÍA	310
10.7. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	311
10.8. NOTAS BIBLIOGRÁFICAS	311
EJERCICIOS	311

Capítulo 11. Diseño por Contrato: construcción de software fiable	313
11.1. MECANISMOS BÁSICOS DE FIABILIDAD	314
11.2. SOBRE LA CORRECCIÓN DEL SOFTWARE	314
11.3. EXPRESAR UNA ESPECIFICACIÓN	316
11.4. INTRODUCIR ASERCIÓNES EN LOS TEXTOS DE SOFTWARE	319
11.5. PRECONDICIONES Y POSTCONDICIONES	319
11.6. CONTRATACIÓN PARA LA FIABILIDAD DEL SOFTWARE	323
11.7. TRABAJO CON ASERCIÓNES	329
11.8. INVARIANTES DE CLASE	343
11.9. ¿CUÁNDO ES CORRECTA UNA CLASE?	349
11.10. LA CONEXIÓN TAD	352
11.11. UNA INSTRUCCIÓN DE ASERCIÓN	357
11.12. INVARIANTES Y VARIANTES DE BUCLE	359
11.13. UTILIZACIÓN DE ASERCIÓNES	366
11.14. DISCUSIÓN	375
11.15. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	383
11.16. NOTAS BIBLIOGRÁFICAS	383
EJERCICIOS	384
APÉNDICE: EL DESASTRE DEL ARIANE 5	386
Capítulo 12. Cuando se rompe el contrato: tratamiento de excepciones	387
12.1. CONCEPTOS BÁSICOS DEL TRATAMIENTO DE EXCEPCIONES	387
12.2. TRATAMIENTO DE EXCEPCIONES	390
12.3. UN MECANISMO DE EXCEPCIONES	394
12.4. EJEMPLOS DE TRATAMIENTO DE EXCEPCIONES	397
12.5. LA TAREA DE UNA CLÁUSULA DE RESCATE	402
12.6. TRATAMIENTO AVANZADO DE EXCEPCIONES	405
12.7. DISCUSIÓN	410
12.8. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	411
12.9. NOTAS BIBLIOGRÁFICAS	412
EJERCICIOS	412
Capítulo 13. Mecanismos de soporte	413
13.1. INTERFAZ CON EL SOFTWARE NO O-O	413
13.2. PASO DE ARGUMENTOS	418
13.3. INSTRUCCIONES	420
13.4. EXPRESIONES	425
13.5. CADENAS DE CARACTERES	429
13.6. ENTRADA Y SALIDA	430
13.7. CONVENCIONES LEXICOGRÁFICAS	430
13.8. CONCEPTOS CLAVE INTRODUCIDOS EN ESTE CAPÍTULO	431
EJERCICIOS	431
Capítulo 14. Introducción a la herencia	433
14.1. POLÍGONOS Y RECTÁNGULOS	433
14.2. POLIMORFISMO	441

14.3. TIPOS Y HERENCIA	445
14.4. LIGADURA DINÁMICA	452
14.5. CLASES Y CARACTERÍSTICAS DIFERIDAS	454
14.6. TÉCNICAS DE REDECLARACIÓN	463
14.7. EL SIGNIFICADO DE LA HERENCIA	466
14.8. EL PAPEL DE LAS CLASES DIFERIDAS	471
14.9. DISCUSIÓN	478
14.10. CONCEPTOS CLAVE INTRODUCIDOS EN ESTE CAPÍTULO	486
14.11. NOTAS BIBLIOGRÁFICAS	487
EJERCICIOS	487
 Capítulo 15. Herencia múltiple	 489
15.1. EJEMPLOS DE HERENCIA MÚLTIPLE	489
15.2. RENOMBRAR CARACTERÍSTICAS	504
15.3. APLANAR LA ESTRUCTURA	510
15.4. HERENCIA REPETIDA	512
15.5. DISCUSIÓN	531
15.6. CONCEPTOS CLAVE INTRODUCIDOS EN ESTE CAPÍTULO	533
15.7. NOTAS BIBLIOGRÁFICAS	534
EJERCICIOS	534
 Capítulo 16. Técnicas de herencia	 537
16.1. HERENCIA Y ASERCIIONES	537
16.2. LA ESTRUCTURA GLOBAL DE LA HERENCIA	547
16.3. CARACTERÍSTICAS CONGELADAS	550
16.4. GÉNERICIDAD RESTRINGIDA	552
16.5. INTENTO DE ASIGNACIÓN	557
16.6. TIPOS Y REDECLARACIÓN	561
16.7. DECLARACIÓN CON ANCLA	564
16.8. HERENCIA Y OCULTACIÓN DE INFORMACIÓN	570
16.9. CONCEPTOS CLAVE INTRODUCIDOS EN ESTE CAPÍTULO	575
16.10. NOTAS BIBLIOGRÁFICAS	575
EJERCICIOS	576
 Capítulo 17. Comprobación estricta de tipos	 577
17.1. EL PROBLEMA DE LA COMPROBACIÓN ESTRICTA DE TIPOS	577
17.2. COMPROBACIÓN ESTÁTICA DE TIPOS: CÓMO Y POR QUÉ	581
17.3. COVARIANZA Y OCULTACIÓN DE INFORMACIÓN EN EL DESCENDIENTE	587
17.4. PRIMERAS APROXIMACIONES A LA VALIDACIÓN DE SISTEMAS	594
17.5. TOMANDO COMO BASE LOS TIPOS CON ANCLA	595
17.6. ANÁLISIS GLOBAL	599
17.7. ¡CUIDADO CON LOS CATCALL POLIMORFOS!	601
17.8. VALORACIÓN	604
17.9. EL AJUSTE PERFECTO	605
17.10. CONCEPTOS CLAVE ESTUDIADOS EN ESTE CAPÍTULO	606
17.11. NOTAS BIBLIOGRÁFICAS	606

Capítulo 18. Objetos globales y constantes	609
18.1. CONSTANTES DE LOS TIPOS BÁSICOS	609
18.2. USO DE LAS CONSTANTES	611
18.3. CONSTANTES DE TIPO CLASE	612
18.4. APLICACIONES DE LAS RUTINAS <i>ONCE</i>	614
18.5. CONSTANTES DE TIPO STRING	619
18.6. VALORES UNIQUE	619
18.7. DISCUSIÓN	621
18.8. CONCEPTOS CLAVE INTRODUCIDOS EN ESTE CAPÍTULO	624
18.9. NOTAS BIBLIOGRÁFICAS	625
EJERCICIOS	625
PARTE D: METODOLOGÍA ORIENTADA A OBJETOS: APLICAR BIEN EL MÉTODO	627
Capítulo 19. Sobre la metodología	629
19.1. METODOLOGÍA DEL SOFTWARE: QUÉ Y POR QUÉ	629
19.2. CONSTRUCCIÓN DE BUENAS REGLAS: ASESORAR A LOS ASESORES	630
19.3. SOBRE LA UTILIZACIÓN DE METÁFORAS	637
19.4. LA IMPORTANCIA DE SER HUMILDE	639
19.5. NOTAS BIBLIOGRÁFICAS	640
EJERCICIOS	640
Capítulo 20. Patrón de diseño: Sistemas interactivos Multi-Panel	641
20.1. SISTEMAS MUTIL-PANEL	641
20.2. UN INTENTO INTUITIVO	643
20.3. UNA SOLUCIÓN FUNCIONAL DESCENDENTE	644
20.4. UNA CRÍTICA DE LA SOLUCIÓN	647
20.5. UNA ARQUITECTURA ORIENTADA A OBJETOS	649
20.6. DISCUSIÓN	657
20.7. NOTAS BIBLIOGRÁFICAS	658
Capítulo 21. Herencia, un caso de estudio: «undo» en un sistema interactivo	659
21.1. DIABOLICUM PERSEVERARE	659
21.2. ENCONTRAR LAS ABSTRACCIONES	662
21.3. DESHACER-REHACER DE MÚLTIPLES NIVELES	667
21.4. ASPECTOS DE IMPLEMENTACIÓN	670
21.5. UNA INTERFAZ DE USUARIO PARA DESHACER Y REHACER	674
21.6. DISCUSIÓN	675
21.7. NOTAS BIBLIOGRÁFICAS	677
EJERCICIOS	678
Capítulo 22. Cómo encontrar las clases	681
22.1. ESTUDIO DE UN DOCUMENTO DE REQUISITOS	681
22.2. SEÑALES DE PELIGRO	687

22.3. HEURÍSTICAS GENERALES PARA ENCONTRAR LAS CLASES	692
22.4. OTRAS FUENTES DE CLASES	696
22.5. REUTILIZACIÓN	701
22.6. EL MÉTODO PARA OBTENER LAS CLASES	702
22.7. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	703
22.8. NOTAS BIBLIOGRÁFICAS	704
EJERCICIOS	704
 Capítulo 23. Principios de diseño de clases	707
23.1. EFECTOS LATERALES DE LAS FUNCIONES	708
23.2. ¿CUÁNTOS ARGUMENTOS DEBE TENER UNA CARACTERÍSTICA?	723
23.3. TAMAÑO DE CLASE: EL ENFOQUE DE LISTA DE LA COMPRA	729
23.4. ESTRUCTURAS ACTIVAS DE DATOS	733
23.5. EXPORTACIONES SELECTIVAS	753
23.6. ABORDANDO LOS CASOS EXCEPCIONALES	754
23.7. EVOLUCIÓN DE CLASES: LA CLÁUSULA OBSOLETE	758
23.8. DOCUMENTACIÓN DE UNA CLASE Y DE UN SISTEMA	760
23.9. CONCEPTOS PRESENTADOS EN ESTE CAPÍTULO	762
23.10. NOTAS BIBLIOGRÁFICAS	762
EJERCICIOS	763
 Capítulo 24. Utilizando bien la herencia	765
24.1. CÓMO NO HAY QUE UTILIZAR LA HERENCIA	765
24.2. ¿PREFIERE USTED COMPRAR O HEREDAR?	768
24.3. UNA APLICACIÓN: LA TÉCNICA HANDLE	773
24.4. TAXONOMÍA	775
24.5. UTILIZACIÓN DE LA HERENCIA: UNA TAXONOMÍA DE LA TAXONOMÍA	777
24.6. ¿UN MECANISMO O MÁS?	788
24.7. HERENCIA DE SUBTIPOS Y OCULTACIÓN EN DESCENDIENTES	790
24.8. HERENCIA DE IMPLEMENTACIÓN	797
24.9. HERENCIA DE FACILIDADES	800
24.10. CRITERIOS MÚLTIPLES Y HERENCIA DE VISTAS	804
24.11. CÓMO DESARROLLAR ESTRUCTURAS DE HERENCIA	810
24.12. VISIÓN RESUMIDA: UTILIZANDO BIEN LA HERENCIA	814
24.13. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	815
24.14. NOTAS BIBLIOGRÁFICAS	815
24.15. APÉNDICE: UNA HISTORIA DE LA TAXONOMÍA	816
EJERCICIOS	821
 Capítulo 25. Técnicas útiles	823
25.1. FILOSOFÍA DE DISEÑO	823
25.2. CLASES	824
25.3. TÉCNICAS DE HERENCIA	825

Capítulo 26. Un sentido del estilo	827
26.1. ¡ASUNTOS COSMÉTICOS!	827
26.2. SELECCIÓN DE NOMBRES CORRECTOS	830
26.3. UTILIZACIÓN DE CONSTANTES	836
26.4. COMENTARIOS DE ENCABEZADO Y CLÁUSULAS DE INDEXACIÓN	837
26.5. DISPOSICIÓN Y PRESENTACIÓN DEL CÓDIGO	843
26.6. TIPOS DE LETRA	851
26.7. NOTAS BIBLIOGRÁFICAS	852
EJERCICIOS	853
Capítulo 27. Análisis orientado a objetos	855
27.1. OBJETIVOS DEL ANÁLISIS	855
27.2. LA NATURALEZA CAMBIANTE DEL ANÁLISIS	858
27.3. LA CONTRIBUCIÓN DE LA TECNOLOGÍA DE OBJETOS	858
27.4. PROGRAMACIÓN DE UNA EMISORA DE TELEVISIÓN	859
27.5. EXPRESIÓN DEL ANÁLISIS: PUNTOS DE VISTA MÚLTIPLES	865
27.6. MÉTODOS DE ANÁLISIS	869
27.7. LA NOTACIÓN DE OBJETOS EN NEGOCIOS	870
27.8. BIBLIOGRAFÍA	873
Capítulo 28. El proceso de construcción del software	875
28.1. CLUSTERS	875
28.2. INGENIERÍA CONCURRENTE	876
28.3. PASOS Y TAREAS	877
28.4. EL MODELO DE CLUSTERS DEL CICLO DE VIDA DEL SOFTWARE	878
28.5. GENERALIZACIÓN	880
28.6. AUSENCIA DE DISCONTINUIDADES Y REVERSIBILIDAD	882
28.7. CON NOSOTROS, TODO ES LA CARA	884
28.8. CONCEPTOS CLAVE ABARCADOS EN ESTE CAPÍTULO	885
28.9. NOTAS BIBLIOGRÁFICAS	885
Capítulo 29. Enseñanza del método	887
29.1. FORMACIÓN INDUSTRIAL	887
29.2. CURSOS INTRODUCTORIOS	889
29.3. OTROS CURSOS	892
29.4. HACIA UNA NUEVA PEDAGOGÍA DEL SOFTWARE	894
29.5. UN PLAN ORIENTADO A OBJETOS	897
29.6. CONCEPTOS CLAVE ESTUDIADOS EN ESTE CAPÍTULO	899
29.7. NOTAS BIBLIOGRÁFICAS	899
PARTE E: TEMAS AVANZADOS	901
Capítulo 30. Concurrencia, distribución, cliente-servidor e Internet	903
30.1. UNA PRESENTACIÓN PREVIA ALGO CHIVATA	903
30.2. EL CRECIMIENTO DE LA IMPORTANCIA DE LA CONCIENCIA	903

30.3. DE PROCESOS A OBJETOS	908
30.4. PRESENTACIÓN DE LA EJECUCIÓN CONCURRENTE	915
30.5. PROBLEMAS DE SINCRONIZACIÓN	927
30.6. ACCESO A OBJETOS SEPARADOS	932
30.7. CONDICIONES DE ESPERA	941
30.8. SOLICITUD DE SERVICIOS ESPECIALES	948
30.9. EJEMPLOS	952
30.10. HACIA UNA REGLA DE DEMOSTRACIÓN	968
30.11. UN RESUMEN DEL MECANISMO	970
30.12. DISCUSIÓN	973
30.13. CONCEPTOS CLAVE PRESENTADOS EN ESTE CAPÍTULO	976
30.14. NOTAS BIBLIOGRÁFICAS	977
EJERCICIOS	979
Capítulo 31. Persistencia de objetos y bases de datos	983
31.1. PERSISTENCIA A PARTIR DEL LENGUAJE	983
31.2. MÁS ALLÁ DEL CIERRE DE PERSISTENCIA	985
31.3. EVOLUCIÓN DE ESQUEMAS	986
31.4. DE LA PERSISTENCIA A LAS BASES DE DATOS	992
31.5. INTEROPERABILIDAD OBJETO-RELACIONAL	993
31.6. FUNDAMENTOS DE LAS BASES DE DATOS ORIENTADAS A OBJETOS	996
31.7. SISTEMAS DE BASES DE DATOS O-O: EJEMPLOS	1001
31.8. DISCUSIÓN: MÁS ALLÁ DE LAS BASES DE DATOS O-O	1003
31.9. CONCEPTOS CLAVE ESTUDIADOS EN ESTE CAPÍTULO	1006
31.10. NOTAS BIBLIOGRÁFICAS	1006
EJERCICIOS	1007
Capítulo 32. Técnicas O-O para aplicaciones gráficas interactivas	1009
32.1. HERRAMIENTAS NECESARIAS	1010
32.2. PORTABILIDAD Y ADAPTACIÓN A UNA PLATAFORMA	1012
32.3. ABSTRACCIONES GRÁFICAS	1014
32.4. MECANISMOS DE INTERACCIÓN	1017
32.5. GESTIONANDO LOS EVENTOS	1018
32.6. UN MODELO MATEMÁTICO	1022
32.7. NOTAS BIBLIOGRÁFICAS	1022
PARTE F: APLICACIÓN DEL MÉTODO A DIFERENTES LENGUAJES Y ENTORNOS	1023
Capítulo 33. Ada y la programación orientada a objetos	1025
33.1. UN POCO DE CONTEXTO	1025
33.2. PAQUETES	1027
33.3. IMPLEMENTACIÓN DE UNA PILA	1027
33.4. OCULTACIÓN DE LA REPRESENTACIÓN: LA HISTORIA DE LA CLÁUSULA PRIVATE	1031
33.5. EXCEPCIONES	1033
33.6. TAREAS	1036

