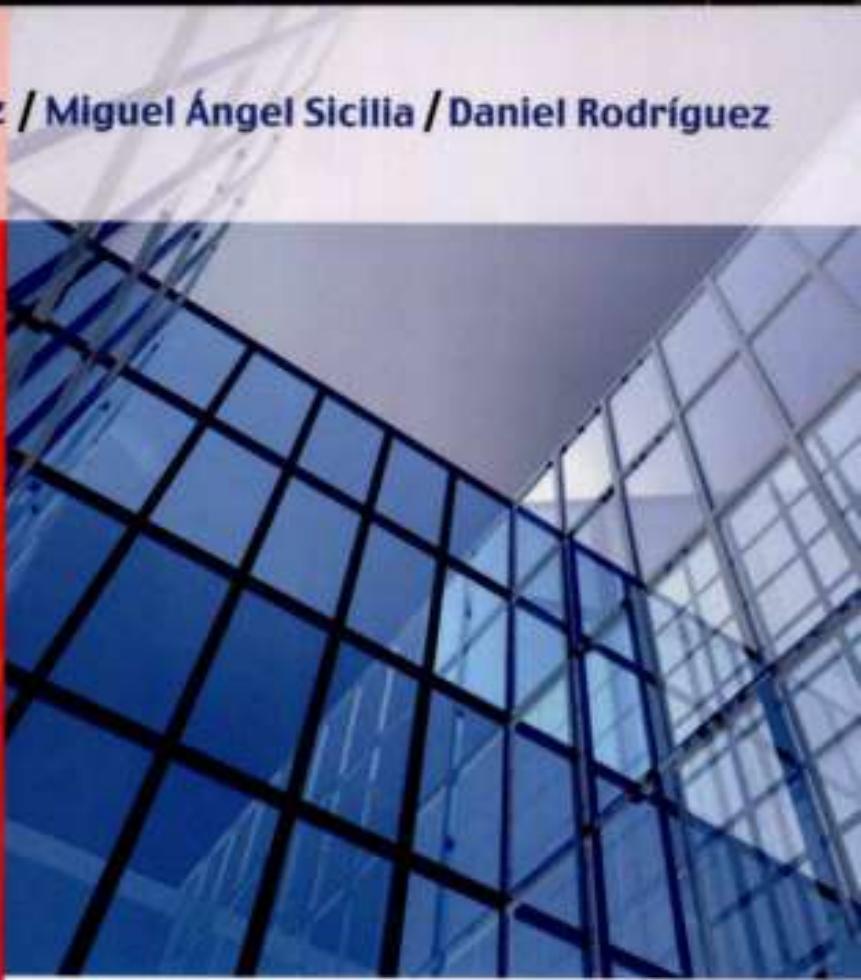


Salvador Sánchez / Miguel Ángel Sicilia / Daniel Rodríguez

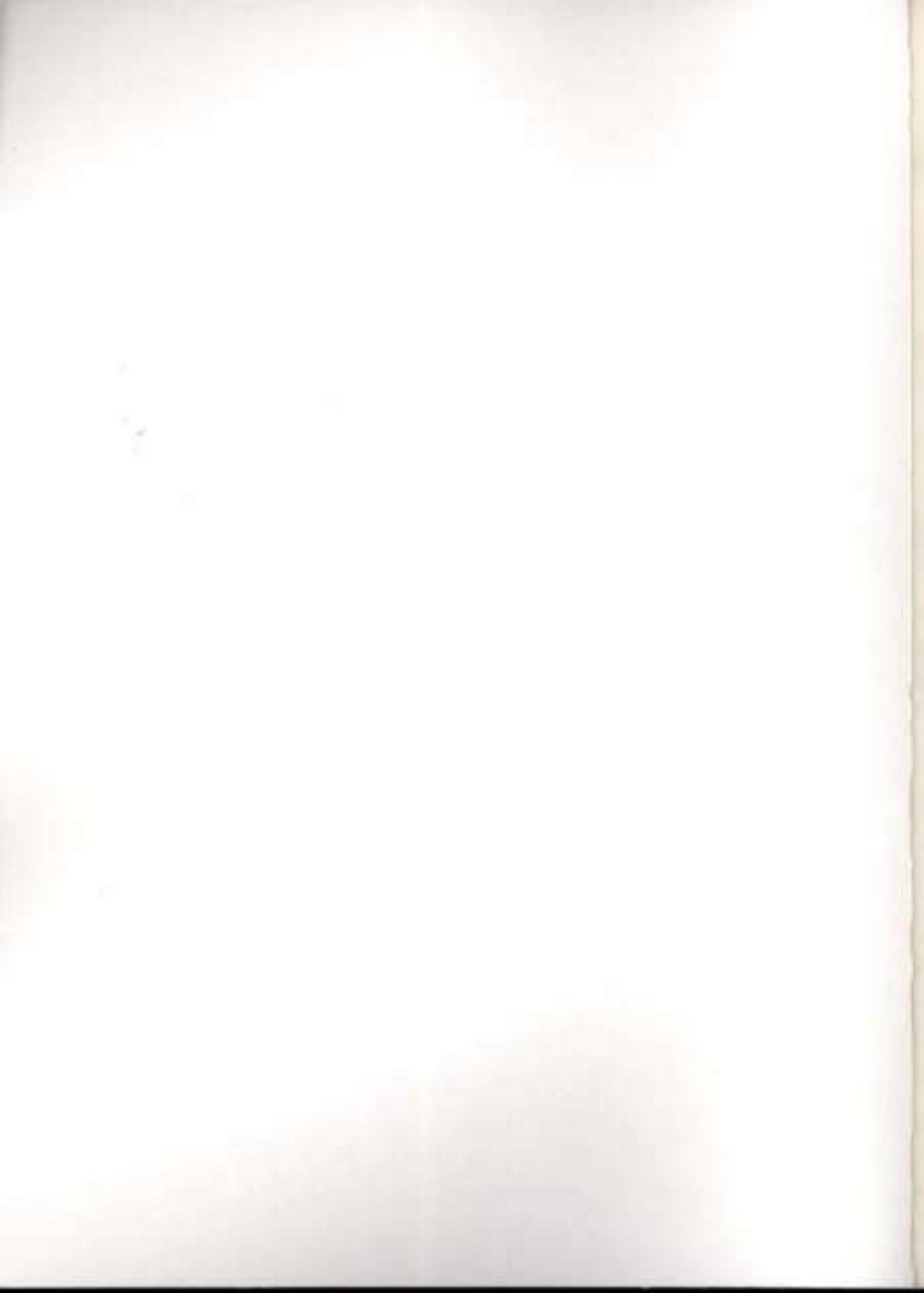


Ingeniería del
Software

Un enfoque desde la guía
SWEBOK

Δ Alfaomega

Garceta
grupo editorial



INGENIERÍA DEL SOFTWARE





INGENIERÍA DEL SOFTWARE

Un enfoque desde la guía SWEBOK

Salvador Sánchez Alonso

Miguel Ángel Sicilia Urbán

Daniel Rodríguez García

Departamento de Ciencias de la Computación

Universidad de Alcalá

 **Alfaomega**

 **Garceta**
grupo editorial

Datos catalográficos

Sánchez, Salvador, Sicilia, Miguel Ángel y
Rodríguez, Daniel
Ingeniería del Software. Un enfoque desde la guía
SWEBOK
Primera Edición

Alfaomega Grupo Editor, S.A. de C.V., México

ISBN: 978-607-707-420-5

Formato: 17 x 23 cm

Páginas: 568

Ingeniería del Software. Un enfoque desde la guía SWEBOK

Salvador Sánchez Alonso, Miguel Ángel Sicilia Urbán y Daniel Rodríguez García

ISBN: 978-84-9281-240-0, edición original publicada por IBERGARCETA PUBLICACIONES, S.L.,
Madrid, España

Derechos reservados © 2011 IBERGARCETA PUBLICACIONES, S.L.

Primera edición: Alfaomega Grupo Editor, México, marzo 2012

Cuarta reimpresión: Alfaomega Grupo Editor, México, agosto 2017

© 2012 Alfaomega Grupo Editor, S.A. de C.V.

Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, 06720 Ciudad de México.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: <http://www.alfaomega.com.mx>

E-mail: atencionalcliente@alfaomega.com.mx

ISBN: 978-607-707-420-5

Derechos reservados:

Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Nota importante:

La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro, ni por la utilización indebida que pudiera dársele.

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, C.P. 06720, Del. Cuauhtémoc, Ciudad de México – Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490.
Sin costo: 01-800-020-4396 – E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Carrera 15 No. 64 A 29, Bogotá, Colombia.
Tel.: (57-1) 2100122 – Fax: (57-1) 6068648 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Dr. La Sierra 1437, Providencia, Santiago, Chile
Tel.: (56-2) 235-4248 – Fax: (56-2) 235-5786 – E-mail: ageschile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Paraguay 1307 PB, Of. 11, C.P. 1057, Buenos Aires,
Argentina. – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaditor.com.ar

Índice

I Fundamentos de la Ingeniería del Software	xix
1 Introducción a la Ingeniería del Software	5
1.1 ¿Arte o ingeniería?	5
1.2 Objetivos	7
1.3 Introducción	7
1.4 ¿Qué es la ingeniería?	8
1.5 Ingeniería y ciencias de la ingeniería	10
1.6 El software como artefacto tecnológico	12
1.6.1 ¿Qué es el software?	13
1.6.2 La complejidad inherente al software	13
1.7 Sistematicidad, disciplina y cuantificación	14
1.8 La Ingeniería del Software como disciplina profesional	16
1.8.1 Breve historia de la Ingeniería del Software	16
1.8.2 Elementos de la Ingeniería del Software como disciplina profesional	17
1.9 Conceptos básicos de la Ingeniería del Software	19
1.9.1 Actividades y artefactos	19
1.9.2 Métodos, especificaciones y modelos	20
1.9.3 Procesos y ciclos de vida	21

2 Modelos y procesos	31
2.1 El proceso del proceso	31
2.2 Objetivos	32
2.3 Introducción	33
2.3.1 Una definición de proceso	34
2.3.2 Modelos del ciclo de vida, marcos de procesos y procesos	35
2.3.3 Características de las definiciones de procesos de software	37
2.3.4 Lenguajes para la especificación de procesos	38
2.4 Modelos de ciclo de vida del software	39
2.4.1 Modelo en cascada	40
2.4.2 Modelo en «V»	41
2.4.3 Modelos de proceso basados en prototipos	42
2.4.4 Modelo en espiral	44
2.5 Procesos de software	46
2.5.1 ¿Qué se define en un proceso de software?	46
2.5.2 El modelo de referencia ISO 12207	48
2.5.3 Iteraciones e incrementos	52
2.6 Algunos tipos de procesos importantes	55
2.6.1 Procesos estructurados y procesos orientados a objetos	56
2.6.2 Procesos ágiles	56
2.6.3 Procesos basados en componentes	58
2.6.4 Especificaciones de proceso abiertas	60
2.7 Resumen	62
2.8 Notas bibliográficas	63
2.9 Cuestiones de autoevaluación	63
2.10 Actividades propuestas	65
3 Medición	67
3.1 La necesidad de medir	67
3.2 Objetivos	68
3.3 Introducción	68
3.3.1 Conceptos básicos	69
3.3.2 Tipos de escalas de medición	70
3.3.3 Clasificación de las medidas	71
3.3.4 Evaluación de las métricas	72
3.3.5 ¿Qué medir en la Ingeniería del Software?	73
3.4 Medidas del producto: atributos internos	75
3.4.1 Medidas del tamaño de los sistemas	75
3.4.2 Medidas de la complejidad del software	76
3.4.3 Medidas de la documentación	80
3.4.4 Medidas de reutilización	81

3.4.5	Medidas de la eficiencia	81
3.5	Medidas del producto: atributos externos	82
3.6	Medidas del proceso y los recursos	84
3.6.1	Medidas relacionadas con el proceso	84
3.6.2	Medidas relacionadas con los recursos	84
3.7	Metodologías y estándares para la medición	87
3.7.1	Método Objetivo-Pregunta-Métrica (GQM)	87
3.7.2	El estándar IEEE 1061-1998	90
3.7.3	PSM y el estándar ISO/IEC 15939	91
3.7.4	Otras metodologías y estándares para la medición	92
3.8	Estudios empíricos	93
3.8.1	Encuestas	95
3.8.2	Casos de estudio	96
3.8.3	Experimentación formal	97
3.9	Resumen	99
3.10	Notas bibliográficas	100
3.11	Cuestiones de autoevaluación	101
3.12	Ejercicios y actividades propuestas	102
3.12.1	Ejercicios resueltos	102
3.12.2	Actividades propuestas	104

II Procesos fundamentales de la Ingeniería del Software

107

4	Requisitos	111
4.1	La difícil tarea de determinar qué debe hacerse	111
4.2	Objetivos	112
4.3	Introducción	113
4.4	Definiciones preliminares y características	115
4.4.1	El concepto de requisito	116
4.4.2	Actividades de requisitos	116
4.4.3	Actores	118
4.4.4	Características de los requisitos	119
4.4.5	El documento de especificación de requisitos	120
4.5	Tipos de requisitos	121
4.5.1	Requisitos funcionales	122
4.5.2	Requisitos no funcionales	123
4.5.3	Otras clasificaciones de los requisitos	125
4.6	Las actividades de requisitos	127
4.6.1	Obtención de requisitos	128
4.6.2	Ánalisis de requisitos	132
4.6.3	Especificación de requisitos	137

4.6.4	Validación de requisitos	143
4.7	Notaciones para el modelado conceptual	147
4.7.1	Casos de uso	147
4.7.2	Modelos entidad-relación	151
4.7.3	Diagramas de clases UML	152
4.7.4	Notaciones formales	155
4.8	Gestión del proceso de requisitos	156
4.8.1	Seguimiento	157
4.8.2	Métricas de los requisitos	158
4.8.3	Herramientas para la gestión de requisitos	160
4.9	Resumen	162
4.10	Notas bibliográficas	163
4.11	Cuestiones de autoevaluación	164
4.12	Ejercicios y actividades propuestas	166
4.12.1	Ejercicios resueltos	166
4.12.2	Actividades propuestas	170
5	Diseño	173
5.1	No es posible construir sin diseñar	173
5.2	Objetivos	174
5.3	Introducción	175
5.4	Conceptos fundamentales de diseño	176
5.4.1	Abstracción	176
5.4.2	Componentes e interfaces	177
5.4.3	Descomposición y modularización	177
5.4.4	Medición de la modularidad	178
5.4.5	Arquitectura de sistemas	180
5.4.6	Notaciones de diseño	184
5.5	Métodos de diseño	184
5.5.1	Métodos estructurados	184
5.5.2	Métodos orientados a datos	189
5.5.3	Diseño orientado a objetos	190
5.6	Otras técnicas relacionadas con el diseño	199
5.6.1	Los patrones de diseño software	199
5.6.2	Software frameworks, plug-ins y componentes	203
5.6.3	Diseño por contrato	205
5.7	Diseño de sistemas distribuidos	207
5.8	Evaluación y métricas en el diseño	209
5.9	Resumen	213
5.10	Notas bibliográficas	214
5.11	Cuestiones de autoevaluación	215

5.12 Ejercicios y actividades propuestas	216
5.12.1 Ejercicios resueltos	216
5.12.2 Actividades propuestas	219
6 Construcción	223
6.1 No da igual cómo esté construido	223
6.2 Objetivos	225
6.3 Introducción	225
6.4 Lenguajes de construcción	227
6.5 Reutilización del código	230
6.6 Principios fundamentales de la construcción de software	232
6.6.1 Minimizar la complejidad	232
6.6.2 Anticipar los cambios	250
6.6.3 Construir para verificar	253
6.6.4 Utilización de estándares	257
6.7 La calidad en la construcción de software	260
6.7.1 Asunciones y diseño por contrato	260
6.7.2 Análisis de rendimiento	262
6.7.3 Depuración	264
6.8 Gestión de la construcción	266
6.8.1 Planificación de la construcción	266
6.8.2 Métricas de construcción	267
6.9 Resumen	268
6.10 Notas bibliográficas	269
6.11 Cuestiones de autoevaluación	269
6.12 Ejercicios y actividades propuestas	271
6.12.1 Ejercicios resueltos	271
6.12.2 Actividades propuestas	276
7 Pruebas	279
7.1 El porqué de las pruebas	279
7.2 Objetivos	280
7.3 Introducción	280
7.3.1 Conceptos fundamentales	284
7.3.2 Limitaciones en la realización de pruebas	286
7.3.3 Las pruebas y el riesgo	287
7.4 Técnicas de prueba	289
7.4.1 Pruebas de caja blanca y de caja negra	290
7.4.2 Clasificación exhaustiva de las técnicas de prueba	295
7.5 Niveles de prueba	298
7.5.1 Pruebas según su objeto	298
7.5.2 Pruebas según el objetivo que persiguen	305

7.6	Pruebas unitarias con JUnit	308
7.6.1	Ejemplo sencillo de uso de JUnit	309
7.6.2	Complicación del ejemplo inicial	312
7.6.3	Colecciones de pruebas	314
7.6.4	JUnit 4	315
7.7	Métricas relacionadas con las pruebas	317
7.7.1	Medidas durante las pruebas	318
7.7.2	Evaluación de las pruebas realizadas	319
7.8	El proceso de prueba	320
7.9	Resumen	322
7.10	Notas bibliográficas	324
7.11	Cuestiones de autoevaluación	325
7.12	Ejercicios y actividades propuestas	326
7.12.1	Ejercicios resueltos	326
7.12.2	Actividades propuestas	330
8	Mantenimiento	333
8.1	La mente de los otros	333
8.2	Objetivos	334
8.3	Introducción	335
8.4	Conceptos fundamentales	336
8.4.1	¿Qué es el mantenimiento del software?	336
8.4.2	La facilidad de mantenimiento	337
8.4.3	Mantenimiento y calidad	339
8.4.4	Aspectos de la facilidad de mantenimiento	340
8.5	La práctica del mantenimiento del software	340
8.5.1	El mantenimiento del software como un caso especial de mantenimiento	342
8.5.2	La evolución del software y sus leyes	342
8.6	El proceso de mantenimiento	344
8.6.1	Las actividades de mantenimiento	345
8.6.2	El mantenimiento como preparación	348
8.7	Técnicas para el mantenimiento del software	348
8.7.1	Ingeniería inversa	350
8.7.2	Reingeniería	352
8.7.3	Reestructuración	354
8.8	Métricas de mantenimiento	357
8.8.1	Métricas del producto	357
8.8.2	Métricas relacionadas con el proceso	360
8.9	Resumen	361
8.10	Notas bibliográficas	363

8.11 Cuestiones de autoevaluación	364
8.12 Ejercicios y actividades propuestas	365
8.12.1 Ejercicios resueltos	365
8.12.2 Actividades propuestas	369
III Gestión y Calidad en la Ingeniería del Software	373
9 Calidad	377
9.1 La especial naturaleza de la calidad	377
9.2 Objetivos	378
9.3 Introducción	379
9.3.1 Cultura y ética de la calidad	379
9.3.2 Valor y costes de la calidad	381
9.3.3 Los múltiples aspectos de la calidad	382
9.4 Calidad del producto	384
9.4.1 El modelo de calidad de McCall	384
9.4.2 El modelo de Boëhm	387
9.4.3 El modelo de calidad ISO/IEC 9126	388
9.4.4 Otros modelos de calidad	391
9.5 Calidad del proceso	392
9.5.1 Aseguramiento de la calidad	392
9.5.2 El modelo CMMI	394
9.5.3 Modelo SPICE: El estándar ISO/IEC 15504	399
9.5.4 Los estándares de la familia ISO 9000	401
9.5.5 Otros modelos, estándares y especificaciones	403
9.6 Resumen	410
9.7 Notas bibliográficas	412
9.8 Cuestiones de autoevaluación	413
9.9 Ejercicios y actividades propuestas	414
9.9.1 Ejercicios resueltos	414
9.9.2 Actividades propuestas	416
10 Gestión	419
10.1 El desarrollo de proyectos no es sólo tecnología	419
10.2 Objetivos	421
10.3 Visión general de la gestión de proyectos	422
10.4 La estimación de coste, plazos y esfuerzo	424
10.4.1 Estimación mediante juicio de expertos	425
10.4.2 Puntos de función	426
10.4.3 Modelos algorítmicos o paramétricos	429
10.4.4 Modelos basados en la inteligencia artificial	436

10.4.5 Sistemas dinámicos	439
10.4.6 Evaluación de modelos	440
10.4.7 Calibración de modelos	443
10.5 Planificación y seguimiento del proyecto	443
10.5.1 Estructura de descomposición del trabajo	444
10.5.2 Los métodos gráficos CPM y PERT	445
10.5.3 Diagramas de Gantt	450
10.5.4 Método del valor conseguido	451
10.6 Revisiones y cierre del proyecto	458
10.7 Gestión de los recursos humanos	459
10.8 Gestión y análisis del riesgo	460
10.9 Resumen	462
10.10 Notas bibliográficas	463
10.11 Cuestiones de autoevaluación	464
10.12 Ejercicios y actividades propuestas	465
10.12.1 Ejercicios resueltos	465
10.12.2 Actividades propuestas	468
11 Gestión de la configuración del software	471
11.1 La importancia de poner las cosas en su sitio	471
11.2 Objetivos	473
11.3 La configuración del software	473
11.4 Actividades de gestión de la configuración del software	477
11.4.1 Identificación de la configuración del software	478
11.4.2 Control de los cambios en el software	481
11.4.3 Gestión de entregas	484
11.5 Planificación y gestión	487
11.5.1 Contabilidad y medición en gestión de la configuración	488
11.5.2 Auditoría de la configuración software	489
11.6 Técnicas y herramientas para el control de versiones	489
11.6.1 Versiones, divisiones y deltas	490
11.6.2 Políticas de control de versiones en grupos de trabajo	491
11.7 Resumen	494
11.8 Notas bibliográficas	495
11.9 Cuestiones de autoevaluación	496
11.10 Ejercicios y actividades propuestas	497
11.10.1 Ejercicios resueltos	497
11.10.2 Actividades propuestas	502

12 Herramientas	505
12.1 Las herramientas nos diferencian	505
12.2 Objetivos	506
12.3 Introducción	507
12.3.1 Justificación de las herramientas CASE	508
12.3.2 Ventajas e inconvenientes del uso de herramientas CASE	509
12.4 Clasificación de las herramientas CASE	510
12.4.1 Herramientas CASE según el ciclo de vida	511
12.4.2 Herramientas CASE según su nivel de integración	519
12.5 Selección y evaluación de herramientas	526
12.5.1 Identificación de las necesidades	527
12.5.2 Selección de herramientas candidatas	528
12.5.3 Evaluación técnica	529
12.5.4 Toma de la decisión final	530
12.6 Resumen	531
12.7 Notas bibliográficas	531
12.8 Cuestiones de autoevaluación	532
12.9 Ejercicios y actividades propuestas	533
12.9.1 Ejercicios resueltos	533
12.9.2 Actividades propuestas	536

c
c
e
j
c

si
te
li
qu
bu
vi
de
tie

es

Prefacio

Si pudiéramos copiar a la industria de la construcción, con su arquitectura, gestión de proyecto, diseño, ingeniería, herramientas, reglas, directrices e incluso sus partes prefabricadas... Si pudiéramos hacer todo eso, exactamente así como lo hemos expresado, entonces resolveríamos todos nuestros problemas con el software.

— M. Bertig

La obra que el lector tiene entre sus manos es el fruto de una convicción. La convicción de sus autores acerca de la importancia de la Ingeniería del Software como parte esencial de la formación de los profesionales de la informática. Porque dado que la ingeniería es el resultado de la objetivación de un oficio, que sedimenta y se estudia científicamente, los ingenieros que crean, estudian, modifican y trabajan con el software necesitan entender cómo se elabora y se mantiene en el tiempo.

La Ingeniería del Software es la disciplina del informático cuando actúa como profesional, en el entorno profesional. Por eso, necesita un tratamiento que la separe de las tecnologías concretas y del estudio de técnicas específicas. Hay algunos (no demasiados) libros de Ingeniería del Software. Pero nosotros sentíamos que hacía falta un libro más, uno que complementase los existentes para centrarse en la ingeniería *per se*. Por ello, hemos buscado separarnos de las técnicas concretas en la medida de lo posible, y proporcionar la visión general que los conceptos en que descansan dichas técnicas proporcionan. Con ello, deseamos que nuestra obra sea, no sólo más universal, sino también más perdurable en el tiempo por estar menos sujeta a las especificidades de cada técnica en particular.

El ingeniero del software debe entender esencialmente por qué un producto software es bueno o no lo es (la calidad), cómo puede saberlo (la medición), cómo pueden hacerse

productos de calidad (el proceso), y cómo pueden ordenarse los recursos humanos y temporales en un proyecto para aplicar correctamente el proceso (la gestión). Además de esos elementos generales, es importante que un ingeniero del software tenga una visión general de las técnicas y métodos disponibles para cada tipo concreto de actividades (que pueden resumirse en: requisitos, diseño, construcción, pruebas, mantenimiento y gestión de la configuración). Por último, hay ciertas herramientas que ayudan a los ingenieros del software a realizar correcta, eficaz y eficientemente las actividades según los procesos. Aunque esas herramientas son muy diversas y evolucionan constantemente, es importante conocer sus tipos para saber seleccionarlas y decidir cuándo aplicarlas o si merece la pena invertir en ellas. Con este párrafo, hemos resumido realmente el contenido del libro y qué es lo que trata de cubrir. La intención es la de resumir todos los elementos que es necesario conocer para el desarrollo profesional del software, sin entrar en el detalle de cada técnica. Siendo más concretos, pretendemos que el lector entienda, por ejemplo, para qué sirven los casos de uso en las actividades de requisitos, pero no pretendemos que el lector domine la técnica ayudado por este libro. Para eso hay textos especializados que podrán servirle en cada caso.

También debemos resaltar que el presente libro pretende ser introductorio. Por eso en muchas ocasiones no hemos profundizado tanto como habríamos deseado, cortando la discusión en partes que realmente nos apetecía tratar en más detalle pero que claramente quedaban fuera del alcance de la obra. Siempre es difícil acertar plenamente en la cobertura que se ofrece, si bien esperamos que el resultado sea de su agrado y, especialmente, que le resulte de utilidad en su trabajo o sus estudios.

¿Por qué un nuevo libro de Ingeniería del Software?

Fundamentalmente porque aún hay sitio para otras formas de introducir la Ingeniería del Software, diferentes a la de los textos que el lector en español puede encontrar a día de hoy. Particularmente, hemos utilizado como guía y como criterio a la hora de decidir qué incluir y qué no la Guía al Cuerpo de Conocimiento de Ingeniería del Software - *Guide to the Software Engineering Body of Knowledge* (SWEBOK). La guía SWEBOK, como nosotros la llamaremos de aquí en adelante, es el fruto de un trabajo de colaboración, redacción y revisión experta de varios años. Como tal, pretende compendiar todo lo que un ingeniero del software debe conocer pues será relevante para su actividad profesional. No obstante, la guía SWEBOK no es un libro de texto, sino una obra de referencia donde se proporciona el esquema de los conocimientos y competencias de los ingenieros del software, junto a listas de referencias a la literatura en las que se basa la guía. Tampoco es una guía curricular, ya que no proporciona indicaciones sobre importancias relativas u objetivos educativos. En definitiva, se puede considerar que este libro es la primera obra que trata de cubrir la Ingeniería del Software desde la estricta observancia de los criterios de quienes elaboraron y dieron escrutinio a la guía SWEBOK.

Los autores son también traductores al español de la guía SWEBOK, y, aunque según el adagio «*traduttore, traditore*», esto nos ha servido como elemento de reflexión y oráculo

cuando elaborábamos los temas. El parecido en la estructura de la guía SWEBOK y esta obra no escaparía a nadie, pero lógicamente, en un libro de texto prima el criterio pedagógico sobre el enciclopédico, y algunos elementos que hemos considerado menos frecuentes o más «avanzados» han tenido que ser sacrificados (no sin pesar) en aras de la concisión. El libro está pensado para cubrir un curso de Ingeniería del Software de carácter elemental, para estudiantes que ya tienen conocimientos de programación al menos a un nivel básico, de modo que pueden contextualizar correctamente las diferentes actividades y comprender las dificultades de la prueba, el desarrollo o la gestión de configuración. El libro puede utilizarse para un curso de un semestre, de unos 6 ECTS, si se complementa con ejemplos y casos prácticos y se introduce en un diseño instruccional apropiado. También puede servir como base para cursos más avanzados, siempre que se complemente con otros textos o recursos que profundicen en un determinado aspecto o categoría de actividades.

Hemos intentado adaptar los contenidos a los «tiempos modernos», aunque, de algún modo sorprendente, si bien las herramientas cambian, la teoría y los conceptos se mantienen más o menos inalterados desde hace ya más de diez años (quizá con la excepción de los procesos, en los que ha habido más movimiento reciente).

Cómo leer esta obra

El libro puede leerse de principio a fin, para obtener una visión completa de la Ingeniería del Software. No obstante, para conocer la disciplina en general de modo rápido, pueden leerse los dos primeros capítulos, en los que se introduce la disciplina y se trata el ciclo de vida del software a vista de pájaro.

El Capítulo 3 sobre medición es muy importante para entender cómo la Ingeniería del Software es una disciplina que puede ser objeto de contraste empírico y de estudio objetivo. Puede obviarse en una primera lectura, o bien dejarse para el final. No obstante, comprender la importancia de la medición y el concepto de métrica es importante en todos los capítulos de la Parte II del libro, por lo que se recomienda al menos leer las primeras secciones del capítulo antes de abordar los siguientes.

Los Capítulos 4 al 8 cubren las principales categorías de actividades en Ingeniería del Software: requisitos, diseño, construcción, pruebas y mantenimiento. Las actividades de gestión de configuración se han dejado para la tercera parte, junto a los temas de calidad y gestión, pues es una actividad de soporte para el resto de las actividades que puede bien posponerse o verse de forma separada. Esta segunda parte del libro también puede leerse por capítulos separados, aunque es recomendable hacerlo en la secuencia que proponemos al tratarse de actividades que normalmente en la práctica conforman un flujo.

La tercera parte (Capítulos 9 al 12) entra en los aspectos relacionados con la gestión y el soporte a las actividades. Hemos decidido comenzar con el tema de la calidad (Capítulo 9), porque ese concepto de calidad determina el canon por el que juzgar si la gestión (Capítulo 10) es buena o mala. El Capítulo 11 trata un aspecto muy concreto de la gestión interna de los proyectos de desarrollo de software, las configuraciones, que por su importancia

merecen capítulo aparte. El Capítulo 12 trata las herramientas para el desarrollo. Hemos evitado entrar a describir productos o herramientas concretas, y en su lugar, hemos intentado proporcionar al lector con elementos de juicio generales para su selección en un proyecto.

Agradecimientos

Es de suponer que la sección de agradecimientos no sea, para muchos lectores, una parte importante de un libro. Para los autores es, sin embargo, la única licencia a la informalidad. Como además se redacta una vez sufrido (literalmente) el proceso de llevar a buen término un proyecto de la envergadura de escribir un libro sobre Ingeniería del Software, resulta especialmente placentero detenerse un momento y mirar el camino que quedó atrás.

La redacción de una obra como ésta es una aventura azarosa, que requiere disciplina y que casi siempre toma como víctima una parte de la vida personal de los autores. Hemos sacrificado muchos días de fiesta y horas que eran deshoras, y esto, lógicamente afecta a las personas que uno tiene más cerca. Testimoniamos aquí que sin el apoyo y ayuda de nuestras familias —que han crecido mucho desde que empezamos el libro hasta hoy— no habríamos llegado a puerto. Especialmente, nuestro agradecimiento de corazón a Charo, Elena y Eva.