33.7. DE ADA A ADA 95	103
33.8. CONCEPTOS CLAVE DEL CAPÍTULO	104
33.9. NOTAS BIBLIOGRÁFICAS	104
EJERCICIOS	104
Capítulo 34. Emulación de la tecnología de objetos en entornos no O-O	104
34.1. NIVELES DE SOPORTE DE LOS LENGUAJES	1045
34.2. ¿PROGRAMACIÓN ORIENTADA A OBJETOS EN PASCAL?	1045
34.3. FORTRAN	1046
34.4. C Y LA PROGRAMACIÓN ORIENTADA A OBJETOS	1050
34.5. NOTAS BIBLIOGRÁFICAS	1057
EJERCICIOS	1057
Capítulo 35. De Simula a Java y más allá: Los principales entornos y lenguajes O-O	1059
35.1. SIMULA	1059
35.2. SMALLTALK	1071
35.3. EXTENSIONES DE LISP	1075
35.4. EXTENSIONES DE C	1075
35.5. JAVA	1075
35.6. OTROS LENGUAJES ORIENTADOS A OBJETOS	1081
35.7. NOTAS BIBLIOGRÁFICAS	1081
EJERCICIOS	1083
PARTE G: HACERLO BIEN	1085
Capítulo 36. Un entorno orientado a objetos	1085
36.1. COMPONENTES	1087
36.2. EL LENGUAJE	1088
36.3. LA TECNOLOGÍA DE COMPILACIÓN	1088
36.4. HERRAMIENTAS	1089
36.5. BIBLIOTECAS	1091
36.6. MECANISMOS DE INTERFAZ	1094
36.7. NOTAS BIBLIOGRÁFICAS	1096
Epílogo, exponiendo con total franqueza el lenguaje	1103
PARTE H: APÉNDICES	1101
Apéndice A. Extractos de las bibliotecas base	1109
Apéndice B. Genericidad frente a herencia	1111
B.1. GENERALIDAD	1111
B.2. HERENCIA	1116
B.3. EMULACIÓN DE LA HERENCIA MEDIANTE GENERICIDAD	1118
B.4. EMULACIÓN DE LA GENERICIDAD MEDIANTE HERENCIA	1119

B.5. COMBINACIÓN DE LA GENERICIDAD Y DE LA HERENCIA	1126
B.6. CONCEPTOS CLAVE PRESENTADOS EN ESTE APÉNDICE	1130
B.7. NOTAS BIBLIOGRÁFICAS	1130
EJERCICIOS	1130
Apéndice C. Principios, reglas, preceptos y definiciones	1133
Apéndice D. Glosario de tecnología de objetos	1137
Apéndice E. Bibliografía	1149
E.1. TRABAJOS DE OTROS AUTORES	1149
E.2. TRABAJOS REALIZADOS POR EL AUTOR DEL PRESENTE LIBRO	1165
Índice	1171

Prólogo

Bien nacido en las azules aguas de la agreste costa noruega; potenciado (por una aberración de las corrientes oceánicas para la cual los oceanógrafos no han hallado aún una explicación convincente) a lo largo de la menos accidentada costa del Pacífico californiano; y visto por algunos como un tifón, por otros como un maremoto, y por otros como una tempestad en un vaso de agua: una ola inmensa asola las costas del mundo de la computación y de la informática.

“Orientado a objetos” es el término más de moda, complementando y en muchos casos sustituyendo a “estructurado” como versión de alta tecnología del término “bueno”. Inevitablemente, como ocurre siempre en estos casos, el término lo emplean distintas personas con distintos significados; igualmente inevitable es la secuencia de reacciones en tres pasos a la que se enfrenta la presentación de cualquier nuevo principio metodológico: (1) “Es una trivialidad”, (2) “No va a funcionar”; (3) “Pero si eso ya lo hacía yo, hombre”. (El orden puede variar.)

Aclaremos esto de inmediato, no sea que el lector piense que el autor se enfrenta al tema a la ligera: no veo que el método orientado a objetos sea una mera moda; creo que no es trivial (aunque me esforzaré por dejarlo tan claro como sea posible); sé que funciona; y creo que no sólo es distinto sino, hasta cierto punto, incompatible con las tecnologías que emplea casi todo el mundo en la actualidad —incluyendo algunos de los principios que se enseñan en muchos textos de ingeniería del software. Además creo que la tecnología de objetos contiene el potencial de unos cambios fundamentales en la industria del software, y que durará mucho tiempo. Por último, espero que el lector, a medida que vaya avanzando por estas páginas, comparta mi emoción respecto a esta prometedora ruta del análisis, diseño e implementación del software.

“Ruta del análisis, diseño e implementación del software”. Para presentar el método orientado a objetos, este libro adopta decididamente el punto de vista de la ingeniería de software —de los métodos, herramientas y técnicas adecuados para desarrollar software de calidad en entornos de producción. Ésta no es la única perspectiva posible, por cuanto también ha habido interés por aplicar los principios orientados a objetos a temas como la programación exploratoria y la inteligencia artificial. Aun cuando la presentación no excluye estas aplicaciones, no son su objetivo principal. Nuestra meta principal en esta discusión es estudiar la forma en que los desarrolladores de software, en entornos tanto industriales como académicos, pueden utilizar la tecnología de objetos para mejorar (drásticamente, en ciertas ocasiones) la calidad del software que producen.

Estructura, fiabilidad, epistemología y clasificación

La tecnología de objetos, en lo fundamental, es la combinación de cuatro ideas: un método de estructuración, una disciplina de fiabilidad, un principio epistemológico y una técnica de clasificación.

El *método de estructuración* se aplica a la descomposición y reutilización del software. Los sistemas software aplican ciertas acciones a objetos de determinados tipos; para obtener sistemas flexibles y reutilizables, es mejor basar su estructura en los tipos de objetos que basarse en las acciones. El concepto resultante es un mecanismo notablemente potente y versátil denominado **clase**, que en la construcción de software orientado a objetos sirve como base tanto para la estructura modular como para el sistema de tipos.

La *disciplina de fiabilidad* es un enfoque radical para el problema de construir software que haga lo que se supone que debería hacer. La idea es tratar al sistema como una colección de componentes que colaboran del mismo modo en que lo hacen los negocios que tienen éxito: respetando unos **contratos** que definen explícitamente las obligaciones y beneficios que incumben a cada una de las partes.

El *principio epistemológico* aborda la cuestión de cómo deberían describirse las clases. En la tecnología de objetos, los objetos que describe una clase suelen estar definidos únicamente por lo que se puede hacer con ellos: las operaciones (que también se conocen con el nombre de características) y las propiedades formales de estas operaciones (los contratos). La idea se expresa formalmente mediante la teoría de los **tipos abstractos de datos**, que se describe detalladamente en un capítulo de este libro. Tiene implicaciones de gran alcance, algunas de las cuales van más allá del software, y explica por qué no hay que detenerse en el concepto *naïve* de “objeto” tomado del significado ordinario de esa palabra. La tradición del modelado de sistemas de información suele suponer la existencia de una “realidad externa” que antecede a cualquier programa que la utilice; para el desarrollador orientado a objetos, esta noción carece de significado, por cuanto la realidad no existe independientemente de lo que se quiere hacer con ella. (Más exactamente, el que existe o no es una cuestión irrelevante, porque sólo conocemos aquello que podemos usar, y lo que sabemos acerca de algo está definido enteramente por la forma en que lo utilizamos.)

La *técnica de clasificación* surge de observar que el trabajo intelectual sistemático en general, y el razonamiento científico en particular, requieren la creación de taxonomías para los dominios que se estudien. El software no es una excepción, y el método orientado a objetos se basa mucho en la disciplina de clasificación conocida como **herencia**.

Los tipos abstractos de datos se discuten en el Capítulo 6, que aborda también algunos de los problemas epistemológicos relacionados.

Sencillo pero potente

Los cuatro conceptos de clase, contrato, tipo abstracto de datos y herencia dan lugar inmediatamente a un cierto número de cuestiones. ¿Cómo se buscan las clases, y cómo se describen? ¿Cómo deben nuestros programas manejar las clases y los objetos correspondientes (las *instancias* o *ejemplares* de esas clases)? ¿Cuáles son las relaciones posibles entre clases? ¿Cómo se pueden aprovechar las similitudes que puedan existir entre diferentes clases? ¿Cómo se relacionan estas ideas con problemas clave de la ingeniería del software, tales como la extensibilidad, la facilidad de uso y la eficiencia?

Las respuestas a estas preguntas se basan en un conjunto pequeño pero potente de técnicas para producir software reutilizable, extensible y fiable: polimorfismo y ligadura dinámica; un nuevo punto de vista de los tipos y de la comprobación de tipos; genericidad restringida y no restringida; ocultación de información; aserciones; tratamiento seguro de excepciones; recolección automática de basura. Se han desarrollado eficientes técnicas de implementación que permiten aplicar con éxito estas ideas a proyectos grandes y pequeños, con las estrictas restricciones del desarrollo de software comercial. Las técnicas orientadas a objetos también han tenido un impacto considerable sobre las interfaces de usuario y los entornos de desarrollo, haciendo que sea posible producir sistemas interactivos mucho mejor de lo que antes era posible. Todas estas ideas tan importantes se estudiarán con detalle, para así equipar al lector con herramientas que sean aplicables de inmediato a una amplia gama de problemas.

Organización del texto

En las páginas siguientes se revisarán los métodos y técnicas de la construcción de software orientado a objetos. La presentación se ha dividido en seis partes.

Capítulos 1 a 2.

La Parte A es una introducción y una visión general. Comienza por explorar la cuestión fundamental de la calidad del software, y prosigue con una breve revisión de las principales características técnicas del método. Esta parte es casi un pequeño libro por sí misma, y proporciona una primera visión del enfoque orientado a objetos para lectores apresurados.

Capítulos 3 a 6.

La Parte B no se apresura. Titulada “El camino a la orientación a objetos”, se toma su tiempo para describir los problemas metodológicos que llevan a los conceptos centrales O-O. Su foco principal es la modularidad: lo que se necesita para inventar estructuras satisfactorias para la construcción de sistemas “al por mayor”. Finaliza con una presentación de los tipos de datos abstractos, la base matemática de la tecnología de objetos. Las matemáticas implicadas son elementales, y los lectores que tengan menores inclinaciones matemáticas pueden conformarse con las ideas básicas, pero la presentación proporciona un trasfondo teórico que será necesario para comprender en su totalidad los principios y problemas O-O.

Capítulos 7 a 18.

La Parte C es el núcleo técnico del libro. Presenta, uno por uno, los componentes técnicos principales del método: clases, objetos y el modelo de ejecución asociado; problemas de gestión de memoria; genericidad y comprobación de tipos; diseño por contrato, aserciones, excepciones; herencia, los conceptos asociados de polimorfismo y ligadura dinámica, y sus múltiples aplicaciones interesantes.

Capítulos 19 a 29.

La Parte D discute la metodología, con especial hincapié en el análisis y el diseño. Mediante varios casos prácticos estudiados en profundidad, se presentan algunos *patrones de diseño* fundamentales, y se abordan cuestiones centrales como la búsqueda de clases, la forma de utilizar la herencia correctamente y la forma de diseñar bibliotecas reutilizables. Comienza con una discusión de metanivel de los requisitos intelectuales para los metodólogos y otras personas que suelen dar consejos; concluye con una revisión del proceso del software (el modelo del ciclo de vida) para el desarrollo O-O y una discusión sobre la mejor manera de enseñar el método tanto en la industria como en la Universidad.

Capítulos 30 a 32.

La Parte E explora temas avanzados: concurrencia, distribución, desarrollo cliente-servidor e Internet; persistencia, evolución de esquemas y bases de datos orientadas a objetos; el diseño de sistemas interactivos con interfaces gráficas modernas (“IGU ó GUI”).

Capítulos 33 a 35.

La Parte F es una revisión de la forma en que se pueden implementar las ideas, o de la forma de emularla en ciertos casos en diferentes lenguajes y entornos. Esto incluye, en particular, un análisis de los principales lenguajes orientados a objetos, centrándose en Simula, Smalltalk, Objective-C, C++, Ada 95 y Java, y una estimación de la forma de obtener algunos de los beneficios de la orientación a objetos en lenguajes no O-O como Fortran, Cobol, Pascal, C y Ada.

Capítulo 36.

La Parte G describe un entorno que va más allá de estas soluciones, y proporciona un conjunto integrado de herramientas que prestan su apoyo a las ideas del libro.

Apéndice A.

Como material complementario de referencia, hay un apéndice que muestra unas cuantas clases de biblioteca que se discuten en el texto, proporcionando un modelo para el diseño de software reutilizable.

Una Web que abarca todo el libro

Puede ser divertido ver a los autores esforzarse por describir las rutas recomendadas para pasar por su libro, a veces con ayuda de sofisticados diagramas del recorrido —como si los lectores fueran a prestar la mínima atención, y no fueran suficientemente inteligentes para trazar su propia ruta. Se permite al autor, sin embargo, decir cuál es el espíritu con el que ha planificado los diferentes capítulos, y la ruta que tiene in mente para lo que Umberto Eco denomina el Lector Modelo —que no debe confundirse con el lector real, conocido con el nombre de “usted”, y que está hecho de carne y hueso, y tiene sus propios gustos.

Aquí la respuesta es la más sencilla posible. Este libro cuenta una historia, y supone que el Lector Modelo seguirá el cuento de principio a fin, aunque se le invita a evitar las secciones más especializadas, que se han marcado para saber que “se pueden obviar en una primera lectura”; y si no tiene inclinaciones matemáticas, se le invita también a que ignore unos cuantos desarrollos matemáticos que también se han marcado explícitamente. El lector real, por supuesto, quizás desee descubrir por anticipado algunos de los desarrollos posteriores de la trama, o limitar su atención únicamente a unas pocas subtramas; por esta razón, cada capítulo se ha hecho tan autocontenido como era posible, de tal modo que sea posible absorber el material en la dosificación exacta que más convenga al lector.

Dado que la historia presenta una visión coherente del desarrollo de software, sus sucesivos temas están muy entrelazados. Las notas al margen ofrecen un subtexto de referencias cruzadas, una Web que abarca todo el libro enlazando las distintas secciones hacia adelante y hacia atrás. Mi consejo para el Lector Modelo es que los ignore en una primera lectura, salvo para asegurarse de que las preguntas que en algunas fases quedan parcialmente abiertas se cerrarán por completo más adelante. El lector real, que quizás no deseé consejo alguno, puede utilizar las referencias cruzadas como guías no oficiales cuando quiera ignorar el orden predefinido de los temas.

Las referencias cruzadas deberían resultar útiles tanto al Lector Modelo como al Lector Real en lecturas subsiguientes, para asegurarse de que ha comprendido realmente un cierto concepto orientado a objetos en profundidad, y de que comprende sus conexiones con los demás componentes del método. Al igual que los vínculos de un documento WWW, las referencias cruzadas deberían hacer posible seguir estas asociaciones de forma rápida y efectiva.

El CD-ROM que acompaña a este libro contiene todo el texto original en inglés y proporciona una forma cómoda de seguir las referencias cruzadas: basta hacer clic en ellas. Se han mantenido todas las referencias cruzadas.

La notación

Quizás en software más que en ningún otro lugar, el pensamiento y el lenguaje están íntimamente conectados. A medida que vayamos avanzando en estas páginas, desarrollaremos cuidadosamente una notación para expresar conceptos orientados a objetos en todos los niveles: modelado, análisis, diseño, implementación y mantenimiento.

Tanto aquí como en los demás lugares del libro, el pronombre “nosotros” no significa “el autor”: tal como en el lenguaje habitual, “nosotros” significa usted y yo —el lector y el autor. En otras palabras, me gustaría que usted esperase, a medida que desarrollemos la notación, estar involucrado en el proceso.

Por supuesto, esto no es totalmente cierto, por cuanto la notación ya existía antes de que el lector empezara a leer estas páginas. Pero tampoco es totalmente disparatado, porque espero que a medida que vayamos explorando el método orientado a objetos, y examinemos cuidadosamente sus implicaciones, la notación que le sirve de apoyo se le vaya ocurriendo de forma más o menos inevitable, para que sienta ciertamente que ha ayudado a diseñarla.

Esto explica por qué, aun cuando la notación existe desde hace más de diez años y de hecho es soportada por varias implementaciones comerciales, incluyendo una de mi empresa (ISE), no la he manifestado como un lenguaje. (Su nombre sí aparece una vez en el texto, y muchas en la bibliografía.) Este libro trata del método orientado a objetos para reutilizar, analizar, diseñar, implementar y mantener software; el lenguaje es una consecuencia importante y (espero) natural de ese método, no una meta en sí misma.

Además, el lenguaje es claro e incluye poco más que un soporte director para el método. Los estudiantes de primer año que lo utilizaban decían que “no es un lenguaje” —lo cual significa que la notación está en correspondencia biyectiva con el método: aprender la una es aprender el otro, y hay muy poca decoración lingüística adicional por encima de los conceptos. Ciertamente, la notación muestra pocas de las peculiaridades (que suelen surgir de circuns-

tancias históricas, restricciones de la máquina o el requisito de ser compatibles con viejos formalismos) que caracterizan a casi todos los lenguajes de programación actuales. Por supuesto, quizás el lector no esté de acuerdo con las palabras reservadas (por qué `do` y no `begin` o quizás `hacer`?), o desee añadir terminadores de punto y coma al final de cada instrucción. (Se ha diseñado la sintaxis de tal modo que el punto y coma fuera opcional.) Pero esto son asuntos secundarios. Lo que cuenta es la sencillez de la notación y hasta qué punto se corresponde directamente con los conceptos. Si comprende la tecnología de objetos, casi la conoce ya.