Además, los borradores de los capítulos de este libro han pasado por el escrutinio desinteresado de expertos en el área de la Ingeniería del Software antes de diarlos por definitivos. Estas contribuciones son especialmente valiosas porque provienen de personas muy ocupadas, cuyo tiempo es muy escaso. Por ello, el regalarnos unas horas para revisar, y hacerlo con la profesionalidad que lo han hecho, es de una generosidad que nosotros valoramos mucho. Dentro de este apartado queremos mencionar a Javier Tuya, de la Universidad de Oviedo, quien hizo una exhaustiva revisión del Capítulo 7, y aportó su amplio conocimiento sobre pruebas de software, dándonos excelentes sugerencias que en su gran mayoría incorporamos en el texto. Abusamos de su confianza hasta el punto de pedirle revisar también del Capítulo 1, donde detectó numerosas incorrecciones y erratas. Javier Dolado, de la Universidad del País Vasco, revisó el Capítulo 3, dándonos indicaciones y consejos que nos fueron muy valiosos para completarlo. Isabel Ramos, de la Universidad de Sevilla, corrigió el Capítulo 9 sobre calidad. Mercedes Ruiz, de la Universidad de Cádiz, revisó exhaustivamente el Capítulo 2, donde detectó numerosas erratas y realizó sugerencias muy interesantes que con gusto incorporamos. Finalmente, a nuestro compañero Ramiro Cano agradecemos el tiempo que dedicó a hacernos sugerencias y correcciones en el capítulo sobre construcción del software: a nosotros también nos tentó hablar del «archiconocido método de depuración por printf» :-). A todos ellos nuestra gratitud y reconocimiento, pues su generosidad ha servido para hacer un poco menos imperfecta esta obra.

No podemos concluir sin hacer mención a nuestro editor, Andrés Otero —cuya familia también ha crecido en este tiempo-. Muchas gracias por creer en el proyecto y por comprender que la escritura es una labor difícil de encorsetar en plazos, hitos y metas inamovibles porque, a diferencia de lo que predicamos en este manual de Ingeniería del Software, a veces han de incumplirse los plazos para conseguir algo de lo que todos estemos orgullosos.

Parte I

Fundamentos de la Ingeniería del Software

Introducción a la Primera Parte

En esta Primera Parte se introduce la Ingeniería del Software como disciplina. Se trata de una ingeniería de un carácter muy particular, dado que el producto resultante, el software, tiene características bien diferentes a las de otras ingenierías. El objetivo del primer capítulo es por tanto delimitar el ámbito de la Ingeniería del Software, y dónde radica la dificultad de hacer buen software.

Como en toda ingeniería, se han de producir artefactos, pero producirlos mediante la aplicación del método y la disciplina, considerando un cierto proceso o forma general de organizar las actividades de ingeniería: qué se debe hacer, en qué secuencia y obteniendo qué productos (intermedios o finales). Así, el segundo capítulo habla del proceso, o mejor, de los procesos en Ingeniería del Software. Como no existe un proceso perfecto, válido para cualquier situación, debemos conocer los diferentes tipos de procesos que se han propuesto en la breve historia de la disciplina, en qué se diferencian, y cómo se especifican. El propósito del segundo capítulo es por tanto proporcionar una visión global que sirva para tomar la decisión de qué proceso elegir para cada situación determinada o, en su caso, qué partes de los procesos existentes tomar y cuáles dejar para diseñar nuestro propio proceso *ad hoc* para un proyecto u organización.

Este bloque termina con un tercer tema dedicado a la medición, donde se introducen los conceptos de medición y de métrica. Si bien no todos los profesionales de la Ingeniería del Software tendrán que véselas con estudios experimentales, sí tendrán que situarse en la posición de establecer un plan de medición y valorar los resultados de esa medición para tomar decisiones y evaluar el progreso de los proyectos y/o los equipos de desarrollo. Por ello, hemos considerado importante conocer la jerga de la medición y entender en qué medida las métricas son fiables y útiles, y cómo deben interpretarse.

1

Introducción a la Ingeniería del Software

El término «Ingeniería del Software» se escogió deliberadamente por ser provocativo, pues implicaba la necesidad de manufacturar software según las bases teóricas y las disciplinas prácticas que son tradicionales en otras ramas de la ingeniería.

— P. Naur y B. Randell

1.1 ¿Arte o ingeniería?

En 1974, el profesor Donald Knuth de la Universidad de Stanford recibió el premio Turing que concede anualmente la asociación ACM, galardón considerado por muchos como el Premio Nobel de la informática. La conferencia que Knuth impartió con motivo de la recepción del premio, comenzaba así:

«Si la programación de computadoras quiere llegar a ser una parte importante del desarrollo e investigación en las ciencias de la computación, deberá transitar desde la programación como arte a la programación como ciencia disciplinada».

Knuth en realidad citaba literalmente una frase acuñada por el comité editorial de la revista *Communications of the ACM*, publicación estandarte de la asociación. Después de tratar en su conferencia diferentes aspectos de los términos *ciencia* y *arte*, Knuth concluyó:

«La programación es un arte, porque aplica conocimiento acumulado, requiere habilidades e ingenio, y especialmente porque produce objetos bellos».

Trascurridas más de tres décadas desde esa conferencia, no hay facultad de ciencias de la computación que suscriba hoy en día un tipo de educación que considere la programación como una actividad de carácter artístico. Dicho de otro modo, el cambio deseado por el comité editorial de la ACM si se ha llevado a cabo, mientras que la consideración del desarrollo de software como arte ha quedado relegada a la esfera de las aficiones personales. Al resultado de ese cambio es a lo que hoy denominamos *Ingeniería del Software*.

No obstante, las diferencias esenciales entre la Ingeniería del Software y otras disciplinas de ingeniería (como veremos más adelante) hacen que aún persista de algún modo la visión del desarrollo de software como una actividad de carácter artesanal (más que artístico), y no como una disciplina ordenada de ingeniería. El desarrollo de programas sigue siendo para muchas personas una actividad vocacional, placentera, y en ocasiones fruto de una formación en su mayor parte autodidacta. No obstante, un ingeniero del software que se enfrenta a un proyecto de desarrollo actúa de acuerdo a un marco de restricciones de carácter económico y organizativo, de plazos, costes y calidades. Ese entorno profesional dista mucho de visiones personalistas como la que D. Knuth relataba en su conferencia:

«El programa del que estoy personalmente más contento y orgulloso es un compilador que escribí para una minicomputadora primitiva que tenía sólo 4096 palabras de memoria, con 16 bits por palabra. Este tipo de cosas hace que uno se sienta como un auténtico virtuoso, al conseguir algo así en unas circunstancias tan estrictas».

Esta visión de la programación como «pasión individual», llevada al terreno del humor, da lugar a posturas como la siguiente (adaptada de una historia publicada por internet):

«[...] las cosas han cambiado mucho en esta era decadente de cerveza sin alcohol, calculadoras de mano y software amigable. En los Gloriosos Viejos Tiempos (con mayúsculas), cuando el término "software" todavía sonaba divertido, y las Computadoras Auténticas estaban hechas de tubos de vacío, los Verdaderos Programadores escribían en código máquina. No en FORTRAN. Ni siquiera en lenguaje ensamblador. Código máquina. Puro, sin adornos. En esos inescrutables números hexadecimales. Directamente».

Los criterios en el desarrollo no son ya la satisfacción personal, la sensación de hacer algo interesante, o la realización de un trabajo brillante. La Ingeniería del Software es hoy en día una actividad de trabajo en grupo y no una pasión individual. En consecuencia, las acciones y decisiones en esta ingeniería no provienen de sentimientos o preferencias personales, sino de la aplicación de métodos y técnicas para racionalizar los recursos de acuerdo con planes y objetivos definidos.

Por todo ello, en este libro trataremos la producción de software como una «ciencia disciplinada», una actividad profesional sometida al estudio científico y objetivada en técnicas y métodos mayoritariamente aceptados por la comunidad profesional en virtud de la

experiencia acumulada. A este respecto, existe un compendio de carácter enciclopédico que recopila aquello que los ingenieros del software —para ser considerados como tales— deben conocer y *saber hacer*. Este compendio es la guía SWEBOK, una obra que se mencionará a partir de aquí en numerosas ocasiones.

El resto de este libro es el intento de los autores de introducir, de manera accesible, los elementos esenciales contenidos en la guía SWEBOK a quienes se inician en la disciplina, o quieren consolidar sus conocimientos con una visión conceptual más amplia.

1.2 Objetivos

El objetivo general de este capítulo es delimitar el concepto de *Ingeniería del Software* como un tipo de disciplina de ingeniería con características especiales, así como proporcionar definiciones de términos fundamentales que se utilizarán a lo largo del libro. Más concretamente, este capítulo pretende que el lector sea capaz de lo siguiente:

- Definir la *Ingeniería del Software*, y comprenderla como una disciplina de ingeniería que trata con un tipo de producto especial, el software.
- Conocer y comprender los conceptos fundamentales que conforman la terminología básica de los ingenieros del software.
- Distinguir entre la Ingeniería del Software como tal disciplina de ingeniería, orientada a la producción de software, y la Ingeniería del Software como *ciencia* que estudia la ingeniería, es decir, como disciplina científica cuyo objeto no es producir software, sino estudiar, comprender, explicar y teorizar sobre la producción de software.
- Conocer los fundamentales enfoques de carácter científico de la Ingeniería del Software entendida como *ciencia de la ingeniería*.

1.3 Introducción

Desde su bautismo oficial en la conferencia promovida por la división de asuntos científicos de la OTAN en 1968, la Ingeniería del Software ha sido objeto de diferentes definiciones. Si bien distintas, todas estas definiciones han compartido la intención de trazar una diferencia entre la ciencia de la computación (*Computer Science*) y la Ingeniería del Software (*Software Engineering*). A este respecto, en la segunda de las conferencias organizadas por la OTAN en 1969, C. Strachey de la Universidad de Oxford hacía la siguiente reflexión:

«Ahora creo que no tendremos un estándar apropiado de programación. Que no haremos posible una disciplina de Ingeniería del Software hasta que podamos tener estándares profesionales apropiados sobre cómo escribir programas. Y esto tiene que hacerse enseñando a la gente, desde el principio, cómo escribir programas de manera correcta.»

El énfasis de Strachey sobre cómo escribir programas refleja la importancia del método en la Ingeniería del Software. Es decir, de los pasos y modos adecuados para desarrollar software. De un modo u otro, esta idea se repite en todas las definiciones de la disciplina, aunque la visión de la misma se haya ampliado notablemente. Actualmente, la Ingeniería del Software se trata desde la perspectiva de grupos de ingenieros (programadores y diseñadores fundamentalmente, pero también otros roles profesionales como gestores del proceso de desarrollo, analistas, etc.) y no desde la perspectiva de un programador aislado.

La definición posiblemente más utilizada de *Ingeniería del Software* es la que propone el Glosario IEEE de Términos de Ingeniería del Software (IEEE, 1990):

1. La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, la operación y el mantenimiento del software. Esto es, aplicar la ingeniería al software.
2. El estudio de enfoques como los mencionados en (1).

Según la primera de estas definiciones, el ingeniero de software es un «desarrollador» en sentido amplio, que desempeña un rol como profesional en la producción de software. Por su parte, la segunda de las definiciones implica la investigación y estudio de las actividades de la Ingeniería del Software, pero no el producir software. Así, define para el ingeniero de software un perfil de «investigador».

Como se ve, estas dos definiciones cubren tanto el aspecto profesional como el aspecto de investigación de la ingeniería. Si bien este libro tratará fundamentalmente de los contenidos y métodos relacionados con la primera definición, también incluye material introductorio para la segunda de las definiciones, ya que en ocasiones el trabajo del ingeniero del software se encuentra en la frontera de ambas.

En el resto del capítulo, estudiaremos en primer lugar qué es la ingeniería en general para, inmediatamente después, definir el objeto de la disciplina, es decir, el *software*. A continuación volveremos la vista atrás para repasar brevemente la historia de la disciplina, sus orígenes y su evolución. Una vez situados en contexto, abordaremos el estudio de los conceptos básicos de la disciplina, para finalmente analizar las tres características fundamentales de la Ingeniería del Software: *sistematicidad*, *disciplina* y *cuantificación*.

1.4 ¿Qué es la ingeniería?

Sunny Auyang definió *ingeniería* de la siguiente manera:

«*La ingeniería es la ciencia de la producción, la cual, junto a la reproducción, es la más fundamental de las actividades humanas*» (Auyang, 2004)

Esta asociación de la ingeniería con la producción es fácilmente identificable en muchas ramas de la ingeniería moderna. Un ingeniero civil, por ejemplo, se especializa en diseñar y construir obras públicas tales como puentes o carreteras, mientras que un ingeniero químico

se especializa en la aplicación industrial de la química. Hoy además se denomina ingeniería a ciertas sub-disciplinas o aspectos muy concretos de disciplinas existentes. Así por ejemplo, dentro de las ciencias de la computación, se han hecho populares la *ingeniería de la usabilidad*, cuyo objeto es diseñar y construir interfaces persona-computadora, y la *ingeniería del conocimiento*, cuyo objeto es producir representaciones del conocimiento para un dominio o propósito dado. En los mismos textos de Ingeniería del Software, suelen encontrarse referencias a la *ingeniería de requisitos* como sub-disciplina.

Como se ve, en todos los casos queda explícito en la propia definición de la disciplina el objeto que se construirá o producirá. De hecho, la historia de la ingeniería como hoy la entendemos es de algún modo la historia de la primera revolución industrial, momento histórico en que surge la mecanización de los medios de producción. En el terreno del conocimiento, la transición de la artesanía a la ingeniería es fundamentalmente el paso del pensamiento práctico desde la intuición hasta el método científico, y de la tutela de aprendices a la educación formal universitaria.

Para continuar el análisis, examinemos las siguientes definiciones comunes del término *ingeniería* (traducidas del diccionario Merriam-Webster):

1. Aplicación de la ciencia y las matemáticas por la cual las propiedades de la materia y las fuentes de energía de la naturaleza se hacen útiles para la gente.
2. Diseño y manufactura de productos complejos.

Realmente, la definición (2) no es suficiente para distinguir la ingeniería de otras actividades humanas, debido a que, por ejemplo, ciertas artesanías diseñan y producen elementos que pueden considerarse como relativamente complejos. Sin embargo, la ingeniería tal y como hoy la entendemos siempre resulta en algún *artefacto* concreto. Estos artefactos pueden ser utilidades finales (por ejemplo, una vivienda o una aplicación software para la gestión empresarial), o elementos para ser reutilizados en otros procesos de ingeniería (por ejemplo, un nuevo material para el aislamiento térmico de viviendas o una biblioteca de funciones para la resolución de ecuaciones). En cuanto a la Ingeniería del Software, su resultado útil son las aplicaciones, de las cuales los usuarios se sirven para hacer más eficaz, controlado o eficiente su trabajo.

Por todo ello, en este libro adoptaremos la siguiente definición de ingeniería:

La ingeniería como actividad humana es la aplicación del conocimiento y los métodos científicos al diseño y la producción de productos complejos

La anterior definición habla de la ingeniería en general, y no de las diversas ingenierías concretas. Si se estudian las diferentes ingenierías concretas, cada una de las cuales tiene un objeto concreto y definido, que han evolucionado hasta alcanzar el reconocimiento como profesión, es relevante el entorno en que éstas se desarrollan. Es importante porque el ejercicio de toda profesión debe ser regulado por un marco jurídico-normativo específico. En

las ingenierías, por ejemplo, es común la existencia de *colegios profesionales*, asociaciones de profesionales que regulan la actividad de un área de conocimiento y ordenan el ejercicio de la profesión.

La estructuración de las diferentes ingenierías como disciplinas profesionales reconocidas se puede encontrar en el trabajo seminal de Paul Starr (1982), donde se enuncian los tres elementos que constituyen una disciplina profesional:

- Aspecto colegial: el conocimiento y competencia del profesional debe haber sido validado por la comunidad de sus pares.
- Aspecto cognitivo: ese conocimiento y competencia consensualmente validado debe descansar en criterios racionales y científicos.
- Aspecto moral: el juicio y los consejos profesionales deben orientarse a un conjunto de valores sustantivos.

La aparición de la guía SWEBOK, de la que hablaremos más adelante, ha constituido un paso más en el aspecto colegial de la Ingeniería del Software, al recopilar lo que se ha denominado «cuerpo de conocimiento de la Ingeniería del Software», es decir, el conocimiento consensualmente aceptado y que descansa en criterios racionales y científicos. Con independencia de la credibilidad o validez que se le dé a la guía SWEBOK —pues existen posturas encontradas en este asunto—, es desde luego un indicador de que la disciplina de la Ingeniería del Software ha entrado en una fase de madurez y de autoconsciencia que la sitúa al mismo nivel de otras ramas tradicionales de la ingeniería.

1.5 Ingeniería y ciencias de la ingeniería

Las diferentes ramas o disciplinas ingenieriles difieren en el objeto de la producción, pero todas ellas tienen en común tres aspectos específicos:

- La *ciencia de la ingeniería*, que se ocupa de los principios y mecanismos subyacentes de la disciplina.
- Procesos de *diseño*, que en general incluyen una fase de conceptualización, y una fase de diseño detallado.
- Aspectos de *gestión y organización*, pues la tecnología que se produce implica tanto a las personas como a las organizaciones. Además, las propias personas que crean tecnología no suelen trabajar aisladas, sino en equipos y organizaciones.

En el caso de la Ingeniería del Software, las actividades de diseño (empleado en el sentido general de la ingeniería) serían asimilables a lo que normalmente conocemos como

"desarrollo"¹. Ahora bien, ¿cuál es la ciencia de la ingeniería que nos interesa cuando estudiamos la Ingeniería del Software? Evidentemente, la ciencia de la computación (*computer science*) está asociada a esta ingeniería, pues abarca los principios matemáticos y físicos, en su sentido más amplio, de los sistemas basados en computadora. No obstante, es importante distinguir claramente entre *ciencia de la computación* e *Ingeniería del Software*, ya que lo específico de esta última es lo concerniente al diseño y uso del software, utilizando el conocimiento que es el objeto de la *ciencia de la computación*. Ahora bien, dentro de la ciencia de la Ingeniería del Software, hay que separar los conocimientos científicos *que se aplican* en la Ingeniería del Software, la ciencia de la Ingeniería del Software *en sí misma*, y la práctica de la ingeniería:

- Las **ciencias que se aplican** en la Ingeniería del Software son la ciencia de la computación y otras ciencias que son de utilidad para aspectos determinados, como las relativas a la organización, la economía, la psicología, y por supuesto, las matemáticas en general. Para dominios muy concretos también se necesitan conocimientos específicos de ciertas ciencias. Así, en la Ingeniería del Software aeroespacial se requieren conocimientos de física, mientras que para el desarrollo de software para biotecnología, serían necesarios ciertos conocimientos de biología.
- La **Ingeniería del Software como ciencia** es la aplicación del método científico a la teorización y creación de conocimiento sobre la propia Ingeniería del Software. Está dedicada al estudio de sus actividades, y centrada en generar teorías, modelos explicativos o enunciados descriptivos sobre la práctica de la ingeniería. Las «*Leyes de la Evolución del Software*» (Lehman y Ramil, 2003), por ejemplo, son enunciados teóricos sobre el mantenimiento del software. Uno de estos enunciados dice que *«a medida que un programa evoluciona, su complejidad aumenta a menos que se dedique esfuerzo específico a reducir o mantener constante dicha complejidad»*. En realidad muchos ingenieros del software desconocen estas leyes aunque en su práctica profesional las tengan en cuenta tácitamente.
- La **práctica de la ingeniería**, que está orientada a prescribir cómo deben realizarse las actividades propias de la disciplina. Es un aspecto complementario con la ciencia de la ingeniería, pues la ciencia necesita de la observación de la práctica, y la práctica a su vez se perfecciona de acuerdo con el conocimiento generado por la ciencia.

A pesar de que este libro trata fundamentalmente de la praxis de la Ingeniería del Software, también se expondrán puntualmente partes de lo que constituye el cuerpo de conocimiento científico de la disciplina.

¹Es importante no confundir el término diseño de software, que es una fase concreta de la Ingeniería del Software, con el uso habitual de la palabra diseño (a secas) que suele utilizarse en un sentido más amplio.



Figura 1.1: Relación entre las ciencias de la computación y la Ingeniería del Software

1.6 El software como artefacto tecnológico

El software es la tecnología o producto resultante de las actividades de Ingeniería del Software. Pero téngase en cuenta que el software tiene una naturaleza que lo diferencia de otros productos de la ingeniería moderna, lo que hace que tenga una problemática también especial. De hecho, desde los años sesenta, se ha venido utilizando el término «crisis del software» para hacer referencia a ciertos problemas específicos y persistentes de la Ingeniería del Software.

Algunos autores han llegado a considerar estos problemas como una enfermedad crónica (en lugar de una simple crisis) para resaltar el carácter persistente de los mismos. Podemos clasificar estos problemas como:

- Problemas asociados al desarrollo, como los retrasos en los plazos de los proyectos, o la baja productividad de los desarrolladores.
- Problemas de uso de los productos finales, como por ejemplo, deficiencias de calidad.

Aparte del evidente problema de carácter económico que provocan los retrasos y las deficiencias, existen además consecuencias de carácter social cuyo resultado es, en último término, una cierta resistencia a la adopción del software, y una merma en la credibilidad de los proyectos. Es un hecho, a la vista de estos problemas, que la Ingeniería del Software es una actividad compleja y difícil de gestionar. Esta complejidad se debe, en buena medida, a la propia naturaleza del software como artefacto. En el resto de esta sección abordaremos la definición del software y estudiaremos su complejidad.

1.6.1 ¿Qué es el software?

El término *software* se suele atribuir a John W. Tukey quien, en un artículo publicado en 1957 en la revista *American Mathematical Monthly*, introdujo por vez primera el término. La idea de software de los años 50 era prácticamente un sinónimo del término «programa de computadora», es decir, un artefacto que proporciona las instrucciones necesarias para que una computadora lleve a cabo una cierta tarea.

Esta definición es, en la actualidad, demasiado específica. Una definición más amplia es la que proporciona el diccionario Merriam-Webster, cuya traducción reproducimos aquí como la definición que consideraremos en este libro.

Software es el conjunto completo de programas, procedimientos y documentación relacionada que se asocia con un sistema, y especialmente con un sistema de computadora. En un sentido específico, software son los programas de computadora

Por tanto, software no son sólo los programas de computadora en sí, sino también los documentos que lo describen (como por ejemplo los manuales de usuario), así como cualquier otro artefacto relacionado con el mismo, como los procedimientos para su instalación o modificación, e incluso los datos necesarios para su operación.

Un aspecto adicional del software es el hecho de que en su mayoría, está destinado a evolucionar. Y de hecho, si no lo hace quedaría obsoleto con el tiempo, al no adaptarse a las cambiantes necesidades de los usuarios. La evolución implica generalmente añadir nuevas funcionalidades o modificar las existentes, si bien la evolución del software es diferente a la de los diseños de otros artefactos ingenieriles.

Precisamente porque el software es fácilmente modificable, y por tanto flexible, es un producto que permite y suele tener muy en cuenta su cambio y evolución en el tiempo. No cabe duda de que esta característica es un elemento que distingue al software de otras creaciones humanas. Otros productos de ingeniería, como los aviones o los automóviles, no cambian tanto ni tan fácilmente durante su vida útil, aunque la actual moda del *tuning* de vehículos pueda hacer pensar lo contrario.

1.6.2 La complejidad inherente al software

En 1987, Frederick Brooks publicó un artículo que ha adquirido una notable popularidad. En dicho artículo, cuyo título puede traducirse como «*No existen balas de plata - Esencia y accidente en la Ingeniería del Software*», Brooks preconizaba que no existiría ninguna herramienta o metodología que consiguiese un incremento notable de la productividad en el desarrollo de software en la década siguiente. Aparte de la anécdota sobre la predicción, lo importante de este artículo es que establece el concepto de complejidad como característica esencial al software, entendiendo esencial en su sentido de «inherente», es decir, propia de la naturaleza del software.

Brooks argumenta que si la complejidad del software fuese *accidental*, podrían crearse herramientas o técnicas para gestionarla y eliminar el problema. Según este autor, la representación de los conceptos en programas (codificación), y la comprobación de su fidelidad (pruebas) son aspectos accidentales, para los cuales sí pueden crearse herramientas que gestionen su complejidad. Sin embargo, la esencia de la Ingeniería del Software es la *especificación, diseño y verificación de un conjunto detallado y muy preciso de conceptos interrelacionados*, tareas sensiblemente más complejas que las anteriores. En definitiva, la traducción final de las especificaciones a código ejecutable no es el problema, el problema es la elaboración de las propias especificaciones.

Además de esa complejidad inherente al software, la Ingeniería del Software, al igual que el resto de las ingenierías, está limitada en la práctica por restricciones económicas, de plazos, regulaciones y otros factores como la gestión de recursos humanos. Profundizando más aún, Brooks menciona otras dos causas de complejidad: la propensión al cambio y la invisibilidad del software. La primera es consecuencia del uso del software, ya que al ser utilizado, debe adaptarse a nuevos requisitos y necesidades. La invisibilidad del software, por su parte, se refiere al hecho de que el software no se pueda representar completamente mediante diagramas, pues dicha representación sería extremadamente compleja, un plus de complejidad en la comprensión del mismo.

Junto con todo lo expuesto, la complejidad del software en sí mismo tiene como reflejo lógico la complejidad en su diseño. Ya en 1968, Peter Naur afirmaba lo siguiente: «*El problema de determinar cuál es el orden adecuado de hacer las cosas durante el diseño es actualmente un tema para la investigación en Ingeniería del Software*».

En conclusión, a pesar de que ha habido y sigue habiendo numerosos intentos de afrontar la complejidad inherente descrita por Brooks (tales como la orientación a componentes o las técnicas de especificación avanzadas), la Ingeniería del Software sigue siendo un proceso con una dificultad intrínseca característica, que debe ser tenida en cuenta al afrontar el diseño y la organización de las actividades de ingeniería.

1.7 Sistematicidad, disciplina y cuantificación

En las anteriores secciones hemos mencionado que la ingeniería, como disciplina de carácter industrial y profesionalizada, se diferencia de la artesanía o de otras actividades humanas en su nivel de estructuración y su carácter auto-reflexivo. En la definición de Ingeniería del Software con que comenzábamos este capítulo, se mencionan tres calificativos que pueden aplicarse a la ingeniería: sistematicidad, disciplina y cuantificación. Detengámonos brevemente para reflexionar sobre estas tres características distintivas:

- Decimos que algo es *sistemático* cuando «sigue un sistema». Así, diremos que una actividad es sistemática cuando es metódica en cuanto al procedimiento o al plan.
- Decimos que una actividad es *cuantificable* si tanto su realización como sus resultados pueden medirse. En cuanto al software, tanto el producto final del desarrollo, como

el propio proceso de desarrollo del software en sí mismo pueden ser sometidos a medición, y generalmente lo son.

- Una actividad es *disciplinada* si está sujeta a control con respecto a ciertos estándares, entendiendo el término «estándar» en su acepción más genérica de *norma* o *patrón*, no como especificación formal respaldada por un organismo de estandarización.

Lo dicho reafirma la noción de que la programación como actividad casual o esporádica no puede ser considerada Ingeniería del Software, lo que no quiere decir que dicha actividad no tenga valor o que no pueda producir resultados interesantes.

PSP: Un ejemplo simple de actividades

Watts S. Humphrey definió el Proceso de Software Personal (PSP) como un método de ingeniería para el trabajo personal propio de cada ingeniero. Inicialmente orientado a principiantes, el autor no lo propuso como sustitución de los métodos que afectan a grupos de ingenieros, sino como una visión personal y complementaria del trabajo diario de cada individuo especialmente adecuada para proyectos no demasiado grandes.

Entre las recomendaciones del PSP, merece la pena mencionar el *control del tiempo*, que se materializa a nivel personal en el registro de los tiempos que se han dedicado a cada tarea, así como de las interrupciones que se han tenido. Un resumen semanal del progreso resulta por tanto una buena forma de hacer un control del progreso (a nivel personal, se entiende). La medición del progreso se puede hacer contando las líneas de código fuente producidas, por ejemplo. De este modo, cada uno puede medir la producción de software, y qué factores hacen que dicha producción avance o se estanque.

El PSP enfatiza la medición detallada del trabajo individual, y su uso para valorar el progreso y la eficiencia en el desarrollo. Su interés radica en que reproduce a escala personal los problemas más importantes de la gestión de la Ingeniería del Software (estimación, planificación, seguimiento, etc.).

En el caso de que la sistematicidad y la disciplina se consideren desde la perspectiva individual, sólo hay ingeniería cuando hay un plan (aunque sea tácito) y una referencia sobre cómo se deben hacer las cosas. Un ejemplo de esta perspectiva individual es el denominado «Proceso de Software Personal» (PSP), un conjunto de buenas prácticas para el desarrollo de software que se centra en la disciplina individual de los ingenieros del software (ver cuadro sobre PSP).

En general, el concepto de **método** en ingeniería captura la sistematicidad y disciplina. En Ingeniería del Software existen métodos para todo tipo de actividades: desde la toma de requisitos hasta la decisión de qué cambios se deben hacer al software después de terminado. No obstante, es importante resaltar que los métodos nunca pueden ser únicos ni definitivos. No pueden ser únicos porque hay diferentes contextos, organizaciones y tipos de aplicaciones, y no existen métodos «de talla única». Tampoco pueden ser definitivos, ya que

la propia tecnología cambia constantemente, y las restricciones existentes, digamos, para la programación de grandes computadoras en los años sesenta, son radicalmente diferentes de las condiciones del desarrollo, por ejemplo, de las aplicaciones para teléfonos móviles.

1.8 La Ingeniería del Software como disciplina profesional

En el informe de la conferencia de la OTAN sobre Ingeniería del Software de 1968, podemos encontrar esta sentencia rotunda sobre el estado de la disciplina:

«Hubo un acuerdo general en que la Ingeniería del Software se encuentra en un estado de desarrollo muy rudimentario en comparación con las ramas de la ingeniería bien establecida».

No obstante, ha habido una considerable evolución en los últimos años. En esta sección repasamos brevemente algunos hitos históricos, y hacemos referencia a los esfuerzos que algunas organizaciones que tienen que ver con la disciplina de la Ingeniería del Software están llevando a cabo.

1.8.1 Breve historia de la Ingeniería del Software

Como se mencionó al principio del capítulo, suele considerarse como evento de bautismo oficial de la disciplina la primera de las conferencias sobre Ingeniería del Software patrocinadas por la OTAN celebrada en 1968. No obstante, el aspecto profesional del desarrollo de programas ya había sido objeto de atención en reuniones y estudios anteriores. La Tabla 1.1 resume en tres períodos la historia de la Ingeniería del Software.

Tabla 1.1: Historia de la Ingeniería del Software

Período	Fase	Descripción
1955–1965	Orígenes de la disciplina	Desarrollo inicial de los principios de la Ingeniería del Software (hasta las conferencias de la OTAN).
1965–1985	Identificación de la crisis	La identificación del problema de la crisis del software se convierte en el tema central de la disciplina.
1985–	Desarrollo de la disciplina	Aproximadamente desde la publicación del artículo de Brooks (1987) se desarrolla la disciplina con la complejidad del desarrollo de software como elemento inherente.

La metodología, es decir, la búsqueda de marcos normativos para las actividades de la Ingeniería del Software, ha sido un elemento central en el desarrollo de la disciplina. En cualquier caso, es importante resaltar que desde los inicios de la disciplina existen recomendaciones curriculares internacionales para estudios de grado que diferencian la Ingeniería del Software de otras disciplinas relacionadas con la computación, como los *Sistemas de*



Figura 1.2: Una de las sesiones de la conferencia de la OTAN de 1968, donde se puede ver a los profesores Dijkstra, Naur y Randell (con gafas y barba, en la parte superior derecha)

Información, las Ciencias de la Computación y otras. En las recomendaciones más ampliamente aceptadas se considera que la Ingeniería del Software es más que codificación, pues incluye calidad, planificación y aspectos económicos, así como el conocimiento y aplicación de principios y disciplina.