La mayoría de los libros de software dan por supuesto el lenguaje, tanto si es un lenguaje de programación como si se trata de una notación para el análisis y el diseño. Aquí el enfoque es diferente; implicar al lector en el diseño significa que uno no sólo debe explicar el lenguaje sino también justificar y discutir las alternativas. La mayor parte de los capítulos de la parte C incluye una sección de "discusión" que explica los problemas hallados durante el diseño de la notación, y la forma en que se resolvieron. Muchas son las veces que he deseado, al leer las descripciones de lenguajes bien conocidos, que los diseñadores me dijeran no sólo las soluciones que habían seleccionado, sino también por qué las habían seleccionado, y las alternativas que habían rechazado. Las discusiones abiertas que se incluyen en este libro deberían, espero, proporcionar valiosas ideas no sólo acerca del diseño del lenguaje sino también acerca de la construcción de software, por cuanto ambas tareas resultan sorprendentemente similares.

Análisis, diseño e implementación

Siempre es arriesgado utilizar una notación que se parezca externamente a un lenguaje de programación, por cuanto esto puede sugerir que la notación sólo abarca la fase de implementación. Esta impresión, pese a ser incorrecta, es muy difícil de corregir, así que con harta frecuencia se dice a los administradores y desarrolladores que existe un hiato de dimensiones cósmicas entre el éter del análisis y diseño y el inframundo de la implementación.

La tecnología de objetos bien entendida reduce ese hiato considerablemente, haciendo hincapié en la unidad esencial del desarrollo del software por encima de las inevitables diferencias entre niveles de abstracción. Este enfoque *sin discontinuidades* para la construcción de software es una de las contribuciones importantes del método, y está reflejado en el lenguaje de este libro, que está destinado al análisis y al diseño, así como a la implementación.

Desafortunadamente, una parte de la evolución reciente de esta disciplina va en contra de estos principios, a través de dos fenómenos igualmente lamentables:

- Los lenguajes de implementación orientados a objetos que no son adecuados para el análisis, para el diseño y en general para el razonamiento de alto nivel.
- Los métodos de análisis o diseño orientados a objetos que no abarcan la implementación (y que se autopronostican "independientes del lenguaje" como si fuera una medalla honorífica y no el reconocimiento de un fracaso).

Estos enfoques amenazan con cancelar gran parte del beneficio potencial de este enfoque. Por contraste, tanto el método como la notación desarrollados en este libro están destinados a ser aplicables a todo el proceso de construcción de software. Hay unos cuantos capítulos que abarcan los problemas de diseño de alto nivel; uno está dedicado al análisis; otros exploran las técnicas de implementación y las implicaciones del tiempo en cuanto a rendimiento.

El entorno

La construcción de software se basa en una tetralogía básica: método, lenguaje, herramientas, bibliotecas. El método es la parte fundamental de este libro; la cuestión del lenguaje se acaba

«Ausencia de discontinuidades y reversibilidad», §28.6.

El último capítulo, el 36, resume el entorno.

de mencionar. De vez en cuando, necesitaremos ver el apoyo que necesitan de las herramientas y bibliotecas. Por razones de comodidad evidentes, estas discusiones aludirán ocasionalmente al entorno orientado a objetos de ISE, con su conjunto de herramientas y bibliotecas asociadas.

El entorno sólo se utiliza como ejemplo de lo que se puede hacer para que los conceptos sean utilizables en la práctica por parte de los desarrolladores. No deje de observar que hay otros muchos entornos orientados a objetos disponibles, tanto para la notación de este libro como para otros métodos y notaciones de análisis, diseño e implementación; y que las descripciones dadas aluden al estado del entorno en el momento de escribir estas líneas, que está sometido, como cualquier otra cosa de nuestra industria, a cambios rápidos —para mejor. También se citan otros entornos, tanto O-O como no O-O, a lo largo de todo el texto.

Agradecimientos (cuasi ausencia de los mismos)

La primera edición de este libro ya contenía una larga lista de agradecimientos. Durante algún tiempo, seguí anotando los nombres de las personas que contribuían con comentarios y sugerencias, y llegó un momento en que perdí el hilo. La lista de colegas que me han ayudado, o de los que he tomado ideas ha crecido tanto que se extendería a lo largo de muchas páginas, e inevitablemente omitiría a personas importantes. Es mejor ofender a todos un poquito que ofender mucho a unos pocos.

Por tanto, estos agradecimientos van a ser colectivos en su mayor parte, lo cual no hace mi gratitud menos profunda. Mis colegas de ISE y de SOL han sido durante años una fuente diaria de ayuda valiosísima. Los usuarios de nuestras herramientas nos han ofrecido generosamente su consejo. Los lectores de la primera edición proporcionaron miles de sugerencias de mejoras. En la preparación de esta nueva edición (realmente debería decir de este nuevo libro), he enviado centenares de mensajes de correo electrónico pidiendo ayuda de muchos tipos: la clarificación de algún asunto delicado, una referencia bibliográfica, un permiso para citar a alguien, los detalles de una atribución, el origen de una idea, el detalle concreto de una notación, la dirección oficial de una página Web, y las respuestas han sido invariablemente positivas. A medida que los borradores de los capítulos iban estando preparados, se hacían pasar por distintos medios, dando lugar a muchos comentarios constructivos (y aquí debo citar por sus nombres a los evaluadores contratados por Prentice-Hall, Paul Dubois, James McKim y Richard Wiener, que me proporcionaron valiosos consejos y correcciones). En los últimos años he impartido innumerables seminarios, conferencias y cursos relativos a los temas de este libro, y en todos los casos he aprendido algo de la audiencia. He disfrutado con el ingenio de mis compañeros de mesa en las conferencias, y me he beneficiado de su sabiduría. Unas breves estancias sabáticas en la University of Technology, Sydney y en la Università degli Studi di Milano me proporcionaron un influjo de nuevas ideas —y en el primer caso con trescientos alumnos de primer curso para validar con ellos mis ideas acerca de la forma en que debería enseñarse la ingeniería de software.

La extensa bibliografía muestra claramente la forma en que las ideas y realizaciones de otros han contribuido a este libro. Entre las influencias conscientes más importantes se encuentra la línea de lenguajes de Algol, con su énfasis en la elegancia sintáctica y semántica; el trabajo seminal sobre programación estructurada, en el sentido serio (Dijkstra-Hoare-Parnas-Wirth-Mills-Gries) del término, y construcción sistemática de programas; las técnicas de especificación formal, en particular las inagotables lecciones de la versión original de Jean-Raymond Abrial del lenguaje de especificación Z (a finales de los años setenta); su diseño más reciente de B, y el trabajo de Cliff Jones sobre VDM; los lenguajes de generación modular (en particular Ada de Ichbiah, CLU de Liskov, Alphard de Shaw, LPG de Bert y Modula de Wirth); y Simula 67, que presentó hace muchos años la mayoría de los conceptos y tenía bien casi todos ellos, lo cual me trae a la cabeza el comentario de Tony Hoare acerca de Algol 60: que era una mejora tan importante respecto a la mayoría de sus sucesores.

Hay unas pocas notas del margen o en las secciones bibliográficas de fin de capítulo que dan crédito a ciertas ideas específicas, frecuentemente no publicadas.

Prólogo a la segunda edición

Muchos son los sucesos acaecidos en el mundo de la orientación a objetos desde que se publicó la primera edición de *CSOO* (nombre que se le daría al libro) en 1988. La explosión de interés a la que se aludía en el Prólogo de la primera edición, reproducida en las páginas anteriores de forma ligeramente expandida, no era nada en comparación con lo que se ha visto desde entonces. En la actualidad, muchas revistas y conferencias tratan la tecnología de objetos; Prentice Hall tiene toda una serie de libros dedicados al tema; se han producido descubrimientos trascendentales en aspectos tales como las interfaces de usuario, la concurrencia y las bases de datos; han aparecido temas completamente nuevos, como el análisis O-O y la especificación formal; la computación distribuida, que antes fuera un tema especializado, se está volviendo relevante para un número cada vez mayor de desarrollos, gracias en parte al crecimiento de Internet; y la Web está afectando al trabajo diario de todo el mundo.

No son estas las únicas noticias interesantes. Resulta gratificante ver cuánto progreso se está produciendo en el campo del software —gracias en parte a la incompleta pero innegable difusión de la tecnología de objetos. Todavía hay demasiados libros y artículos de ingeniería de software que empiezan con el obligatorio lamento acerca de la “crisis del software” y el lamentable estado de nuestra industria en comparación con las *verdaderas disciplinas* de la ingeniería (que, como todos sabemos, nunca estropean las cosas). No hay razón para esta condena. Bueno, es bien cierto que todavía nos queda un largo camino por recorrer, como muy bien sabe cualquiera que utilice productos software; pero dados los retos a que nos enfrentamos, no hay razón para avergonzarnos de nosotros mismos como profesión; y además lo vamos haciendo mejor día a día.

La ambición de este libro, como lo fuera la de su predecesor, es ayudar en el proceso. Esta segunda edición no es una actualización, sino el resultado de una reconstrucción exhaustiva. No se ha dejado intacto ni un solo párrafo de la versión original. (De hecho, prácticamente ni una línea.)

Se han añadido muchos temas nuevos, incluyendo todo un capítulo dedicado a la concurrencia, distribución, computación cliente-servidor y programación para Internet; uno sobre el ciclo de vida del software, muchas tramas de diseño y técnicas de implementación, la forma de usar bien la herencia y de evitar su mala utilización; discusiones sobre muchos otros temas de metodología orientada a objetos; una extensa presentación de los tipos abstractos de datos —la base matemática de nuestra materia, indispensable para una completa comprensión de la tecnología de objetos y que sin embargo se trata raras veces con detalle en los libros de texto y en los cursos programados; una presentación del análisis orientado a objetos; centenares de referencias bibliográficas y de sitios Web nuevos; la descripción de todo un entorno de desarrollo orientado a objetos (que se incluye en el CD-ROM adjunto para disfrute del lector) y de los conceptos subyacentes; y docenas de ideas, principios, advertencias, explicaciones, ejemplos, comparaciones, citas, clases y rutinas nuevas.

La reacción frente a *CSOO-1* fue tan satisfactoria que sé que los lectores tenían grandes esperanzas. Espero que encuentren en *CSOO-2* algo desafiante, útil, y a la altura de sus estándares.

Santa Barbara

B.M.
Enero de 1997

El CD-ROM que viene con este libro contiene todo el texto original en inglés con vínculos en formato Adobe Acrobat. También contiene el software Acrobat Reader de Adobe, lo cual permite leer ese formato; las versiones que se ofrecen abarcan las principales plataformas de la industria. Si todavía no tiene Acrobat Reader en su equipo, puede instalarlo siguiendo las instrucciones. El autor y la editorial no pretenden representar en modo alguno al propietario de Acrobat y de las herramientas asociadas; el Acrobat Reader se proporciona únicamente como un servicio para los lectores de este libro, y cualquier pregunta sobre Acrobat debería dirigirse a Adobe. Quizá desee consultar a Adobe acerca de las posibles versiones del Reader que puedan haber aparecido después del libro.

Para comenzar con el CD-ROM, abra el archivo de Acrobat *README.pdf* del directorio OOSC_2, que le orientará hacia la tabla de contenidos y el índice. Solo se puede abrir ese archivo desde Acrobat Reader; si el Reader no está instalado en su computadora, examine la versión de formato texto que se encuentra en el archivo *readme.txt* del directorio raíz.

La presencia de una versión electrónica será de especial utilidad para aquellos lectores que deseen aprovechar los miles de referencias cruzadas presentes en este libro (véase «Una Web que abarca todo el libro», página vii). Tener la versión electrónica en una computadora permite seguir los vínculos de vez en cuando sin tener que pasar páginas hacia adelante y hacia atrás. La forma electrónica resulta especialmente cómoda para una lectura, en la que posiblemente se desee explorar los vínculos de manera más sistemática.

Todos los vínculos (referencias cruzadas) aparecen en azul en el formulario de Acrobat, según se ha ilustrado más arriba en dos ocasiones. Para seguir un vínculo, basta hacer clic en la parte azul.

Si la referencia alude a otro capítulo, el capítulo aparecerá en una nueva ventana. La orden de Acrobat Reader para volver a la situación anterior es normalmente Control-sígueme menos (esto es, pulsar —mientras se mantiene pulsada la tecla CONTROL). Consulte la documentación en línea de Acrobat Reader para hallar otros mandatos útiles para la navegación.

Las referencias bibliográficas también aparecen como enlaces, tal como [Knuth 1968], en el formulario de Acrobat, así que es posible hacer clic en cualquiera de ellas para visualizar la entrada correspondiente en la bibliografía del Apéndice E.

El CD-ROM contiene también:

- Unos componentes de biblioteca que aportan un extenso material para el Apéndice A.
- Un capítulo del manual de un constructor gráfico de aplicaciones, que proporciona complementos matemáticos para el material del Capítulo 32.

Además, el CD-ROM contiene una versión limitada en tiempo de un entorno de desarrollo orientado a objetos avanzado para Windows 95 o Windows NT, según se describe en el Capítulo 36, proporcionando una excelente oportunidad práctica para probar las ideas desarrolladas a lo largo del libro. El archivo «Readme» indica la posición de las instrucciones de instalación, y los requisitos del sistema.

Agradecimientos: La preparación del libro en hipertexto la hizo posible la ayuda de varias personas de Adobe Inc., en particular Sandra Knox, Sarah Rosenbaum y el grupo de apoyo al cliente de FrameMaker. Debo especial agradecimiento —tanto por el libro como por el CD— a Russ Hall y a Eileen Clark de Prentice Hall.

Sobre la bibliografía, las fuentes de Internet y los ejercicios

La bibliografía comienza en la página 1149.

Este libro se basa en contribuciones anteriores de múltiples autores. Para facilitar la lectura, la discusión de las fuentes no aparece en la mayoría de los casos en la discusión misma, sino en las secciones de "Notas Bibliográficas" del final del capítulo. Asegúrese de leer esas secciones, para comprender el origen de muchas ideas y averiguar la forma de aprender más. Las referencias tienen la forma [Nombre 19xx], donde *Nombre* es el nombre del primer autor, y se refiere a la bibliografía del Apéndice E. Esta convención sólo se sigue por legibilidad, y no pretende subestimar la contribución de los autores distintos del primero. La letra M en lugar del *Nombre* alude a publicaciones del autor de este libro, que se enumeran por separado en una segunda parte de esta bibliografía.

Aparte de la bibliografía en sí, algunas de las referencias aparecen en el margen, al lado de los párrafos que las citan. La razón de este tratamiento por separado es hacer que la bibliografía sea utilizable por sí misma, como colección de referencias importantes acerca de la tecnología de objetos y de los temas relacionados con ella. La aparición en el margen y no en la bibliografía no constituye un juicio de valor desfavorable; la división es únicamente una estimación pragmática de lo que debe constar en una lista fundamental de referencias orientadas a objetos.

Aun cuando las referencias electrónicas se considerarán algo habitual en unos pocos años, éste debe ser uno de los primeros libros técnicos (salvo los libros dedicados a temas relacionados con Internet) que hace un uso extenso de referencias a páginas Web, foros de discusión y grupos de noticias de Usenet y otros recursos de Internet.

Las direcciones electrónicas son notablemente volátiles. He intentado obtener de los autores una cierta seguridad de que las direcciones dadas seguirían siendo válidas durante varios años. Ni ellos ni yo, por supuesto, podemos proporcionar una garantía absoluta. En caso de dificultad, tenga en cuenta que en la Red las cosas se trasladan en lugar de desaparecer: las herramientas de búsqueda basadas en palabras clave pueden servirle de ayuda.

Casi todos los capítulos contienen ejercicios de distintos grados de dificultad. Me he abstenido de proporcionar soluciones, aunque muchos ejercicios contienen pistas bastante precisas. Habrá quien lamente la ausencia de soluciones completas; sin embargo, espero que sepan apreciar las razones de esta decisión: el miedo a destruir el disfrute del lector; darse cuenta de que hay muchos ejercicios que son problemas de diseño, para los cuales hay más de una buena respuesta; y el deseo de proporcionar una fuente de problemas preparados para su uso a los instructores que hagan uso de este libro como texto.

Por brevedad y sencillez, el texto sigue la tradición imperfecta pero secular de utilizar palabras tales como "el" o "los" como referencias de personas indeterminadas, y como abreviatura de "el o la" y de "la o las", sin que haya connotación alguna de género.

*Asombran estos objetos al alma modesta
Y cuantos valiosos extraños traen a nuestra testa.*

Molière, *El Tartufo*, Acto Tercero.