1.8.2 Elementos de la Ingeniería del Software como disciplina profesional

En muchas disciplinas de ingeniería, la acreditación de los profesionales y la existencia de directrices comunes para la elaboración de currículos y planes de estudio son asuntos a los que se presta especial atención. El reconocimiento de un cuerpo de conocimiento para la Ingeniería del Software, así como la creación de mecanismos de acreditación, era una asignatura pendiente hasta que las dos organizaciones más activas y relevantes en el área, ACM e IEEE Computer Society, comenzaron a promover activamente su puesta en práctica.

Estas dos organizaciones, reconocidas a nivel internacional, incluyen dentro de su ámbito de interés la Ingeniería del Software, si bien no se limitan exclusivamente a ella. Lo importante es que aunaron esfuerzos para desarrollar una recomendación curricular conjunta para titulaciones de Ingeniería del Software, conocida como CCSE (*Computing Curriculum Software Engineering*) y oficialmente denominada SE2004 (*Software Engineering 2004*). SE2004 complementa a la guía SWEBOK, no orientada al diseño de programas universitarios, sino a delimitar los conocimientos necesarios para el ejercicio profesional más allá de la educación formal.

Los dos documentos son, cada uno en su terreno, el resultado de un proceso gradual y mantenido de consolidación de la disciplina:

- El **SE2004** es un esfuerzo conjunto de la ACM y la IEEE Computer Society que forma parte de un conjunto de directrices curriculares, en forma de volúmenes separados, uno por cada área importante de la computación. Así, existe un volumen de ciencias de la computación, otro de la ingeniería de la computación y otro volumen de sistemas de información. El SE2004, que no es sino el volumen dedicado a la Ingeniería del Software, gira alrededor de tres elementos: el desarrollo de directrices curriculares, la diseminación y especificación del conocimiento a incluir en los planes de estudio de Ingeniería del Software, y la construcción de un conjunto de recomendaciones que describen cómo estructurar un currículo de Ingeniería del Software.
- La guía **SWEBOK** surge por el deseo de IEEE de crear una acreditación para la profesión de ingeniero del software en Estados Unidos, distinta de la certificación para la profesión de *informático o científico de la computación*. Con este objetivo se recopilaron los esfuerzos realizados para definir el cuerpo de conocimiento de la Ingeniería del Software durante sus cuatro décadas de existencia y se plasmaron en esta guía, verdadero compendio del conocimiento asociado a la disciplina de Ingeniería del Software. En esencia, los conocimientos compilados corresponden a lo que un profesional titulado en Ingeniería del Software debería tener tras cuatro años de experiencia profesional. Los objetivos fundamentales la guía SWEBOK son:
 - Caracterizar los conocimientos del cuerpo de conocimiento de la Ingeniería del Software.
 - Promover una visión consistente y universal de la disciplina.
 - Establecer las diferencias entre la Ingeniería del Software y otras disciplinas relacionadas, como las ciencias de la computación, la gestión de proyectos, o las matemáticas.
 - Servir de base para la certificación de profesionales.

En cuanto a su contenido, la guía SWEBOK describe las áreas principales de la Ingeniería del Software, a las que denomina «áreas de conocimiento». A cada área le dedica un capítulo separado, en el que además de abordar los contenidos fundamentales del área se resumen las referencias clave de la misma.

A esta consolidación se tiene que añadir la existencia de numerosas conferencias específicas de la Ingeniería del Software, así como de revistas de investigación especializadas sobre el tema. No obstante todo lo anterior, la profesión de ingeniero del software aún se percibe como diferente a otras ingenierías. Esto es así incluso dentro de la misma profesión. El siguiente texto, adaptado del original en inglés de John McCormick, es ilustrativo de ciertas percepciones de la disciplina:

«En una conferencia de desarrolladores de software, uno de los ponentes lanzó la siguiente pregunta a los asistentes: —“Si estuviesen ustedes subiendo a un avión y les dijeran que el software de control del aparato fue desarrollado por el equipo de programadores de su empresa, ¿quién de los presentes desembarcaría de inmediato?”. Entre el bosque de manos alzadas sólo una persona permaneció sin moverse. El ponente se dirigió a esa persona para preguntarle qué haría él, quien con un aire tranquilo replicó: —“No me preocuparía en absoluto, con el software de mi empresa es muy poco probable que el avión pudiese ni siquiera despegar”».

Todos los esfuerzos mencionados (la guía SWEBOK y las directrices curriculares del SE2004, así como la existencia de una comunidad científica activa en torno a las revistas y conferencias del área), junto con la consolidación de las titulaciones en Ingeniería del Software, han servido para que poco a poco la profesión vaya teniendo reconocimiento en la sociedad y para que la industria demande específicamente este tipo de expertos.

1.9 Conceptos básicos de la Ingeniería del Software

Como toda ingeniería, donde se crean objetos con una cierta función, la Ingeniería del Software trata fundamentalmente de **actividades** llevadas a cabo por **personas** (ingenieros, pero también en cierta medida, usuarios u otros intervinientes) que producen, usan o modifican **artefactos**. Esas actividades no son espontáneas sino que responden a planes parcial o totalmente prescritos (esto es, son *sistemáticas* y *disciplinadas*, según la definición propuesta anteriormente). Por ello, hay considerar también elementos tales como **métodos**, **especificaciones** y **modelos**, entre otros.

En esta sección se describe la terminología más básica de la Ingeniería del Software con el objeto de dar coherencia al resto del libro. Aunque muchas de las definiciones pueden parecer en este punto obvias o ingenuas, es importante darles un significado preciso para su uso y aplicación de aquí en adelante.

1.9.1 Actividades y artefactos

La ingeniería se desarrolla en la forma de *actividades* de ingeniería. Toda actividad, por ejemplo consultar el email por la mañana, sucede en un intervalo temporal determinado y se distingue por los elementos que intervienen en la misma. En el ejemplo, la computadora y el programa gestor del email distinguen esta actividad de otras. Las actividades también tienen lugar en una ubicación concreta del espacio, y tienen unos elementos participantes.

Una actividad es un proceso que tiene lugar en el tiempo y en el espacio, y en el cual un agente actúa con unos objetivos determinados

Las actividades en la Ingeniería del Software abarcan por tanto cualquier acción con un propósito claro dentro de esta ingeniería, lo que incluye actividades de gestión, producción, comunicación y documentación. Existen diversos términos relacionados que se usan con sentidos parecidos al de actividad en la Ingeniería del Software, tales como proceso, acción, evento o tarea. Por el momento, el término *actividad* es suficientemente genérico para referir *aquellos que se hace* en la Ingeniería del Software. Cuando más adelante se considere necesario, se introducirán términos más específicos tales como actividades de prueba, de obtención de requisitos, etc.

Muchas de las actividades en la Ingeniería del Software están orientadas a obtener un producto concreto, tales como una especificación, un documento o código fuente. El término *artefacto* se utiliza cada vez con más frecuencia para denotar los elementos de información que se usan o producen en la Ingeniería del Software.

Un **artefacto** es algo tangible creado con un propósito práctico

Son artefactos de la Ingeniería del Software todos aquellos elementos creados en actividades propias de la disciplina, tales como el código, los documentos o los diagramas, entre otros. Todos los artefactos tienen un carácter de «elementos de información», ya que todos son susceptibles de proporcionar información en el proceso de ingeniería. La especificación del lenguaje UML, por ejemplo, de la que se hablará especialmente en los Capítulos 4 y 5, considera que son artefactos «los modelos, las descripciones y el software», de manera muy genérica. Es importante diferenciar entre el resultado conceptual y el documento en que aparece, ya que en ocasiones los documentos finales incluyen solamente una parte de las informaciones generadas durante el desarrollo.

Por tanto, la realidad de la Ingeniería del Software se materializa en términos de actividades, de sus participantes y de los artefactos que producen, transforman o utilizan. Esta caracterización, no obstante, no tiene en cuenta que las actividades no se producen de manera anárquica o casual, pues siguen un guión establecido. Según la definición dada al principio de este capítulo, debe haber un enfoque sistemático y disciplinado. A continuación se discuten conceptos relacionados con esos aspectos.

1.9.2 Métodos, especificaciones y modelos

Las actividades que tienen lugar y los artefactos que se crean en la ingeniería en general tienen asociadas una serie de prescripciones, es decir, están sujetos a normas que dictan cómo deben hacerse. Estas normas provienen o bien del conocimiento científico (por ejemplo, la necesidad de hacer las pruebas de una manera determinada), o bien de la experiencia, o bien de la intuición o el sentido común de una persona o grupo. Vengan de donde vengan, es claro que la actividad de ingeniería está sujeta a ciertas prescripciones; el término *método* es uno de los más utilizados para referirse a ellas. Según la guía SWEBOK, los métodos

«imponen estructura a la actividad de Ingeniería del Software con el objetivo de hacerla más sistemática y finalmente más exitosa».

Un **método**, en sentido general, es la especificación de una secuencia de acciones orientadas a un propósito determinado. En la Ingeniería del Software, los métodos determinan el orden y la forma de llevar a cabo las actividades.

El término *metodología* hace referencia al estudio de los métodos (sea general o específico de una disciplina), aunque también puede utilizarse para hacer referencia a un conjunto coherente de métodos.

En la Ingeniería del Software, se denomina **metodología** a un conjunto de métodos coherentes y relacionados por unos principios comunes

Es habitual en Ingeniería del Software hablar de términos tales como «metodologías estructuradas» o «metodologías orientadas a objetos», haciendo referencia a la primera de las acepciones del término.

La propia definición de método introduce otro término importante en la ingeniería: el término *especificación*. Las especificaciones, como elemento de información, son una parte fundamental de toda disciplina de ingeniería.

Una **especificación** es una descripción detallada y precisa de algo existente (o que existirá) o de una cierta situación, presente o futura

Para elaborar las especificaciones se emplean lenguajes o notaciones de diferente tipo. En muchas ocasiones, se utiliza el lenguaje natural (como el español o el inglés), pero esto a veces no es adecuado, bien por facilidad de comunicación, o bien por falta de precisión en la expresión. Por ello, hay lenguajes «visuales», que emplean diagramas e iconos para facilitar la comunicación. También existen lenguajes «formales», en los que se utiliza notación matemática para alcanzar un grado mayor de precisión y eliminar las ambigüedades. En la Ingeniería del Software, una especificación del software que se desea construir da lugar a especificaciones ejecutables denominadas programas de computadora.

1.9.3 Procesos y ciclos de vida

El término *actividad*, tal y como se ha descrito, proporciona una descripción muy general de lo que se hace en la Ingeniería del Software. Aunque finalmente todo se reduce a actividades, cada método o modelo emplea su propia jerga.

Un concepto muy utilizado es el de *ciclo de vida del software*. La definición más común lo retrata como el período de tiempo «que comienza cuando se concibe un producto software y termina cuando el producto deja de usarse», definición relativa a un proyecto determinado que bien puede reformularse en referencia a las actividades. La utilidad de este concepto es resaltar que el ciclo de vida no se restringe a las actividades de desarrollo previas al uso del software, sino que abarca también su evolución y mantenimiento. Así, puede también abarcar una fase de concepción previa al mismo inicio de las actividades de ingeniería.

El ciclo de vida de un producto o proyecto software es la evolución del mismo desde el momento de su concepción hasta el momento en que deja de usarse, y puede describirse en función de las actividades que se realizan dentro de él

En la literatura sobre Ingeniería del Software es frecuente encontrar menciones al «ciclo de vida en cascada» o al «ciclo de vida en espiral». Debe quedar claro que en estos casos no se está haciendo referencia al ciclo de vida de un software determinado, sino a una especificación de cuáles deben ser las fases o el curso general de la Ingeniería del Software. En realidad sería más conveniente emplear la locución **modelo de ciclo de vida del software** cuando se desea utilizar esta última acepción.

A lo largo de la historia han surgido diferentes modelos generales de ciclo de vida del software, con una complejidad creciente en el desarrollo de las secuencias de actividades. El modelo más antiguo era en cierto modo heredero de la forma de producción en las ingenierías tradicionales. Este modelo, denominado «ciclo de vida en cascada», proponía una secuencia de fases claramente reconocibles por los desarrolladores, a saber, estudio de viabilidad, requisitos, análisis, diseño, codificación, pruebas y mantenimiento. El supuesto de este modelo de ciclo de vida es la progresión secuencial, lo cual se basa en el hecho de que las fases no se solapan. En modelos de ciclo de vida modernos, tales como el *Proceso Unificado*, y debido fundamentalmente a la naturaleza intangible y fácilmente modificable del software, las fases se solapan y el software se produce en un proceso iterativo e incremental.

Un término relacionado que también se usa con profusión es el de *proceso*. Dado que el término *proceso software* ha adquirido un significado especial, parece conveniente aportar una definición:

Un proceso software es un conjunto coherente de políticas, estructuras organizativas, tecnologías, procedimientos y artefactos que se necesitan para concebir, desarrollar, implantar y mantener un producto software

Por su parte, el Glosario IEEE de Términos de Ingeniería del Software describe proceso como «una secuencia de pasos llevados a cabo para un propósito específico; por ejemplo, el proceso de desarrollo de software». Lo cierto es que muchas de las definiciones del término hacen referencia a la finalidad como elemento fundamental del proceso, según el cual todo

proceso tiene unos objetivos definidos. Sin embargo, un proceso según la definición del glosario IEEE no es otra cosa que una secuencia de actividades que comparten un propósito. En la definición que proponemos se recoge dicho significado.

Una vez más, hay que distinguir claramente entre el proceso como realización de ciertas actividades, y el modelo de proceso, que es la especificación de una manera concreta de proceder en la ingeniería. En realidad, puede afirmarse que un modelo de proceso es equivalente a una metodología en el terreno de la Ingeniería del Software. De hecho, los modelos de ciclo de vida no son otra cosa que modelos de procesos, pero con una diferencia de énfasis, pues un modelo de ciclo de vida solamente describe las fases fundamentales, y es por tanto una abstracción de muy diversos procesos. Es importante resaltar que no existe un único proceso correcto para la Ingeniería del Software, y sí un cierto número de procesos concretos, de muy diferente índole. Por eso, cuando se hable en términos genéricos de «el proceso software», no se hará referencia a uno en concreto sino a cualquiera de ellos.

Resumen

En este capítulo introductorio hemos situado la Ingeniería del Software en el contexto del resto de las disciplinas de ingeniería, resaltando aquellos aspectos peculiares de la misma que tienen su origen en el software, un producto de características especiales.



Figura 1.3: Principales conceptos tratados en el capítulo

La disciplina puede verse como un conjunto de actividades de propósito específico, que dan como resultado ciertos artefactos. Estas actividades no se desarrollan de manera casual, sino que siguen métodos que prescriben qué formas de hacer las cosas serán más efectivas según las circunstancias. Hemos visto además los fundamentos científicos que hacen de la Ingeniería del Software no sólo una ingeniería sino también una *ciencia de la ingeniería*.

Notas bibliográficas

Brian Randell ha dedicado muchos años de esfuerzo a investigar la historia de la computación, y estuvo presente en las conferencias seminales organizadas por la OTAN en 1968 y 1969 que dieron lugar al nacimiento de la Ingeniería del Software. Actualmente profesor emérito de la Universidad de Newcastle, ofrece a través de su página web la posibilidad de consultar las actas de las reuniones de la OTAN en 1968 (Naur y Randell, 1969) y 1969 (Randell y Buxton, 1970), lo cual resulta una lectura muy enriquecedora e interesante.

En «*Engineering – an endless frontier*», Sunny Auyang realiza un interesante análisis de las diferentes ramas de la ingeniería (Auyang, 2004). Muy completo, incluye la definición de ingeniería que se propone en el capítulo. Resultará interesante a todos los que deseen comprender la importancia de la ingeniería en la sociedad humana. Se recomienda asimismo leer la introducción de la guía SWEBOK para adquirir una idea general de los contenidos de la práctica profesional de la Ingeniería del Software.

Para aquellos lectores interesados en saber más sobre el proceso de software personal (PSP), «*Introduction to the Personal Software Process*» (Humphrey, 1996) resultará sin duda de gran utilidad, ya que proporciona no sólo las bases teóricas del método sino también ejercicios prácticos para ayudar a adquirir las buenas prácticas de gestión del tiempo y aseguramiento de la calidad que el método propone.

Cuestiones de autoevaluación

1.1 Cuando se habla de crisis del software ¿cuál es la fecha de comienzo y fin de esa crisis?

R. *La crisis del software no es un evento histórico concreto, sino un fenómeno asociado a la disciplina en sí misma. Dicho fenómeno se identificó en la década de los sesenta pero persiste en nuestros días.*

1.2 Rzone si la siguiente afirmación es o no cierta: «La Ingeniería del Software es una disciplina que trata la optimización del rendimiento de los sistemas informáticos».

R. *La Ingeniería del Software es una disciplina que estudia cómo desarrollar software de manera metódica de acuerdo con ciertas restricciones. En este sentido, el conocimiento sobre cómo optimizar el rendimiento de los sistemas informáticos es útil, pero no es el aspecto central de la disciplina.*

1.3 Indique si es cierta la siguiente afirmación y rzone su respuesta: «La Ingeniería del Software es la aplicación de las ciencias matemáticas al diseño de programas, por lo que la verificación de la corrección de los mismos se lleva a cabo de manera formal».

R. *El uso de técnicas formales en Ingeniería del Software es útil, especialmente en casos en los que se requiere una alta fiabilidad del diseño. No obstante, los métodos matemáticos son conocimiento y técnicas auxiliares para la disciplina.*

1.4 Siguiendo los razonamientos de Brooks ¿puede la Ingeniería del Software llegar a tener un método de desarrollo óptimo, que mejore de manera significativa la eficacia y productividad del desarrollo?

R. Si presuponemos que el software es un producto inherentemente complejo, no puede existir un método que elimine totalmente el esfuerzo necesario para tratar esa complejidad. Posiblemente aparecerán técnicas novedosas que proporcionarán mejores resultados, pero la complejidad inherente al software seguirá haciendo de la Ingeniería del Software una actividad con características especiales.

1.5 La definición de los roles profesionales en una organización de desarrollo ¿es un aspecto de la Ingeniería del Software?

R. Si lo es, dado que la Ingeniería del Software no concierne solamente a los aspectos técnicos del desarrollo sino también a los organizativos.

1.6 ¿Qué diferencia una medida de una métrica en Ingeniería del Software?

R. Esencialmente, una métrica es la interpretación de una o varias mediciones de un cierto atributo del software (el producto) o de las actividades de Ingeniería del Software (el proceso). Esta interpretación habitualmente se basa en estudios estadísticos que relacionan las medidas con el atributo observado.

1.7 ¿Cuáles son las tres características fundamentales de la Ingeniería del Software?

R. Sistematicidad (sus actividades siguen un procedimiento o sistema), disciplina (está sujeta al control con respecto a ciertos estándares) y cuantificación (tanto su realización como sus resultados pueden medirse).

1.8 En el contexto de la Ingeniería del Software ¿Qué diferencia un proceso de un método?

R. Un método es una forma de realizar una cierta actividad de Ingeniería del Software de manera sistemática y siguiendo pasos establecidos, mientras que un proceso se aplica a todo el ciclo de vida del software, o a conjuntos de actividades dentro del mismo.

1.9 ¿Puede considerarse que el registro de las horas de trabajo en cada módulo de los programadores en un proyecto de desarrollo de software es un artefacto de Ingeniería del Software?

R. Si puede considerarse como tal, ya que es un producto tangible de una actividad (de control, concretamente). Además, ese tipo de información es útil para la ingeniería por varios motivos, por ejemplo para medir cuantitativamente el esfuerzo con el fin de conocer más sobre la productividad de la organización.

1.10 El tamaño (en bytes) de los ficheros fuente en un desarrollo es una medida del tamaño del software, pero también puede utilizarse como medida la cuenta de las líneas de código o de las sentencias del lenguaje de programación. ¿Cuál de estas mediciones podría tener más sentido como métrica de complejidad del software?

R. El tamaño del software es importante como medición de la complejidad y el esfuerzo de desarrollo. El tamaño en bytes de los ficheros no es una medida muy interesante, ya que depende de factores como el estilo de codificación o la cantidad de comentarios, por ejemplo. La cuenta de las líneas de código adolece de similares problemas, pero la cuenta de las sentencias, por contra, sí resulta significativa, pues es independiente de los comentarios y la longitud de los identificadores.

Ejercicios y actividades propuestas

Ejercicios resueltos

- 1.1 El proceso de software personal (PSP) es una definición de proceso de Ingeniería del Software orientada a un uso individual, que se utiliza en ocasiones para aprender conceptos de procesos software. Una de las guías que proporciona es la de desarrollo. El siguiente es un fragmento del proceso personal.

Propósito: Guiarle en el desarrollo de programas pequeños.

- Entradas necesarias
 - Relación de requisitos.
 - Resumen del plan del proyecto con el tiempo planificado de desarrollo.
 - Cuadernos de registro del tiempo y de defectos.
 - Estándar de tipos de defectos.
- Diseño
 - Revisar los requisitos y realizar un diseño que los cumpla.
 - Registrar el tiempo en el cuaderno de registro del tiempo.
- Código
 - Implementar el diseño.
 - Registrar en el diario de registro de defectos cualquier defecto de requisitos o diseño encontrado.
 - Registrar el tiempo en el cuaderno de registro del tiempo.
- Compilar
 - Compilar el programa hasta que esté libre de errores.
 - Corregir todos los defectos encontrados.
 - Registrar los defectos el cuaderno de registro de defectos.
 - Registrar el tiempo en el cuaderno de registro del tiempo.
- Probar
 - Probar hasta que los casos de prueba diseñados funcionen sin error.
 - Corregir todos los defectos encontrados
 - Registrar los defectos el cuaderno de registro de defectos.
 - Registrar el tiempo en el cuaderno de registro del tiempo.
- Criterios de salida
 - Un programa completamente probado.
 - El cuaderno de registro de defectos completado.
 - El cuaderno de registro del tiempo completado

A partir de lo anterior, conteste a las siguientes preguntas. ¿Puede considerarse como un método el registro de defectos? ¿Son las pruebas un artefacto en el proceso? ¿Se puede considerar lo anterior como un método de Ingeniería del Software?

Solución propuesta: el registro de defectos es una política de control que resulta útil para el seguimiento del estado del código. Las pruebas son un artefacto, aunque no se especifican como salida ya que son un elemento de uso interno en el tipo de actividad descrito.

La guía anterior puede considerarse un método, en este caso para una actividad muy específica: la codificación y realización de pruebas unitarias. Pero es por supuesto una guía de pasos que indica cómo se debe hacer una actividad de Ingeniería del Software.

- 1.2 La métrica DIT (*Depth of Inheritance Tree*, profundidad del árbol de herencia) está asociada al diseño orientado a objetos, y se define de la siguiente forma:

«*El DIT de una clase A es su profundidad en el árbol de herencia. Si A se encuentra en situación de herencia múltiple, el DIT de A será la longitud máxima desde A hasta la raíz.*»

La siguiente descripción, adaptada de la que proporciona la herramienta *Project Analyzer* de Aivosto detalla cómo puede interpretarse la métrica:

«*Se sabe que un DIT elevado incrementa los fallos. De todos modos, no son necesariamente las clases que están más abajo en la jerarquía las que son origen de la mayor parte de los fallos. Algunos autores han demostrado que las clases más propensas a fallos son las que están en la zona intermedia del árbol de jerarquía. De acuerdo a sus hallazgos, las clases que están en la raíz o en la parte más baja del árbol se consultan con frecuencia, y debido a que son más familiares [a los desarrolladores], tienen menor tendencia a fallos si se las compara con las clases que están en la zona intermedia de la jerarquía.*»

De acuerdo a la descripción ¿a qué actividades de la Ingeniería del Software proporciona información la métrica?

Solución propuesta: la métrica puede utilizarse para el diseño del software, como indicador para tratar de reorganizar el diseño. También puede utilizarse en el mantenimiento que se lleva a cabo después de la entrega, para detectar las clases con más posibilidades de tener defectos, o incluso en las actividades de prueba para decidir qué clases deben recibir más atención durante el proceso de prueba.

- 1.3 Lea el siguiente texto sobre los *procesos ágiles* para la Ingeniería del Software y haga una valoración del mismo de acuerdo con la definición de la Ingeniería del Software.

«*Dé más valor a los individuos y a sus interacciones que a los procesos y las herramientas*»: Éste es posiblemente el principio más importante [...]. Por supuesto que los procesos ayudan al trabajo. Son una guía de operación. Las herramientas mejoran la eficiencia, pero sin personas con conocimiento técnico y actitud adecuada, no producen resultados. [...] Los procesos deben ser una ayuda y un soporte para guiar el trabajo. Deben adaptarse a la organización, a los equipos y a las personas; y no al revés. La defensa a ultranza de los procesos lleva a postular que con ellos se pueden conseguir resultados extraordinarios con personas mediocres, y lo cierto es que este principio es peligroso cuando los trabajos necesitan creatividad e innovación (adaptado de Wikipedia).

Solución propuesta: en ocasiones se considera equivocadamente que los postulados de los *procesos ágiles* como el anterior son contrarios a la consideración de la Ingeniería del Software como tal ingeniería. No obstante, afirmaciones como las de este fragmento de texto no cuestionan el carácter de la Ingeniería del Software, sino únicamente la efectividad de ciertos procesos, o el énfasis en ciertos aspectos de los mismos. Es importante resaltar que la efectividad de un proceso determinado es difícil de justificar de manera científica, por la dificultad de acumular evidencia fiable sobre la misma. Por ello, la discusión sobre qué proceso es mejor para qué situaciones es un tema abierto, y obliga a considerar las condiciones de cada proyecto a la hora de seleccionar el proceso y los métodos para su desarrollo.

- 1.4 ¿Cómo pueden definirse las relaciones entre los conceptos de actividad, proceso y método en la Ingeniería del Software?

Solución propuesta: las actividades son los sucesos que realmente ocurren en el trabajo de los ingenieros del software. Los métodos son especificaciones sobre *cómo* realizar esas actividades. Por tanto, los métodos añaden la dimensión normativa a las actividades. Por otro lado, los procesos son conjuntos de métodos para diferentes actividades, organizados según principios coherentes en un sistema que abarca varias fases de la ingeniería. En consecuencia, los procesos *contienen* métodos para diferentes aspectos de la Ingeniería.

Actividades propuestas

- 1.1 Buscar en la guía SWEBOK la clasificación de métodos de Ingeniería del Software.
- 1.2 Analizar los diferentes tipos de actividades de Ingeniería del Software que un estudiante ha de llevar a cabo cuando desarrolla un programa para una práctica de laboratorio. Clasificarlos en función del artefacto que crea o modifica cada uno de los tipos de actividades identificados.
- 1.3 Leer el artículo seminal de Brooks sobre la complejidad del software (Brooks, 1987) es una actividad muy recomendable. Después de esa lectura, se proponen las siguientes actividades:
 - Hacer un esquema de los elementos de la complejidad mencionados por Brooks.
 - Analizar esos elementos de complejidad mediante ejemplos cercanos.
- 1.4 Supongamos que estamos interesados en analizar para nuestra organización la eficacia y rapidez en la resolución de defectos encontrados después de la entrega de aplicaciones software. Proponga posibles medidas cuantitativas para esos dos aspectos del mantenimiento.
- 1.5 El siguiente ejemplo está adaptado de una experiencia real reseñada en Navegapolis.net, un interesante blog sobre Ingeniería del Software. Coméntelo en grupo:

«Miguel forma parte del equipo de programación de un sistema en entorno Visual Basic .NET. Ha trabajado durante dos días en una función –de ciento treinta líneas de código– capaz de comprobar la validez de una cadena como fecha, pero su función contiene dos errores: Considera como fechas válidas las cadenas vacías, y su cálculo de los años bisiestos no es correcto. [...] Miguel trabaja para una empresa que desarrolla software desde la perspectiva de los procesos, donde la premisa de producción industrial es que el valor del producto es resultado, en su mayor parte, de los procesos que siguen las personas que lo desarrollan. Así pues, asumiendo

que su empresa sigue PSP (*Personal Software Process*), modelo que controla el tiempo, el tamaño del trabajo y el número de errores para medir la eficiencia y calidad de las personas y de los equipos, veámos cómo se determinaría la eficiencia del trabajo realizado por este trabajador. La información objetiva es clara: 130 líneas de código en 16 horas (2 días completos). Lo mismo ocurre con la calidad de su trabajo: dos errores en 130 líneas de código. Con estas informaciones los gestores de la empresa determinarán si Miguel es más o menos eficiente que la media, si la calidad de su trabajo está dentro del rango admisible, etc. Lo que no pueden descubrir es que Visual Basic incorpora de forma nativa la función `IsDate()`, y que si Miguel lo hubiera sabido, en lugar de dos días hubiera tardado un minuto en realizar el mismo trabajo, que además no contendría ningún error. Este hecho, basado en un caso real, pone de manifiesto cómo para algunas empresas es beneficioso este método de trabajo. La empresa que desarrolló este proyecto, concretamente, facturaba 250 dólares por hora de trabajo, por lo que programar esta función mediocre le aportó una facturación de 4,000 dólares».

- 1.6 Encuentre en internet las actas –editadas por P. Naur y B. Randell– de la conferencia promovida por la división de asuntos científicos de la OTAN en 1968. Acceda a las mismas, seleccione un artículo al azar y analice el estado de los problemas planteados en el mismo a día de hoy.
- 1.7 Estudie las recomendaciones curriculares del SE2004 y compárelas alguna titulación superior en computación de universidades en su país. ¿Qué grado de cobertura de las mismas existe?
- 1.8 Repita el ejercicio anterior en grupo. Cada persona estudiará la cobertura de una universidad diferente, tras lo cual se hará una puesta en común con las distintas conclusiones individuales.
- 1.9 Lea el artículo clásico de Dijkstra «*El humilde programador*» (Dijkstra, 1972), con especial atención a las conclusiones que sobre la crisis del software proporciona al final del mismo.
- 1.10 En «*Ingeniería del Software: ¿una idea cuyo momento llegó y ya se fue?*» (deMarco, 2009), Tom deMarco, uno de los autores que han hecho contribuciones fundamentales a la disciplina, revisa la historia de la misma, mostrando cómo su percepción de las cosas ha cambiado a lo largo de los años. Su lectura puede servir de base pura una interesante discusión.

Modelos y procesos

Tal y como Brooks sugiere, «la complejidad del software es una propiedad esencial, no accidental». Observamos que esta complejidad inherente tiene que ver con cuatro factores: la complejidad del dominio del problema, la dificultad para gestionar el proceso de desarrollo, la flexibilidad que el software ofrece y los problemas que caracterizan el comportamiento de los sistemas discretos.

— Grady Booch

2.1 El proceso del proceso

Javier Sánchez «el soriano» asistía a sus clases de Ingeniería del Software durante el último año de carrera y disfrutaba de la sensación de que cada vez le costaba menos aprender cosas nuevas. Había sudado tratando de dominar primero los fundamentos de la programación y de las estructuras de datos, y después las diferentes complicaciones de ambas disciplinas, como la programación distribuida o los programas concurrentes. Pero ahora, era el año 1996, parecía que por fin marchaba con viento a favor. Ya había interiorizado la necesidad de documentar bien los programas, así como la importancia de acompañarlos de algún tipo de explicación del diseño. Lo que no se esperaba es lo que, perplejo, estaba escuchando en los últimos días: toda una catarata de fases, productos, requisitos y criterios de entrada y salida de actividades. Todo ello enmarcado en lo que sus profesores denominaban «metodologías», todas ellas parecidas, si bien finalmente diferentes: SSADM, Merise, Métrica... La imagen mental de un desarrollo siguiendo al pie de la letra todo aquello se le antojaba lo más parecido al trabajo burocrático que hubiera visto hasta la fecha. Muy lejos en todo caso del creativo mundo del programador que él tanto apreciaba. En este nuevo mundo de la Ingeniería del Software el mensaje era claro: el proceso es la clave. El orden y el estricto seguimiento de las actividades conducen a la calidad.