Prólogo a la edición española

Estado del arte en construcción de software

La década de los años 90, y especialmente los últimos años de la misma y por extensión el Tercer Milenio, se están caracterizando por el predominio de las *Tecnologías de Objetos* (TO) y su generalización más completa, las *Tecnologías de Componentes* (TC). Así, no sólo se puede ver toda suerte de libros, revistas, artículos junto a la proliferación de seminarios, congresos, simposiums, etc. relativos a TO, sino también relativos a TC (ese es el caso de CORBA, COM, DCOM, ActiveX, etc.), incluso ya es frecuente oír hablar de *ingeniería de componentes*. Los “objetos”, así como su manifestación más práctica, “los componentes”, están impregnando toda la industria de construcción de software. ¿Cuál es entonces el estado del arte en construcción de software? Evidentemente, la respuesta es difícil; pero qué duda cabe que los objetos son parte esencial de cualquier estudio serio –profesional o científico– que se realice sobre esa cuestión.

¿Cuáles son las preocupaciones actuales de las empresas productoras y consumidoras de tecnologías de la información, de los desarrolladores de software, de las universidades, de los centros de investigación, etc.? Nada mejor para contestar a esta pregunta que consultar las revistas de software de prestigio, la Web (Internet), organizaciones internacionales (ACM, OMG, etc.), conferencias y congresos sobre Tecnologías Orientadas a Objetos (TOOLS, OOPSA, Object Expo, Object World, etc.), programas y currícula de universidades, etc. De esas consultas se deduce qué en uno u otro orden, los temas más sobresalientes y de mayor preocupación dentro del mundo del software, son:

- Cliente / Servidor.
- Internet.
- Metodología de análisis y diseño.
- Interfaces gráficas de usuario.
- Diseño de bibliotecas.
- Conurrencia.
- Persistencia.
- Ciclo de vida del software.
- Calidad del software.
- *Reutilización (reusabilidad)*.
- Etc.

Tal vez haciéndose estas preguntas u otras similares Bertrand Meyer comenzó a configurar la 2.^a edición de *Object-Oriented Software Construction* pensando, como ya hizo en la 1.^a edición, que lo importante para la comunidad informática o de computación era proporcionarles criterios rigurosos y eficientes para construir software de calidad. Así, el ya lejano 18 de

Febrero de 1996¹ a la pregunta del newsgroups comp.lang.eiffel “;When is the next version of OOSC due out? Bertrand Meyer contestó (una síntesis):

“El libro ha sido totalmente reescrito. Ni un sólo párrafo se ha dejado sin modificar. Hay una considerable cantidad de nuevo material:

- *Concurrencia, persistencia.*
- *Metodología para usar herencia.*
- *Un capítulo sobre el rol (papel) de la metodología.*
- *Ánálisis.*
- *El diseño de interfaces gráficas de usuario.*
- *Una discusión más amplia sobre el Diseño por Contrato.*
- *Una discusión completa sobre tipos abstractos de datos.*
- *Principios de entorno.*
- *Tipificación (typing).*
- *Muchos ejemplos, ejercicios, discusiones.*
- *Un glosario de tecnología de objetos.*
- *Un epílogo.*
- *¡Puede haber alguna sorpresa!*
- ...”

Pues bien, tras haber leído el libro de principio a fin, coordinar y revisar su traducción y seguir consultándolo con mucha frecuencia para mis clases, seminarios, conferencias, etc., podemos asegurar que Meyer no sólo cumplió su promesa al grupo de noticias de Internet que se lo solicitaba (comp.lang.eiffel), sino que con creces nos sorprendió gratamente y, naturalmente, añadió más de una sorpresa para satisfacción de su legión de seguidores; por ejemplo añadió:

- *Informática/computación Cliente/Servidor.*
- *Programación en Internet.*
- *El ciclo de vida del software.*
- *Muchos patrones de diseño y técnicas de implementación.*
- *Temas metodológicos.*
- *Una amplia teoría matemática sobre tipos abstractos de datos.*
- *Distribución.*
- *Una nueva metodología.*
- *Aplicación del método en diversos lenguajes y entornos, tales como Ada-83, Ada-95, Java, Smalltalk, etc., así como una emulación de tecnologías de objetos en entornos no orientados a objetos.*

Es decir, Bertrand Meyer, ha realizado lo que parecía imposible, analizar y describir con maestría prácticamente todos los temas que el estado del arte actual exigía en la construcción de software, especialmente orientado a objetos. Se podría decir, sin temor a equivocarnos, que —la comunidad informática² creemos así lo reconoce— *Construcción de Software Orientado a Objetos, 2.ª edición (OOSC-2)* es la Biblia de las Tecnologías de Objetos. La comunidad informática, el mundo universitario, investigador y empresarial, pensamos, así considerarán a esta magna y enciclopédica OOSC-2. La primera edición OOSC-1 ha sido considerada por muchas personas como un trabajo científico y riguroso de máximo nivel que se ha convertido en una obra clásica citada continuamente por profesores universitarios e investigadores. Las más afamadas personalidades del mundo de las Tecnologías de Objetos y de la ingeniería de

¹ Newsgroup:com.lang.eiffel. <http://www.prog.soc.uts.edu.au/gedridge/ef9598.htm>.

² El libro ha recibido, entre otros honores, el excelente premio internacional “1997 Jolt Award”.

software, en general, citan su obra con gran frecuencia en sus propios trabajos. OOSC-1 es una de las obras clásicas de la literatura informática y de computación, OOSC-2, con escasamente un año de vida, se puede considerar ya también una obra clásica y que paradójicamente –parece que así lo ha querido Meyer, ha modificado todos y cada una de sus páginas– convivirá con la 1.^a edición, de modo que tenemos la suerte la comunidad informática de disfrutar de dos obras clásicas para consulta: OOSC-1 y –naturalmente y sobre todo– OOSC-2.

Factores externos de la calidad del software

Bertrand Meyer siempre se ha preocupado, en sus obras como en sus artículos en la prensa científica y profesional por la construcción de software de calidad y, para ello siempre ha tratado de proporcionar –y justificar– criterios para su consecución. Así lo ha ido plasmando en sus numerosas obras, de modo que de los cinco primeros principios originales de 1988 –factores externos–: *Corrección (correctness)*, *robustez (robustness)*, *extensibilidad (extensibility)*, *reutilización o reusabilidad (reusability)* y *compatibilidad (compatibility)*, añadió en 1995³: “*Fiabilidad (reliability)*, *portabilidad (portability)* y *Eficiencia (efficiency)*”, para finalmente hacer una declaración de principios de calidad de software a modo de decálogo⁴ que entiendo como máxima a seguir por los profesionales de la construcción del software, así como por los docentes universitarios que nos dedicamos a la disciplina de ingeniería de software y, por descontado, los analistas e ingenieros de software encargados de velar por la calidad del software en sus empresas y en los productos que crean. Esta especie de decálogo, que Meyer denomina los factores externos de calidad y cuya consecución es la tarea central de la construcción de software orientado a objetos⁵, es:

- *Corrección.*
- *Robustez.*
- *Extensibilidad.*
- *Reutilización o reusabilidad.*
- *Compatibilidad.*
- *Eficiencia.*
- *Portabilidad.*
- *Facilidad de uso.*
- *Funcionalidad.*
- *Oportunidad.*

Meyer añade también otras cualidades: *Verificabilidad* (facilidad de verificación), *Integridad*, *Reparabilidad* (facilidad de reparación de defectos) y *Economía*.

La orientación a objetos como técnica de ingeniería de software: la reutilización

Meyer, señalaba en 1995⁶ que la orientación a objetos es una técnica de ingeniería de software, para acto seguido definir la ingeniería de software como el estudio de métodos y herramientas que se pueden utilizar para producir software práctico de calidad. Los dos aspectos clave de esta definición son el rol de la calidad y el énfasis en software práctico –software

³ Meyer. *Object Success*. Prentice Hall, 1993, p. 3.

⁴ Aunque no es definida así por Meyer, entendemos que constituyen formalmente un Decálogo de la Calidad del Software, y así proponemos su aceptación por la comunidad informática: “*Decálogo de la calidad del software de Meyer*”.

⁵ OOSC-2, 1997, pp. 4-16.

operacional desarrollado bajo principios económicos y organizativos de empresa-. El deseo subyacente y de modo obsesivo en la obra de Meyer que continúa en OOSC-2 es la construcción de software correcto (fiel, exacto) y por ello alienta continuamente al desarrollo basado en componentes de software correcto y reutilizables, y esta característica, que la mantiene en todo el libro por su importancia, le dedica un capítulo completo (más de 30 páginas) en el que describe los componentes de software reutilizables (Capítulo 4) con especial énfasis en la documentación adecuada y que completa con el magnífico Capítulo 11 sobre el diseño por contrato en el que ya da los principios fundamentales para la construcción de software fiable.

Reutilizar software es una condición requerida (exigida) para cualquier progreso en el desarrollo de software⁷. La reutilización requiere la construcción de componentes, considerando como tales los módulos de software reutilizables que abarcan algunos de los patrones fundamentales del desarrollo de software en áreas de aplicación. Meyer sigue insistiendo en que la experiencia no sólo le ayudará a utilizar estos módulos adecuadamente, sino también a desarrollar sus propias bibliotecas robustas y reutilizables, asegurándose de que tengan éxito.

Se podría decir que reutilizar software es el proceso de crear sistemas software a partir de otro ya existente en lugar de construir a partir de un borrador o proyecto inicial⁸. La reutilización de software es todavía una disciplina emergente, pero no por ello madura. Meyer⁹ cita completa, un artículo clásico de McIlroy que señalaba, ya, en 1968, las características que debía tener el software reutilizable: “*clases, precondiciones, postcondiciones, invariantes de clases, formatos cortos y categóricos –llano(s)–, herencia en sus formatos simple, múltiple y repetida, ocultación de la información y manejo disciplinado de excepciones*”. Condiciones que Meyer usa además de otras nuevas, que añade en un lenguaje de programación que utiliza en toda la obra y que constituye el núcleo fundamental de la notación y del lenguaje empleado en OOSC-2, que en sí es una propia metodología para desarrollo de software.

La orientación a objetos y Meyer enseña este concepto de modo general y riguroso es el mejor medio posible para construir componentes de calidad. El libro y el lenguaje que utiliza han sido escritos para explotar al máximo la reutilización de componentes de software. Otra técnica que todos los informáticos siempre agradeceremos es el principio *cliente-proveedor y la programación por contrato*. Es decir la relación que liga a un componente que hace uso de las facilidades proporcionadas por otros componentes, de modo que los componentes existentes se denominan proveedores y los nuevos componentes, clientes mientras que en la programación por contrato muestra las responsabilidades de un componente software que ha de producir resultados correctos y previstos.

Los ingenieros de software siempre han soñado con llegar a conseguir una situación que emulara en contrapartidas al hardware en las que los módulos de software se trataran como componentes hardware, tal como circuitos integrados (el IC-software de Brad Cox). Esto es, los módulos que puedan ser visualizados como componentes software, sean diseñados, construidos y comprobados por su uso como componentes de software en otros productos.

Así, una vez que se construye un componente y se visualiza como inalterable, el proceso de construir un nuevo producto consistirá en encontrar una colección de componentes que cuando “se enchufan juntos” producen la aplicación requerida. Por consiguiente, la idea fundamental es disponer de un catálogo de componentes de software entre los cuales elegir y la tarea de los diseñadores de aplicaciones es determinar cómo hacer que los componentes seleccionados interactúen para resolver un problema específico. A veces, naturalmente, se necesitará construir nuevos componentes; sin embargo, construir componentes correctos puede ser extremadamente costoso y consume tiempo y tiene justificación reutilizar componentes comprobados ya existentes siempre que sea posible.

⁷ Meyer. *Reusable software*. Prentice Hall, 1994, p. vii (1.^a línea del prólogo).

⁸ Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, pp. 24: 131-83, June 1992. Citado por Johannes Sametinger en *Software Engineering with Reusable Components*. Springer-Gerlag Berlin Heidelberg, 1997, p. 10.

⁹ Meyer. *Reusable software*. Prentice Hall, 1994, p. viii.

En esencia, OOSC-2 persigue y —pensamos lo consigue— producir programas fiables, fáciles de cambiar, fáciles de explicar a otras personas, fáciles de transportar a otras máquinas. Enseña las técnicas para construir software combinando módulos existentes del mismo modo en que los ingenieros electrónicos construyen circuitos a partir de chips ya existentes.

Construcción de software utilizando una metodología orientada a objetos

OOSC-2 no es un libro sobre un lenguaje de programación específico, sino sobre construcción de software utilizando una metodología orientada a objetos. Sin embargo, difícilmente se podrá enseñar programación sin presentar código fuente y el código se ha de presentar en algún lenguaje de programación. Meyer utiliza una sencilla notación fácilmente legible que evoluciona a medida que se examina y profundiza en las técnicas orientadas a objetos. Este análisis es tan completo que la notación termina siendo robusta y útil como lenguaje de programación.

El libro trata sobre un método orientado a objetos para reutilizar, analizar, diseñar, implementar y mantener software; el lenguaje es importante y es consecuencia natural de ese método y no una meta u objetivo en sí mismo. Realmente el lenguaje resultante es el que puede considerarse, y así lo definen muchos expertos, el mejor lenguaje de programación orientado a objetos disponible hoy día.

De hecho, tal como advierte Meyer, se puede considerar un lenguaje anónimo, o es, como diríamos muchos profesores de programación, un pseudocódigo ideal con su propio intérprete/compilador incorporado. En realidad, aunque subyace el nombre Eiffel en todo el texto, es cierto que la palabra Eiffel no aparece *nunca* en todo el texto; de hecho la primera vez y única —si exceptuamos el índice alfabético del final— que aparece la palabra Eiffel es en el Epílogo, del libro original inglés, precisamente es la última palabra del citado epílogo.

A medida que se va desarrollando gradualmente la notación a través del libro, no se toma ninguna decisión sin una explicación, ninguna selección se realiza sin una discusión inteligente. Con bastante frecuencia se presentan los pros y los contra de cada opción. El lector puede no estar de acuerdo con alguna de las discusiones o planteamientos, pero al menos Meyer siempre muestra porqué ha tomado esa elección y, al contrario que otros métodos, siempre tiene presente las prestaciones o rendimiento de la opción considerada.

Unas notas sobre la organización del libro

El libro se divide en ocho partes que contienen 32 capítulos y cinco apéndices. La parte A se dedica a la calidad del software y a los criterios de orientación a objetos qué, aunque los lectores conocedores de las tecnologías de objetos tengan la tentación de saltárselas, les recomendamos, muy al contrario, su atenta lectura, dado que está llena de conceptos, sugerencias, consejos, propuesta que siempre le vendrán bien para consolidar sus conocimientos o a modo de “refresco” de información, por emplear un término informático.

Meyer hace a lo largo de la segunda parte una serie de observaciones importantes que muchos ingenieros de software con experiencia estarán de acuerdo. Por ejemplo, en el tema de la documentación de software externa: “*En lugar de considerar la documentación como un producto independiente del propio software es preferible construir el software tan autodocumentado como sea posible*”. Es fácil de decir, pero difícil de ejecutar. Durante años se han realizado esfuerzos infructuosos para alcanzar el objetivo de disponer de software autodocumentado. Meyer lo consigue merced a la notación, que incluye en sí misma un sistema de autodocumentación totalmente integrado, completo, útil y fácil de utilizar, que realmente hace el código más sencillo de crear, y de comprender, en vez de dejar esa tarea en manos del programador.

La parte C constituye por sí sola un auténtico curso de técnicas orientadas a objetos. Son alrededor de 500 páginas que ellas solas justifican el título del libro. La parte D es la metodología orientada a objetos, tal vez una de las prestaciones más solicitadas por los lectores y seguidores de la primera edición.

La parte E "Temas avanzados" trata sobre nuevos enfoques orientados a objetos, concurrencia, computación distribuida, tecnologías cliente/servidor, persistencia de objetos, bases de datos orientadas a objetos, etc. Estos temas están ausentes en la mayoría de los libros de ingeniería de software o de metodologías, de ahí su gran valor.

A pesar de no sugerir el uso de ningún lenguaje de programación, la lectura del libro puede convertir al lector en un excelente programador. El libro examina alternativas, populares hoy día, como Ada 95, C++, Java, Smalltalk, etc.

Otra de las grandes virtudes que ofrece la estructura del libro es el carácter autónomo de sus diferentes partes. De hecho, Meyer recomienda, en algún caso, la lectura del libro de un modo no lineal. Una versión con hiperenlaces, se incluye en el CD ROM y además, muchas partes del mismo se pueden consultar en Internet en la página de la empresa ISE Inc., así como en centenares de enlaces (sitios Web) que aparecen en la Red cuando se solicita en los buscadores más populares términos tales como "OOSC-2" o bien "Bertrand Meyer".

El libro como texto profesional y ayuda a la investigación

Todos los planes de estudio (*curricula*) de carreras universitarias en España de Ingeniería Informática e Ingeniería Técnica en Informática (especialidades de Sistemas y Gestión), así como en carreras de Ingeniería en Sistemas Computacionales y Licenciatura en Informática en Latinoamérica, y también en otras ingenierías como Industriales o Telecomunicaciones, o Licenciaturas en Matemáticas o Físicas en las especialidades de Ciencias de la Computación, contemplan el estudio de asignaturas tales como Programación, Tecnología de la Programación, Algoritmos y Estructuras de datos –con énfasis especial en tipos abstractos de datos–, Metodología de la Programación e Ingeniería de Software, entre otras. Por otra parte, comienza a ser también más frecuente la inclusión de asignaturas específicas tales como Algoritmia avanzada, Programación Orientada a Objetos, Análisis y Diseño Orientado a Objetos, Ingeniería de Software Orientada a Objetos, Interfaz Hombre-Máquina, Calidad de Software, etc. (este es el caso cada día más numeroso de universidades españolas y latinoamericanas, por ejemplo la Universidad Pontificia de Salamanca en el campus de Madrid tiene incorporadas todas esas asignaturas en su curricula).

Resulta paradójico, pero es una realidad tangible que OOSC-2 sirve como texto de referencia en casi todas las asignaturas citadas, en unas como texto base, en otras como libro de consulta o complementario. Éstas son las razones por las que OOSC-2 ha sido considerada con anterioridad como *la Biblia de los objetos y componentes*.

Sin embargo, también es justo reconocer que OOSC-2 se ajusta fielmente a materias troncales que se incluyen en los planes de estudios que señala el Consejo de Universidades de España. Estas materias a que nos referimos son, fundamentalmente, *Metodología de la programación, Estructura de datos e Ingeniería de Software*; los descriptores recomendados para esas materias están cubiertos de una u otra manera en OOSC-2, desde una perspectiva de objetos. Como ya se comentó anteriormente, cada día es más frecuente la aparición en los planes de estudio de asignaturas tales como Programación, Análisis y Diseño e Ingeniería de Software Orientadas a Objetos; en este caso *Construcción de Software Orientado a Objetos, 2.ª edición* se adapta perfectamente a esas asignaturas.

Nuestra experiencia en las muchas visitas que hemos realizado en los últimos diez años a un gran número de universidades e institutos tecnológicos de Hispanoamérica (Méjico, Cuba, Venezuela, Guatemala, etc.) nos permite confirmar también el interés creciente en las universidades e institutos tecnológicos de todos los países hispanoamericanos por todo lo relativo a

tecnologías orientadas a objetos. Por todo lo anterior, entendemos que OOSC-2 será una referencia obligatoria en universidades españolas y latinoamericanas en los próximos años, al menos, y como ha sucedido con la primera edición, como mínimo otros diez años.

El libro como texto profesional y ayuda a la investigación

La primera edición de OOSC ha sido considerada por muchos profesionales e investigadores como un clásico en la construcción de software. Desde que Prentice Hall publicó en 1988 OOSC-1, esta obra ha sido considerada referencia obligatoria por una gran cantidad de profesionales e investigadores. Raros son los libros de iniciación, medios o avanzados, de cualquier autoridad informática que se precie, que no cite a Bertrand Meyer y su primera edición (naturalmente, también sus otras obras de prestigio).

Por estas circunstancias y dado que OOSC-2 ha introducido una vasta y extensa colección de conceptos, innovaciones tecnológicas, etc., es de prever —y seguros estamos de ello— que OOSC-2 será de gran utilidad para el desarrollo profesional de software tanto para programadores como analistas e ingenieros de software, así como investigadores y científicos de todo tipo.

Epílogo

La traducción de esta excelente e imprescindible última obra de Bertrand Meyer ha sido posible gracias al esfuerzo coordinado y común que hemos realizado varios profesores de Facultades de Informática y de Ciencias de diferentes universidades españolas y latinoamericanas; me corresponde como coordinador de la traducción y revisor técnico, *escribir este prólogo, cosa que hago como un gran honor profesional*. Todos los profesores citados impartimos disciplinas de las áreas de la programación e ingeniería de software, y naturalmente asignaturas de tecnologías orientadas a objetos, y hemos utilizado de una u otra forma la primera edición de OOSC y ya utilizamos la versión en inglés de OOSC-2.

Por esta circunstancia hemos podido apreciar la magnitud de esta nueva obra de Meyer, a la que hemos calificado, pensamos que con justicia y sin riesgo de exageración, como “*La Biblia de las Tecnologías de Objetos*”. Si la primera edición fue y sigue siendo referencia continua de profesores, alumnos y profesionales de informática y ciencias de la computación, y en general, ingeniería esta segunda edición será la referencia obligatoria para los años venideros y estamos seguros de que será obra emblemática en el tercer milenio durante muchos años.

Los principios y conceptos clásicos de objetos —en particular— y de construcción de software —en general— han tenido una influencia considerable y difícil de medir, precisamente por su magnitud, en la comunidad que desarrolla software orientado a objetos. Los principios y conceptos nuevos introducidos en esta segunda edición, tanto los innovadores tecnológicamente hablando, como los clásicos, estudiados bajo la rigurosa y acertada visión de Meyer, están teniendo y tendrán una influencia muy considerable en numerosos estudiantes y profesores de ingeniería informática, ingeniería en sistemas computacionales y licenciatura de informática, además de otras carreras ya citadas de habla española, así como en numerosos cursos de masters, maestrías, doctorados y cursos de reciclaje para postgraduados en nuevas tecnologías orientadas a objetos.

Esta segunda edición llena numerosas lagunas en la literatura de habla española ya que no sólo actualiza todos los contenidos de la 1.^a edición, sino y sobre todo extiende esos contenidos a todos aquéllos que se utilizan en la construcción de software y, en general, en ingeniería de software en esta década y con seguridad en la siguiente del tercer milenio.

El excelente e inmenso trabajo de Bertrand Meyer permitirá por un lado seguir formando buenos y numerosos profesionales informáticos y de computación y, por otro, formar a espe-

cialistas en las modernas tecnologías orientadas a objetos con un énfasis especial en las ideas clave de estos años y del siglo XXI: *componentes* y *reutilización (reusabilidad)*. La obra que hoy prologamos será la referencia obligatoria en construcción de software orientado a objetos de los años previos al tercer milenio y durante el tercer milenio. Haciendo honor a su origen francés, sólo cabe decir, aunque evidentemente no sea una “expresión universitaria rigurosa” *“Chapeau Monsieur Meyer”*, *OOSC-2*, no sólo es *un libro mágico, magnífico e imprescindible* para aprender a construir software orientado a objetos, sino que su lectura conducirá al lector *a conseguir la excelencia en el mundo del software*.

Madrid, Agosto de 1988

Luis Joyanes Aguilar

Director del Departamento de Lenguajes,
Sistemas Informáticos e Ingeniería de Software
Facultad de Informática

Universidad Pontificia de Salamanca, campus Madrid (España)

Intentamos mejorar este libro

El objetivo de Prentice Hall es mejorar continuamente los libros que publica con el fin de mantener siempre el mayor rigor científico y técnico en sus obras. Como el lector podrá comprobar fácilmente a medida que vaya avanzando por los capítulos de este libro, ésta es una obra con cierto grado de dificultad tanto en el contenido y desarrollo de los temas como en la elaboración de los mismos. Por ello, es más susceptible que otros libros de contener alguna incorrección; de hecho, en el libro original publicado en inglés a finales de 1997 ya se han detectado algunos errores que se han procedido a corregir en esta edición en español.

Por este motivo se ha habilitado la siguiente dirección de correo electrónico con el fin de que cualquier lector de la obra pueda enviar sus opiniones y comentarios, lo cual nos ayudará a mejorar el resultado de la misma:

CSOOErrata@mail.yahoo.com

Todas las mensajes con sugerencias y comentarios serán bien recibidos y tenidos en cuenta por la editorial, que agradece de antemano el interés y la ayuda mostrados por los lectores.

EL EDITOR

Parte A

Los problemas

La parte A define los objetivos que perseguimos, examinando con detalle el concepto de calidad del software, y para aquellos lectores que no les disgusta conocer la trama de la película antes de verla, ofrece una visión preliminar concentrada de los aspectos más relevantes de la tecnología de objetos.

Calidad del software



*E*n la ingeniería se busca la calidad; la ingeniería del software es la producción de software de calidad. Este libro introduce un conjunto de técnicas para lograr mejoras significativas en la calidad de los productos software.

Antes de estudiar estas técnicas se deben clarificar sus objetivos, describiendo la calidad del software como una combinación de varios factores. Este capítulo analiza algunos de estos factores, muestra dónde se necesitan mejoras urgentes y señala las direcciones donde se buscarán soluciones a lo largo del camino que recorre este libro.

1.1. FACTORES EXTERNOS E INTERNOS

Todos deseamos que nuestros sistemas de software sean rápidos, fiables, fáciles de usar, legibles, modulares, estructurados y así sucesivamente. Pero estos adjetivos describen dos tipos de cualidades diferentes.

Por una parte, se consideran cualidades tales como la velocidad o la facilidad de uso, cuya presencia o ausencia en un producto de software puede ser detectada por sus usuarios. Estas propiedades pueden ser denominadas factores de calidad externos.

Bajo el término “usuarios” incluiríamos no sólo a las personas que realmente interactúan con los productos finales, como un empleado de una compañía aérea que usa un sistema de reserva de vuelos, sino también a aquellos que compran el software o contratan su desarrollo, tales como un ejecutivo de la compañía aérea encargado de adquirir o contratar el desarrollo del sistema de reserva de vuelos. De modo que una propiedad como la facilidad con la que el software se adapta a los cambios en las especificaciones —definida más adelante como *extensibilidad*— formaría parte de la categoría de factores externos aun cuando no sea de interés inmediato para los “usuarios finales” tales como el empleado que hace las reservas.

Otras cualidades aplicables a un producto de software, como la modularidad o legibilidad son factores internos, perceptibles sólo por profesionales de la informática que tienen acceso al código fuente.

En última instancia, sólo importan los factores externos. Si se usa un navegador Web o se vive cerca de una planta nuclear controlada por computadora, importa poco que el software sea legible o modular si los gráficos tardan años en cargarse o si la introducción de datos erróneos hace explotar la planta. La clave para obtener los factores externos radica en los internos, para que los usuarios disfruten de las cualidades visibles, los diseñadores y los implementadores deben aplicar técnicas internas que aseguren las cualidades ocultas.

Los capítulos siguientes presentan un conjunto de técnicas modernas para obtener la calidad interna. Sin embargo, no se debe perder de vista la perspectiva global; las técnicas internas no son un fin en sí mismas sino un medio para alcanzar las cualidades externas del software. Por lo tanto, comenzaremos analizando los factores externos. El resto de este capítulo examina dichos factores.

1.2. UNA REVISIÓN DE LOS FACTORES EXTERNOS

Definiremos ahora el más importante de los factores externos de calidad, cuya obtención es la tarea central de la construcción de software orientado a objetos.

Corrección

Definición:

Corrección es la capacidad de los productos software para realizar con exactitud sus tareas, tal y como se definen en las especificaciones.

La corrección es la cualidad principal. Si un sistema no hace lo que se supone que debe hacer, poco importan el resto de consideraciones que hagamos sobre él —si es rápido, si tiene una bonita interfaz de usuario...

Pero esto es más fácil de decir que de lograr. Incluso el primer paso hacia la corrección es ya difícil: debemos ser capaces de especificar los requisitos del sistema en una forma precisa, lo que es en sí mismo una ardua tarea.

Los métodos que aseguran la corrección serán usualmente **condicionales**. Un sistema de software importante, incluso uno pequeño según los estándares de hoy, implica a tantas áreas que sería imposible garantizar su corrección manejando todas las componentes y propiedades en un solo nivel. En cambio, es necesario una solución multinivel, en la que cada nivel confía en la corrección de los inferiores:

*Capas
en el desarrollo
de software*



En la solución condicional de la corrección, sólo hay que preocuparse en garantizar que cada nivel sea correcto *bajo el supuesto* de que los niveles inferiores son correctos. Ésta es la única técnica realista, puesto que consigue una separación de áreas de interés y permite concentrarse en cada etapa en un conjunto limitado de problemas. No se puede comprobar de un modo provechoso que un programa escrito en un lenguaje de alto nivel X es correcto a menos que seamos capaces de asumir que el compilador que tenemos a mano implementa correctamente a X. Esto no significa que hay que creer ciegamente en el compilador, sino que sencillamente se separan las dos componentes del problema: la corrección del compilador y la corrección del programa relativa a la semántica del lenguaje.

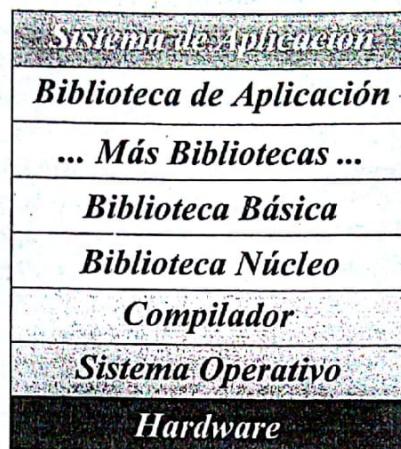
En el método descrito en este libro todavía intervienen más niveles: el desarrollo de software se basará en bibliotecas de componentes reutilizables, las cuales pueden utilizarse en muchas aplicaciones diferentes.

La solución condicional también se aplicará aquí: hay que asegurarse de que las bibliotecas sean correctas y, de forma separada, que la aplicación es correcta asumiendo que las bibliotecas lo son.

Muchos que empiezan en esto de la programación, cuando se les presenta el problema de la corrección del software, piensan en la prueba y depuración. Se puede ser más ambicioso: en capítulos posteriores se explorarán varias técnicas, en particular los tipos de datos

y las aserciones, que ayudan a construir software correcto desde el principio —en lugar de depurarlo para hacerlo correcto. Por supuesto, la depuración y la prueba siguen siendo indispensables como medio de doble comprobación del resultado.

Niveles en un proceso de desarrollo que incluye reutilización



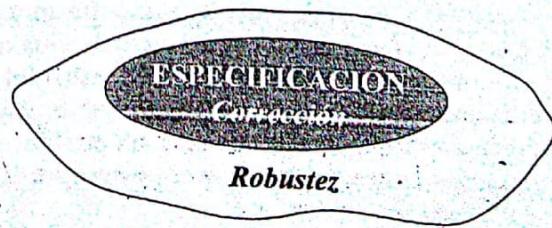
Es posible ir más allá y adoptar un enfoque completamente formal en la construcción de software. Este libro se queda corto en tal objetivo, como se sugiere por algunos términos tímidos como “prueba”, “garantiza” y “asegura” utilizados anteriormente con preferencia a la palabra “demostración”. De todos modos, muchas de las técnicas que se describen en capítulos posteriores provienen directamente del trabajo sobre técnicas matemáticas para la especificación y verificación formal de programas y van por el largo camino que pretende asegurar el ideal de corrección.

Robustez

Robustez es la capacidad de los sistemas software de reaccionar apropiadamente ante condiciones excepcionales.

La robustez complementa la corrección. La corrección tiene que ver con el comportamiento de un sistema en los casos previstos por su especificación; la robustez caracteriza lo que sucede fuera de tal especificación.

Robustez versus corrección



Como se refleja en las palabras de esta definición, la robustez es por naturaleza una noción más difusa que la corrección. Puesto que tiene que ver aquí con casos no previstos por la especificación, no es posible decir, como con la corrección, que el sistema debería “realizar sus tareas” en tal caso; donde las tareas son conocidas, el caso excepcional formaría parte de la especificación y regresaríamos al terreno de la corrección.

Sobre manejo de excepciones, véase Capítulo 12.

Esta definición de “caso excepcional” será útil de nuevo cuando se estudie el manejo de excepciones. Esto implica que las nociones de normal y excepcional son siempre relativas a cierta especificación; un caso excepcional es simplemente un caso no previsto por la especificación. Si se amplía la especificación, los casos excepcionales se convierten en normales —incluso si corresponden a eventos tales como entrada de datos errónea por parte del usuario que sería preferible que no ocurriesen. “Normal” en este sentido no significa “deseable” sino simplemente “planificado durante el diseño del software”. Aunque en principio puede parecer paradójico que una entrada errónea sea llamada un caso normal, cualquier otro planteamiento tendría que basarse en un criterio subjetivo y sería por tanto poco útil.

Siempre habrá casos en que la especificación no contemple explícitamente. El papel del requisito de robustez es asegurar que si tal caso surgiese el sistema no causará eventos catastróficos; debería producir mensajes de error apropiados, terminar su ejecución limpiamente o entrar en el llamado modo de “degradación elegante”.

Extensibilidad

Extensibilidad es la facilidad de adaptar los productos de software a los cambios de especificación

El software se supone que es *soft* (blando), y realmente lo es en un principio; nada es más fácil de cambiar que un programa si se tiene acceso a su código fuente. Sólo hay que usar el editor de textos favorito.

El problema de la extensibilidad es un problema de escala. Para programas pequeños realizar cambios no es normalmente una tarea difícil; pero a medida que el software crece comienza a ser cada vez más difícil de adaptar. A menudo, un gran sistema software es visto por las personas encargadas de su mantenimiento como un castillo gigante de naipes en el que sacar un elemento puede provocar que todo el edificio se derrumbe.