Años después, «el soriano» trabaja en una empresa dedicada a desarrollar aplicaciones de comercio electrónico. Todo avanza muy rápido. Su trabajo consiste esencialmente en construir nuevo software a partir de componentes para la Web previamente desarrollados. De los componentes, se sabe que funcionan, pues están siendo ejecutados millones de veces diariamente como parte de otras aplicaciones, aunque se echa en falta algo de documentación adicional. A veces tiene que imaginarse para qué sirve un parámetro de un componente o preguntárselo a un compañero de una oficina en otra provincia, que fue miembro del equipo que lo desarrolló. Si bien es cierto que existen algunos documentos de diseño (diagramas UML en su mayoría), no hay mucha preocupación por mantenerlos actualizados. Sentado cerca de las máquinas de café instantáneo se pregunta: ¿dónde está el proceso que me enseñaron? Parecen quedar pocos vestigios. Quizá fuese algo pasajero, o necesario para otro contexto. Quién sabe si el terremoto de internet ha hecho que se lo trague la tierra.

Cuando días después llega un nuevo jefe de proyecto a la empresa, «el soriano» asiste a una sesión informal donde se habla de metodologías «ágiles». Procesos que abogan por construir las pruebas unitarias prácticamente antes que el código, o a la vez, y ejecutarlas de manera incluso obsesiva. La documentación queda apenas relegada al código y los roles se difuminan. El nuevo jefe de proyecto habla de negociar pequeños incrementos de la aplicación con los clientes, para producir el software en pequeñas entregas, pero todo ello rápidamente. Si las metodologías que estudió en la carrera le parecieron semejantes a la burocracia de la Administración Pública, éstas nuevas le parecen las normas de una tribu. Rígidas pero escuetas. Esto es claramente un nuevo proceso, aunque de un tipo muy distinto. Ensimismado en estas reflexiones, piensa cuál de los dos enfoques es el correcto, el antiguo o el nuevo. Nunca encontrará respuesta.

2.2 Objetivos

El objetivo general de este capítulo es introducir el concepto de *proceso de software* en el sentido de su definición, uso, evaluación, medida, gestión, cambio y mejora. Los contenidos concretos de los procesos, es decir, las actividades y técnicas a utilizar en cada momento del ciclo de vida del software, son objeto de los capítulos que tratan cada uno de dichos momentos por lo que no serán analizados con detalle aquí. Los objetivos específicos son:

- Conocer el concepto de proceso de software y distinguirlo de otros relacionados, como el de actividad, metodología o ciclo de vida.
- Conocer los modelos de ciclo de vida del software, entendiéndolos como procesos genéricos.
- Saber diferenciar diferentes metodologías o métodos de Ingeniería del Software.

2.3 Introducción

El software se ha convertido en las últimas décadas en un elemento fundamental en las sociedades modernas. Junto a esa importancia creciente, se ha generalizado una preocupación por la calidad del software que ha fomentado el desarrollo de diferentes áreas de estudio en la Ingeniería del Software. Una de esas áreas es la de los *procesos software*, que aborda el análisis de las mejores formas de organizar y llevar a cabo el desarrollo de software. Puesto que el desarrollo de software es una actividad muy compleja, se asume que el orden y el método en las fases y actividades del desarrollo redundarán en un mayor y mejor control del mismo, permitiendo obtener mejores productos software. No obstante, a día de hoy esa hipótesis no puede contrastarse de una forma sencilla, por lo que los procesos software siguen siendo un área de intenso estudio.

Un proceso, entendido de manera general, es una serie de pasos que incluyen actividades, restricciones y recursos que resultan en un producto determinado con ciertas características

Otra definición general de proceso es la que proporciona el Glosario IEEE de Términos de Ingeniería del Software (IEEE, 1990):

Un proceso es una secuencia de pasos que se lleva a cabo para un propósito determinado

Los procesos pueden describirse a diferentes niveles, por lo que muchos a su vez se descomponen en subprocessos, organizándose en una secuencia jerárquica de subprocessos intermedios. Hay procesos simples y otros muy detallados y complejos. Antes de continuar, merece la pena reflexionar un instante sobre la importancia de seguir un proceso, pues resulta evidente que ello supone un cierto esfuerzo.

En los comienzos de la informática, el proceso era simplemente la forma personal y particular de actuar de cada desarrollador, es decir, los procesos eran todos *ad hoc*. No había normas escritas, lo cual acarrea al menos tres problemas:

- El software se hace difícil de mantener. Si un desarrollador abandona el proyecto, por ejemplo, otro desarrollador tendrá que completar su trabajo. En el peor de los casos, no habrá ningún tipo de plan ni documentación, ya que todo estaba en la cabeza del autor original. Aun no suponiendo el peor de los escenarios, el impacto de la eventualidad implica siempre un riesgo. Pensando en casos menos extremos, la mera ausencia de un proceso definido dificulta la comprensión del software desarrollado.
- Si no hay proceso es difícil establecer medidas de calidad. Si cada desarrollador actúa de una forma diferente, no tendremos forma de comparar sus resultados.

- La ausencia de proceso no permite reutilizar experiencias pasadas. La ausencia de notaciones y técnicas comunes dificulta la comunicación de los desarrolladores, lo cual podría conducir a documentaciones o incluso diseños incompatibles entre sí.

Como acabamos de ver, un proceso proporciona, además de otros beneficios, *fácilidad de mantenimiento, capacidad de seguimiento y consistencia*. De los beneficios de la facilidad de mantenimiento se habla con mayor detalle en el Capítulo 8. La capacidad de seguimiento, por su parte, hace que se puedan medir los productos de las actividades, y la consistencia facilita la homogeneidad en la documentación y el diseño. Puesto que generalmente se considera que dichos atributos contribuyen a la calidad de un software, podríamos resumir lo dicho en que la *calidad en el proceso resulta en calidad en el producto* para ciertos aspectos concretos como los que acabamos de describir.

2.3.1 Una definición de proceso

Un proceso de software es una estructura impuesta al desarrollo del software. Una definición más detallada sería la siguiente:

Un proceso de software es un conjunto coherente de políticas, estructuras organizativas, tecnologías, procedimientos y artefactos que se necesitan para concebir, desarrollar, implantar y mantener un producto software

Esta definición habla de la *especificación o definición* de un proceso de software («lo que debe hacerse»), pero no del proceso finalmente llevado a cabo en un proyecto concreto («lo que realmente se hace»). Éste es el sentido en que emplearemos en adelante el término, teniendo en cuenta que esta definición abarca todo el ciclo de vida del software, desde la concepción hasta el mantenimiento.

Es una confusión muy común el no diferenciar entre proceso de software y proyecto de software, e incluso ciclo de vida del software. Lo cierto es que, dependiendo del contexto, todos ellos pueden ser aplicados a diferentes conceptos, lo cual resulta ciertamente confuso. En consecuencia, y para delimitar mejor estos términos, comenzaremos por establecer qué se entiende por proyecto:

Un proyecto es un esfuerzo que se lleva a cabo una sola vez, que tiene objetivos bien definidos y que se produce dentro de un plazo determinado

Como veremos, todo proyecto viene definido por un plan de proyecto que se prepara a tal efecto, por las personas que están asignadas al mismo, por los recursos asignados y por la especificación de criterios de éxito. Sin embargo, nuestro interés en el presente libro no es el estudio de los proyectos en abstracto, sino el análisis de los proyectos de desarrollo

de software, y por tanto nos interesa especialmente conocer cómo se define un proyecto de Ingeniería del Software:

Un proyecto de Ingeniería del Software es un proyecto cuyo objetivo es obtener un producto de software que satisfaga ciertos requisitos, en el plazo previsto y dentro del presupuesto

La inclusión de las definiciones anteriores (ambas tomadas de la Enciclopedia de Ingeniería de Software de Marciniak) pretende ayudar a relacionar los conceptos proceso y proyecto de desarrollo de software (o proyecto de Ingeniería del Software), y a diferenciarlos desde el principio. Para ello, resulta útil recordar las diferencias entre el ciclo de vida de un proyecto de desarrollo y el ciclo de vida de un proceso. Según el Glosario IEEE de Términos de Ingeniería del Software:

El ciclo de vida de un desarrollo de software es el periodo de tiempo que comienza cuando se toma la decisión de desarrollar un producto de software y que concluye cuando se entrega el software

Dado lo habitual de considerar que proceso de software, proceso de desarrollo y proceso de ciclo de vida del software son términos sinónimos, en el presente libro los tomaremos como tales. En cualquier caso, un proceso de software incluye, entre otros, los siguientes elementos:

- Métodos y técnicas empleados como guías para llevar a cabo las actividades que componen un proyecto software. En el resto de los capítulos de este libro se describen diferentes métodos y técnicas específicos de ciertos tipos de actividades de Ingeniería del Software. Pues bien, los procesos ordenan y agrupan dichas técnicas en fases coherentes entre sí.
- Aspectos de gestión de equipos y personas, dado que el desarrollo habitualmente se hace en equipos y se gestiona dentro de una estructura organizativa.

Es importante, por tanto, entender que cuando hablamos del *proceso de software* no nos referiremos a ningún método en concreto (por ejemplo un método de análisis orientado a objetos), sino a aspectos de gestión de las actividades y su secuenciación.

2.3.2 Modelos del ciclo de vida, marcos de procesos y procesos

La especificación de los procesos se puede hacer a diferentes niveles de detalle. Mientras algunas especificaciones simplemente nos hablan de las fases generales, otras llegan a detallar los roles, tareas y productos de cada actividad dentro de un proyecto.

Método vs. metodología

Aunque método y metodología son generalmente utilizados como sinónimos, el término *metodología* hace en realidad referencia al estudio de los métodos. El concepto de *método* en Ingeniería del Software es muy general, mucho más que el de proceso de ciclo de vida. Un método en Ingeniería del Software impone una estructura a ciertas actividades de ingeniería con el objetivo de hacer esas actividades sistemáticas y, en definitiva, más eficaces. Los métodos por lo general proporcionan ciertas notaciones, un vocabulario específico, procedimientos para llevar a cabo ciertas tareas, y directrices para evaluar tanto la ejecución de las actividades como el producto. Pero hay métodos que cubren todo el ciclo de vida, y otros que son específicos de una fase o de una actividad en concreto. Por ello, diferenciaremos el concepto de método del de proceso de ciclo de vida.

Los *modelos de proceso del ciclo de vida* son definiciones de alto nivel de las fases por las que transcurren los proyectos de desarrollo software. No son guías concretas y detalladas, sino definiciones generales que muestran las dependencias entre las fases. Son ejemplos de estos modelos el *ciclo de vida en cascada*, el *ciclo de vida en espiral* o el *desarrollo basado en prototipos*. A partir de uno de estos modelos, se pueden especificar procesos de ciclo de vida del software más concretos y detallados.

Además del concepto de modelo de proceso, un concepto muy utilizado actualmente es el de *marco de proceso*. La idea de un marco de proceso es dar una definición genérica de un proceso que puede aplicarse a muchas situaciones, de modo que para especificar nuestros procesos, lo podemos utilizar como una «biblioteca de procesos» reutilizable, adaptándolo. La diferencia con un modelo de proceso es que los marcos suelen proporcionar un nivel de detalle mayor que los modelos, y se expresan de manera más formal.

Possiblemente el marco de proceso más conocido es el *Proceso Unificado*, que define apenas unos cuantos principios generales y esboza las fases y actividades fundamentales. A partir del Proceso Unificado se puede adaptar el proceso de cualquier organización en particular siempre que éste cumpla con los principios generales del proceso unificado. Un aspecto importante del Proceso Unificado, y de otros marcos de proceso similares, es que es independiente de las técnicas de desarrollo e incluso de la metodología de análisis y diseño utilizada, ya que se centran especialmente en los aspectos de gestión de las actividades. Otro marco de proceso muy popular es OpenUP (*Open Unified Process*, proceso unificado abierto), marco de proceso de fuente abierta desarrollado por la fundación Eclipse.

La adaptación o instanciación de un *proceso* específico para cada organización a partir de un marco de proceso es especialmente importante. El contexto de la organización, e incluso el de cada proyecto concreto, aportan características específicas que muy difícilmente podría cubrir un proceso concreto con aspiraciones de erigirse en «el proceso universal». Como indica la guía SWEBOK, la naturaleza del trabajo a realizar, el dominio de aplicación, el modelo de ciclo de vida y la madurez de la organización influyen decisivamente en el tipo de definición de proceso que resulta más útil en cada caso.

Principios del Proceso Unificado

El Proceso Unificado es un marco general de procesos basado en un pequeño conjunto de principios que se consideran fundamentales en todo proceso:

- **Iterativo e incremental.** Las actividades se realizan en ciclos de desarrollo y el resultado de cada uno de esos microciclos es un incremento del sistema. Dicho de otro modo: en cada microciclo se añaden nuevas funcionalidades al sistema.
- **Dirigido por los casos de uso.** Los casos de uso, especificaciones de requisitos funcionales al fin y al cabo, se utilizan como guía de todas las actividades del proceso.
- **Centrado en la arquitectura.** La arquitectura software es el diseño de alto nivel que constituye el armazón del sistema. El proceso se centra en la construcción temprana de un prototipo arquitectónico que si bien implementa de forma completa sólo un pequeño subconjunto de la funcionalidad, permite descubrir los problemas técnicos fundamentales al principio del proyecto.
- **Orientado a los riesgos.** El proceso requiere identificar y priorizar los riesgos del proyecto. Una vez hecho esto, las actividades de desarrollo se orientarán a enfrentarse cuanto antes a los riesgos más importantes.

2.3.3 Características de las definiciones de procesos de software

Antes de estudiar con profundidad los procesos de software es necesario tener en cuenta algunos aspectos importantes sobre los mismos. Así por ejemplo:

- No existe un único proceso de software que determine la forma correcta de hacer las cosas independientemente de la organización, el proyecto y las circunstancias. De hecho, hay un considerable grado de diversidad en los procesos, y si hoy se utiliza mayoritariamente un determinado proceso, el futuro probablemente haya otras variantes, tal vez motivadas por cambios en la tecnología o en la forma de desarrollar. En conclusión podemos afirmar que hay muchos «tipos de proceso».
- Hay que diferenciar los procesos entendidos como «las actividades de desarrollo que realmente ocurren en una organización X» de la discusión general sobre cómo son y qué propiedades tienen los distintos tipos de procesos. Este segundo sentido es el que utilizamos en este capítulo.
- Los procesos no son sólo para las grandes organizaciones. Los procesos son útiles también para las pequeñas organizaciones y, debidamente adaptados, también lo son para los pequeños desarrollos. Incluso son útiles para equipos formados por una sola persona, como los procesos personales que se estudian en la Sección 9.5.5.

Al margen de las consideraciones anteriores, existe toda una serie de atributos de calidad de un proceso que pueden considerarse como requisitos deseables, o como criterios para comparar los distintos procesos. La Tabla 2.1 resume algunos de los más importantes.

Tabla 2.1: Atributos deseables para un proceso de software

Atributo	Descripción
Efectividad	Un proceso es efectivo si realmente conduce a la construcción de un producto correcto. Es de especial importancia que el proceso permita recolectar correctamente los requisitos de usuario, trasladarlos al software, y finalmente, verificar que realmente están en el producto final.
Predictibilidad	Un aspecto fundamental de los procesos es que permitan predecir el esfuerzo y tiempo necesarios para realizar los proyectos, así como la calidad del producto. Además, la consistencia del proceso permite reutilizar la experiencia de otros proyectos para predecir qué sucederá en los que ahora estamos comenzando.
Repetibilidad	Si un proceso funciona bien, se repetirá en futuros proyectos. Los procesos <i>ad hoc</i> no se pueden replicar porque sólo pueden volverlos a seguir las mismas personas; por lo que un proceso exitoso se ha de documentar y hacerse sistemático.
Adaptabilidad	La implantación de un proceso en una organización no da resultados de forma inmediata. Al contrario, se necesita tiempo y experiencia para que el proceso dé los frutos deseados. Puesto que la tecnología y las herramientas cambian, los procesos deben evolucionar y adaptarse. Es recomendable por tanto, que el proceso no sea excesivamente rígido ni en su aplicación ni en su forma, para facilitar la adaptación y mejora continua.
Seguimiento	El proceso debe facilitar la gestión. Esto es, debe permitir el seguimiento del estado de los proyectos, su medición y su comparación. Sin los elementos suficientes para un buen seguimiento, los proyectos nunca serán predecibles.
Facilidad de mantenimiento (del producto)	Prácticamente todo software evolucionará después de su entrega. Un buen proceso hace que el diseño del software se exponga de forma que sea fácil de comprender por personas diferentes a las que lo desarrollaron. Esto reduce significativamente el esfuerzo necesario para hacer correcciones o cambios.
Calidad (del producto)	Un buen proceso se dirige a que el producto final se ajuste a los requisitos para los que fue concebido, y debe poder evolucionar si esas necesidades cambian. La calidad es un aspecto transversal al proceso, y de hecho, la facilidad de mantenimiento puede considerarse uno de los aspectos de la calidad.

2.3.4 Lenguajes para la especificación de procesos

Por todo lo anteriormente expuesto sabemos que existen y existirán múltiples definiciones de proceso. Y lógicamente, hay también notaciones que pueden utilizarse para plasmar esas definiciones. La especificación «*Software & Systems Process Engineering Meta-Model Specification (SPEM)*» publicada por el consorcio OMG contiene una notación de procesos específica del software que puede expresarse en UML.

SPEM 2.0 define fase como «*un periodo significativo en un proyecto que termina en un hito de gestión, un conjunto de productos o un punto de comprobación significativo*». Así, se pueden especificar modelos de proceso mediante diagramas como el de la Figura 2.2. En él se especifican las fases principales de un proceso ficticio, incluyendo una de ellas (la fase «Prueba») en la forma de *iteración*. Eso indica que esa fase será repetitiva.

Se especifica además un hito entre la fase de «Requisitos de Usuario» y la fase de «Análisis», que es la producción de la especificación de requisitos. En este lenguaje, cualquier fase debe tener algún tipo de hito o producto final que permita saber cuándo se ha terminado.

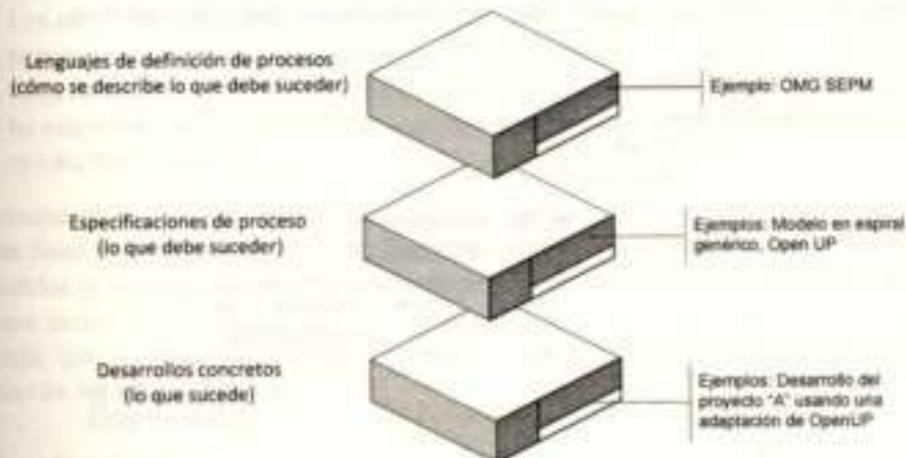


Figura 2.1: Lenguajes de definición de procesos

La especificación de la Figura 2.2 es de alto nivel, y corresponde por tanto a un modelo de proceso o a un proceso genérico. Las notaciones como SPEM pueden además especificar los roles profesionales, las actividades y su secuencia para actividades muy específicas, como por ejemplo, una prueba de integración o una revisión de código. Para ello, se podría descomponer alguna de las fases en subfases o actividades.

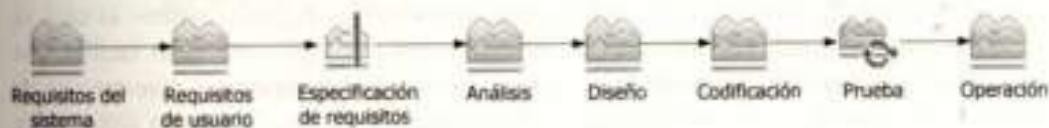


Figura 2.2: Ejemplo de definición de modelo de procesos en notación SPEM 2.0

2.4 Modelos de ciclo de vida del software

En esta sección se describen varios modelos de proceso o de ciclo de vida del software. Los modelos que describiremos a continuación no son los únicos que existen, sino aquellos que han tenido mayor relevancia histórica y aún son útiles como referencia o como contraste.

2.4.1 Modelo en cascada

El modelo del ciclo de vida en cascada (del inglés *waterfall*) es tal vez el más ampliamente difundido dentro de los métodos clásicos, y si bien no existe una única formulación del mismo, su característica distintiva es la de llevar a cabo distintas fases de desarrollo en secuencia, comenzando cada una de ellas en el punto en que terminó la anterior.

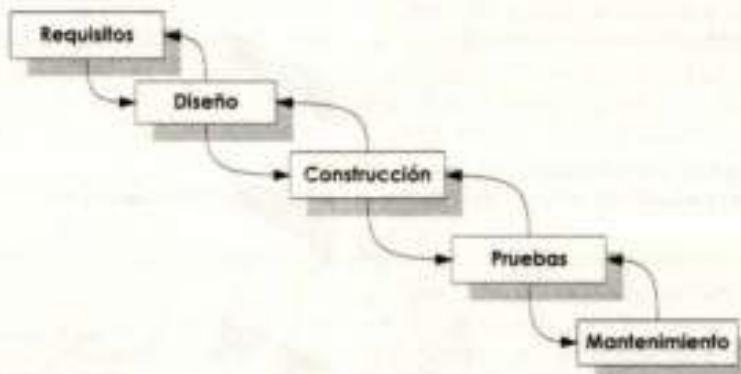


Figura 2.3: Una formulación del modelo de ciclo de vida en cascada

La formulación original de este modelo, tomada y adaptada de las ingenierías clásicas, consistía en una serie de fases que debían realizarse en una secuencia estricta. Con posterioridad se definió una versión más flexible, que permitía la iteración entre fases para la corrección de errores. Fuera cual fuese la formulación, uno de los aspectos fundamentales de este modelo es que el sistema se entrega completo y de una sola vez al final del proyecto, lo que –entre otras cosas– facilita la gestión de los contratos entre proveedores y clientes del software. Sin embargo, este modelo de ciclo de vida asume requisitos estables que se fijan al principio del proyecto –son difficilmente modificables posteriormente– lo cual es un grave inconveniente. La existencia de fronteras bien definidas entre fases facilita, por otra parte, la evaluación del grado de avance del proyecto. La Figura 2.3 muestra una de las formulaciones típicas de este modelo, aquella donde se permite la «vuelta atrás» entre fases.

Las ventajas del modelo en cascada incluyen las siguientes:

- El estado del proyecto se hace visible, dado que hay una progresión secuencial por fases perfectamente diferenciadas.
- Los entregables pueden asociarse a las fases, facilitando así la gestión.

Entre las desventajas de este modelo tenemos las siguientes:

- Asume la estabilidad de los requisitos durante el desarrollo, lo que en la mayoría de los casos resulta poco realista.

- Los límites entre las fases son demasiado rígidos, y esto hace que la revisión de los entregables e hitos entre fases sea difícil. A menudo estas decisiones son tan complejas que obligan a paralizar el desarrollo, por ejemplo cuando se ha de dar por válido lo obtenido y proseguir, o no.
- Los errores en el análisis –la primera de las fases– se propagan al resto del desarrollo, puesto que en las siguientes fases no hay más análisis. Lo mismo ocurre con otros tipos de errores, y como sólo se detectan errores en la fase de pruebas, cuando ya se ha empleado mucho esfuerzo en construir el software, la subsanación de los mismos resulta muy costosa.

Aunque el modelo en cascada ha sido muy criticado, hay que tener en cuenta que la idea de fases sigue presente de algún modo en modelos posteriores. La diferencia es que en modelos posteriores las actividades de desarrollo de cada tipo no se limitan a una fase, sino que tienen lugar en varias de ellas, avanzando mediante *iteraciones*. El concepto de iteración, que se describe más adelante, permite además adaptar mejor el proceso a los cambios en los requisitos.

2.4.2 Modelo en «V»

El modelo del ciclo de vida «con forma de V» es una evolución del modelo en cascada en el que se enfatizan las actividades de verificación y validación. La Figura 2.4 muestra una formulación típica de ese modelo.

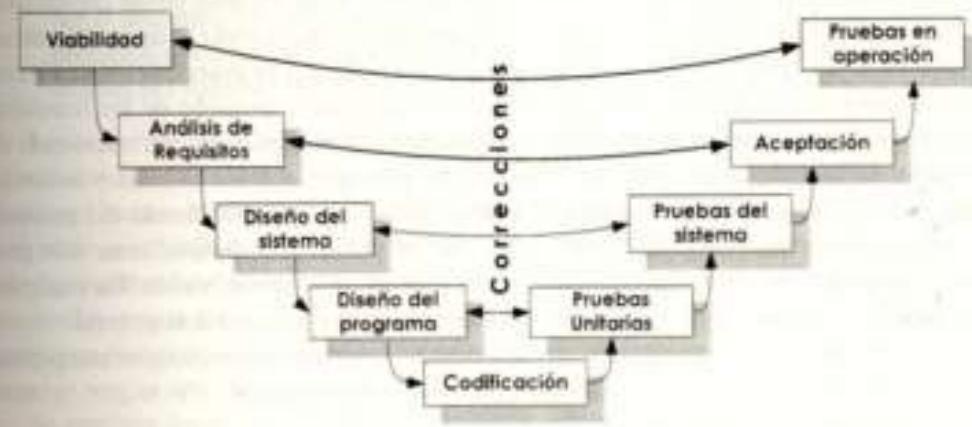


Figura 2.4: Modelo en «V»

La parte izquierda de la «V» representa las fases de desarrollo previas a las actividades de prueba. En este caso, se considera que la prueba unitaria corresponde al diseño detallado o de «bajo nivel», aunque en otros casos la prueba unitaria se considera como la prueba de la codificación. De hecho en las metodologías ágiles, se considera que prueba unitaria y

codificación deberían ser actividades quasi-simultáneas. Subiendo por la parte derecha, las pruebas del sistema serían las encargadas de evaluar el diseño de alto nivel, y las pruebas de aceptación serían la evaluación de que los requisitos que se tomaron reflejaban fielmente las necesidades de los usuarios. Por último, la puesta en marcha del sistema es el contraste de que el proyecto era viable.

Este modelo de ciclo de vida tiene como principal ventaja el énfasis en la prueba. Aún así, las fases siguen siendo de carácter secuencial hacia adelante. De hecho, si añadiésemos flechas de conexión entre las fases en el sentido contrario –por ejemplo desde el diseño de bajo nivel hacia el diseño de alto nivel– obtendríamos un modelo secuencial con «vuelta atrás», lo que nos lleva a afirmar que el modelo en «V» es en realidad una variante del modelo de ciclo de vida en cascada, más que un modelo completamente nuevo.

2.4.3 Modelos de proceso basados en prototipos

Un *prototipo software* es un modelo ejecutable de un sistema futuro que implementa sólo una pequeña parte de la funcionalidad, pero permite a los clientes, usuarios y desarrolladores adquirir experiencia con la arquitectura y la funcionalidad. Esto posibilita la evaluación temprana de riesgos evitando que los fallos o discordancias con las necesidades reales se manifiesten al final del proceso, cuando se ha gastado la mayor parte del esfuerzo del mismo.

Los prototipos permiten desarrollar en fases tempranas del proyecto una visión y un entendimiento común entre los clientes, los usuarios y los desarrolladores. Esto es especialmente útil en desarrollos de tamaño considerable, para alcanzar lo más pronto posible una «visión compartida» del proyecto. En realidad, este tipo de modelo puede combinarse con otros, incluso con un modelo en cascada, simplemente incluyendo el desarrollo de un prototipo en alguna fase temprana del proceso. En la Figura 2.5 se muestran los tres pasos básicos de creación de un prototipo: identificar un subconjunto pequeño de los requisitos que serán implementados en el prototipo, implementar el prototipo como un microciclo de actividades de Ingeniería del Software, y revisar el prototipo con los clientes y usuarios. Tipicamente, este último paso conlleva una toma de decisión. Dependiendo del proceso, esa decisión puede ser continuar con el desarrollo del proyecto completo, crear otro prototipo o eventualmente, cancelar el proyecto si se considera que no es viable. En cualquier caso, la creación del prototipo es un proceso empotrado en un proceso más general.

Diferentes modelos o especificaciones de procesos consideran los prototipos como cosas bien distintas. Las diferencias se pueden resumir en dos tipos básicos:

- **Prototipos desechables** o «de usar y tirar». También se conoce como «prototipado rápido». En este caso, se construye un prototipo muy básico que probablemente se descartaría. Por ello, estos prototipos no suelen hacerse con todos los criterios técnicos de Ingeniería del Software ya que lo importante aquí es construir el prototipo lo más rápido posible; de hecho, en ocasiones el software no es completamente funcional. La selección de los requisitos suele estar orientada a las interfaces que se enseñarán a los usuarios.

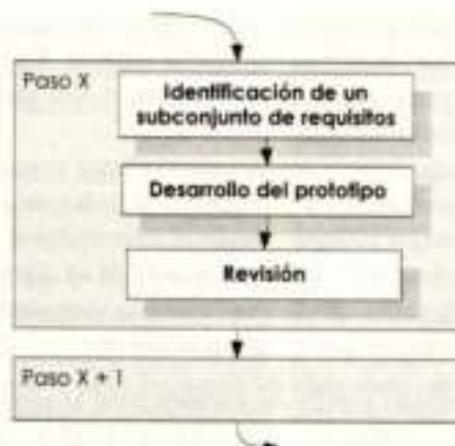


Figura 2.5: El modelo básico de un microciclo de prototipado

- **Prototipos evolutivos.** A diferencia de los anteriores, un prototipo evolutivo es un software en el que se han empleado todos los criterios de calidad en el desarrollo, y de hecho, no se diseña para «tirarlo después de usarlo». Estos prototipos pueden verse como esqueletos del sistema final, en los que se prueba la arquitectura final con sólo unos pocos requisitos funcionales, pero con un diseño definitivo. El concepto *evolución* recuerda que se desarrollará el sistema mediante incrementos sucesivos sobre el prototipo original, por lo que los requisitos que se suelen tomar para el primer prototipo son aquellos en los cuales entran en juego todos o la mayor parte de los elementos del sistema. En muchos casos, suelen ser las funcionalidades más complejas.

Los prototipos desecharables se construyen con diferentes niveles de calidad, desde software realmente funcional a software que «simula que funciona» en la interfaz gráfica. En ocasiones incluso el entorno de desarrollo para los prototipos no es el mismo que se utilizará para el desarrollo final. En el extremo de estas posibilidades podemos situar algunos procesos centrados en el usuario, en los que la usabilidad de las interfaces es la consideración primordial, y para los cuales se utilizan con mucha frecuencia los escenarios (dibujados en papel, por ejemplo). Los escenarios no son software en sí mismos, pero sirven para que los usuarios tengan una visión clara de cómo será la interfaz.

Se emplean los prototipos evolutivos cuando además de obtener realimentación de los usuarios y clientes, se desean minimizar los riesgos para la arquitectura del software. Dado que en estos prototipos se implementan funcionalidades significativas en cuanto a su complejidad, las dificultades en el desarrollo se harán patentes al principio del proyecto, cuando aún hay tiempo para reaccionar, probar con arquitecturas alternativas, o incluso cancelar el proyecto tempranamente si no es viable.

En algunos procesos no está claro si la aproximación del prototipado es incremental en el sentido que hemos descrito. Por ejemplo, hay procesos orientados al desarrollo de

aplicaciones web que propugnan construir el software en tres pasos. Primero, se construyen interfaces «huecas» en HTML, sólo para estudiar la interfaz. En un segundo paso, se programan todas las páginas, pero se usa un servidor que simula las respuestas. Sólo en una tercera fase se termina la implementación.

En general, el uso de prototipos es útil cuando el cliente conoce los objetivos generales para el software, pero no identifica los requisitos detallados de entrada, procesamiento o salida. También ofrece un mejor enfoque cuando el responsable del desarrollo del software no está seguro de la arquitectura del sistema, la eficacia de un algoritmo, la adaptabilidad de un sistema operativo o de la forma que debería tomar la interacción persona-computadora. En suma, los beneficios fundamentales de este modelo son:

- Mejora del entendimiento común de los requisitos al principio del desarrollo, evitando que fallos en su identificación se propaguen hasta el código y sólo se descubran al final del proyecto.
- Facilidades para que los desarrolladores estimen mejor los plazos y el esfuerzo necesario para realizar el proyecto.

Es importante reflexionar que el uso de prototipos conlleva una implicación temprana de clientes y usuarios, lo cual en todos los casos redundará en una mejor valoración del servicio prestado por la organización de desarrollo.

2.4.4 Modelo en espiral

A menudo se considera que el modelo del ciclo de vida en espiral reúne las mejores características del uso de prototipos y de los modelos de ciclo de vida en cascada. Definido por Boehm a finales de los años 80, la idea fundamental es que la gestión de riesgos debe guiar el proceso de desarrollo. De este modo, propugna evaluar los riesgos de forma regular en el proyecto y, si procede, tomar acciones para mitigar el impacto de esos riesgos a medida que avanza el desarrollo. La formulación original se representa en la Figura 2.6. La esencia del modelo es la división en cuadrantes del gráfico, dado que cada cuadrante representa un tipo de actividad diferente:

- *Identificar objetivos, alternativas y restricciones.* El objetivo es la especificación de la siguiente porción del sistema a desarrollar, incluyendo requisitos funcionales y no funcionales. Se identifican las posibles opciones de diseño para resolver esos objetivos y las restricciones de cada uno. Resultan especialmente reseñables entre estas últimas las restricciones de calendario y coste.
- *Evaluuar las alternativas mediante la identificación y resolución de riesgos.* Esta fase se centra en las áreas de incertidumbre del proyecto, potenciales fuentes de riesgo. A partir del análisis de los riesgos se derivará una estrategia que puede incluir el hacer prototipos, simulaciones, modelos o *benchmarks* (comparativas entre distintas posibles opciones).

- *Desarrollar y verificar el siguiente nivel del sistema.* Al depender de resultados de fases anteriores, esta fase puede implicar bien actividades completas o bien un ciclo de ingeniería completo (diseño, construcción, pruebas e implantación).
- *Planificar las siguientes fases.* Aquí se incluyen una evaluación y la preparación del plan para el siguiente ciclo de la espiral.

Es importante resaltar que el número de vueltas de la espiral no queda determinado por el modelo. El modelo solamente describe las fases en general, pero es en cada proyecto donde se determina el número de ciclos. Además, la dimensión radial del gráfico representa el coste acumulado al llevar a cabo los pasos en cada punto de la espiral. La Figura 2.6 muestra un ejemplo concreto con un número de vueltas determinado que no tiene por qué ser el mismo para todo proyecto.

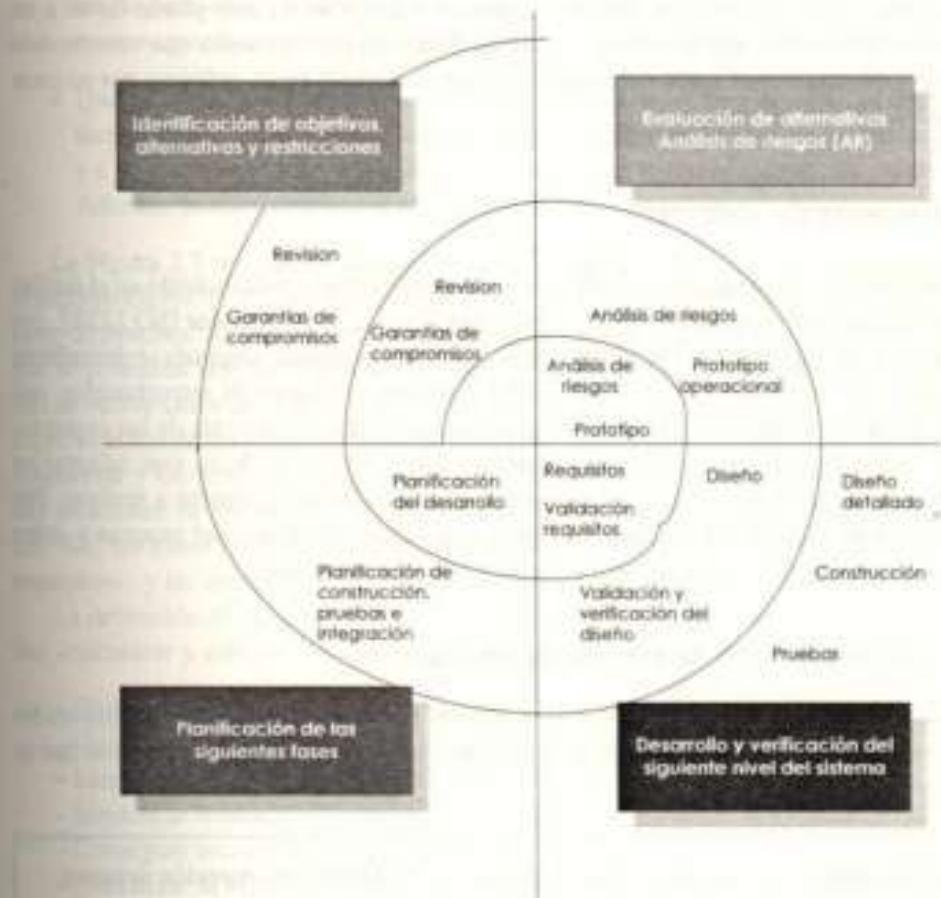


Figura 2.6: Modelo de proceso en espiral

La espiral tiene un punto crítico en el comienzo de cada nueva vuelta. En ese punto se lleva a cabo una comprobación de la viabilidad del proyecto, y se busca expresamente una ratificación del compromiso por todas las partes. Es por ello por lo que, teóricamente, el proyecto puede terminar en ese punto en cualquier vuelta.

Los puntos fuertes de la aproximación en espiral son los siguientes:

- La aplicación se desarrolla de forma progresiva, con la colaboración de los usuarios.
- Es más probable que se detecten los riesgos y defectos a medida que el desarrollo evoluciona, en lugar de al final del proyecto.
- La aceptación del software es progresiva.

Una crítica común a este modelo es que, dado que el número de vueltas queda indefinido, y que en cada vuelta se deciden y ajustan los objetivos, esto puede llevar a un alargamiento innecesario del proyecto. Además, debe tenerse en cuenta que este modelo requiere una elevada implicación del usuario, lo que se traduce en un esfuerzo por su parte para el que debe estar preparado.

2.5 Procesos de software

En esta sección se proporciona una visión general de qué es lo que está incluido en el ámbito de los procesos de software. Nos apoyaremos en primer lugar en el estándar ISO 12207, que proporciona un abanico de tipos de proceso de una cobertura amplia. Después se describirán algunos aspectos importantes en la definición de procesos y la forma de estructurarlo, que van más allá de los modelos de proceso generales. Hoy en día, la mayoría de las organizaciones no se plantea *adoptar* un proceso «prefabricado» podríamos decir, sino *adaptar* un marco de proceso general a sus propias características o a las del proyecto a realizar. Por eso es importante saber hasta dónde podemos cubrir con la definición del proceso y cómo abordar la especificación de un proceso para necesidades particulares.

2.5.1 ¿Qué se define en un proceso de software?

Las especificaciones de procesos de software definen cómo se deben hacer las actividades de Ingeniería del Software. Pero antes de proseguir vamos a detenernos a examinar qué se entiende por *actividad*:

Una **actividad** es la unidad mínima de trabajo, la cual tiene una duración definida, está relacionada lógicamente con otras actividades del proyecto, consume recursos y tiene generalmente un coste asociado

Lo cierto es que las especificaciones de procesos de software, si bien no detallan técnicas concretas para las diferentes actividades, sí indican en qué secuencia se deben hacer las mismas. Por ejemplo, algunos procesos establecen que los casos de prueba han de hacerse en la primera fase, al principio del proyecto, aunque no dicen con qué técnicas o herramientas debe hacerse dicha especificación. Las definiciones de proceso tampoco suelen prescribir ni los lenguajes de programación ni las técnicas de modelado de datos.

Los procesos se pueden describir en torno a tres elementos fundamentales:

- *Actividades*. Las actividades, de diversos tipos, pueden definirse a diferentes niveles de detalle. Una fase, o incluso un proceso entero, es una actividad de larga duración. Estas actividades se descompondrán generalmente en subactividades. Las actividades puntuales dentro del proyecto se denominan *hitos*.
- *Quién hace las actividades*, es decir, los roles de los implicados (analistas, arquitectos de software, clientes, usuarios, etc.) Dependiendo del tamaño del proyecto, una persona puede tener más de un rol en diferentes momentos o actividades.
- *Qué se utiliza para hacer esas actividades y qué se obtiene de ellas*, es decir los productos de las actividades. Estos productos pueden ser salidas de ciertas actividades, y a su vez entradas en otras, de modo que actúan como «parámetros» de las mismas. Además, pueden definir cuándo un producto se considera concluido.

La Figura 2.7 muestra la definición de la actividad «identificar y refinar requisitos» en el proceso OpenUP. Allí pueden verse los roles principales implicados: el analista y el ingeniero de pruebas. Si bien hay otros implicados definidos, incluyendo a los clientes, estos dos que se muestran son los responsables. El ingeniero de pruebas es responsable de la actividad de «crear casos de prueba», que toma como entrada los casos de uso y da como salida la especificación de los casos de prueba. El rol de analista es responsable de dos actividades, «encontrar y esbozar los requisitos» y «detallar los requisitos». Aunque aparentemente no hay relaciones de secuencia entre las actividades, éstas están determinadas por los productos. Así, los casos de uso se crean como resultado de la actividad «encontrar y esbozar los requisitos», y las otras dos tareas requieren los casos de uso como entrada.

La definición de las actividades en ocasiones se detalla aún más. Por ejemplo, la actividad «encontrar y esbozar los requisitos» en OpenUP tiene una serie de pasos:

- Obtener información.
- Identificar términos del dominio.
- Identificar los tipos de requisitos relevantes del dominio.
- Identificar requisitos de soporte.
- Conseguir una visión compartida.
- Actualizar la «lista de tareas», esencialmente un documento de planificación.

Para algunos de los pasos el proceso proporcionará directrices o listas de comprobación, que permiten verificar si la actividad se ha llevado a cabo eficazmente.

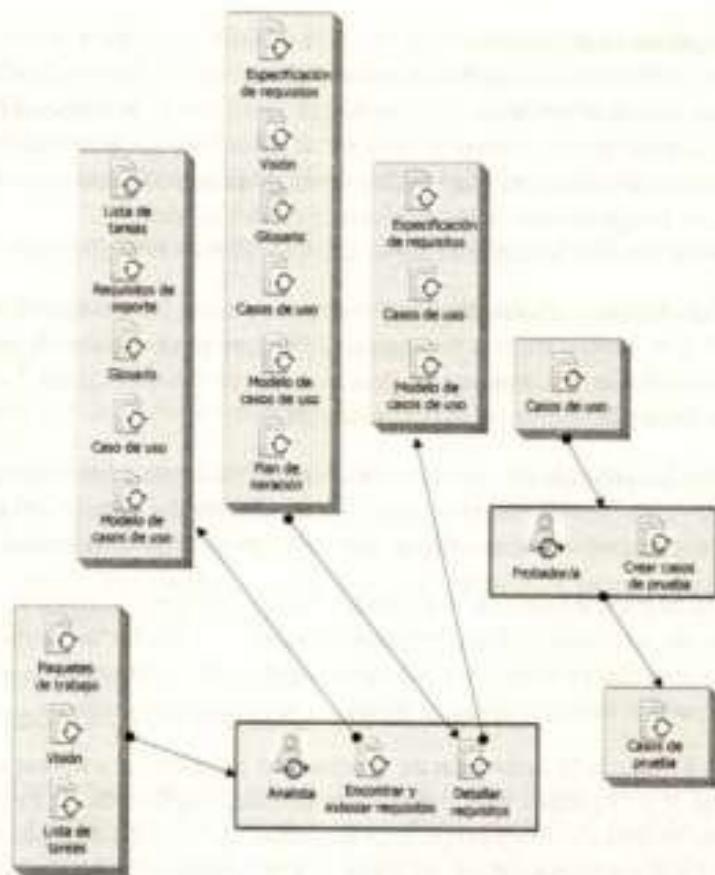


Figura 2.7: Vista de una actividad de «identificar y refinar requisitos» en OpenUP

2.5.2 El modelo de referencia ISO 12207

El estándar ISO/IEC 12207 es el modelo de referencia de procesos del ciclo de vida del software. En la edición de 2008, los procesos descritos se dividen en dos categorías:

- Procesos de **contexto del sistema**: *procesos de acuerdo, procesos organizativos habilitadores del proyecto, procesos de proyecto y procesos técnicos* (ver Figura 2.8). Se trata de actividades que no son específicas de la Ingeniería del Software, pues o bien se encargan de la sistematización de las actividades previas al desarrollo en sí mismo, o bien se trata de actividades que están en el contexto organizativo de ese desarrollo. Las Tablas 2.2 y 2.3 describen los procesos en los dos primeros grupos.
- Procesos **específicos del software**: *implementación del software, soporte y reutilización*. La Figura 2.9 muestra los diferentes tipos de procesos en cada categoría.



Figura 2.8: Procesos de contexto del sistema

Tabla 2.2: Procesos de acuerdo en el estándar ISO/IEC 12207

Proceso	Descripción
Adquisición	Es el proceso de adquisición de un bien o servicio para satisfacer unas necesidades. Este proceso implica una preparación, eventualmente analizando los requisitos y considerando diferentes opciones como la de comprar un software pre-empaquetado o hacer un desarrollo a medida. Puede implicar la selección de proveedores entre diferentes ofertas, y terminará en la negociación de un contrato. Finalmente, implica el seguimiento de lo acordado.
Provisión	Es el proceso desde la perspectiva opuesta, la del proveedor de un producto o servicio. Implica la identificación de una oportunidad, la negociación del contrato y la realización del plan y del proyecto. También implica la entrega y la provisión de soporte.

Tabla 2.3: Procesos organizativos habilitadores del proyecto (ISO/IEC 12207)

Atributo	Descripción
Gestión de modelos de ciclo de vida	Este proceso implica la definición y mantenimiento de modelos de ciclo de vida, modelos de proceso y políticas para los procesos. Las actividades fundamentales son el establecimiento del proceso, su evaluación y la mejora del mismo. Se trata de un «proceso de gestión del proceso».
Proceso de gestión de infraestructura	La infraestructura incluye las instalaciones, herramientas y recursos tecnológicos necesarios en la organización para soportar la ejecución de los proyectos. Este proceso se encarga de la implantación, establecimiento y mantenimiento de la infraestructura.
Proceso de gestión del portafolio de proyectos	Este proceso tiene como objetivo la selección de un número de proyectos suficiente y adecuado para la organización. Entre sus actividades está la identificación de oportunidades de negocio, la evaluación de la viabilidad de los proyectos y su terminación bien prematura o bien según el calendario.
Proceso de gestión de recursos humanos	Incluye las actividades necesarias para que la organización cuente con el personal suficiente y competente para la realización de los proyectos, identificando las competencias necesarias, desarrollándolas cuando sea necesario, o contratando nuevo personal. También, las actividades de gestión del conocimiento y fortalecimiento de los equipos de trabajo.
Proceso de gestión de la calidad	Incluye la gestión de la calidad del proceso y del producto y la implementación de acciones correctivas en los casos en que se necesite.

El estándar considera que el software es parte integral de un sistema, por lo cual los procesos del sistema son de un ámbito más general. Así, el estándar contiene procesos específicos de la realización de proyectos: planificación, evaluación y control, toma de decisiones, gestión de riesgos, gestión de la configuración, gestión de la información y medición. Estos procesos de gestión de proyecto aparecen de algún modo en los modelos y marcos de proceso. El Proceso Unificado, por ejemplo, incluye actividades específicas de análisis y gestión de riesgos. El estándar tiene la virtud de enumerarlos y categorizarlos, sirviendo así como modelo de referencia cuando se quiere implantar un proceso que considere todos los posibles aspectos.

En cuanto a los procesos técnicos, éstos incluyen la definición de requisitos, el análisis de requisitos, el diseño arquitectónico, la implementación, la integración del sistema, las pruebas del sistema, la instalación del software, su aceptación, operación, mantenimiento y cese de operación. Estos procesos son genéricos al sistema, no al software que los incluye, por lo que el estándar define además procesos específicos del software. Dentro de éstos, se incluyen algunos de implementación, tales como la implementación del software, el análisis de requisitos software, el diseño de la arquitectura, el diseño detallado o la construcción del software, su integración y prueba, y otros de soporte, como la gestión de la documentación, la gestión de la configuración, el aseguramiento de la calidad del software, su verificación y revisión, la auditoría del software y la resolución de problemas del mismo.

Tabla 2.4: Procesos de reutilización del software en el estándar ISO/IEC 12207

Atributo	Descripción
Ingeniería del dominio	Desarrollar y mantener modelos del dominio, arquitecturas de dominio y recursos de dominio. La ingeniería del dominio también se conoce como ingeniería de líneas de producto, y consiste en analizar y diseñar los aspectos reutilizables de una familia de productos software relacionados. Por ejemplo, la ingeniería del dominio para el dominio bancario identificaría componentes comunes a diferentes tipos de aplicaciones bancarias, que podrían reutilizarse y/o venderse por separado llamados recursos o assets.
Gestión de recursos reutilizables	Este proceso incluye la gestión de los recursos reutilizables, desde su concepción hasta su retirada. Esto incluye su catalogación, almacenamiento, gestión y control.
Gestión de programas de reutilización	Los programas de reutilización en las organizaciones son planes para maximizar las posibilidades de reutilización de recursos. Esto incluye la identificación de los dominios, la evaluación y planificación de la reutilización, y la revisión del programa en sí.

**Figura 2.9:** Procesos específicos del software en el estándar ISO/IEC 12207

Los procesos específicos del software del estándar incluyen un tercer grupo denominado procesos de reutilización del software, que se describen en la Tabla 2.4. Éstos son

especialmente importantes en aquellas especificaciones de proceso en las que se quiera dar una importancia especial a las economías relacionadas con la reutilización, como en el caso de los métodos de desarrollo basados en componentes.

2.5.3 Iteraciones e incrementos

La mayoría de los procesos de desarrollo actuales se autocalifican como *iterativos*, y muchos de ellos también como *incrementales*. La noción de iteración e incremento ya aparecía de algún modo en el modelo en espiral. Podríamos decir que cada vuelta de la espiral es una iteración, y que el software se construye por partes denominadas incrementos. No obstante, es importante comprender bien el concepto de iteración e incremento y cómo encajan en los procesos. Antes, hay que comprender que el incremento y la iteración son estrategias para la planificación y la ordenación del desarrollo, no un modelo de proceso en sí mismo.

- En el *desarrollo incremental* se construyen diferentes partes del sistema en distintos momentos, o a diferentes velocidades, y después se integran. Un incremento es, por tanto, un subconjunto de la funcionalidad del sistema. Si no se adopta una posición incremental en el desarrollo, toda la funcionalidad se desarrolla «a la vez», independientemente de que el sistema se descomponga en subsistemas que probablemente sean llevados a cabo por equipos separados, lo que hace que la integración sea problemática. Con el enfoque incremental, las integraciones implican sólo dos módulos, pues los incrementos se desarrollan casi siempre en secuencia.
- El *desarrollo iterativo*, por el contrario, divide el desarrollo en varios pasos o iteraciones, de tal modo que se pueda revisar el trabajo anterior de forma planificada. Aunque a menudo se asocia el término *iterativo* con el calificativo *incremental*, en principio, puede haber desarrollo iterativo sin tener desarrollo incremental. Puede que se planifique una primera iteración cubriendo todas las funcionalidades (quizá con algún recorte o restricción), para utilizar la segunda y tercera iteraciones, por ejemplo, para mejorar, ampliar o corregir la funcionalidad.

Desde el punto de vista del proceso, un incremento no debería ser revisado necesariamente, mientras que una iteración sí. En la práctica, el enfoque iterativo se combina con el incremental, y de hecho en muchas ocasiones se habla de iteraciones asumiendo que el software se construye por incrementos distribuidos entre las iteraciones. La Figura 2.10 muestra la estructura en fases iterativas del proceso unificado abierto (OpenUP), en notación SPEM 2.0. En dicha figura se muestran las fases así como los hitos intermedios entre ellas.

El que se hagan iteraciones no implica que deje de haber fases en el sentido tradicional, aunque pueda parecerlo a primera vista. Dicho de otro modo, en el enfoque iterativo e incremental no se trata solamente de hacer una secuencia de procesos de desarrollo completos, donde cada uno de los procesos de la secuencia se ocupa de un incremento. Lo que sí sucede es que las actividades de Ingeniería del Software no quedan ligadas a una fase

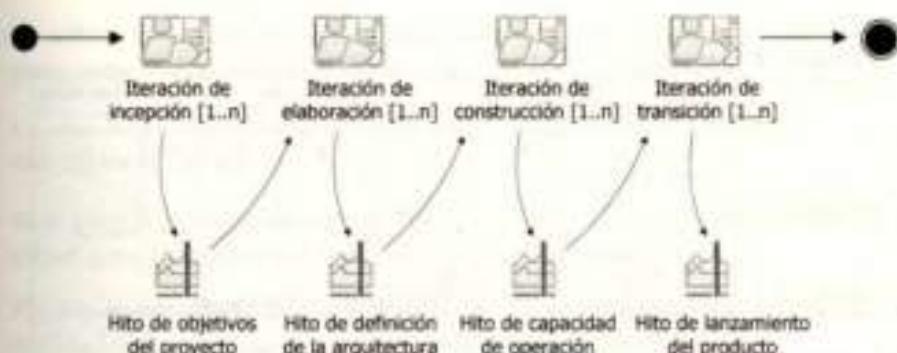


Figura 2.10: Estructura en iteraciones de OpenUP

particular del proceso, tal y como sucedía en el modelo en cascada. Por el contrario, las fases y sus iteraciones contienen diferente intensidad de los diferentes tipos de actividades.

La Figura 2.11 muestra la relación entre las diferentes actividades (requisitos, diseño, construcción, prueba e implantación), las cuales tienen lugar durante varias de las fases con diferente intensidad. Por tanto, el proceso sigue teniendo fases. Si nos fijamos en el eje horizontal, que representa el paso del tiempo, aparece algo que nos recuerda al modelo en cascada. No obstante, aquí las actividades no se confinan a una fase determinada, y hay una organización explícita en iteraciones. Puede haber fases, como la fase de inicio por ejemplo, que sólo tengan una iteración, mientras que en otros casos pueden existir varias iteraciones.

Las mismas actividades de ingeniería tienen lugar en diferentes fases cuando tenemos un proceso iterativo. No obstante, la forma de realizar esas actividades puede variar. La Figura 2.12 muestra diagramas de actividad que resumen las fases de inicio y construcción en OpenUP. En ambas fases hay una actividad denominada «identificación y refinamiento de requisitos». No obstante, en la fase de construcción, la subactividad de «encontrar y esbozar requisitos» tiene menos pasos que la misma subactividad en la fase de inicio. Las actividades son por tanto las mismas, pero tanto sus pasos como las entradas y salidas de éstas varían en cada fase, para ajustarse al estado de desarrollo general del proyecto.

El desarrollo iterativo e incremental se adapta bien a proyectos en los que los requisitos no están del todo claros al principio, pero si se amplían los requisitos excesivamente, puede que la estructura base proporcionada por el primer incremento no sea suficiente, en cuyo caso, habrá que rehacer el diseño. Por ello, en este tipo de desarrollo se hace un énfasis especial en el diseño arquitectónico como tarea esencial de las primeras iteraciones. Los primeros incrementos en ocasiones se denominan prototipos, si bien se trata de prototipos evolutivos. Los beneficios de los procesos iterativos e incrementales se pueden resumir en los siguientes puntos:

- El software se entrega en etapas, por lo que el cliente recibe al menos parte de lo encargado mucho antes que con un modelo en cascada.

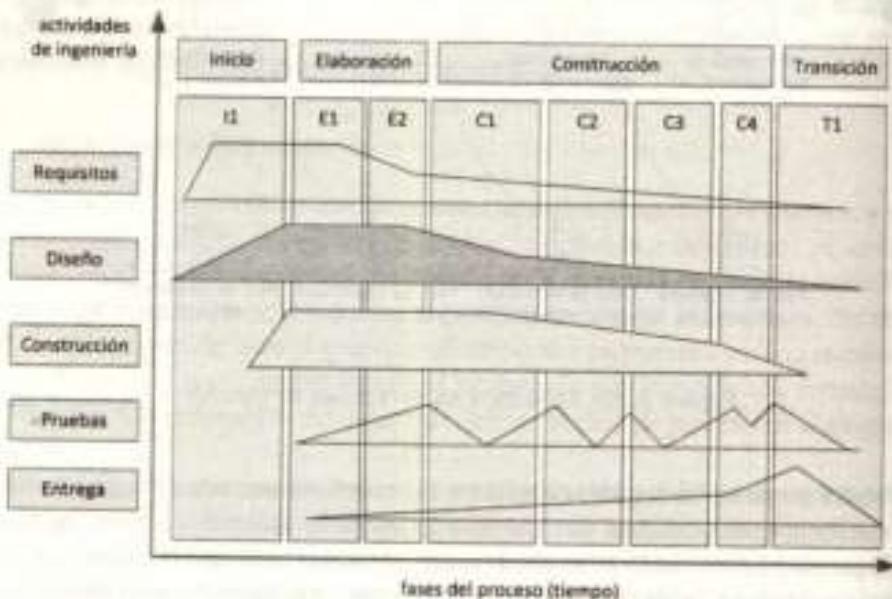


Figura 2.11: Modelo iterativo e incremental

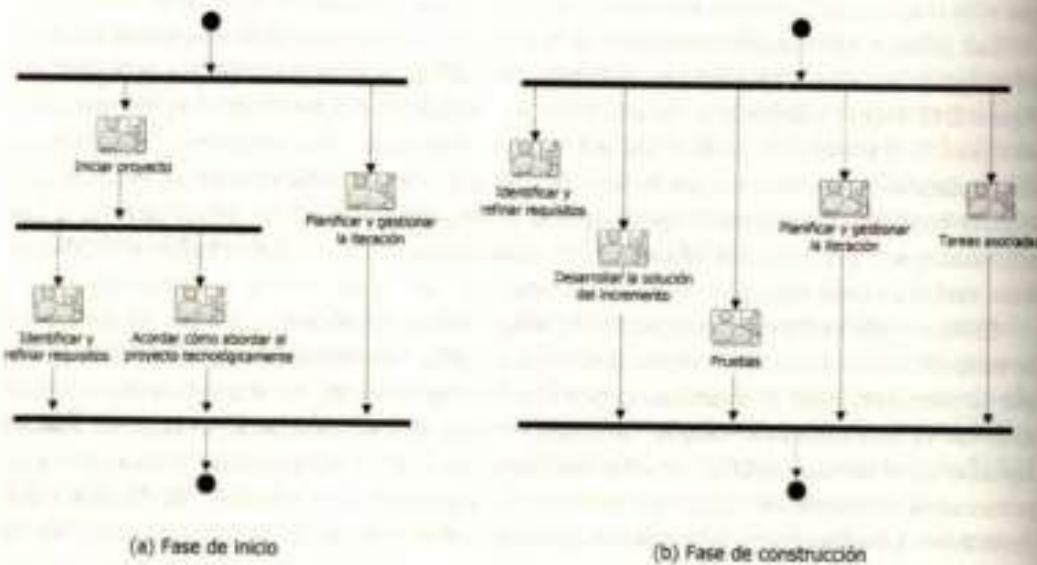


Figura 2.12: La actividad de identificación y refinamiento de requisitos en OpenUP

- La selección correcta de las funcionalidades a incluir en las primeras iteraciones funciona como mecanismo de control del riesgo en el diseño.
- La calidad y la estabilidad del software progresan con las iteraciones, y hay oportunidad para la mejora.
- Si se planifican las iteraciones de forma progresiva, es muy probable que las estimaciones para iteraciones subsiguientes sean más acertadas.
- Facilita la incorporación de personal durante el desarrollo de un proyecto, pues la realización de diferentes ciclos hace más sencillo el aprendizaje.
- Ayuda en proyectos que utilizan tecnologías nuevas o en fase de aprendizaje por parte de la organización.

Sin embargo, y a pesar de las ventajas enunciadas anteriormente, el desarrollo iterativo e incremental también tiene ciertas desventajas:

- No es útil en proyectos cortos en duración o ámbito, ni en los que la funcionalidad a implementar esté perfectamente definida, ya que el coste extra de planificación por incremento no merece la pena.
- Se corre el riesgo de que la funcionalidad de un proyecto crezca continuamente. El problema parte del hecho de que cuando no hay requisitos bien definidos para la terminación del número de iteraciones, el proyecto se puede alargar indefinidamente.
- Algo similar a lo indicado en el punto anterior ocurre en el caso de funcionalidades complejas, pues a menudo en un desarrollo se pospone constantemente una funcionalidad debido a su alto nivel de complejidad (procrastinación).
- Otro problema importante está en que el diseño del primer incremento se convierte en un elemento crítico. Si no es adecuado para los incrementos subsiguientes, se corre el riesgo de que éstos se desarrollen con un diseño inadecuado simplemente por tratar de ajustarlos al diseño base original.

Una pieza clave en el desarrollo iterativo e incremental es saber cómo seleccionar la parte del sistema que se desarrollará primero. También es fundamental saber seleccionar los incrementos de tal modo que aquellos que son seleccionados sean significativos para los usuarios. Estas preguntas, y otras como éstas, son las que los procesos de software deben ayudar a responder.

2.6 Algunos tipos de procesos importantes

En esta sección se introducen algunos conceptos adicionales relativos a los procesos de software, importantes para comprender el estado actual de la cuestión.

2.6.1 Procesos estructurados y procesos orientados a objetos

Antes de que se generalizase el uso de la orientación a objetos como conjunto de métodos de Ingeniería del Software, existía algo denominado *enfoque estructurado*. Las actividades de análisis, diseño y construcción en el enfoque estructurado se basan en el concepto de abstracción funcional, en contraposición a la abstracción de datos propia de la orientación a objetos.

A pesar de la gran importancia que tradicionalmente se ha concedido a ambos enfoques en los libros de Ingeniería del Software, esta diferencia afecta a los métodos utilizados para la Ingeniería del Software, pero no al proceso en sí mismo como ordenación y guía de esas actividades. Por ello no trataremos lo específico de ambas metodologías, ya que realmente son contenidos concretos de actividades vistas en otros capítulos de este libro.

2.6.2 Procesos ágiles

Se denominan *métodos de desarrollo ágiles* a un conjunto de métodos que enfatizan el enfoque iterativo, la adaptabilidad del proceso y la colaboración. Los métodos ágiles se caracterizan además por el hecho de que reducen la documentación y los procedimientos al mínimo. Por ello, se les suele contraponer con métodos anteriores en los que se produce una gran profusión de documentos y comprobaciones formales, a los que a veces se les aplica la metáfora de métodos «de gran ceremonia».