La extensibilidad es necesaria porque en la base de todo software encontramos algún fenómeno humano y de ahí su volatilidad. El caso obvio del software de gestión o software de negocios (MIS, Management Information Systems), donde la aprobación de una ley o una adquisición de la compañía pueden invalidar de pronto los supuestos sobre los que el sistema se apoyaba, no es especial; incluso en la programación científica, donde podemos suponer que las leyes de la física no cambian de un mes para otro, cambiará la interpretación y modelado de los sistemas físicos.

Las técnicas tradicionales de ingeniería de software no tienen suficientemente en cuenta el cambio y se basan en una visión ideal del ciclo de vida del software donde en la etapa inicial de análisis se congelan los requisitos y el resto del proceso se dedica al diseño y la construcción de una solución. Esto es comprensible: en la evolución de la disciplina, la primera tarea fue desarrollar buenas técnicas para establecer y resolver problemas particulares, antes de lamentarnos sobre qué hacer si el problema cambia hay que dedicarse a resolverlo. Pero ahora que las técnicas básicas de ingeniería de software están en su verdadero sitio se ha convertido en esencial reconocer y señalar este aspecto central. El cambio es omnipresente en el desarrollo del software: cambios de los requisitos, de nuestra comprensión de los requisitos, de los algoritmos, de la representación de los datos, de las técnicas de implementación. Ofrecer soporte para los cambios es un objetivo básico de la tecnología de objetos y un tema recurrente a lo largo de este libro.

Aunque muchas de las técnicas que mejoran la extensibilidad se pueden explicar con pequeños ejemplos o en cursos introductorios, su relevancia sólo se ve con claridad en los grandes proyectos. Hay dos principios esenciales para mejorar la extensibilidad:

- *Simplicidad del diseño*: una arquitectura simple siempre será más fácil de adaptar a los cambios que una compleja.
- *Descentralización*: cuanto más autónomos sean los módulos, más alta es la probabilidad de que un cambio simple afecte a un solo módulo, o a un número pequeño de módulos, en lugar de provocar una reacción en cadena de cambios en el sistema completo.

El método orientado a objetos es, antes que cualquier otra cosa, un método de arquitectura de sistemas que ayuda a los diseñadores a producir sistemas cuya estructura es a un mismo tiempo simple (incluso para grandes sistemas) y descentralizada. Simplicidad y descentralización serán temas recurrentes en las discusiones que conducirán a los principios orientados a objetos en los capítulos siguientes.

Reutilización

Definición: reutilización

Reutilización es la capacidad de los elementos de software de servir para la construcción de muchas aplicaciones diferentes.

La necesidad de la reutilización surge de la observación de qué los sistemas software a menudo siguen patrones similares; debería ser posible explotar esta similitud y evitar reinventar soluciones a problemas que ya han sido encontradas con anterioridad. Capturando tal patrón, un elemento de software reutilizable se podrá aplicar en muchos desarrollos diferentes.

La reutilización tiene una influencia sobre todos los demás aspectos de la calidad del software, ya que al resolver el problema de la reutilización se tendrá que escribir menos software y en consecuencia se podrán dedicar entonces mayores esfuerzos (por el mismo costo total) a mejorar los otros factores, tales como la corrección y la robustez.

Aquí tenemos de nuevo un aspecto que la visión tradicional del ciclo de vida del software no ha reconocido adecuadamente y por la misma razón histórica: deben encontrar primero las formas de resolver un problema antes de preocuparse por aplicar la solución a otros problemas. Pero con el crecimiento del software y sus intentos de convertirse en una verdadera industria, la necesidad de la reutilización se ha convertido en un asunto que apremia.

La reutilización desempeñará un papel central en las discusiones de los capítulos siguientes, uno de los cuales está de hecho dedicado por completo a un examen en profundidad de este factor de la calidad, sus beneficios concretos y las cuestiones que plantea.

Capítulo 4.

Compatibilidad

Definición: compatibilidad

Compatibilidad es la facilidad de combinar unos elementos de software con otros.

La compatibilidad es importante debido a que los sistemas software no se desarrollan en el vacío: necesitan interactuar con otros. Pero con mucha frecuencia los sistemas tienen

Aunque muchas de las técnicas que mejoran la extensibilidad se pueden explicar con pequeños ejemplos o en cursos introductorios, su relevancia sólo se ve con claridad en los grandes proyectos. Hay dos principios esenciales para mejorar la extensibilidad:

- *Simplicidad del diseño*: una arquitectura simple siempre será más fácil de adaptar a los cambios que una compleja.
- *Descentralización*: cuanto más autónomos sean los módulos, más alta es la probabilidad de que un cambio simple afecte a un solo módulo, o a un número pequeño de módulos, en lugar de provocar una reacción en cadena de cambios en el sistema completo.

El método orientado a objetos es, antes que cualquier otra cosa, un método de arquitectura de sistemas que ayuda a los diseñadores a producir sistemas cuya estructura es a un mismo tiempo simple (incluso para grandes sistemas) y descentralizada. Simplicidad y descentralización serán temas recurrentes en las discusiones que conducirán a los principios orientados a objetos en los capítulos siguientes.

Reutilización

Definición: Reutilización
Reutilización es la capacidad de los elementos de software de servir para la construcción de muchas aplicaciones diferentes.

La necesidad de la reutilización surge de la observación de que los sistemas software a menudo siguen patrones similares; debería ser posible explotar esta similitud y evitar reinventar soluciones a problemas que ya han sido encontradas con anterioridad. Capturando tal patrón, un elemento de software reutilizable se podrá aplicar en muchos desarrollos diferentes.

La reutilización tiene una influencia sobre todos los demás aspectos de la calidad del software, ya que al resolver el problema de la reutilización se tendrá que escribir menos software y en consecuencia se podrán dedicar entonces mayores esfuerzos (por el mismo costo total) a mejorar los otros factores, tales como la corrección y la robustez.

Aquí tenemos de nuevo un aspecto que la visión tradicional del ciclo de vida del software no ha reconocido adecuadamente y por la misma razón histórica: deben encontrar primero las formas de resolver un problema antes de preocuparse por aplicar la solución a otros problemas. Pero con el crecimiento del software y sus intentos de convertirse en una verdadera industria, la necesidad de la reutilización se ha convertido en un asunto que apremia.

La reutilización desempeñará un papel central en las discusiones de los capítulos siguientes, uno de los cuales está de hecho dedicado por completo a un examen en profundidad de este factor de la calidad, sus beneficios concretos y las cuestiones que plantea.

Compatibilidad

Definición: compatibilidad

Compatibilidad es la facilidad de combinar unos elementos de software con otros.

La compatibilidad es importante debido a que los sistemas software no se desarrollan en el vacío: necesitan interactuar con otros. Pero con mucha frecuencia los sistemas tienen

dificultades para interactuar porque hacen suposiciones contradictorias sobre el resto del mundo. Un ejemplo es la amplia variedad de formatos de archivos soportados por muchos sistemas operativos. Un programa puede usar directamente como entrada los resultados de otro sólo si los formatos de archivos son compatibles.

La falta de compatibilidad puede ocasionar desastres. He aquí un caso extremo:

*San José (Calif.)
Mercury News,
20 de julio de 1992.
Señalado
en el grupo
de noticias
«comp.risks»
de Usenet, 13
de julio de 1992.*

DALLAS — La semana pasada, AMR, la compañía matriz de American Airlines, Inc., dijo que había recibido una estocada tratando de desarrollar un moderno sistema a escala industrial que pudiera manejar también reservas de hoteles y automóviles.

AMR cortó el desarrollo de su nuevo sistema de reservas Confirm sólo unas semanas después de cuando se suponía que iba a comenzar a ocuparse de las transacciones de sus patrocinadores Budget Rent-A-Car, Hilton Hotels Corp. y Marriot Corp. La suspensión de un proyecto de 125 millones de dólares y de 4 años de trabajo se tradujo en pérdidas por valor de 165 millones de dólares y echó por tierra la reputación de la compañía como líder en tecnología de viajes. [...]

A finales de enero, los responsables de Confirm descubrieron que la labor de más de 200 programadores, analistas de sistemas e ingenieros había sido aparentemente en balde. Las principales piezas del impresionante proyecto —que requerían 47.000 páginas de descripción— habían sido desarrolladas de forma separada por métodos diferentes. Cuando se unieron, no funcionaban unas con otras. Cuando los desarrolladores intentaban conectar las partes entre sí, no podían. Los diferentes “módulos” no podían tomar la información que necesitaban de la otra parte de la conexión.

AMR Information Services despidió a ocho jefes de proyecto, incluyendo el líder del equipo. [...] El pasado junio, Budget y Hilton anunciaron que se retiraban.

La clave de la compatibilidad recae en la homogeneidad del diseño y en acordar convenciones estándares para la comunicación entre programas. Los enfoques incluyen:

- Formatos de archivo estándares, como en el sistema Unix, donde cualquier archivo de texto es simplemente una secuencia de caracteres.
- Estructuras de datos estándares como en los sistemas Lisp, donde tanto los datos como los programas, se representan mediante árboles binarios (llamados listas en Lisp).
- Interfaces de usuario estándares, como en las diferentes versiones de Windows, OS/2 y MacOS donde todas la herramientas utilizan un solo paradigma para la comunicación con el usuario, basado en componentes estándares tales como ventanas, iconos, menús, etcétera.

Sobre tipos abstractos de datos, véase capítulo 6.

Se han obtenido soluciones más generales definiendo protocolos estándares de acceso a todas las entidades importantes manipuladas por el software. Ésta es la idea que está detrás de los tipos de datos abstractos y del enfoque orientado a objetos, así como de los denominados protocolos *middleware* tales como CORBA y Microsoft OLE-COM (ActiveX).

Eficiencia

Definición: eficiencia

Eficiencia es la capacidad de un sistema software para exigir la menor cantidad posible de recursos hardware tales como tiempo del procesador, espacio ocupado de memoria interna y externa o ancho de banda utilizado en los dispositivos de comunicación.

Casi sinónimo de eficiencia es la palabra “rendimiento”. La comunidad del software muestra dos tipos de actitud con relación a la eficiencia:

- Algunos desarrolladores¹ tienen una obsesión con las cuestiones del rendimiento y le dedican una gran cantidad de esfuerzos a presuntas optimizaciones.
- Por otro lado, existe la tendencia de soslayar las cuestiones de eficiencia, como se evidencia en las frases de la industria “hágalo correcto antes de hacerlo rápido” y “de todos modos los modelos de computadoras del año que viene van a ser un 50% más rápidos”.

No es extraño ver a la misma persona mostrando estas dos actitudes en momentos diferentes, como un caso de desdoblamiento de la personalidad en el dominio del software (Dr. Abstracto y Mister Microsegundo).

¿Dónde está la verdad? Está claro que los desarrolladores a menudo muestran una preocupación exagerada con la micro-optimización. Como ya se ha señalado, la eficiencia no se preocupa mucho si el software no es correcto (lo que sugiere un nuevo refrán, “*no importa cuán rápido es hasta que no esté correcto*”, que es parecido al anterior pero no exactamente lo mismo). De manera más general, la preocupación por la eficiencia debe sopesarse con otros objetivos tales como la extensibilidad y la reutilización; optimizaciones extremas pueden hacer al software tan especializado que limite el cambio y la reutilización. Es más, la potencia creciente del hardware de las computadoras nos permite tener una actitud más relajada con respecto a tratar de ganar hasta el último byte o microsegundo.

Sin embargo, todo esto no disminuye la importancia de la eficiencia. A nadie le gusta esperar para obtener las respuestas de un sistema interactivo o tener que comprar más memoria para poder ejecutar un programa. En realidad las actitudes poco ceremoniosas con respecto al rendimiento incluyen muchas posturas; si el sistema final es tan lento o pesado que impide su utilización, aquellos que declaran que la “velocidad no es tan importante” no serán precisamente los últimos en quejarse.

Estas cuestiones reflejan lo que yo considero la característica principal de la ingeniería de software, y que no es probable que se quite de en medio pronto: la construcción de software es difícil precisamente porque requiere tener en cuenta muchos requisitos diferentes, algunos de los cuales, como la corrección, son abstractos y conceptuales, mientras que otros, como la eficiencia, son concretos y acotados por las propiedades del hardware.

Para algunos del lado de las ciencias, el desarrollo de software es una rama de las matemáticas; para algunos ingenieros, es una rama de la tecnología aplicada. En realidad es ambas cosas. El desarrollador de software debe reconciliar los conceptos abstractos con sus implementaciones concretas, la matemática de la computación correcta con las restricciones de tiempo y espacio que se derivan de las leyes físicas y de las limitaciones de la tecnología actual del hardware. Esta necesidad de reconciliar los ángeles y las bestias puede ser el desafío central de la ingeniería del software.

El constante aumento de potencia de las computadoras, siendo impresionante, no es excusa para infravalorar la eficiencia, al menos por tres razones:

- El que compra una computadora más grande y más rápida desea ver algunos beneficios de esa potencia extra —como poder tratar nuevos problemas, procesar problemas anteriores más rápidamente o procesar versiones mayores de los problemas anteriores en la misma cantidad de tiempo. Usar la nueva computadora para procesar los problemas anteriores en el mismo tiempo, ¡no!
- Uno de los efectos más visibles de los avances en la potencia de las computadoras está en *incrementar* la ventaja de los buenos algoritmos sobre los malos. Supongamos que una nueva máquina es dos veces más rápida que la anterior. Sea n el tamaño del problema a resolver y N el máximo n que puede manejar un cierto algoritmo en un tiempo dado. Entonces si el algoritmo es $O(n)$, es decir, se ejecuta en un tiempo proporcional a n , la

¹ El autor utiliza el término *developer* con frecuencia en este libro para nombrar a las personas que se dedican a las diferentes actividades involucradas en el desarrollo de software. Traducirlo por *programador* puede hacer pensar al lector que se refiere sólo a escribir código en un lenguaje de programación, por lo que, aunque no es una acepción formalmente aceptada en castellano, se ha utilizado en esta traducción el término *desarrollador* (N. del T.).

nueva máquina será capaz de tratar problemas de tamaño 2^*N . Para un algoritmo que se ejecute en tiempo $O(n^2)$ la nueva máquina sólo permitirá un aumento del 41% en el tamaño N . Un algoritmo que sea $O(2^n)$, como algunos algoritmos combinatorios para búsquedas exhaustivas, sólo añadirían uno al N —lo que no es mucho aumento por su dinero.

- En algunos casos la eficiencia puede afectar a la corrección. Una especificación puede establecer que la respuesta de la computadora a un cierto evento debe ocurrir antes de cierta cantidad de tiempo especificada; por ejemplo, una computadora de vuelo debe estar preparada para detectar y procesar un mensaje del sensor de una válvula lo suficientemente rápido como para llevar a cabo una acción correctiva. Esta conexión entre eficiencia y corrección no está sólo restringida a las aplicaciones conocidas como de “tiempo real”; pocas personas estarían interesadas en un modelo de predicción climática que tarde veinticuatro horas en pronosticar el tiempo del día siguiente.

Otro ejemplo, aunque por supuesto menos crítico, me ha irritado frecuentemente: un sistema de ventanas que estuve usando durante un tiempo era a veces tan lento en detectar que el cursor del ratón se había movido de una ventana a otra, que los caracteres tecleados para una cierta ventana podían ocasionalmente aparecer en la otra.

En este caso una limitación del rendimiento provoca una violación de la especificación, es decir, de la corrección, pues lo que puede parecer inocuo en las aplicaciones cotidianas puede causar desagradables consecuencias: piense en lo que pasaría si las dos ventanas se usan para enviar mensajes electrónicos a dos destinatarios diferentes. Por menos de esto se han roto matrimonios, e incluso se han declarado guerras.

Debido a que este libro se centra en los conceptos de la ingeniería de software orientada a objetos, no en aspectos de implementación, solamente unas pocas secciones tratan explícitamente los costes de rendimiento asociados. Pero el asunto de la eficiencia estará ahí a lo largo de todo el libro. Siempre que la discusión presente una solución orientada a objetos a algún problema, se puede estar seguro que la solución será no sólo elegante sino también eficiente; siempre que se introduce un nuevo mecanismo de O-O, ya sea la recolección de basura (y otras técnicas de gestión de memoria en los sistemas para la computación orientada a objetos), ligadura dinámica, genericidad o herencia repetida, estará basado en el conocimiento de que dicho mecanismo se puede implementar a un coste razonable en tiempo y espacio; siempre que sea apropiado se mencionarán las consecuencias en el rendimiento de las técnicas estudiadas.

La eficiencia es sólo uno de los factores de la calidad; no deberíamos permitirle (como hacen algunos de la profesión) gobernar nuestras vidas de ingenieros. Pero es un factor que debe tomarse en cuenta, lo mismo en la construcción de un sistema software que en el diseño de un lenguaje de programación. Si usted descarta el rendimiento, el rendimiento lo descartará a usted.

Portabilidad (transportabilidad)

Definición: portabilidad

Portabilidad (transportabilidad) es la facilidad de transferir los productos software entre sistemas hardware y software diferentes.

La portabilidad tiene que ver con las variaciones no sólo del hardware físico sino más generalmente de la **máquina hardware-software**, la que realmente programamos y que incluye el sistema operativo, el sistema de ventanas (si se emplea) y otras herramientas fundamentales. En el resto del libro la palabra “plataforma” se usará para denotar un tipo de máquina hardware-software. Un ejemplo de plataforma es “Intel X86 con Windows NT” (conocida como “Wintel”).

Muchas de las incompatibilidades existentes entre las plataformas son injustificadas, y para un observador ingenuo a veces la única explicación parece ser una conspiración para martirizar a la humanidad en general y a los programadores en particular. Sin embargo, independientemente de las causas, esta diversidad convierte la portabilidad en un asunto primordial tanto para los que desarrollan como para los que usan el software.

Facilidad de uso

Facilidad de uso es la facilidad con la cual personas con diferentes formaciones y aptitudes pueden aprender a usar los productos software y aplicarlos a la resolución de problemas. También cubre la facilidad de instalación, de operación y de supervisión.

La definición insiste en los diferentes niveles de experiencia de los posibles usuarios. Este requisito plantea uno de los mayores retos de los diseñadores de software preocupados por la facilidad de uso: cómo proporcionar explicaciones y guías detalladas a los usuarios novatos sin fastidiar a los usuarios expertos que quieren ir directos al grano.

Al igual que con muchas de las otras cualidades discutidas en este capítulo, una de las claves de la facilidad de uso es la simplicidad estructural. Un sistema bien diseñado, construido de acuerdo a una estructura clara y bien pensada, tiende a ser más fácil de aprender y usar que uno confuso. Por supuesto, la condición no es suficiente (lo que es simple y claro para el diseñador puede ser difícil y oscuro para los usuarios, especialmente si se expone en términos de diseño en lugar de en términos comprensibles para el usuario), pero ayuda considerablemente.

Ésta es una de las áreas donde el método orientado a objetos es particularmente productivo; muchas técnicas O-O, que en principio están dirigidas al diseño y a la implementación también producen ideas nuevas eficaces sobre interfaces que ayudan a los usuarios finales. En capítulos posteriores se mostrarán varios ejemplos.

Los diseñadores de software preocupados por la facilidad de uso harán bien en considerar con cierto recelo el precepto usado con más frecuencia en la literatura sobre interfaces de usuario, de un artículo inicial de Hansen: *conocer al usuario*. El argumento radica en que un buen diseñador debe hacer un esfuerzo para comprender a la comunidad de usuarios a la que se destina el sistema. Este punto de vista ignora una de las características de los sistemas que gozan de mayor éxito: que siempre sobrepasan la audiencia inicial. (Dos ejemplos viejos y famosos son Fortran, concebido inicialmente como una herramienta para resolver el problema de una pequeña comunidad de ingenieros y científicos de la programación de la IBM 704, y Unix concebido para uso interno de los Laboratorios Bell.) Un sistema diseñado para un grupo específico se basará en supuestos que simplemente no se cumplen para una gran audiencia.

Los buenos diseñadores de interfaces siguen una política más prudente. Hacen las menos suposiciones posibles sobre los usuarios. Cuando se diseña un sistema interactivo, se debe esperar que los usuarios sean miembros de la raza humana y que sepan leer, mover un ratón, presionar un botón y teclear (lentamente); no mucho más. Si el software está dirigido a un área especializada de aplicación, se puede dar por supuesto que los usuarios están familiarizados con sus conceptos básicos. Pero incluso esto es arriesgado. Parafraseando a la inversa la advertencia de Hansen:

Principio de diseño de la interfaz de usuario

No suponga que conoce al usuario mejor que él mismo.

Funcionalidad

Funcionalidad es el conjunto de posibilidades que proporciona un sistema.

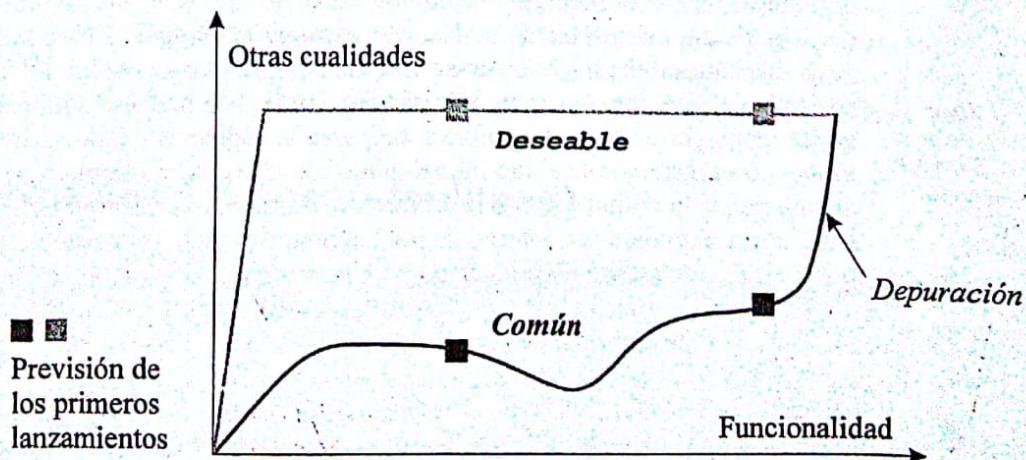
Uno de los problemas más difíciles a los que se enfrenta un jefe de proyecto es conocer cuánta funcionalidad es suficiente. La presión para ofrecer más facilidades, conocida en el lenguaje de la industria como *featurism*², está constantemente presente. Sus consecuencias son malas para los proyectos internos, donde las presiones vienen de los usuarios de la misma compañía, y son peores para los productos comerciales, ya que la parte más destacada de los análisis comparativos suele ser una tabla donde se enumeran una por una las propiedades que ofrecen los distintos productos analizados.

El *featurism* es realmente la combinación de dos problemas, uno más difícil que el otro. El problema más fácil es la pérdida de consistencia como consecuencia de estar añadiendo nuevas propiedades, lo que puede afectar a su facilidad de uso. Los usuarios se quejan con razón de que toda la parafernalia que acompaña a una nueva versión de un producto lo hace tremadamente complejo. Tales comentarios no deberían preocuparnos en exceso, puesto que las nuevas propiedades no surgen de la nada: la mayor parte de las veces han sido solicitadas por los usuarios —otros usuarios. Lo que a unos les puede resultar algo superfluo puede ser una facilidad indispensable para otros.

La solución aquí es trabajar una y otra vez sobre la consistencia del producto global, tratando de que todo encaje en un molde general. Un buen producto de software está basado en un número pequeño de potentes ideas; incluso si tiene muchas propiedades especializadas, éstas deberían explicarse como consecuencia de los conceptos básicos. El “gran plan” debe estar visible y todo debería ocupar su sitio dentro de él.

El problema más difícil es evitar centrar tanto la atención en las propiedades hasta el punto de olvidar otras cualidades. Es común que los proyectos cometan tal error, una situación expresada gráficamente por Roger Osmond mediante los dos caminos posibles para completar un proyecto:

Curvas de Osmond; en [Osmond 1995]



La curva inferior es bastante común: en la frenética carrera por añadir más propiedades, el desarrollo se sale de la pista de la calidad global. La fase final que intenta que las cosas

² El término en inglés es *featurism* que significa «muchos rasgos distintos». Este término parodia fonéticamente al término *futurism* (futurismo). Este parecido fonético no se puede traducir al castellano (N. del T.).

queden bien, puede ser larga y estresante. Si bajo la presión de los usuarios o los competidores usted está forzado a lanzar el producto prematuramente —en las etapas marcadas en la figura con cuadrados en negro— el resultado puede ser perjudicial para su reputación.

Lo que Osmond sugiere (la curva superior) es, ayudado por las técnicas de desarrollo O-O que aumentan la calidad, mantener constante el nivel de calidad a lo largo del proyecto para todos los aspectos incluyendo la funcionalidad. No se deben comprometer la fiabilidad, la extensibilidad o los demás factores: hay que rechazar el pasar a considerar nuevas propiedades hasta que no se esté satisfecho con las que se tiene.

Este método es difícil de aplicar en el día a día debido a las presiones mencionadas, pero nos conduce a un proceso de software más efectivo y a menudo hacia un mejor producto final. Incluso si el resultado final es el mismo, como se asume en la figura, éste debe alcanzarse antes (aunque la figura no muestra el tiempo). Seguir el camino sugerido significa también que la decisión de lanzar una versión inicial —en uno de los puntos marcados por los dos cuadrados superiores de la figura— será, si no más fácil, al menos más simple: estará basado en valorar si el producto cubre un área lo suficientemente amplia del conjunto completo de características para atraer a los consumidores previstos en lugar de alejarlos. La cuestión “¿es lo suficientemente bueno?” (al igual que “el sistema no fallará”) no debería ser un factor.

Como conoce cualquier lector que haya dirigido un proyecto de software, es más fácil estar de acuerdo con este consejo que aplicarlo. Pero todo proyecto debería seguir el enfoque representado por la mejor de las dos curvas de Osmond. Funciona bien con el *modelo de agrupación* introducido en un capítulo posterior como esquema general para un desarrollo orientado a objetos disciplinado.

Oportunidad

• Oportunidad es la capacidad de un sistema de software de ser lanzado cuando los usuarios lo desean, o antes.

La oportunidad es una de las mayores frustraciones de nuestra industria. Un gran producto software que aparece demasiado tarde puede no alcanzar su objetivo. Esto es cierto en otras industrias también, pero pocas evolucionan tan rápidamente como el software.

La oportunidad es todavía, para grandes proyectos, un fenómeno poco común. Cuando Microsoft anunció que la última versión de su principal sistema operativo, que llevaba realizando varios años, saldría al mercado un mes antes de lo previsto, el suceso fue lo suficientemente relevante como para encabezar los titulares de *Computer World* (a la cabeza de un artículo que llamaba la atención sobre las largas demoras que afectaron a proyectos anteriores).

Otras cualidades

Además de las cualidades analizadas hasta ahora, existen otras que afectan a los usuarios de sistemas software y a la gente que compra estos sistemas o encarga su desarrollo. En particular:

- **Verificabilidad** es la facilidad para preparar procedimientos de aceptación, especialmente datos de prueba y procedimientos para detectar fallos y localizar errores durante las fases de validación y operación.
- **Integridad** es la capacidad de los sistemas software de proteger sus diversos componentes (programas, datos, etc.) contra modificaciones y accesos no autorizados.

- **Reparabilidad** es la capacidad para facilitar la reparación de los defectos.
- **Economía**, junto con la oportunidad, es la capacidad que un sistema tiene de complementarse con el presupuesto asignado o por debajo del mismo.

Sobre la documentación

En una lista de los factores de calidad del software, uno podría esperar encontrar la presencia de una buena documentación como uno de los requisitos. Pero éste no es un factor de calidad separado; la necesidad de documentación es una consecuencia de otros factores de calidad vistos anteriormente. Se pueden distinguir tres tipos de documentación:

- La necesidad de documentación *externa*, que permite a los usuarios conocer la potencia de un sistema y usarlo convenientemente, es una consecuencia de la definición de facilidad de uso.
- La necesidad de documentación *interna*, que permite a los desarrolladores de software comprender la estructura e implementación de un sistema, es una consecuencia del requisito de extensibilidad.
- La necesidad de documentación de la *interfaz de un módulo*, que permite a los desarrolladores de software comprender las funciones proporcionadas por un módulo sin tener que comprender su implementación, es una consecuencia del requisito de reutilización. También se desprende de la extensibilidad, ya que una documentación de la interfaz de un módulo permite determinar cuándo cierto cambio necesario afecta a un determinado módulo.

En lugar de tratar la documentación como un producto propio del software, es preferible producir software lo más autodocumentado posible. Esto se aplica a los tres tipos de documentación:

- Incluyendo facilidades de “ayuda” en línea y siguiendo normas para interfaces claras y consistentes, se alivia la tarea de los autores de los manuales de usuario y de otras formas de documentación externa.
- Un buen lenguaje de implementación puede eliminar muchas de las necesidades de documentación interna si favorece la claridad y la estructura. Éste será uno de los requisitos principales de la notación orientada a objetos que se desarrolla a lo largo de este libro.
- La notación soportará la ocultación de información y otras técnicas (tales como las aserciones) para separar la interfaz de los módulos de su implementación. Será posible entonces utilizar herramientas para producir automáticamente documentación de la interfaz del módulo a partir del texto de los módulos. Esto también es uno de los temas que se estudian en detalle en capítulos posteriores.

Todas estas técnicas reducen la importancia de la documentación tradicional, aunque no podemos esperar que la eliminen completamente.

Compromisos³

En esta revisión de los factores externos de calidad del software se han encontrado requisitos que pueden entrar en conflicto unos con otros.

³ El término inglés «tradeoffs» hace referencia a aquellas situaciones en las que al solucionar un problema, entran en conflicto varios factores y es necesario optar por alguno de ellos (*N. del T.*).

¿Cómo se puede tener *integridad* sin introducir protecciones de varias clases, lo cual inevitablemente perjudica la *facilidad de uso*? La *economía*, a menudo va contra la *funcionalidad*. La *eficiencia* óptima requeriría una adaptación perfecta a un entorno particular de hardware y software, lo cual es contrario a la *portabilidad*. Una perfecta adaptación a una especificación puede ir en contra de la *reutilización*, que incita a resolver problemas más generales que el planteado inicialmente. Las presiones de *oportunidad* pueden tentarnos para que utilicemos técnicas de Desarrollo Rápido de Aplicaciones (Rapid Application Development, RAD), cuyos resultados pueden no hacerle mucha gracia a la *extensibilidad*.

Aunque en muchos casos es posible encontrar una solución que reconcilie factores aparentemente en conflicto, a veces es necesario tomar soluciones de compromiso. Con bastante frecuencia, los desarrolladores realizan estas decisiones implícitamente, sin tiempo para examinar las características involucradas y las diversas opciones disponibles; la eficiencia tiende a ser el factor dominante en este tipo de decisiones silenciosas. Un verdadero enfoque de ingeniería de software implica un esfuerzo por establecer los criterios con claridad y tomar las decisiones conscientemente.

Si es necesario decidirse por algún factor de calidad, uno sobresale del resto: la corrección. Nunca hay justificación para comprometer la corrección en aras de otras cuestiones tales como la eficiencia. Si el software no lleva a cabo su función el resto es inútil.

Cuestiones clave

Todas las cualidades discutidas antes son importantes. Pero en el estado actual de la industria del software sobresalen cuatro:

- *Corrección y robustez*: es bastante difícil producir software sin defectos (*bugs*) y muy difícil corregir los defectos una vez que están ahí. Las técnicas para mejorar la corrección y la robustez son similares: enfoques más sistemáticos para la construcción de software; más especificaciones formales; más comprobaciones integradas a lo largo del proceso de construcción del software (no sólo la prueba y depuración después de hecho); mejores mecanismos en los lenguajes tales como la comprobación estática de tipos, las aserciones, la gestión automática de memoria y un tratamiento disciplinado de las excepciones, permiten a los desarrolladores asegurar los requisitos de corrección y de robustez y posibilitan herramientas para detectar inconsistencias antes de que se conviertan en defectos. Debido al parecido entre la corrección y la robustez, es conveniente usar un término más general, **fiabilidad** para cubrir ambos factores.
- *Extensibilidad y reutilización*: el software debe ser fácil de cambiar; los elementos de software que se produzcan deben ser aplicables de la forma más general posible, debiendo existir un gran inventario de componentes de propósito general que se puedan reutilizar cuando se vaya a desarrollar un nuevo sistema. Aquí de nuevo son útiles ideas similares para mejorar ambas cualidades: cualquier idea que ayude a producir arquitecturas más descentralizadas, cuyos componentes sean autocontenidos y se comuniquen sólo a través de canales restringidos y claramente definidos, será de ayuda. El término **modularidad** abarcará la reutilización y la extensibilidad.

Como se estudia en detalle en los capítulos posteriores, el método orientado a objetos puede mejorar significativamente estos cuatro factores de calidad —por eso es tan atractivo. También tiene contribuciones significativas que hacer en relación a los otros aspectos, en particular:

- *Compatibilidad*: el método promueve un estilo común de diseño y módulos e interfaces de sistema estándares, que ayudan a producir sistemas que pueden interactuar entre sí.
- *Portabilidad*: con el énfasis en la abstracción y la ocultación de información, la tecnología de objetos estimula a los diseñadores a distinguir entre las propiedades

de especificación y de implementación, facilitando los esfuerzos de portabilidad. Las técnicas de polimorfismo y ligadura dinámica harán posible escribir sistemas que se adapten automáticamente a distintos componentes tanto hardware como software de la máquina, por ejemplo diferentes sistemas de ventanas o diferentes sistemas de gestión de bases de datos.