Aunque no todos los procesos ágiles son iguales, la siguiente lista de características generales puede aplicarse a la mayoría de ellos, si bien es posible que no todos los métodos ágiles las cumplan todas:

- La medida del progreso es el software desarrollado y funcional. La clave es que al final de cada iteración el cliente tenga una versión funcional del software. Así, los clientes pueden reevaluarlo y decidir qué les interesa a partir de ese punto.
- La documentación es la documentación del código junto con el diseño de dicho código. Es decir, no se producen documentos de análisis o de especificaciones.
- No hay una estructura organizativa rígida en el equipo de desarrollo. Los equipos suelen ser pequeños e incluyen a un representante del cliente. Este representante se considera crítico para el éxito del proyecto y debe estar disponible para las aclaraciones y decisiones tácticas del día a día.
- El proceso debe ser capaz de cambiar de dirección para adaptarse a necesidades o requisitos nuevos que aparecen entre las iteraciones. De hecho, idealmente el cliente decide al principio de cada iteración las funcionalidades a añadir en función del valor que estas tengan para él.

Los métodos ágiles comenzaron denominándose «métodos ligeros» en clara contraposición a los métodos anteriores. No siguen ciertas normas de disciplina pero sí utilizan la

planificación. No obstante, la diferencia está en cómo y cuándo se planifica. En un método ágil, la planificación es a muy corto plazo, lo que permite adaptarse a los cambios, mientras que en los métodos más tradicionales la planificación se realiza utilizando plazos más largos, por lo que la reacción a los cambios es más lenta.

Las ideas principales de estos métodos quedaron plasmadas en una declaración conocida como «Manifiesto ágil». Algunos de los principios en ese manifiesto son los siguientes:

- Satisfacer al cliente con entregas rápidas y continuas de software útil.
- El software se entrega de forma frecuente (semanas en lugar de meses).
- El software es la principal medida del progreso.
- Incluso los cambios tardíos en los requisitos son bienvenidos.
- Colaboración directa, cercana y diaria entre los clientes y los desarrolladores.
- Un proyecto se construye con personas motivadas, en las que se puede confiar.
- Debe haber una atención continua a la excelencia técnica y el buen diseño.
- Simplicidad.
- Los equipos se autoorganizan.
- Hay una adaptación regular a las circunstancias cambiantes.

No hay un consenso claro sobre cuándo son apropiados los métodos ágiles. Se sabe que funcionan bien para equipos pequeños, y son útiles cuando realmente se requiere que el proceso se pueda adaptar rápidamente, por ejemplo, cuando se debe reaccionar a la competencia o no se sabe qué los requisitos de los usuarios están cambiando.

En otras áreas y tipos de desarrollo aún se están recolectando experiencias que permitan determinar si los métodos ágiles son igualmente apropiados. En todo caso, parece existir consenso acerca de los tres factores críticos para el éxito de los proyectos de desarrollo con métodos ágiles: la estrategia de entregas cortas, las técnicas ágiles de Ingeniería del Software y la capacidad de los equipos. En la medida que una organización sea capaz de gestionar estos tres factores adecuadamente, contará con mayores posibilidades de alcanzar el éxito en el desarrollo.

Uno de los métodos ágiles más populares es *Extreme Programming* (XP) propuesta por Kent Beck (2000). Como método ágil, se basa en un conjunto reducido de *prácticas* que reflejan una serie de *valores*. Los valores de XP son comunicación (utilizar técnicas que la fomenten), simplicidad (comenzar por las soluciones simples), retroalimentación (de los clientes, del equipo de desarrollo, y del sistema en sí a través de las pruebas), coraje (que implica no dudar, si es necesario, en cambiar el diseño) y respeto (buscar siempre la calidad). En sintonía con estos valores, XP propone una serie de prácticas y un proceso simple. Entre las prácticas innovadoras podemos citar la «programación en parejas», que consiste en que la codificación la realizan dos programadores sentados frente a la misma máquina, uno que escribe y el otro que revisa lo que se escribe, programadores que además intercambian sus papeles con frecuencia. Otras prácticas también interesantes son la integración continua o el desarrollo estructurado en pequeñas entregas.

Un ejemplo de método ágil: SCRUM

SCRUM es un proceso iterativo que denomina *sprints* a las iteraciones. Dichas iteraciones tienen la característica distintiva de que son muy cortas, típicamente de dos a cuatro semanas. Tras cada una, como es común en los métodos ágiles, se produce una versión funcional y potencialmente entregable del producto. El método se articula alrededor de 3 roles, 3 ceremonias y 3 artefactos:

- Roles: el propietario del producto, responsable del valor de negocio del producto; el gestor SCRUM, responsable de que el equipo sea funcional y productivo; y por último, los integrantes del equipo de desarrollo, que se auto-organizan.
- Ceremonias: la reunión de planificación del siguiente *sprint*, las reuniones SCRUM diarias y las reuniones de revisión *sprint*.
- Artefactos: la bitácora del producto, la bitácora del *sprint* y el diagrama de progreso. La bitácora del producto es, básicamente, la lista priorizada de requisitos. La bitácora del *sprint* contiene la planificación de las tareas de la siguiente iteración, mientras que el diagrama de progreso se utiliza para el seguimiento de los *sprints*.

En el proceso minimalista de SCRUM, el énfasis está en adaptarse a la noción de valor que el propietario del producto crea con cada iteración.

2.6.3 Procesos basados en componentes

Cuando se compara la industria del software con otras, como la electrónica, resalta la diferencia en el grado de estandarización de componentes. Los catálogos de componentes electrónicos describen una multitud de elementos con funcionalidades comunes, perfectamente definidas, y requisitos de funcionamiento bien especificados. Algunos creen que el software también puede dividirse en componentes reutilizables, con interfaces bien definidas, y éste es el objeto de lo que se conoce como *desarrollo basado en componentes*.

Un **componente** es una pieza de software auto-contenida, que ofrece unos servicios definidos en sus interfaces y que está preparado para integrarse en otras aplicaciones

El uso y desarrollo de componentes tiene beneficios importantes. Los componentes pueden venderse de forma separada, o, dentro de una misma organización, puede reutilizarse un componente desarrollado en proyectos anteriores para así reducir el tiempo de desarrollo de los proyectos en curso. Además, un componente que se ha reutilizado varias veces ha pasado también varias veces por procesos de prueba del sistema y ha sido expuesto a la valoración de los usuarios en varias ocasiones por lo que, en general, tienen menos defectos y funcionalidades más afinadas.

Por tanto, el desarrollo basado en componentes tiene un hilo director adicional a las necesidades del cliente, que no es otro que el intento de reducir costes, mejorar la calidad

o buscar nuevas oportunidades de negocio. Esto podría verse como un intento de «servir a la vez a dos señores», por lo que los procesos que tienen en cuenta el desarrollo basado en componentes tienen que tener detrás una estructura de gestión donde las prioridades sean claras, así como los criterios de qué es reutilizable y qué no. En muchos casos, conseguir que se reutilice más es simplemente un problema de comunicación interna en la organización, dado que muchos desarrolladores pueden no saber que en otros proyectos pasados en los que ellos no participaron se desarrollaron funcionalidades similares.

La idea del desarrollo basado en componentes es una realización de los procesos de reutilización del software en el estándar ISO/IEC 12207. En cuanto a las actividades técnicas del desarrollo basado en componentes, éstas pueden resumirse en las cuatro siguientes:

- Evaluación de la adecuación. Esta actividad implica seleccionar componentes o comprarlos, evaluando que son apropiados para el uso concreto en el sistema que se está desarrollando. Por lo tanto, hay dos subactividades: descubrimiento y evaluación.
- Adaptación de los componentes. En muchas ocasiones no pueden utilizarse los componentes directamente y necesitan adaptarse. Hay tres posibles tipos de adaptación:
 - De caja blanca, cuando el código fuente está disponible, y se puede modificar para adaptarlo. Es la posibilidad más flexible, obviamente, aunque con este enfoque se pierden las ventajas económicas del uso de componentes cerrados.
 - De caja gris, cuando no se modifica el código fuente, pero el componente tiene interfaces de programación específicas para extender o adaptar su funcionalidad.
 - De caja negra, cuando no se modifica el código fuente ni hay interfaces especiales. En este caso las adaptaciones posibles serían adaptadores software que toman el componente como una caja negra y pre o post-procesan sus salidas.
- Ensamblaje de los componentes. Debe haber algún tipo de tecnología para permitir el ensamblaje de los componentes. Algunos ejemplos de ello son el modelo COM de Microsoft, o modelos distribuidos como Java RMI o CORBA.
- Evolución para usar componentes actualizados. Los componentes evolucionan con el tiempo, es decir, aparecen versiones nuevas, y aunque los buenos componentes deberían ser compatibles «hacia atrás», éste no es siempre el caso. Por ello, la evolución debe gestionarse adecuadamente.

Un desarrollo basado en componentes en un caso ideal sustituiría el desarrollo por la integración de componentes preexistentes. En ocasiones se pueden desarrollar componentes para «líneas de productos», cuando se asume que hay una familia de aplicaciones que comparte una arquitectura común, de modo que se pueden desarrollar piezas y después combinarlas de diferentes formas. El método para la creación de componentes para líneas de productos es a lo que comúnmente se denomina *ingeniería del dominio*.

Es importante distinguir el desarrollo de componentes del diseño y la programación orientada a objetos, aunque esta última se utiliza en la mayoría de los casos de creación de componentes. La orientación a objetos es una aproximación al diseño de programas que se basa en modelar los objetos del mundo real en las entidades del software (las clases). Sin embargo, el diseño de componentes no es en todos los casos un buen diseño orientado a objetos, dado que priman otros criterios. Por eso, la programación basada en componentes y la programación orientada a objetos no siempre coinciden.

2.6.4 Especificaciones de proceso abiertas

Algunas especificaciones de proceso son en sí mismas productos comerciales, o requieren comprar el documento o el software en el que se especifican. El Proceso Unificado de IBM-Rational, por ejemplo, es un producto registrado. Otros procesos desde el principio han sido descritos y difundidos a través de la Web, sin ningún tipo de restricción, pero no se han documentado, desarrollado y mantenido de forma planificada y sostenida, como es el caso de la programación extrema. Ambos casos plantean una restricción o riesgo en su adopción.

La existencia de riesgos en los dos tipos de procesos existentes hasta un determinado momento ha hecho que la idea del código de fuente abierto, que ha tenido un notable auge en los últimos años, se haya aplicado finalmente a la especificación de los procesos software. Un proceso abierto es gratuito y libremente disponible a través de la Web, pero además cuenta con una estructura predefinida, procedimientos específicos para su evolución y una comunidad de usuarios que lo sostiene.

El Proceso Unificado Abierto (OpenUP) es un proceso abierto que se basa en muchas de las ideas del Proceso Unificado. Este proceso tiene una especificación formal en el lenguaje SPEM 2.0, y está incluido en una herramienta software para la gestión de modelos de proceso denominada EPF. El *Eclipse Process Framework* (EPF) es un marco para la edición de modelos y especificaciones de proceso basado en SPEM 2.0. EPF implementa el concepto de «biblioteca de procesos», de modo que cuando una organización quiere desarrollar el suyo puede, si lo desea, tomar partes o patrones de proceso de esa biblioteca.

OpenUP es un ejemplo interesante de proceso definido en varios «niveles». Concretamente, OpenUP considera las siguientes capas:

- Los **incrementos** son procesos orientados al trabajo personal. Así, un incremento es el trabajo de un desarrollador en unas horas o días.
- Las **iteraciones** son intervalos planificados típicamente de unas pocas semanas, en las que el equipo de desarrollo debe producir un resultado tangible (como un prototipo, o una nueva versión de cierta parte del sistema).
- Las **fases** del proceso son comienzo, elaboración, construcción y transición, que ya eran fases en el Proceso Unificado. La división en fases está orientada a los clientes, usuarios y directivos implicados en el desarrollo. El objetivo es tener una visión del progreso general del proyecto. Cada una de las fases se compone de una serie

de iteraciones del mismo orden, y las iteraciones varían en las diferentes fases. Por ejemplo, en la fase de comienzo, una de las tareas es «identificar y refinar los requisitos», mientras que en la fase de transición esa tarea no aparece, pero sí aparecen otras que no aparecen en el comienzo, como «probar la solución». No obstante, esta última tarea sí aparece también en la fase de «construcción». Por lo tanto, hay un cambio progresivo en el tipo de las tareas que se realizan, cuando se avanza por las fases.

Según esa descripción, OpenUP se parece mucho al Proceso Unificado. No obstante, OpenUP pretende ser un proceso ágil, de modo que incorpora un conjunto básico de actividades y está pensado para que se adapte a las necesidades de cada desarrollo concreto. De hecho, entre sus principios destaca «aplicar el proceso mínimo necesario que aporte valor» y «evitar la sobrecarga con productos de trabajo formales que no sean productivos».

OpenUP ofrece un conjunto de prácticas técnicas y de gestión definidas en detalle, que pueden o no tomarse de acuerdo a lo que se considere apropiado en un cierto desarrollo. Por ejemplo, la práctica de «desarrollo dirigido por las pruebas» es una práctica técnica consistente en desarrollar primero el código de las pruebas, y después el código con la funcionalidad. Para esta práctica, OpenUP define:

- Las entradas –que en este caso serían el diseño y la implementación–, y las especificaciones técnicas.
- El propósito y los objetivos de la práctica.
- La descripción de la práctica, que en este caso es una actividad de «implementar las pruebas de desarrollo», seguido de una actividad de «ejecutar el test de desarrollo» que se ejecuta para ver qué falla como comprobación previa, seguido de «implementar la solución», que es desarrollar el código en sí. Después de esta última actividad, se vuelve a la anterior, que en este caso debería ejecutarse correctamente. En caso de que no lo haga, se pasa a implementar –corregir en este caso– de nuevo, y se volverá a ejecutar la prueba. Según OpenUP, este ciclo dura «minutos» y permite al desarrollador concentrarse en qué es el comportamiento correcto del software, que es la medida no ambigua que permite avanzar.
- Información adicional, por ejemplo, referencias bibliográficas.

Además de lo anterior, OpenUP define conceptos y directrices que sirven para entender las prácticas. Por ejemplo, para la práctica anterior, se puede consultar, entre otras, la directriz sobre «pruebas hechas por el desarrollador», que contiene recomendaciones sobre cómo hacer las pruebas. También está relacionada con conceptos como la «propiedad colectiva del código», que se asume en el desarrollo dirigido por las pruebas. Los marcos de proceso como OpenUP se asemejan mucho a repositorios de prácticas, de los cuales podemos tomar o dejar algunas para configurar nuestro propio proceso.

2.7 Resumen

La siguiente nube de palabras resume visualmente los principales conceptos del capítulo, mostrando de manera sencilla e intuitiva la importancia relativa de cada uno.



Figura 2.13: Principales conceptos tratados en el capítulo.

Los procesos software son modelos y especificaciones sobre qué actividades de ingeniería deben realizarse y en qué orden. Desconocemos si existe o no una forma correcta de hacer las cosas, pero la experiencia acumulada ha llevado a la formulación de procesos de diferente índole, que hacen hincapié en determinados aspectos, como puede ser la gestión de los riesgos, la adaptabilidad a los cambios o la disminución del esfuerzo de desarrollo.

Los modelos de proceso, como los basados en prototipos, o los modelos en cascada, o en espiral, son descripciones de alto nivel, no especificaciones detalladas, como sí son el Proceso Unificado y el Proceso Unificado Abierto. Los modelos de proceso más antiguos enfatizaban el trabajo en una secuencia estricta de fases que permitía un control riguroso del estado del proyecto. No obstante, propuestas posteriores han permitido la adaptación a requisitos poco claros o cambiantes, introduciendo el concepto de iteración e incremento para hacer el proceso progresivo, tomando partes de las funcionalidades (incrementos) que se desarrollarán en sucesivos pasos (iteraciones). Aún más allá: los métodos ágiles hacen el proceso más flexible al cambio en los requisitos, por lo que es el cliente quién decide qué se implementa en una secuencia de iteraciones de duración corta.

Hemos visto finalmente que los procesos deben distinguirse de los métodos o técnicas concretas. La orientación a objetos, por ejemplo, es una filosofía de desarrollo que se concreta en diferentes métodos de análisis, diseño, construcción y prueba, si bien los métodos orientados a objetos pueden utilizarse con cualquier tipo de proceso, desde un proceso en cascada a un proceso ágil.

2.8 Notas bibliográficas

Para los interesados en las formulaciones originales de algunos de los modelos de ciclo de vida descritos en el capítulo, pueden resultar de interés las siguientes referencias:

- En «*A Spiral Model of Software Development and Enhancement*» (Boehm, 1998) se describe el modelo en espiral, y aparece por primera vez la idea de iteración en el modelo de ciclo de vida de los desarrollos de software.
- Una buena lectura sobre las diferencias entre los métodos clásicos y los enfoques evolutivos es el artículo «*Evolutionary delivery versus the waterfall model*» (Gilb, 1985), que a pesar de los años transcurridos desde su publicación, aún aporta interesantes reflexiones sobre el desarrollo de software.
- Una interesante reflexión sobre la vigencia del modelo en cascada es la que se plantea en «*El deceso del modelo en cascada y otras leyendas urbanas*» (Laplante y Neill, 2004), argumentando que la muerte de este modelo está aún lejana por motivos diversos, entre otros porque es el modelo de elección de muchos desarrolladores aún hoy. Otra reflexión general interesante y hasta cierto punto complementaria es la que aparece en el artículo «*Puren el ciclo de vida, que yo me bajo*» (Gladden, 1982).

Quienes deseen saber más sobre los métodos ágiles deberían, si no lo han hecho ya, leer el «*Manifiesto ágil*». Disponible en <http://agilemanifesto.org/>, fue publicado en 2001 como respuesta a las tradicionalmente demasiado pesadas metodologías de desarrollo de software. Sobre la experiencia en desarrollos con métodos ágiles, un interesante artículo blanco escrito por Schwaber (Martin y Schwaber, 2004) pone de manifiesto las dificultades de desarrollar software con un modelo clásico y cómo SCRUM, o cualquier otro método ágil, puede resultar una buena idea (aunque no exenta de esfuerzo, ciertamente).

Finalmente, mencionar que muchas de las definiciones del capítulo se han extraído de la Enciclopedia de la Ingeniería del Software de Marcinia (1994), obra de referencia importante para aclarar conceptos y consultar definiciones.

2.9 Cuestiones de autoevaluación

- 2.1 Razonar si la siguiente afirmación es cierta o falsa: «Solamente en el modelo de proceso en cascada hay una división en fases secuenciales del desarrollo, el resto de los procesos simplemente consideran una secuencia de iteraciones».
- R La afirmación es falsa. Muchos otros modelos de proceso y especificaciones de proceso concretos definen fases diferenciadas en el desarrollo. Por ejemplo, el Proceso Unificado define toda una serie de fases, cada una de las cuales puede tener una o más iteraciones.
- 2.2 Razonar si la siguiente afirmación es cierta o falsa: «En los procesos ágiles, se prescinde de la planificación en beneficio de una mayor rapidez de desarrollo, no teniendo los desarrolladores un plazo estricto para cada iteración».

- R. La afirmación es falsa. Los procesos ágiles planifican sus iteraciones, aunque éstas suelen ser más cortas que en otros tipos de proceso. Es más, en algunos métodos la duración máxima de cada iteración está restringida a priori. Estos procesos no consideran que la ausencia de planificación sea una ventaja de ningún tipo.
- 2.3 Razonar si la siguiente afirmación es cierta o falsa: «Los modelos en cascada son propios de las metodologías precedentes a la orientación a objetos».
- R. La afirmación es falsa. Las metodologías de desarrollo no están ligadas irremediablemente a ciertos tipos de proceso. Históricamente hay una cierta coincidencia en la emergencia de modelos posteriores al modelo en cascada y el auge de la orientación a objetos, pero esto no establece una relación esencial entre ellas.
- 2.4 Razonar si la siguiente afirmación es cierta o falsa: «El desarrollo basado en componentes tiene su sentido como método de recortar costes en las empresas de desarrollo».
- R. La afirmación es falsa. Es cierto que el desarrollo basado en componentes tiene como una de sus motivaciones el reducir costes de desarrollo mediante la reutilización, no obstante hay otros factores, como la calidad que se asocia a componentes probados.
- 2.5 El modelo de desarrollo en espiral ¿planifica los ciclos de espiral en el inicio del proyecto?
- R. En la esencia del modelo de proceso en cascada está el que el proyecto se replantea en cada ciclo de la espiral, evaluando la eventual viabilidad del mismo. Por ello, no hay una planificación a priori de las partes que se desarrollarán en cada ciclo.
- 2.6 Dado un proyecto corto, bien definido y teniendo en cuenta que el equipo de desarrollo tiene experiencia previa en proyectos similares. ¿Cree más adecuado un modelo en cascada o un método iterativo e incremental?
- R. En este caso posiblemente el modelo en cascada sería más adecuado, pues se evitaría la planificación extra necesaria en cada iteración.
- 2.7 Indique si es correcta o no la siguiente afirmación: «El modelo de desarrollo en cascada es negativo desde todos los puntos de vista».
- R. Es falsa, ya que existen ciertas características positivas en dicho modelo. Por ejemplo, la existencia de fronteras bien definidas entre fases facilita la evaluación del grado de avance de los proyectos según el modelo en cascada.
- 2.8 ¿Qué es un prototipo evolutivo?
- R. Es un prototipo que, a diferencia de otros, no se diseña para «tirarlo después de usarlo». Se trata de prototipos donde se prueba un «esqueleto» del sistema final con unos pocos requisitos funcionales sobre una arquitectura definitiva.
- 2.9 En el proceso iterativo e incremental puede haber diferentes iteraciones por fase. Así, por ejemplo, la fase de inicio sólo tiene una iteración, mientras que en otras pueden existir varias iteraciones. ¿Cuál es la diferencia entre esto y el modelo en cascada?
- R. En realidad, lo descrito es muy diferente a la posibilidad de «vuelta atrás» del modelo en cascada. En dicho modelo, los ciclos de desarrollo se repiten de forma parcial o completa a lo largo de las iteraciones.

2.10 ¿Cuál de las siguientes no es una ventaja del uso de iteraciones e incrementos en el desarrollo de software?

- El cliente recibe parte de lo encargado antes que con un modelo en cascada.
- Es útil en todo tipo de proyectos.
- La calidad y la estabilidad del software progresan con las iteraciones.
- Facilita la incorporación de personal durante el desarrollo.

R. *La utilización de incrementos e iteraciones no es completamente óptima en todo tipo de proyectos. De hecho, no es útil en proyectos cortos en duración o ámbito, ni en los que la funcionalidad a implementar esté perfectamente definida, ya que en estos existe un sobre-coste de planificación por incremento.*

2.10 Actividades propuestas

- 2.1** En el capítulo hemos estudiado someramente los métodos ágiles. Acceda al «*Manifiesto agil*» (disponible en <http://agilemanifesto.org/>) y discuta en grupo sobre las diferencias y novedades que dicho manifiesto propugna con respecto a los métodos anteriores.
- 2.2** Busque información sobre el proceso de desarrollo para los proyectos de desarrollo colaborativo de código abierto y compare dichos procesos con los métodos vistos en el capítulo.
- 2.3** Construx es una compañía de consultoría software fundada por Steve McConnell para ayudar a otras compañías a incorporar buenas prácticas en el desarrollo. Acceda a su web en <http://www.construx.com/> y tras analizar las prácticas descritas en su proceso «CxOne» a la luz de lo tratado en el capítulo, discuta en grupo sobre ellas.
- 2.4** Busque artículos donde se analice el desarrollo de código abierto y compare esta forma de desarrollo con algunos de los procesos descritos en este capítulo.
- 2.5** Utilizando el entorno de modelado de procesos Eclipse Modeling Framework (EMF), cree una definición de proceso para alguno de los modelos de proceso vistos en el capítulo.
- 2.6** El Proceso Unificado es un marco general de procesos que como hemos visto, promueve un conjunto de principios que se consideran fundamentales en todo proceso. Busque un ejemplo de proyecto que se haya guiado por el marco definido por el proceso unificado y analice cómo se ha adaptado el marco general a ese proyecto en concreto.
- 2.7** La forma en que diferentes procesos enfocan las actividades de *prueba* del software es muy diversa. Por ejemplo, la guía metodológica METRICA-3 (*Metodología de Planificación, Desarrollo y Mantenimiento de sistemas de información, versión 3, especificación del Ministerio de Administraciones Públicas español*), dentro de su «*Proceso de Construcción del Sistema de Información*» (CSI) define una serie de actividades relacionadas con la codificación y la prueba, concretamente las siguientes:
 - Generación del código de los componentes y procedimientos (CSI actividad 2), que se hace según las especificaciones de construcción del sistema de información, y conforme al plan de integración del sistema de información.

- Ejecución de las pruebas unitarias (CSI actividad 3), donde se llevan a cabo las verificaciones definidas en el plan de pruebas para cada uno de los componentes.
- Ejecución de las pruebas de integración (CSI actividad 4), que incluye la ejecución de las verificaciones asociadas a los subsistemas y componentes, a partir de los componentes verificados individualmente, y la evaluación de los resultados.

Por otro lado, el proceso de prueba en el método Extreme Programming (XP) promueve crear el código de las pruebas unitarias antes de la codificación en sí misma.

En el caso de METRICA-3, lo que se definió con antelación es el «plan de pruebas para cada uno de los componentes». En esa especificación previa de las pruebas, se incluye, según la especificación: «Casos de prueba asociados: se definen en detalle los casos de prueba y se detalla cómo proceder en la ejecución de dichos casos, describiendo todas las entradas necesarias para ejecutar la prueba, y las relaciones de secuencialidad existentes entre las entradas, así como todas aquellas salidas que se espera obtener una vez ejecutado el caso de prueba, y las características especiales requeridas, como por ejemplo, tiempo de respuesta».

Considerando cómo afrontan las pruebas los dos procesos (que en ocasiones se mencionan como antagónicos), analizar las similitudes y diferencias de ambos en cuanto a cómo y cuándo se deben realizar las pruebas unitarias: su preparación, su codificación y su evaluación. Toda la documentación sobre METRICA-3 está disponible en la web del Ministerio de Administraciones Públicas de España (<http://www.csi.map.es/csi/metrica3/>).

- 2.8 Busque en internet otros ejemplos de métodos ágiles, y describa brevemente sus características a la luz de lo estudiado en el capítulo.
- 2.9 Investigue en la literatura y busque la formulación original del modelo de ciclo de vida en cascada. Realmente, ¿no existe una única formulación original del mismo?
- 2.10 Busque en internet algunos otros lenguajes para la especificación de procesos y comparelos con lo visto sobre SPEM.

Medición

In God we trust, all others bring data.
— W. Edwards Deming

3.1 La necesidad de medir

La Ingeniería del Software es una disciplina que está aún aprendiendo a medir, a estimar y a mejorar la calidad de sus productos y procesos. Y puesto que uno de los primeros pasos para progresar en la ciencia es tomar medidas e interpretarlas, habrá que comenzar –como sugirió Galileo Galilei– por «*medir lo que sea medible, y hacer medible lo que no lo es*».

La historia de la humanidad ha demostrado cuan difícil resulta establecer el modo correcto de medir ciertas magnitudes. A veces nos ha llevado años, o incluso siglos, conseguirlo. Un buen ejemplo fue la determinación exacta de la longitud en la navegación marítima. Desde que comenzaron los viajes transoceánicos (a finales del siglo XV), hasta que se supo medir adecuadamente la longitud (a finales del siglo XVIII), fueron muchas las pérdidas personales y económicas debidas a naufragios. Numerosos buques se extraviaron con preciosas cargas al navegar sin rumbo desorientados tras una tormenta, o simplemente por la larga duración de los viajes. La causa de ello era la incapacidad para determinar con exactitud las coordenadas de longitud. Tal era la necesidad de llegar al conocimiento de un método fiable para calcular la longitud, que en 1714 el Gobierno Británico ofreció un premio de 20.000 libras de la época a quien encontrara un modo de situar la longitud geográfica con un error menor de 30 millas náuticas (unos 56 kilómetros). La enorme dotación del premio hizo que muchos dedicasen toda su vida a resolver el problema, pero fueron necesarios 60 años para que John Harrison ganara el premio con su cronómetro marino en 1773.

Si comparamos la Ingeniería del Software con la navegación, podríamos decir que estamos en la época en que podemos construir barcos (quizás pequeños si los comparamos

con los que sabremos hacer en el futuro), pero todavía navegamos perdiendo el rumbo en demasiadas ocasiones, no somos capaces de hacer estimaciones precisas, a menudo no alcanzamos la calidad suficiente y nos pasamos de largo con el presupuesto. En definitiva, todavía existen muchas métricas incorrectas desde el punto de vista técnico, así como creencias erróneas sobre la bondad de ciertos métodos o herramientas. Como decíamos al principio, aún estamos aprendiendo qué medir y cómo hacerlo, descubriendo las leyes de la Ingeniería del Software y formalizando la investigación empírica dentro de la disciplina, entre otras cosas. Al fin y al cabo, no podemos esperar alcanzar la madurez de la industria naval en los escasos 60 años de historia de la Ingeniería del Software.

3.2 Objetivos

El objetivo de este capítulo es poner los cimientos de una actividad, la medición, necesaria para todas las demás actividades del ciclo de vida del software. Para ello:

- Aprenderemos los fundamentos de básicos de la medición en general y su aplicación a la Ingeniería del Software.
- Conoceremos las entidades y sus métricas principales.
- Estudiaremos algunos procesos básicos para la aplicación de métricas en la disciplina.
- Presentaremos una visión general de la experimentación en la Ingeniería del Software, un campo de creciente actividad que ayudará a establecer la Ingeniería del Software como una actividad madura de ingeniería.

En otros capítulos se explican actividades y métricas relacionadas con cada fase específica del ciclo de vida, y se muestran ejemplos aplicados. En éste nos limitamos a tratar conceptos generales de medición aplicables a todas las fases, y a poner los cimientos sobre los que se sustentan las actividades de medición. Igualmente, presentaremos los conceptos fundamentales y estableceremos las guías para llevar a cabo dichas actividades.

3.3 Introducción

Como vimos en el Capítulo 1, el Glosario IEEE de Términos de Ingeniería del Software define la Ingeniería del Software como la «*aplicación de una aproximación sistemática, disciplinada y cuantificable al desarrollo del software...*». La medición está pues ligada inexorablemente a nuestra disciplina como una actividad necesaria a lo largo de todo el ciclo de vida del software. En aspectos clave tales como la planificación y gestión de proyectos, la medición resulta fundamental para la estimación de recursos, coste y esfuerzo, la evaluación del personal o el cómputo de la productividad. La medición permite además, durante la ejecución de un proyecto, conocer el estado del mismo para realizar ajustes o mejoras en los

procesos, si fuera necesario. Finalmente, la medición de los productos y sus características hace posible la mejora de su calidad. Por ejemplo, se pueden estudiar qué características del código fuente se dan más en los módulos con errores.

Es importante resaltar que al igual que ocurre con otras áreas de la Ingeniería del Software, la medición está todavía madurando. Por ello, es frecuente objeto de críticas, muchas de ellas relacionadas con la falta de adherencia a la teoría.

3.3.1 Conceptos básicos

La teoría de la representación de la medición establece tanto los principios generales de la medición como su validez. Esta teoría trata de expresar de forma numérica (mundo formal) las entidades del mundo real (o mundo empírico) y la correspondencia entre ambos mundos. Como veremos más adelante, las entidades en la Ingeniería del Software son los procesos, los recursos (personal, oficinas, etc.) y todos los artefactos (código, documentación, etc.) generados durante el ciclo de vida del software. El estándar ISO/IEC 15939 –que veremos con más detalle en la Sección 3.7.3– define entidad del siguiente modo:

Se denomina **entidad** a un objeto que va a ser caracterizado mediante una medición de sus atributos

Los atributos, por otra parte, son las características de las entidades. Por ejemplo, algunos atributos del código fuente pueden ser las líneas de código o su complejidad.

Un **atributo** es una característica medible de una entidad

Para el estudio de la medición, es importante clarificar la diferencia entre *medición* y *medida*, dos términos que utilizaremos mucho en el resto del capítulo.

Medición es el proceso por el que se asignan números o símbolos a atributos de entidades del mundo real para describirlos según unas reglas definidas de antemano

que es la definición clásica de Fenton y Pfleeger (1998), quienes definen *medida* así:

Medida es la asignación de un símbolo o número resultado de una medición a una entidad para caracterizar un atributo

Otra definición más específica de nuestra disciplina para el término *medida* es la que proporciona el Glosario IEEE de Términos de Ingeniería del Software:

Medida es la evaluación cuantitativa del grado en el cual un producto o proceso software posee un atributo determinado

Nos encontramos por tanto con elementos reales, por una parte, y con otros formales o matemáticos, por otra, entre los cuales existe una relación. Así por ejemplo, al atributo *número de líneas de código* del código fuente se le puede asignar un número entero. A otro atributo del código fuente, la *facilidad de mantenimiento*, podríamos asignarle subjetivamente el valor *alto, medio o bajo*, etc.

Las medidas han de satisfacer la denominada «condición de representación», que establece que las relaciones empíricas deben preservar las relaciones numéricas y viceversa. Por tanto, la magnitud A sólo será mayor que la magnitud B si las medidas que tomemos de A son mayores que las que tomemos de B ($A > B$ si y sólo si $M(A) > M(B)$). La función *DuracionProyecto* (por ejemplo), definida como «el número de días transcurridos desde el inicio de un proyecto», cumple la condición de representación pues para todo par de proyectos P_1 y P_2 , siendo P_1 más corto que P_2 , $DuracionProyecto(P_1) < DuracionProyecto(P_2)$.

Otro concepto importante en el mundo de la medición es la noción de escala, que podría definirse del siguiente modo:

Una **escala** de medición es un conjunto de valores que permite establecer relaciones entre medidas. Con frecuencia dicho conjunto es continuo, está ordenado y viene delimitado por un punto inicial y otro final

Dado que se emplean diferentes escalas de medición según la magnitud a medir, es lógico que existan también diferentes tipos de escalas tal y como se detalla a continuación.

3.3.2 Tipos de escalas de medición

Cada tipo de escala engloba a todas las escalas que admiten las mismas *transformaciones admisibles*, es decir, los tipos de operaciones matemáticas que se pueden aplicar a una escala garantizando que se conserva la condición de representación. Cada atributo debe poder ser medido con un tipo de escala determinada y si bien es posible modificar los valores de la misma, el tipo debe mantenerse inalterable. A continuación se listan los diferentes tipos de escalas de menor a mayor complejidad:

- **Escala nominal.** Aquella formada por categorías entre las cuales no existe ningún orden, por lo que la única relación que se puede aplicar es la de igualdad. Por ejemplo, la escala nominal para determinar el sexo, que tiene únicamente 2 valores: {*masculino, femenino*}. O la clasificación de los módulos según su lenguaje de programación: {*Java, C++, Phyton, COBOL, Ruby...*}

- **Escala ordinal.** Aquella en la que se definen categorías pero, a diferencia de la escala nominal, existe una relación de orden «es menor que» entre ellas. Un ejemplo de escala nominal es la escala de Likert, muy utilizada en cuestionarios, en la que se asignan valores enteros entre 1 y 5 que se hacen corresponder generalmente con los valores (*Muy poco, Poco, Medio, Bastante, y Mucho*). El tipo de error en un programa, que podría clasificarse como (*Leve, Moderado o Grave*), es otro ejemplo de escala ordinal.
- **Escala intervalo.** En este tipo de escala la distancia entre intervalos es conocida y siempre la misma, si bien no tienen un valor inicial de referencia o cero absoluto. El ejemplo clásico es el de la escala centígrada o Celsius para la temperatura. En esta escala la diferencia entre 12°C y 13°C es la misma que entre 24°C y 25°C, pero no podemos decir que a 24°C haga «el doble de calor» que a 12°C. Una escala de este tipo en la Ingeniería del Software es la *duración de un proyecto* en días: podemos decir que un proyecto está en el día 200, pero no tiene ningún sentido decir que un proyecto va a empezar el «doble de tarde» que otro.
- **Escala de ratio.** Este tipo de escalas es el que más información proporciona y por tanto el que permite llevar a cabo análisis más interesantes y completos. Tienen un valor inicial de referencia o cero absoluto, y permiten definir ratios coherentes con los valores de la escala, por lo que se pueden comparar los valores estableciendo proporciones. El ejemplo clásico es el de la escala Kelvin de medición de temperaturas. En la Ingeniería del Software, un ejemplo de este tipo de escalas es la longitud de un programa en líneas de código, que permite decir que un programa es *el doble de largo o la mitad de largo* que otro.
- **Escala absoluta.** Escalas con las características de las escalas anteriores, si bien consisten simplemente en la cuenta sin transformación del número de entidades. El número de programadores involucrado en un desarrollo, algo que sólo se puede medir contando, es un ejemplo de escala absoluta.

En la Tabla 3.1 se muestran los distintos tipos de escalas con las diferentes operaciones matemáticas y estadísticas que se pueden aplicar a cada una de ellas. Téngase en cuenta que para cada una de ellas no sólo se pueden aplicar las operaciones que se indican, sino también todas las aplicables a las escalas de menor complejidad que la preceden.

3.3.3 Clasificación de las medidas

Una vez definidos los conceptos fundamentales, trataremos las medidas con mayor profundidad. Vaya por delante el hecho de que no existe una única clasificación de las medidas y que, por el contrario, se suelen clasificar según distintos criterios. Una clasificación habitual consiste en dividir las medidas de un atributo en dos tipos, directas e indirectas:

Tabla 3.1: Resumen de escalas de medición y operaciones permitidas

Escala	Operaciones estadísticas	Operaciones matemáticas
nominal	moda	igualdad (=)
ordinal	mediana	orden ($<$, $>$)
intervalo	media, desviación estándar	+,-
ratio	media geométrica, coeficiente de variación	+,-, \times , \div

- **Medidas directas:** las medidas directas de un atributo son aquellas que pueden ser obtenidas directamente de la entidad sin necesidad de ningún otro atributo. Ejemplos de medidas directas en la Ingeniería del Software serían la *longitud* del código, el *número de defectos* durante los 6 primeros meses del sistema en producción o las *horas de trabajo* de un programador en un proyecto.
- **Medidas indirectas:** son aquellas que se derivan de una o más medidas de otros atributos. Se definen y calculan, por tanto, a partir de otras medidas. Un ejemplo de medida indirecta es la densidad de defectos de un módulo, que se define como el número de defectos del módulo dividido por su tamaño.

Otra clasificación diferencia entre medidas objetivas y subjetivas, pues dependiendo de uno u otro tipo éstas se emplean como base de muy diferentes estudios:

- **Medidas objetivas:** una medida cuyo valor no depende del observador.
- **Medidas subjetivas:** son aquellas en las que la persona que realiza la medición puede introducir factores de juicio en el resultado. No obstante, el interés de estas medidas está en que incluyen percepciones, opiniones y juicios que en ciertos casos resultan valiosos.

Aunque en alguna literatura sobre medición de software se da un significado específico y distinto a los términos *medida* y *métrica*, habitualmente se usan de manera indistinta. Ambos términos tienen las siguientes acepciones: (i) como medida, es decir la asignación de un número a una entidad; (ii) como escala de medición; y (iii) como atributo de las entidades, por ejemplo, métricas orientadas a objetos. Para los propósitos de este libro, consideraremos medida y métrica términos sinónimos.

3.3.4 Evaluación de las métricas

La definición de métricas y modelos de medición como correspondencia entre entidades del mundo real y números no es trivial. Una métrica debe medir adecuadamente el atributo de la entidad a medir, pero también definir inequívocamente cómo se va a realizar la medición.

Es por tanto necesario, para que las métricas y modelos de medición cobren sentido, que sean evaluados tanto teórica como experimentalmente.

Desde el punto de vista teórico, las métricas deben cumplir ciertas propiedades para ser consideradas válidas. Aunque existen estudios donde se enumeran detalladamente dichas propiedades, aquí no las estudiaremos en profundidad, pero sí enumeraremos algunas para ilustrar al lector. En el caso de las *métricas directas* (aquellas que se obtienen directamente de la entidad sin ningún otro atributo intermedio) es necesario por ejemplo que la métrica permita distinguir diferentes entidades entre sí, que cumpla la *condición de representación* o que permita que diferentes entidades puedan tener el mismo valor, entre otros.

Pero además de la evaluación teórica, es importante que una métrica sea evaluada de manera experimental. Para corroborar la validez de las métricas desde este punto de vista se emplean métodos empíricos, que pueden clasificarse en encuestas, casos de estudio y experimentación formal. Así, usando métodos estadísticos y experimentales, se evalúan la utilidad y relevancia de las métricas. En la Sección 3.8 se describen en más detalle las distintas posibilidades a la hora de evaluar las métricas de Ingeniería del Software.

3.3.5 ¿Qué medir en la Ingeniería del Software?

Una vez introducidos los fundamentos de la teoría de la medición, necesitamos definir los tipos de entidades que encontramos en la Ingeniería del Software, para después definir sus correspondientes atributos, que es sobre los que se llevan cabo las mediciones. En concreto son tres los tipos de entidades:

- **Productos.** Cualquier artefacto, entregable o documento que resulta de cualquiera de las actividades (procesos) del ciclo de vida del software. El código fuente, las especificaciones de requisitos, los diseños, el plan de pruebas y los manuales de usuario son algunos ejemplos de productos.

Tabla 3.2: Ejemplos de métricas de productos, adaptado de Fenton y Pfleeger (1998)

Producto	Atributos internos	Atributos externos
Especificaciones	Tamaño, reutilización, etc.	Calidad, legibilidad
Diseño	Tamaño, acoplamiento, cohesión, complejidad, etc.	Calidad, complejidad
Código	Tamaño, complejidad...	Facilidad de mantenimiento, fiabilidad
Casos de prueba	Número de casos, %cobertura...	Calidad

- **Procesos.** Todas las actividades del ciclo de vida del software: requisitos, diseño, construcción, pruebas, mantenimiento, etc. Las mediciones en los procesos van encaminadas, en primer lugar, a conocer el estado de los procesos y cómo se llevan a cabo, para después mejorarllos. Algunas de las métricas relacionadas con los procesos son el tiempo invertido en las actividades o el tiempo para reparar un defecto.

Tabla 3.3: Ejemplos de métricas de procesos, adaptado de Fenton y Pfleeger (1998)

Proceso	Atributos internos	Atributos externos
Requisitos	Tiempo, esfuerzo, número de requisitos	Calidad, coste, estabilidad
Diseño	Tiempo, esfuerzo, número de errores, etc.	Calidad, coste, estabilidad
Pruebas	Tiempo, esfuerzo, número de errores, etc.	

- Recursos.** Cualquier entrada de una actividad. Por ejemplo, el número de personas por actividad o proyecto, las herramientas utilizadas (herramientas para requisitos, compiladores, etc.), oficinas, computadoras, etc.

Tabla 3.4: Ejemplos de métricas de recursos, adaptado de Fenton y Pfleeger (1998)

Recurso	Atributos internos	Atributos externos
Personal	Edad, salario	Productividad, experiencia
Equipos	Número de personas, estructura del equipo	Productividad, experiencia
Software	Coste, número de licencias, etc.	Usabilidad, fiabilidad
Hardware	Marca, coste, especificaciones técnicas, etc.	Usabilidad, fiabilidad

Como hemos comentado anteriormente, a las entidades se les asignan atributos a medir. En la literatura se distingue entre atributos internos y atributos externos:

- Atributos internos.** Los atributos internos de un producto, proceso o recurso son aquellos que se pueden medir directamente a partir de dicho producto, proceso o recurso. En un módulo software, por ejemplo, podemos medir directamente el número de defectos que se han encontrado en el mismo. Sin embargo, es importante resaltar que el principal uso de los atributos internos es la medición de los atributos externos, como a continuación veremos.
- Atributos externos.** Los atributos externos de productos, procesos o recursos son aquellos que los relacionan con el entorno. Se miden por medio de métricas indirectas y se deducen en función de atributos internos. Como atributos externos se suelen considerar los relacionados con la calidad. Aunque lo veremos en más detalle en el Capítulo 9, dedicado exclusivamente a la calidad, aquí podemos adelantar que la calidad suele dividir en factores que no pueden medirse directamente, como por ejemplo, la fiabilidad, la eficiencia, la usabilidad o la facilidad de mantenimiento. A cada uno de estos factores se les asigna una o varias métricas. Por ejemplo, al atributo *facilidad de mantenimiento* de una entidad software, se le pueden asignar métricas como el tiempo medio para reparar un defecto, el número de errores no resueltos, el porcentaje de modificaciones que introducen errores, etc.

En las siguientes secciones estudiaremos las diferentes medidas según su objeto, primero las medidas del producto, luego las medidas de los procesos y finalmente las de los recursos.

3.4 Medidas del producto: atributos internos

3.4.1 Medidas del tamaño de los sistemas

La cuenta de las *líneas de código* (*LoC – Lines of Code*) es una de las métricas más usadas, esencialmente por su facilidad y simplicidad de cálculo. Sin embargo, aunque *LoC* puede parecer una métrica sencilla e intuitiva, su definición no es trivial. Y lo que es todavía más importante, su definición y modo de cómputo debe homogeneizarse para que todos los interesados entiendan y apliquen la métrica del mismo modo. Porque por ejemplo, una persona podría considerar las líneas en blanco y/o los comentarios como líneas válidas, mientras que otra podría no tenerlos en cuenta.

También es discutible si debe considerarse que las declaraciones de variables hayan de formar parte de la cuenta de instrucciones del lenguaje de programación o no. Por todo lo anterior, es frecuente toparse con variantes de la métrica *LoC* tales como *NCLoC* (*Non-Comment Lines of Code*, líneas de código sin comentarios), *CLoC* (*Comment Lines of Code*, líneas de código con comentarios) o *DSI* (*Delivered Source Instructions*, número de sentencias o instrucciones). Se debe por tanto saber exactamente a qué nos referimos cuando usamos las líneas de código para comparar tamaños de programas y sistemas¹ o cuando hacemos estimaciones de tamaño y productividad (que se pueden medir en *LoC/día*).

Dada la variabilidad de los resultados de utilizar como medida de tamaño las líneas de código (algunos autores han estimado que puede llegar a diferencias del 500%), y puesto que esta métrica es además muy dependiente del lenguaje de programación usado, han aparecido métricas de tamaño más perfeccionadas. Los *puntos de función*, originalmente descritos a finales de los años 1970 por Allan Albrecht, miden el tamaño del software por la cantidad de funcionalidad que proporciona a los usuarios (sin considerar el código fuente). Se basan fundamentalmente en la cuenta de entradas, salidas, accesos y modificaciones a las bases de datos y ficheros ponderada por la complejidad de cada uno de ellos (ver Sección 10.4.2, donde se ofrece una visión más amplia de los puntos de función).

Todas las métricas tratadas hasta el momento son métricas directas de tamaño. Un ejemplo de métrica indirecta en el tamaño del software es la *densidad de comentarios* de un programa, que se calcula dividiendo el número de líneas con comentarios entre el número de líneas totales, entendiendo aquí *LoC* en su acepción más común (cualquier sentencia, salvo las de comentario y las líneas en blanco):

$$DoC = CLoC / (LoC + CLoC)$$

Esta métrica suele utilizarse como indicador de la legibilidad de un código o de su facilidad de mantenimiento ya que, en teoría, cuantos más comentarios haya en el código fuente, más fácil será de entender y en consecuencia mantener.

Las métricas de tamaño se utilizan a menudo como umbrales e indicadores de posibles problemas. Como ejemplo de ello se pueden citar los estudios que corroboran el límite de

¹Para referirse a este tipo de comparaciones se utiliza a menudo el término inglés *benchmarking*.

líneas a partir del cual un código es menos legible. McCabe, concretamente, establece que si una función excede de 60 líneas, que es más o menos lo que entra en una página impresa, la función es más propensa a tener errores.

3.4.2 Medidas de la complejidad del software

En esta sección estudiaremos los dos conjuntos de métricas clásicas para medir la complejidad del software: las métricas de McCabe y la ciencia del software de Halstead. Ambas fueron desarrolladas como indicadores para la estimación del coste, esfuerzo, número de defectos y facilidad de mantenimiento del software, entre otros atributos. Hoy en día son aún populares tanto por el hecho de que no son específicas de ningún lenguaje de programación en concreto, como por la existencia de numerosas herramientas software que las implementan y dan soporte.

McCabe y las métricas de complejidad

La **complejidad ciclomática** de McCabe $v(g)$ se basa en la cuenta del número de caminos lógicos individuales contenidos en un código de programa. Para calcular la complejidad ciclomática, el programa (o fragmento de programa) se representa como un grafo cuyos nodos son las instrucciones, y los posibles caminos sus aristas. Una vez representado de este modo, la complejidad ciclomática se calcula como:

$$v(g) = e - n + 2$$

donde e representa el número de aristas y n el número de nodos, esto es, el número de posibles caminos del código. Teniendo en cuenta que la psicología establece el límite normal de elementos simultáneos en la memoria de trabajo humana en 7 ± 2 , diremos que un módulo es más complejo, y por tanto potencial causa de problemas, en función de la cercanía de su complejidad ciclomática a dicho límite.

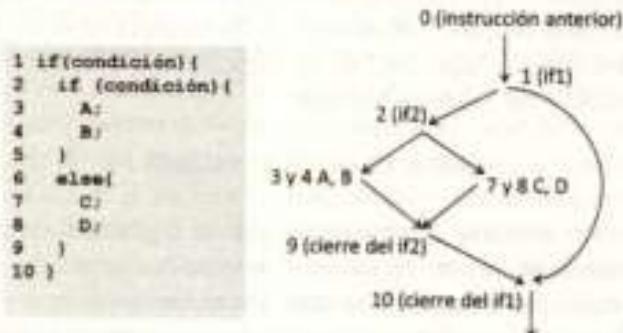


Figura 3.1: Cómputo de la complejidad ciclomática

La parte izquierda de la Figura 3.1 muestra un fragmento de código con 2 sentencias selectivas (`if`) anidadas, que se representan según el grafo de la parte derecha. Dado que en dicho grafo se identifican 3 posibles caminos, se dice que la complejidad ciclomática es 3 (aplicando la fórmula vista para su cálculo, se obtiene $v(g) = 9 - 8 + 2 = 3$).

La complejidad ciclomática se utiliza, por ejemplo, en la *metodología de pruebas estructurada* (ver Capítulo 7). Otros usos de la complejidad de McCabe son medir la complejidad de la integración de módulos, evaluar la dificultad de automatizar las pruebas de un código o incluso medir su fiabilidad. En la Tabla 3.5 se muestran los umbrales comúnmente aceptados para clasificar módulos de software, datos basados no sólo en la psicología humana sino también en estudios realizados por el propio McCabe a partir de cierto número de proyectos de software. En dicha tabla se indica qué rangos de complejidad son potencial causa de problemas y qué tipo de módulos software, por tanto, deberían ser probados más cuidadosamente.

Tabla 3.5: Clasificación de los módulos según su complejidad ciclomática

Complejidad ciclomática	Complejidad del código
1-10	Simple, sin riesgos
11-20	Algo complejo, riesgo moderado
21-50	Complejo, riesgo elevado
51+	Muy difícil de probar, riesgo muy alto

Si bien la complejidad ciclomática mide la cantidad del código, poco o nada dice acerca de su calidad. Para medir específicamente dicho atributo, y con el objetivo final de evitar lo que comúnmente se conoce como «código spaghetti» (código no estructurado), McCabe definió la denominada **complejidad esencial**. La complejidad esencial $ev(g)$ se calcula de modo similar a la complejidad ciclomática, pero utilizando un grafo simplificado en el que se eliminan las construcciones básicas de la programación estructurada (secuencias, estructuras selectivas e iteraciones). Dijkstra demostró que cualquier programa puede expresarse utilizando únicamente estas construcciones. Por tanto, el grafo resultante una vez eliminadas dichas estructuras mostrará el grado de alejamiento con respecto al canon o, en otras palabras, su grado de imperfección.

La Figura 3.2 muestra cómo una estructura selectiva pura (en la parte izquierda del diagrama) se simplifica, pero no es posible hacer lo mismo con la estructura selectiva que aparece en la parte inferior de ese mismo grafo ya que existen saltos (`goto`) desde el interior de la misma a otras zonas del código, incumpliendo así la regla de Dijkstra. El rango de esta métrica es $1 \leq ev(g) \leq v(g)$, por lo que cuanto más cercano a uno sea su valor, más estructurado será el código.

La **complejidad del diseño del módulo** $iv(g)$ es otra medida de complejidad que utiliza un grafo simplificado para sus cálculos. En este caso, la reducción se aplica en las llamadas a otros módulos. Se trata de una medida muy útil en las pruebas de integración.

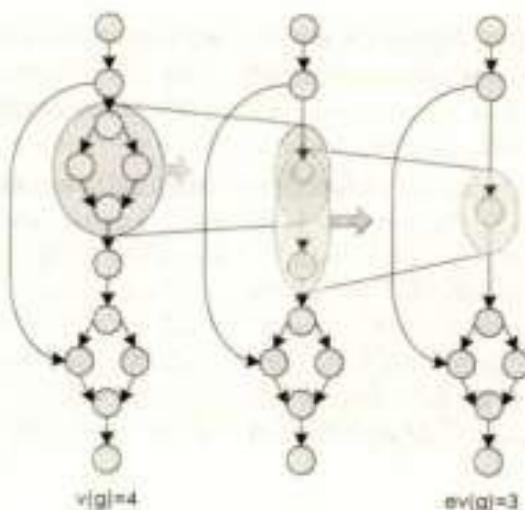


Figura 3.2: Complejidad ciclomática ($v(g)$) vs. complejidad esencial ($ev(g)$)

Halstead y la ciencia del software

A finales de 1970, Halstead desarrolló un conjunto de métricas conocidas en su conjunto como la *ciencia del software de Halstead*. Estas métricas se basan en último término en computar los *operadores* y *operandos* de un programa:

- Los operadores son las palabras reservadas del lenguaje (tales como `if`, `while` o `for`), los operadores aritméticos (+, -, *, etc.), los de asignación (=, +=, *=, etc.) y los operadores lógicos (AND, OR, etc.)
- Los operandos son las variables, los literales y las constantes del programa.

Halstead propone varias medidas diferentes que basan en el cálculo previo del número de operadores y operandos únicos, y del número total de operadores y operandos. Para calcular todos estos valores utiliza la siguiente notación:

n_1 – número de operadores únicos que aparecen en un código.

N_1 – número total de ocurrencias de operadores.

n_2 – número de operandos únicos que aparecen en un código.

N_2 – número total de ocurrencias de operandos.

Así, si por ejemplo consideramos el siguiente fragmento de programa:

```
if (MAX < 2){
    a = b * MAX;
    System.out.print(a);
}
```

obtendremos los siguientes valores:

$$n_1 = 6 \text{ (if, [], System.out.print(), =, *, <)}$$

$$N_1 = 6 \text{ (if, [], System.out.print(), =, *, <)}$$

$$n_2 = 4 \text{ (MAX, a, b, 2)}$$

$$N_2 = 6 \text{ (MAX, 2, a, b, MAX, a)}$$

A partir de estos cuatro parámetros, Halstead elabora diferentes métricas para diversas propiedades de los programas, independientemente —como hemos dicho— del lenguaje de programación utilizado. Las más relevantes son las siguientes:

- **Vocabulario** ($n = n_1 + n_2$). El vocabulario es una medida de la complejidad de las sentencias de un programa a partir del número de operadores y operandos únicos. Se basa en el hecho de que un programa que utiliza un número reducido de elementos muchas veces será, según Halstead, menos complejo que un programa que emplea un mayor número de elementos.
- **Longitud** ($N = N_1 + N_2$). La longitud mide el tamaño de un programa: cuanto más grande, mayor será la dificultad para comprenderlo. Se trata de una medida alternativa a la simple cuenta de líneas de código y también fácil de calcular. N es sin embargo más sensible a la complejidad, porque no asume que todas las instrucciones son igualmente fáciles o difíciles de entender.
- **Volumen** ($V = N \cdot \log_2(n)$). El vocabulario se define como el número de bits necesarios para codificar un programa en un alfabeto que utiliza un único carácter para representar todo operador u operando. Cuando la longitud es una simple cuenta del total de operadores y operandos, el volumen da un peso extra al número de operadores y operandos únicos. Por ejemplo, si dos programas tienen la misma longitud N pero uno de ellos tiene un mayor número de operadores y operandos únicos, que naturalmente lo hacen más difícil de entender y mantener, éste tendrá un mayor volumen.
- **Esfuerzo mental** ($E = V/L$). El esfuerzo proporciona una estimación del trabajo requerido para desarrollar un programa, dividiendo su volumen por el nivel del lenguaje, L (este indicador L varía en función de si se está utilizando un lenguaje de alto o bajo nivel). El esfuerzo es proporcional al volumen, pero decrece cuando se utiliza un lenguaje de mayor nivel. Así por ejemplo, una llamada a un procedimiento podría tener un valor $L = 1$ en Java, mientras que en COBOL podría ser 0,1 y en lenguaje ensamblador 0,01. Según estudios empíricos, E es una mejor medida de la facilidad para comprender un programa de lo que lo es N . La motivación original de Halstead al crear esta métrica fue representar el esfuerzo mental necesario (en términos de operaciones mentales de discriminación) para escribir un programa de longitud N .

Las métricas de Halstead son tan ampliamente utilizadas como criticadas por su comprometida validación empírica, ya que muchas de las métricas se definen en función del

número de discriminaciones mentales, un valor ciertamente difícil de definir y evaluar. Debe tenerse en cuenta además que se trata de métricas pensadas para medir programas una vez se tiene el código completo, lo cual impide utilizarlas para realizar estimaciones, por ejemplo. Son, sin embargo, útiles durante las actividades de prueba pues permiten identificar aquellos módulos potencialmente problemáticos de acuerdo a su complejidad.

Existen muchas otras métricas relacionadas con el tamaño y la complejidad del código. En general estas métricas son utilizadas para predecir esfuerzo en las etapas de construcción y mantenimiento, para detectar posibles módulos defectuosos y por tanto para determinar dónde invertir el esfuerzo de pruebas. Dado que se trata de decenas de métricas, describir las todas se escapa a la intención de esta obra. Se recomienda al lector interesado en profundizar en el estudio de estas métricas, acudir a los trabajos de Fenton y Pfleeger o de Zuse incluidos en la sección de referencias bibliográficas al final de este capítulo.

3.4.3 Medidas de la documentación

Además de las métricas estudiadas en las secciones anteriores, centradas en la medición de atributos del propio código en sí, existen otras que miden aspectos relacionados con el mismo. Las métricas de la documentación, por ejemplo, permiten precisar la documentación generada en cada una de las distintas fases del ciclo de vida. Así, en la fase de requisitos es posible contar el número de requisitos, el número de casos de uso o el número de cambios en los requisitos por mes, entre otros. Existen además otras métricas provenientes de la lingüística, más generales y no específicas de la Ingeniería del Software, que permiten medir por ejemplo la legibilidad de un documento. Entre este tipo de métricas se incluyen las siguientes:

Índice de Gunning-Fog: se trata de una medida del nivel de estudios que una persona debe tener para comprender un texto. Esta métrica, que se diseñó para textos en inglés y específicamente para el sistema de educación americano, se basa en el siguiente proceso. En un texto de aproximadamente 100 palabras, se calcula así:

$$Fog = 0.4 \times \left(\frac{N^o \text{ Palabras}}{N^o \text{ Frases}} + 100 \times \left(\frac{N^o \text{ Palabras Complejas}}{N^o \text{ Palabras}} \right) \right)$$

donde se consideran *Palabras Complejas*, las de 3 o más sílabas. La métrica toma valores entre 1 y 12 para representar el nivel de educación necesario –en el sistema norteamericano– para comprender un texto. Aún no conociendo este sistema la interpretación es sencilla: cuanto menor el índice, más fácil de entender el texto.

Índice de facilidad de lectura de Flesch: señala cómo de fácil es la comprensión de un texto a través del cómputo de la siguiente ecuación:

$$Flesch = 206,835 - 1,015 \times \left(\frac{\text{total palabras}}{\text{total frases}} \right) - 84,6 \times \left(\frac{\text{total sílabas}}{\text{total palabras}} \right)$$

Este índice, cuyo rango oscila entre 0 y 100, se interpreta del siguiente modo: cuanto menor es el índice, más difícil será la comprensión del texto. Valores superiores a un 80% o 90% indican que el texto puede ser comprendido por la práctica totalidad de la población.

Aunque existen muchas otras métricas para evaluar la facilidad de comprensión de textos, las dos descritas se encuentran entre las más utilizadas. Son de hecho tan populares, que están implementadas en procesadores de texto ampliamente conocidos y utilizados como Microsoft Word y Google Docs.

3.4.4 Medidas de reutilización

La reutilización de documentación, diseños, código, casos de prueba, etc. es de primordial importancia a la hora de desarrollar nuevos proyectos. Como sabemos, la reutilización afecta de forma positiva a la calidad y productividad de un proyecto. Desde el punto de vista de la gestión de un proyecto de desarrollo resulta esencial conocer el grado de reutilización, para así poder saber cuánto código será producido y cuánto será reutilizado a partir de desarrollos anteriores, de bibliotecas externas, etc. Pese a todo, la definición y medición del grado de reutilización no es trivial.

Es frecuente clasificar los módulos, clases, etc. mediante rangos nominales con valores como *completamente reutilizado*, *ligeramente modificado* (si el número de líneas modificadas es menos del 25%), *muy modificado* (si el número de líneas modificadas es más del 25%) y *nuevo* (cuando el módulo, función o clase es completamente nuevo). Sin embargo, es más común emplear como métrica general de la reutilización el *porcentaje de reutilización* de líneas de código, que se calcula de la siguiente manera:

$$\text{Porcentaje de reutilización} = \frac{\text{LoC reutilizadas}}{\text{Total LoC}} \times 100$$

3.4.5 Medidas de la eficiencia

Muchas veces necesitamos medir la eficiencia de un software. En sistemas de tiempo real, por ejemplo, es obligatorio garantizar una respuesta dentro de un determinado rango de tiempo y por tanto, el modo natural y habitual de evaluar estos programas es medir el tiempo que demoran en ejecutarse. Pero el tiempo de ejecución es una métrica externa, ya que un cierto programa puede ejecutarse en más o menos tiempo dependiendo de la entrada que se le proporcione, del compilador utilizado o de la plataforma sobre la que se ejecute. Si lo que se desea es medir la eficiencia examinando únicamente el código del programa (es decir, mediante métricas internas), podemos contar el número de operaciones que efectúa el algoritmo dada una entrada. Esta métrica se ha formalizado matemáticamente en lo que se denomina *orden* (O grande) de un algoritmo. Veamos un ejemplo.

Existen diferentes algoritmos de búsqueda. La *búsqueda lineal* depende sólo del número de elementos a buscar, n , lo que matemáticamente se denota como $O(n)$. Este $O(n)$ —que se lee «complejidad de orden n »— representa el hecho de que, en el peor de los casos, necesitaremos recorrer los n elementos de la entrada para encontrar el elemento buscado.