- *Facilidad de uso:* la contribución de las herramientas O-O a los sistemas interactivos modernos y especialmente a sus interfaces de usuario es bien conocida, hasta el punto de que a veces oscurece otros aspectos (hay gente que llama "orientado a objetos" a cualquier sistema que use iconos, ventanas y entradas controladas por el ratón).
- *Eficiencia:* como se señala anteriormente, aunque a primera vista el poder extra de las técnicas orientadas a objetos parecen pasarnos factura, utilizar componentes reutilizables de alta calidad profesional, a menudo conlleva mejoras significativas en el rendimiento.
- *Oportunidad, economía y funcionalidad:* las técnicas O-O permiten a aquellos que las dominan producir software más rápidamente y a menor coste; facilitan la adición de funciones y pueden en sí mismas sugerir nuevas funciones a añadir.

A pesar de todos estos avances debemos tener presente que el método orientado a objetos no es una panacea y que muchos de los problemas habituales de la ingeniería de software se mantienen. Ayudar a señalar un problema no es lo mismo que resolverlo.

1.3. SOBRE EL MANTENIMIENTO DEL SOFTWARE

La lista de factores no incluye una cualidad que se menciona con frecuencia: *facilidad de mantenimiento*. Para entender por qué, se debe analizar más en detalle el concepto que subyace: el mantenimiento.

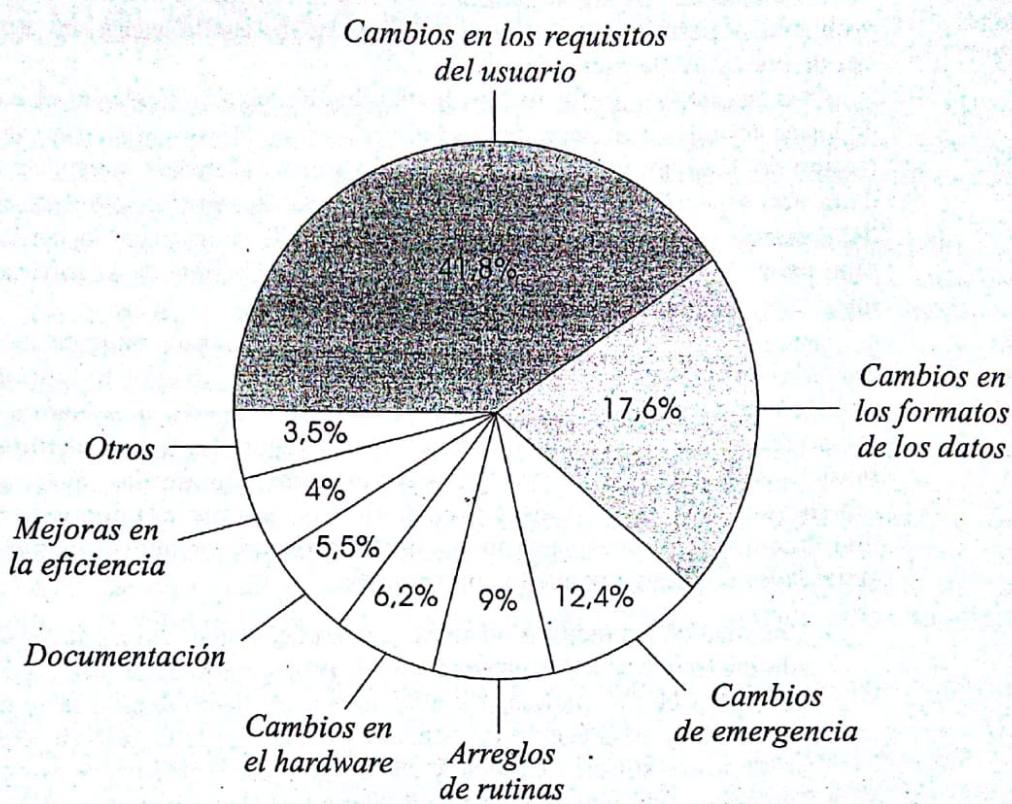
Mantenimiento es lo que sucede después de que se ha distribuido un producto de software. Las discusiones sobre la metodología de software tienden a centrar la atención en la fase de desarrollo; esto es lo que hacen los cursos introductorios de programación. Pero generalmente se estima que el 70% del coste del software se dedica al mantenimiento. Ningún estudio de la calidad del software puede ser satisfactorio si ignora este aspecto.

¿Qué significa "mantenimiento" en el caso del software? Un minuto de reflexión nos muestra que este término es inapropiado: un producto de software no se gasta por el uso repetido, y por tanto no necesita ser "mantenido" en la forma que lo es un automóvil o un televisor. En realidad, la palabra es utilizada en el mundillo del software para describir algunas actividades nobles y otras no tan nobles. La parte noble es la modificación: en la medida que las especificaciones de los sistemas informáticos cambian, reflejando los cambios en el mundo exterior, los sistemas deben cambiar también. La parte menos noble es la depuración a posteriori: quitar los errores que en principio nunca deberían haber existido.

La figura de la página siguiente, dibujada a partir de un estudio histórico desarrollado por Lientz y Swanson, arroja alguna luz sobre qué es lo que realmente cubre el término mantenimiento. El estudio realizó una encuesta en 487 instalaciones que desarrollaban software de todo tipo; aunque es algo antiguo, publicaciones más recientes confirman los mismos resultados en términos generales. Esta figura muestra el porcentaje de los costes de mantenimiento en cada una de las actividades identificadas por los autores.

Más de las dos quintas partes del coste está dedicado a extensiones y modificaciones solicitadas por los usuarios. Ésta es la que se ha llamado más arriba la parte noble del mantenimiento, que es una parte inevitable. La pregunta sin respuesta es qué parte del esfuerzo total de la industria se podría ahorrar si se construyese el software desde un principio dedicando más atención a la extensibilidad. Podemos con legitimidad esperar que la tecnología orientada a objetos nos ayude en esto.

Gráfica de los costes de mantenimiento.
Fuente:
[Lientz 1980]



Para otro ejemplo, véase «¿Cuántos problemas nos puede traer la inicial intermedia?». §6.2.

El segundo elemento en orden decreciente de porcentaje de coste resulta particularmente interesante: el efecto de los cambios en los formatos de los datos. Siempre que cambia la estructura física de los archivos y otros elementos de datos se deben adaptar los programas. Por ejemplo, cuando el servicio postal de los Estados Unidos, hace unos pocos años, introdujo el código postal “5 + 4” para las grandes compañías (usando nueve dígitos en lugar de cinco) hubo que reescribir numerosos programas que trabajaban con las direcciones y que contemplaban un código postal de cinco dígitos, un esfuerzo que la prensa cifró en varios cientos de millones de dólares.

Muchos lectores habrán recibido bonitos folletos sobre un conjunto de conferencias —no un sólo evento sino un ciclo de sesiones en muchas ciudades— dedicadas al “problema del año 2000”: cómo actualizar el sinfin de programas dependientes de fechas cuyos autores ni por un momento imaginaron que podía existir una fecha más allá del siglo veinte. El esfuerzo de adaptación del código postal palidece en comparación con esto. A Jorge Luis Borges le habría gustado la idea: puesto que presumiblemente pocas personas se preocuparán por lo que sucederá el 1 de enero del año 3000, éste debe ser el asunto más pequeño en la historia de la humanidad al cual se le ha dedicado o se le dedicará una serie de conferencias: *un simple dígito decimal*.

La cuestión no es que alguna parte del programa conozca la estructura física de los datos: esto es inevitable puesto que los datos deben ser accedidos finalmente para su manipulación interna, sino que con las técnicas tradicionales de diseño este conocimiento se distribuye sobre muchas partes del sistema, provocando modificaciones injustificadamente grandes en los programas en el momento en que cambia alguna de las estructuras físicas —como inevitablemente ocurrirá. En otras palabras, si el código postal pasa de cinco a nueve dígitos, o las fechas requieren un dígito más, es razonable suponer que un programa que manipule los códigos o las fechas necesite ser adaptado; lo que no es aceptable es que esa necesidad se extienda a todo el programa, de tal modo que un cambio en dicha longitud cause cambios en los programas en una magnitud desproporcionada con el tamaño conceptual del cambio en la especificación.

El capítulo 6 trata en detalle los tipos abstractos de datos.

La teoría de los tipos abstractos de datos proporcionará la clave para resolver este problema, al permitir que los programas accedan a los datos por sus propiedades externas en vez de por su implementación física.

Otro elemento significativo en la distribución de actividades es el bajo porcentaje (5,5%) de los costes de documentación. Debe recordarse que éstos son costes de tareas realizadas en tiempo de mantenimiento. La observación aquí —al menos la especulación en ausencia de datos más específicos— es que un proyecto o presta atención a la documentación como parte del desarrollo o no la considera para nada. Aprenderemos a usar un estilo de diseño en el cual gran parte de la documentación está realmente embebida en el software, existiendo herramientas especiales para extraerla cuando se necesite.

Los siguientes elementos en la lista de Lientz y Swanson también son interesantes, si bien menos relevantes con relación a los temas que se tratan en este libro. Las correcciones de emergencia ante fallos (hechas con precipitación cuando los usuarios informan de que el programa no está produciendo los resultados esperados o se comporta de modo catastrófico) cuestan más que las correcciones rutinarias planificadas. Esto no es sólo porque se tienen que realizar bajo altas presiones, sino porque rompen el proceso ordenado de distribución de nuevas versiones y pueden introducir nuevos errores. Las dos últimas actividades llevan pequeños porcentajes:

- Una trata de las mejoras de eficiencia; parece que una vez que un sistema funciona, los administradores de los proyectos y los programadores se muestran reacios a interrumpirlo para buscar mejoras de eficiencia y prefieren dejarlo tal y como está. (Cuando se considera el precepto de “hágalo correcto antes de hacerlo rápido”, para muchos proyectos es suficiente felicidad el cumplir con el primero de estos pasos.)
- También implicado en un pequeño tanto por ciento nos encontramos con “transferir a nuevos entornos”. Una posible interpretación (de nuevo una conjetura en ausencia de datos más precisos) es que hay dos clases de programas con respecto a la portabilidad, con pocas diferencias entre ellos: algunos programas están diseñados con la portabilidad en mente y cuesta relativamente poco adaptarlos a diferentes plataformas; otros están tan sujetos a la plataforma original, y serían tan difíciles de transferir, que los desarrolladores ni siquiera lo intentan.

1.4. CONCEPTOS CLAVE INTRODUCIDOS EN ESTE CAPÍTULO

- El propósito de la ingeniería de software es encontrar modos de construir software de calidad.
- En lugar de un único factor, la calidad del software se ve mejor como una cuestión de equilibrio entre un conjunto de objetivos diferentes,
- Los factores externos, perceptibles por los usuarios y clientes, deben distinguirse de los factores internos, perceptibles por diseñadores e implementadores.
- Lo importante son los factores externos pero éstos sólo se pueden alcanzar a través de los factores internos.
- Hemos visto una lista de los factores de calidad externos. Aquellos para los que el software actual está mal, necesitando de métodos mejores que son abordados por el método orientado a objetos, son los factores de corrección y robustez relacionados con la seguridad, conocidos en conjunto como fiabilidad. Los factores que requieren arquitecturas de software más descentralizadas, en conjunto conocidos como modularidad.
- El mantenimiento del software, que consume una gran parte de los costes, está penalizado por la dificultad de hacer cambios en los productos software y por la dependencia que los programas tienen de la estructura física de los datos que manipulan.

1.5. NOTAS BIBLIOGRÁFICAS

Varios autores han propuesto definiciones sobre la calidad del software. De entre los primeros artículos sobre el tema, hay dos que en particular mantienen su vigencia hoy día: [Hoare 1972], un editorial por invitación, y [Boehm 1978], el resultado de uno de los primeros estudios sistemáticos realizado por un grupo de TRW.

La distinción entre los factores externos e internos fue introducida en un estudio de 1977 de la General Electric encargado por la US Air Force [McCall 1977]. McCall emplea los términos "factores" y "criterio" para lo que este capítulo ha denominado factores externos e internos. Muchos (aunque no todos) de los factores introducidos en este capítulo corresponden a algunos de los de McCall; se ha quitado uno de sus factores, la facilidad de mantenimiento, porque como ya se explicó se cubre adecuadamente con la extensibilidad y la verificabilidad. El estudio de McCall analiza no sólo los factores externos sino también un buen número de factores internos ("criterios"), así como también las *métricas*, o técnicas cuantitativas para evaluar el logro de los factores internos. Sin embargo, con la tecnología orientada a objetos muchos de los estudios de factores internos y métricas, tan arraigados en los viejos modos de hacer software, han quedado obsoletos. Llevar a cabo esta parte del trabajo de McCall con las técnicas desarrolladas en este libro sería un proyecto útil; véase la bibliografía y los ejercicios del capítulo 3.

El argumento del efecto relativo que tienen las mejoras de la máquina dependiendo de la complejidad de los algoritmos ha sido extraído de [Aho 1974].

Una referencia estándar sobre la facilidad de uso es [Shneiderman 1987] que es una extensión de [Shneiderman 1980] que está dedicada al vasto tema de la psicología del software. La página Web del laboratorio de Shneiderman en <http://www.cs.umd.edu/projects/hcil/> contiene muchas referencias bibliográficas sobre estos temas.

Las curvas de Osmond provienen de un tutorial impartido por Roger Osmond en TOOLS USA [Osmond 1995]. Obsérvese que la forma en que se ha expuesto en este capítulo no muestra el tiempo, propiciando una visión más directa del equilibrio entre la funcionalidad y otras cualidades en las dos curvas alternativas, pero no refleja el potencial de la curva en negro para retrasar un proyecto. Las curvas originales de Osmond fueron dibujadas con respecto al tiempo en vez de respecto a la funcionalidad.

La gráfica de los costes de mantenimiento ha sido obtenida del estudio de Lientz y Swanson, basado en un cuestionario sobre el mantenimiento enviado a 487 organizaciones [Lientz 1980]. Véase también [Boehm 1979]. Aunque algunos de los datos de entrada pueden considerarse demasiado especializados y ahora obsoletos (se basaba en aplicaciones MIS⁴ de trabajo "por lotes" de un tamaño promedio de 23.000 instrucciones, lo cual es grande pero no en los estándares de hoy en día), los resultados siguen siendo aplicables por norma general. La Asociación de Gestión del Software (*Software Management Association*) lleva a cabo un estudio anual del mantenimiento; véase [Dekleva 1992] para encontrar un informe sobre estos estudios.

La expresión *programming-in-the-large* y *programming-in-the-small*⁵ fue introducida por [DeRemer 1976].

Para un análisis general sobre aspectos de la ingeniería de software, véase el libro de Ghezzi, Jazayeri y Mandrioli [Ghezzi 1991]. Un texto sobre lenguajes de programación de uno de estos mismos autores [Ghezzi 1997], proporciona una formación complementaria de algunos de los aspectos discutidos en el presente libro.

⁴ Siglas en inglés de *Management Information System* (Sistemas de Gestión de Información) (N. del T.).

⁵ Programando en lo grande, programando en lo pequeño (N. del T.).

Criterios de la orientación a objetos

Iniciábamos en el capítulo anterior la exposición del método orientado a objeto explorando sus objetivos. Como preparación para las partes B y C, en las cuales se describirán en detalle las técnicas del método, es útil echar una ojeada rápida pero amplia a los aspectos clave del desarrollo orientado a objetos. Éste es el objetivo de este capítulo.

Uno de los beneficios será obtener un breve recordatorio de qué es lo que hace que un sistema sea orientado a objetos. Esta expresión se está empleando de modo tan indiscriminado que se necesita una lista de propiedades precisas con respecto a las cuales se pueda valorar cualquier método, lenguaje o herramienta que proclame ser O-O.

Este capítulo limita las explicaciones al mínimo, de modo que si ésta es su primera lectura, no debe esperar comprender con detalle todos los criterios que se mencionan; explicarlos será la tarea del resto del libro. Debe considerarse esta discusión como una visión previa —no la película real sino un avance.

Realmente se impone una advertencia porque en términos cinematográficos este capítulo puede considerarse, a diferencia de un buen avance de una película, como una de esas personas que en el cine gustan de descubrirle la trama a los demás. Como tal este capítulo rompe la progresión paso a paso del libro, especialmente de la parte B, que construye pacientemente los argumentos para la orientación a objetos, analizando problema tras problema antes de deducir y justificar las soluciones. Si tiene la intención de leer una amplia visión de conjunto antes de meterse en más profundidad, éste es el capítulo. Pero si lo que se prefiere es *no* estropear el placer de ver como se revelan los problemas e ir descubriendo las soluciones una a una, entonces debe saltar este capítulo. No es necesario leer este capítulo para entender los siguientes.

Advertencia:
¡AQUÍ SE
CUENTA
EL FINAL!

2.1. SOBRE LOS CRITERIOS

Primero examinaremos la selección de criterios para evaluar si se es o no orientado a objetos¹.

¿Hasta qué punto hay que ser dogmático?

La lista que se presenta a continuación incluye todas las capacidades que este autor considera esenciales para la producción de software de calidad empleando el método orientado a objetos. Es ambiciosa y puede parecer que no hace concesión alguna, e incluso dogmática. ¿Qué conclusión implica esto para un entorno que satisfaga algunas de estas condiciones,

¹ Objectness en el original. Desafortunadamente no existe en castellano un término equivalente, traducirlo por *objetividad* provocaría confusión por las otras acepciones de esta palabra (*N. del T.*).