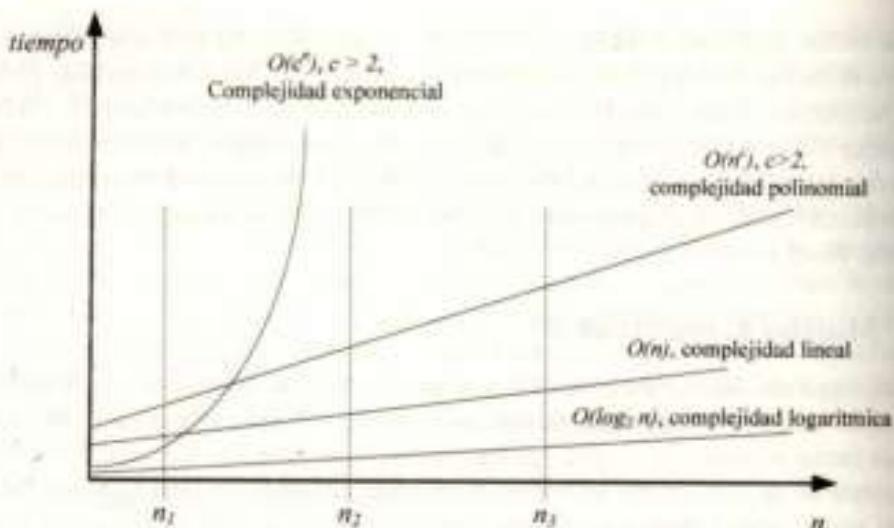


Figura 3.3: Ejemplos de orden de función

Sin embargo, un algoritmo de *búsqueda binaria* en conjuntos de elementos ordenados, el cual utiliza un método de búsqueda similar al modo en que los humanos buscamos una palabra en un diccionario, establece la cota del número de elementos analizados necesarios para encontrar el buscado sensiblemente menor: $\log_2 n$. Es decir, el orden del algoritmo de búsqueda binaria es $O(\log_2 n)$, lo que indica que este tipo de búsqueda es significativamente más eficiente que la búsqueda lineal.

Aunque el análisis de algoritmos se escapa al ámbito de este libro (ya que es parte de lo que se denomina *algorítmica*), desde el punto de vista de la Ingeniería del Software es importante para estimar y acotar el tiempo de ejecución de los programas. Así, y pese a que existen casos en que con entradas muy pequeñas un algoritmo exponencialmente acotado puede ser más eficiente que uno polinomial (aquejlos cuyo tiempo de ejecución depende de una función polinómica), casi siempre preferiremos algoritmos polinomialmente acotados. La Figura 3.3 muestra gráficamente cómo para una entrada pequeña n_1 el tiempo de ejecución de un algoritmo exponencialmente acotado es menor que el de algoritmos menos complejos, si bien para valores grandes (n_2 , n_3 y siguientes) los tiempos de respuesta devienen inaceptables conforme se incrementa el tamaño de la entrada.

3.5 Medidas del producto: atributos externos

Los atributos externos tratan de medir características que dependen de la visión externa del producto, asociándose generalmente esta visión con la calidad del producto. Entre los modelos de calidad más conocidos se encuentran los de McCall y Bohem, el estándar ISO



Figura 3.4: Factores, subfactores y métricas en modelos de calidad

9126 y su evolución en el estándar ISO/IEC 14598 (ver Sección 9.4). En dichos modelos, a cada entidad (producto, proceso o recurso) se asignan unos atributos externos que en el mundo de la calidad se denominan *factores de calidad*. Algunos de estos factores son la *corrección*, la *fiabilidad*, la *facilidad de uso* o la *eficiencia*. Los factores de calidad se pueden a su vez dividir en otros subfactores (atributos internos) que se pueden calcular directamente de las entidades, por ejemplo la legibilidad, el acoplamiento o la eficiencia, por nombrar sólo unos pocos. La Figura 3.4 resume cómo se definen los modelos de calidad citados en función de sus atributos externos e internos.

Toda entidad generada durante el ciclo de vida puede tener su modelo específico de calidad generado a partir de modelos de calidad como los mencionados anteriormente. Por ejemplo, a la hora de medir requisitos podemos considerar la legibilidad de los documentos, pero no la eficiencia, mientras que a la hora de definir un modelo de calidad para el código, podríamos considerar tan importante la legibilidad como la eficiencia. Además, muchas entidades serán artefactos generados y después refinados en sucesivas iteraciones a lo largo del ciclo de vida. Por ejemplo, los modelos de diseño de alto nivel se transformarán en modelos de diseño de bajo nivel y estos últimos pueden tener distintas versiones según se van realizando iteraciones.

El Capítulo 9 está exclusivamente dedicado al estudio de la calidad y define y estudia los modelos más conocidos, proporcionando algunas métricas relevantes. Por ello, y porque resulta más propio de un capítulo que trate de la calidad que de uno que trate las métricas, no estudiaremos con profundidad los atributos externos.

3.6 Medidas del proceso y los recursos

A diferencia de las métricas vistas hasta el momento, basadas en la medición de atributos del producto, las métricas que veremos en esta sección evalúan el proceso de desarrollo y los recursos empleados en el mismo.

3.6.1 Medidas relacionadas con el proceso

Existen dos tipos de métricas del proceso, las internas –que incluyen el tiempo de desarrollo del producto, el esfuerzo que conlleva dicho desarrollo o el número de incidentes, defectos o cambios en las distintas fases del ciclo– y las externas –que incluyen la facilidad de observación, la estabilidad del proceso o su coste, por citar sólo algunas–.

Hoy en día se dedican importantes esfuerzos a la mejora (y por tanto evaluación) de los procesos, pues mejorando la calidad del proceso se mejora la calidad del producto. Entre los más conocidos modelos de mejora de procesos se encuentran CMMI, SPICE, ISO 9000, Six Sigma y COBIT, de los que trataremos con más detalle en el Capítulo 9. Algunos de estos modelos, como CMMI y SPICE, agrupan los procesos con sus respectivas métricas en distintos niveles de madurez, con la idea de que una organización vaya progresivamente mejorando sus procesos.

3.6.2 Medidas relacionadas con los recursos

Resultan de gran utilidad en la Ingeniería del Software las medidas que se llevan a cabo sobre los recursos. Dentro de esta categoría veremos métricas específicas para cada uno de los principales recursos, que son el *personal*, las *herramientas* utilizadas en el proceso de desarrollo, los *materiales* y los *métodos*. Para todos ellos podremos medir el coste, utilizado principalmente por los gestores de proyectos para decidir en qué recursos invertir el presupuesto disponible. Con el propósito de que la inversión lleve a la máxima productividad –definida genéricamente como la relación entre la cantidad de trabajo obtenido y el esfuerzo invertido– y siempre dependiendo de las circunstancias del proyecto, se invertirá más en un tipo de recurso (por ejemplo en personal) o en otro (por ejemplo herramientas).

El concepto de productividad es muy utilizado en las medidas relacionadas con los recursos. La definición general que hemos esbozado no aspira a definir el término de manera cuantificable, aunque sin embargo, a menudo será necesario definirla de este modo. De aquí en adelante determinaremos así la productividad en Ingeniería del Software:

$$\text{Productividad} = \frac{\text{Tamaño}}{\text{Esfuerzo}}$$

donde el *tamaño* vendrá dado normalmente bien por el número de líneas de código (*LoC*), o bien por el número de puntos de función, mientras que el *esfuerzo* se medirá casi siempre en personas por mes. En este cálculo de la productividad obviamente influyen todos los tipos de recursos.

En el caso particular de las herramientas esto resulta especialmente evidente –por ejemplo diferentes compiladores pueden tardar distinto tiempo en compilar el mismo programa– pero también es cierto para el resto de recursos materiales, sobre todo para el personal. A continuación se describen algunos ejemplos de medidas de recursos.

Medición del personal

Dentro de las métricas relacionadas con el personal podemos distinguir entre aquellas que miden atributos de la persona como unidad individual y aquellas cuyo objeto de medición son los diferentes grupos o equipos que pueden formarse dentro del personal. A nivel individual, las medidas más utilizadas son el coste (el salario), la productividad individual, y la experiencia. A la hora de medir esta última, por ejemplo, se deben tener en cuenta distintas habilidades, tanto técnicas como de gestión. Desde una perspectiva colectiva, es posible tomar, dentro de un equipo de personal, medidas indirectas como la media o la moda de alguna de las métricas individuales. La media de edad de un equipo, por ejemplo, pertenece a este tipo pues se obtiene a partir de medidas directas, en este caso la edad de cada uno de los individuos que conforman el equipo.

Además de las métricas indirectas, están las métricas relacionadas con la *estructura y comunicación de los equipos*, muy a tener en cuenta durante la gestión del proyecto. A este respecto, es interesante mencionar las métricas Hewlett-Packard para medir la *complejidad de las comunicaciones*. Dichas métricas se basan en un grafo de comunicaciones en el que se representan los miembros del equipo como nodos y las comunicaciones entre ellos como aristas, el cual sirve de base para la definición de las siguientes métricas:

- El *tamaño* es el número de individuos del equipo.
- La *densidad de comunicaciones* es el ratio entre el número de arcos y nodos, es decir, la proporción entre el tamaño del equipo y el número de comunicaciones que se producen entre ellos.
- El *número de líneas de comunicación* es la cuenta de aristas del grafo.
- El *nivel de comunicaciones* es una medida de la impureza del árbol –un indicador de cuán lejos (o cerca) está un cierto grafo de ser un árbol-. Matemáticamente, dado un grafo G , su impureza $m(G)$ viene dada por la ecuación:

$$m(G) = \frac{2 \cdot (e - n + 1)}{(n - 1) \cdot (n - 2)}$$

donde e es el número de aristas y n el número de nodos.

- El *nivel individual de comunicaciones* es el número de individuos con el que se comunica un determinado miembro del equipo.
- El *nivel medio de comunicaciones* es la media de los niveles individuales.

La Figura 3.5 muestra un ejemplo de grafo y las métricas de complejidad de las comunicaciones obtenidas a partir de la información en el mismo. En este tipo de grafos, el incremento del número de vías de comunicación, indica una mayor complejidad. En un grafo con forma de lista, que representaría a un miembro del equipo que sólo se comunica con otro miembro del equipo, el número de líneas de comunicación sería $n - 1$. Según se vayan incrementando las vías de comunicación, se puede llegar al máximo nivel de comunicación, que es aquél en el que todos los miembros del equipo se comunican con el resto de miembros, en cuyo caso las líneas de comunicación serían $n \cdot (n - 1)/2$.

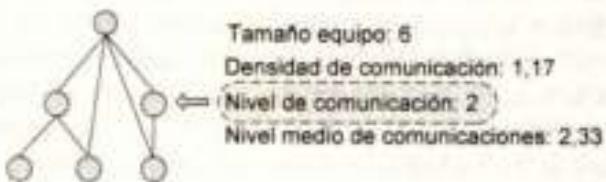


Figura 3.5: Ejemplo de métricas de complejidad de las comunicaciones

Medición de las herramientas

Como veremos en el Capítulo 12, en los desarrollos modernos se emplean gran número de herramientas. Los encargados de seleccionar las herramientas hardware y software necesarias para llevar a cabo las distintas actividades del ciclo de vida del software, han de tener en cuenta multitud de variables pero, demasiado a menudo, la única variable que se tiene en cuenta es el coste de adquisición. Sin embargo, para asegurar el éxito en la decisión deberían sopesarse factores importantes como la productividad de las herramientas o el tiempo que las personas que van a utilizarlas necesitarán para aprender a usarlas adecuadamente todas las cuales pueden (y deben) ser medidas.

Entre las métricas a tener en cuenta a la hora de comprar o seleccionar hardware, se encuentran la potencia de cálculo o la capacidad de almacenamiento, por citar sólo dos de las más habituales. En el caso del software, podríamos tener en cuenta el tipo de licencia (código abierto o propietario), su facilidad de uso, el tiempo de aprendizaje o la posibilidad de reutilizarlo en otros proyectos, entre otros.

Medición de los materiales

Los costes relacionados con las oficinas, el material fungible, los desplazamientos a las sedes de nuestros clientes, y otros como éstos, influyen en el coste final del producto y por tanto, en la productividad. Incluso la temperatura ambiente en la oficina trabajo o el tipo de oficina (colectivas vs. individuales), son métricas que podrían utilizarse para analizar la productividad en función de los diferentes entornos.

Medición de los métodos

Dentro de los recursos, es importante tener en cuenta la clasificación de los métodos usados en un proyecto, por ejemplo si se utilizaron métodos formales para los requisitos, o si se siguieron métodos estructurados o métodos orientados a objetos en el diseño. Los datos medidos a este respecto pueden ayudar a la selección de los métodos más apropiados en futuros proyectos. De hecho, es común que las organizaciones de desarrollo con altos niveles de madurez lleven a cabo lo que comúnmente se conoce como análisis *post-mortem*, es decir, un análisis de cómo ha ido el proyecto una vez éste ha finalizado.

3.7 Metodologías y estándares para la medición

Siempre se mide con la idea de mejorar, de aumentar la calidad de las entidades involucradas en los procesos de producción del software. Sin embargo, es imposible (o al menos poco práctico) medir todos los atributos de todas las entidades. Además, como se verá en el Capítulo 9 sobre la calidad en la Ingeniería del Software, programadores, gestores de proyectos y usuarios pueden tener diferentes puntos de vista de lo que significa *calidad*. Es por ello por lo que debemos determinar qué medir basándonos en metodologías y modelos de calidad bien definidos. En otras palabras, necesitamos saber qué queremos mejorar para saber qué medir. Entre los métodos más conocidos se encuentra GQM (*Goal-Question-Metric*) y el estándar de IEEE 1061-1998, que junto con otros de más reducida difusión, se explican a continuación.

3.7.1 Método Objetivo-Pregunta-Métrica (GQM)

El método Objetivo-Pregunta-Métrica, más conocido por GQM (*Goal-Question-Metric*), fue desarrollado para alcanzar los objetivos de calidad requeridos por la NASA en los años 1970. GQM ayuda a identificar, centrar, documentar y analizar un número reducido de métricas con la intención de mejorar un objetivo que puede ser tanto del producto como del proceso o sus recursos. El método se basa en los 3 niveles representados en la Figura 3.6:

- En el *nivel conceptual*, se identifica un objetivo de calidad que puede estar relacionado con la evaluación o la mejora y que será el propósito de la medición en relación a una entidad -producto, proceso o recurso- desde un punto de vista específico (gestor, desarrollador, operador, mantenimiento, etc.)
- El segundo nivel, llamado *nivel operacional*, divide el objetivo en una serie de preguntas que caracterizan a la entidad.
- Finalmente, el *nivel cuantitativo* especifica el conjunto de métricas necesarias para poder responder a las preguntas planteadas en el segundo nivel.

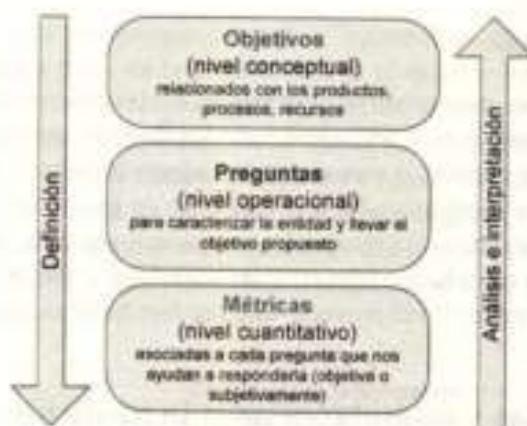


Figura 3.6: Niveles GQM

Estos 3 niveles se pueden resumir en un árbol de objetivos donde se muestran objetivos, preguntas y métricas (véase el ejemplo de la Figura 3.7). El funcionamiento del método se basa en el refinamiento progresivo de un conjunto de objetivos de negocio (*G-Goals*) que se establecen como partida. Tomando dichos objetivos como entrada y mediante el planteamiento de preguntas (*Q-Questions*), se obtienen finalmente un conjunto de métricas (*M-Metrics*) específicas que permitirán medir los objetivos enunciados.

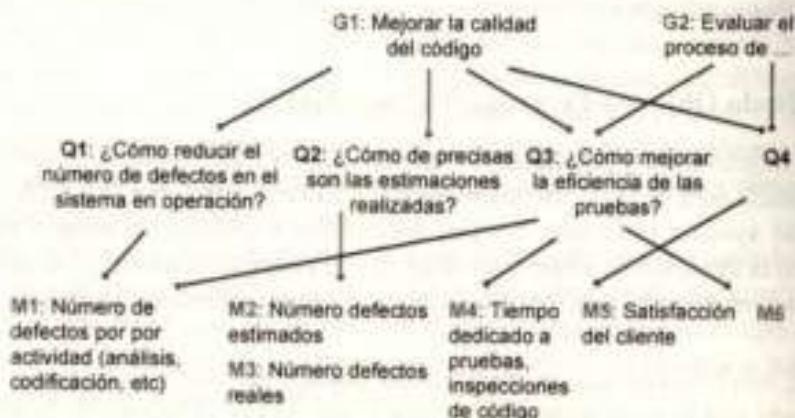


Figura 3.7: Ejemplo de árbol GQM

Más en detalle, diremos que es necesario:

1. Desarrollar un conjunto de objetivos tanto de la organización y de sus divisiones organizativas, como del proyecto. Establecer métricas que permitan evaluar la productividad y calidad deseadas para cada uno de ellos.

2. Generar preguntas que cubran los objetivos desarrollados con la mayor compleción y siempre de forma cuantificable, todo ello basándose en modelos de calidad.
3. Especificar las métricas necesarias a recabar para poder responder a las preguntas y asegurar que los procesos y productos se ajustan a los objetivos.
4. Desarrollar los mecanismos que permitan llevar a cabo las mediciones.
5. Realizar las mediciones y evaluar y analizar los datos durante la ejecución del proyecto para poder llevar a cabo acciones correctivas si fuera necesario.
6. Una vez terminado el proyecto, analizar los datos para evaluar el cumplimiento de los objetivos y recomendar mejoras basadas en las lecciones aprendidas (a esto se le suele denominar *análisis post-mortem*).

Finalmente, para que los objetivos estén bien especificados y puedan ser entendidos por todas las personas involucradas en un proyecto, éstos se suelen especificar en una plantilla con los siguientes puntos:

- *Objeto*: es la entidad que se va a estudiar. Los posibles valores a marcar en la plantilla –los más comunes– serían *proceso*, *producto*, *recurso*, *modelo* o *métrica*.
- *Propósito*: indica cuál es la motivación para medir dicho objetivo. El porqué puede ser *analizar*, *evaluar*, *predecir*, *entender*, *mejorar*, etc. Por ejemplo: medición de un cierto modelo (objeto de la medición) con objeto de estudiar posibles mejoras en el mismo (propósito).
- *Perspectiva*: informa sobre cuál es el atributo de calidad propósito del objeto, o lo que es lo mismo, con respecto a qué se toma la medida. Algunos valores posibles son la *corrección*, la *fiabilidad* o el *coste*. Por ejemplo: medición de la corrección (perspectiva de la medición) del producto (objeto de la medición).
- *Punto de vista*: hace referencia a la perspectiva desde la que se lleva a cabo el estudio. Así, es posible estudiarlo desde el punto de vista del usuario, del cliente, del gestor del proyecto o de los programadores, por poner sólo algunos ejemplos.
- *Entorno*: es el contexto en el que se ejecuta. Este entorno puede hacer referencia a factores relacionados con el contexto e incluiría factores de personal, del proyecto, de los métodos, de las herramientas, etc.

En la Tabla 3.6 se muestra, como ejemplo de todo lo anterior, una plantilla para un caso de mejora de la calidad del código.

Tabla 3.6: Ejemplo de plantilla GQM

Objeto	Calidad del software producido
Con el propósito de	Entender
Con respecto a	Facilidad de mantenimiento
Desde el punto de vista de	Personal/Equipo mantenimiento
En el contexto de	Proyecto XYZ

3.7.2 El estándar IEEE 1061-1998

El estándar IEEE 1061-1998, denominado Metodología para Métricas de Calidad del Software (*IEEE Standard for a Software Quality Metrics Methodology*), define una metodología para llevar a cabo la identificación, implementación, y evaluación de métricas para procesos y productos software válida para todas las fases del ciclo de vida independientemente del modelo utilizado (cascada, iterativo e incremental, etc.). El estándar comprende cinco pasos que deben realizarse de manera iterativa, ya que los resultados de uno de ellos pueden necesitarse en pasos subsiguientes. El conjunto de tareas a llevar a cabo es el siguiente:

1. Establecer los requisitos de calidad del software, abordando las siguientes tareas:
 - (a) Elaborar un listado con los posibles requisitos de calidad, utilizando para ello toda la información disponible: estándares, requisitos de los contratos o compras (tales como garantías o fechas de entrega), etc.
 - (b) Evaluando la importancia de cada requisito y eliminando posibles contradicciones, identificar la lista de requisitos de calidad a partir del listado anterior.
 - (c) Cuantificar los factores de calidad. A cada factor se le debe asignar al menos una métrica con sus respectivos valores. Por ejemplo, si un requisito fundamental es la *fiabilidad* en un sistema web, una métrica podría ser la «Disponibilidad diaria» con un valor del 95%, o lo que es lo mismo, disponibilidad de 1.368 minutos sobre los 1.440 minutos que componen las 24 horas de un día.
2. Identificar las métricas de calidad del software mediante la aplicación de un marco de métricas de calidad, la realización de un análisis de costes y beneficios, y establecer las garantías necesarias para implantar dichas métricas.
3. Implementar las métricas de calidad. En este paso se definen los procedimientos de medición, se hace un prototipo del proceso de medición y finalmente se lleva a cabo la medición en sí.
4. Analizar los resultados. Consiste en interpretarlos, analizando si los requerimientos de calidad se ajustan a los requisitos definidos en la especificación.
5. Evaluar los resultados aplicando criterios y metodologías de evaluación.

Nótese que el estándar no prescribe ninguna métrica específica, limitándose a facilitar y promover el uso de métricas dentro de las organizaciones en el contexto de proyectos con el objetivo final de mejorar la calidad del software producido.

3.7.3 PSM y el estándar ISO/IEC 15939

Tanto el estándar ISO/IEC 15939 (Proceso de Medición del Software) como el PSM (*Practical Software Measurement*, Medición Práctica del Software) en el cual está basado, establecen las actividades y tareas necesarias del proceso de medición, ayudando a identificar, definir, seleccionar, aplicar y mejorar la medición de software dentro de un proyecto general o de la estructura de medición de una empresa. El estándar se compone de dos aspectos, (i) el modelo de información de la medición y (ii) el proceso de medición propiamente dicho, de forma que pueda ser integrado en procesos generales.

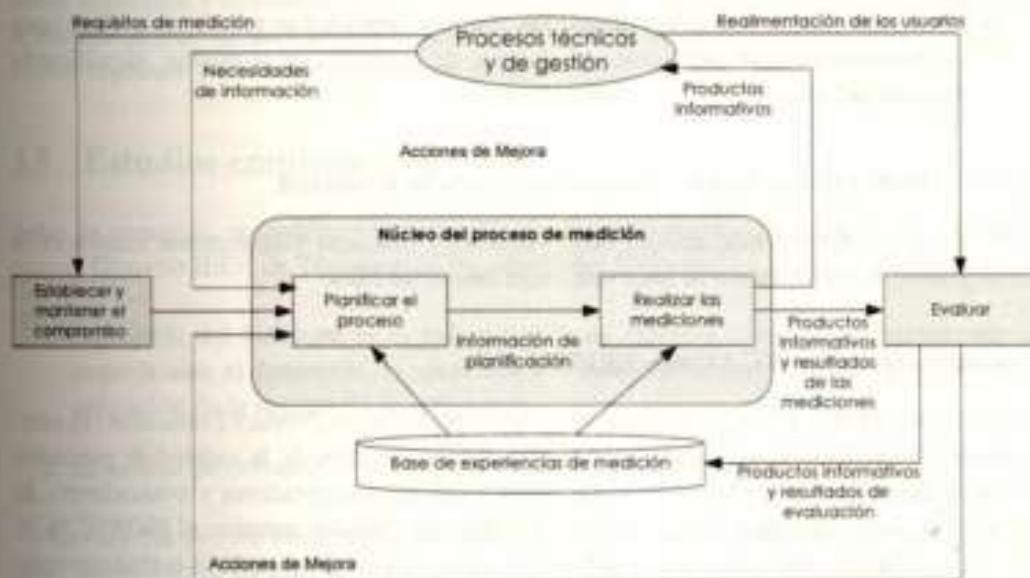


Figura 3-8: Actividades ISO/IEC 15939

El modelo de información de la medición define los términos de uso común relativos a la medición del software y la relación entre las necesidades de información, los tipos de medidas o métricas (por ejemplo los indicadores necesarios, métricas directas e indirectas) y las entidades a medir (en este estándar son procesos, productos, proyectos y recursos). Como muestra la Figura 3.8, el proceso de medición se compone de 4 actividades fundamentales:

- **Establecer y mantener el compromiso de medición.** Esta actividad implica el establecimiento de un compromiso empresarial para realizar mediciones, compromiso que será necesario establecer al más alto nivel y con pocas o ninguna dependencia

de personas concretas. Esto es así porque para recuperar la inversión realizada será necesario mantener este compromiso durante un cierto tiempo durante el que muchas cosas podrían cambiar en la organización.

- **Planificar el proceso de medición.** Esta actividad consiste básicamente en la selección de las medidas de acuerdo con los objetivos y características de la organización, definir los procedimientos de cómo y cuándo se van a realizar las medidas seleccionadas, quién las va a llevar a cabo, la frecuencia de medición de las mismas y cómo se analizarán.
- **Ejecutar el proceso de medición.** En esta actividad se realizan y almacenan las mediciones realizadas, para posteriormente analizarlas. Con los datos obtenidos se elaboran informaciones que puedan ser utilizadas en procesos de decisión.
- **Evaluación de las medidas obtenidas.** En esta actividad se pretenden evaluar tanto las mediciones realizadas como el proceso de medición en sí mismo, identificando mejoras potenciales.

3.7.4 Otras metodologías y estándares para la medición

Además de los dos citados, existen otros modelos y prácticas comúnmente citados en la bibliografía. A continuación se hace una breve reseña de ellos.

Los estándares ISO/IEC 14598 e ISO/IEC 9126

El estándar ISO/IEC 14598 (*Information Technology-Software Product Evaluation*) se compone de seis volúmenes dedicados a la medición y evaluación de la calidad de productos software desde distintos puntos de vista (desarrolladores, compradores y evaluadores). El estándar incluye actividades de proceso y se basa en un estándar anterior, el ISO/IEC 9126, que se estudia con mayor detalle en la Sección 9.4.3. Actualmente, ambos estándares comparten la misma terminología y están en proceso de convergencia hacia un único estándar que los englobe.

Vocabulario internacional de metrología – VIM

En su última edición, el estándar ISO/IEC Guide 99:2007 (más comúnmente conocido como VIM) recopila un conjunto de definiciones y términos relacionados con la ciencia de la medición (metrología) en general y no sólo de la Ingeniería del Software. Además de esto, incluye diagramas conceptuales que muestran de manera más evidente las relaciones entre términos, ejemplos y todo tipo de información complementaria. El objetivo último es que sirva como referencia para otros estándares y metodologías relacionadas con la medición, armonizando así la dispersa nomenclatura actual.

AMI (*Aplicación de métricas en la industria*)

El método AMI, publicado por Kuntzmann-Combelle y sus colaboradores en 1996, combina el modelo de madurez CMM con el método GQM con objeto de crear un marco de trabajo para la mejora de procesos. Se trata de un método orientado a objetivos de claro enfoque iterativo y cuantitativo, que involucra a todos los miembros de la organización integrando a su vez a los gestores, de quienes necesita un compromiso firme. Cubre todo el ciclo de mejora del proceso, y utiliza CMM (pero también otras normas de calidad como Bootstrap, SPICE o ISO 9001) para identificar posibles áreas débiles en el proceso de desarrollo. Esta información, junto con información sobre el entorno de la empresa y los objetivos específicos se utiliza para definir objetivos del proceso de software. Estos objetivos se evalúan, se refinan en subobjetivos, de los cuales finalmente se obtienen métricas. El método establece además un plan de medición para recolectar datos, que serán posteriormente analizados y contrastados con el objetivo original. En función de los datos obtenidos se elabora un plan de acción para mejorar el proceso de desarrollo, para lo cual se definen nuevos objetivos y se vuelve a repetir así el ciclo.

3.8 Estudios empíricos

Antes de proseguir, recordemos la definición del término *Ingeniería del Software* que propone el Glosario IEEE de Términos de Ingeniería del Software:

1. Ingeniería del Software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, la operación y el mantenimiento del software. Esto es, la aplicación de la ingeniería al software.
2. El estudio de enfoques como los mencionados en el punto (1).

La mayoría de este libro cubre aspectos relacionados con la primera parte de la definición. Sin embargo, en esta sección daremos una visión general de la segunda: el estudio de la Ingeniería del Software en sí misma. Mostraremos cómo la evaluación de nuevas técnicas, procesos, herramientas o métricas de Ingeniería del Software se puede llevar a cabo a través de estudios empíricos, algo ampliamente utilizado en otras disciplinas como la Química, la Biología o la Psicología.

Los estudios empíricos nos ayudan a demostrar la mejoría introducida por el uso de una nueva técnica o herramienta y a responder preguntas del estilo Realmente „funciona mejor Java que C++ en entornos de robótica? „Con qué método de especificación de requisitos se genera un código con menor cantidad de errores en sistemas de tiempo real, con los casos de uso o con los métodos formales? Lo que buscamos con los métodos empíricos son respuestas cuantitativas a este tipo de preguntas.

La experimentación es una disciplina relativamente joven dentro de la Ingeniería del Software. Por tanto, no es extraño ver que el conocimiento en la Ingeniería del Software

se adquiere todavía, desde el punto de vista epistemológico, bien por influencia de una cierta autoridad o persona prestigiosa (*«si tal persona lo dice, será la mejor forma»*), o bien por inercia (*«si IBM lo hace así, es seguro que funciona»*) o bien por parecer que se utiliza lo más actual o lo que está de moda (*«si todo el mundo ha desecharado el enfoque estructurado y adoptado el modelo de orientación a objetos, nosotros también lo haremos aunque no entendamos muy bien las razones para hacerlo»*). Sin embargo, esta situación no ayuda a la consolidación de la Ingeniería del Software como disciplina de ingeniería de pleno derecho. En la Ingeniería del Software, como en otras disciplinas, las nuevas teorías deberían venir respaldadas por datos experimentales que las sustenten y confirmen su validez. Todo ello, por supuesto, asumiendo las características y limitaciones propias de la Ingeniería del Software con respecto a disciplinas como la Física o la Biología donde la experimentación tiene siglos de historia.

Poco a poco, pero inexorablemente, la importancia de la experimentación dentro de la Ingeniería del Software está ganando terreno. Ya se han empezado a publicar libros al respecto, y comienzan a celebrarse cada vez con mayor frecuencia conferencias dedicadas. Este hecho queda patente en varios estudios:

- En un estudio llevado a cabo en 1997, en el que se analizaron cerca de 600 artículos científicos del área de la informática, se demostró que sólo el 10% de los mismos incluía algún tipo de experimentación controlada. Aproximadamente el 30% de los artículos no incluía experimentación ninguna.
- Tichy y sus colaboradores realizaron un estudio similar en 1995 sobre 400 artículos. Sus conclusiones indican que aproximadamente un 40% de los artículos no incluye experimentación, lo cual es muy elevado en comparación con otras disciplinas de ingeniería en las cuales este número desciende hasta el 10-15%.

Además, y hasta la fecha, se han formulado muy pocas leyes en la Ingeniería del Software, siendo las más conocidas las leyes de la evolución del software de Lehman (ver Sección 8.5.2 para una descripción más completa). Recientemente se han planteado diversas leyes, teorías e hipótesis para tratar de acercar la Ingeniería del Software al método científico. Según dicho método la adquisición de conocimiento se basa en la obtención de información mediante observación, a lo cual sigue la propuesta de teorías e hipótesis, la evaluación de las hipótesis y si es posible, la replicación de estudios anteriores. No obstante, otros métodos de experimentación son igualmente aplicables a la Ingeniería del Software, como el método ingenieril, el método empírico y el método analítico.

Para llevar a cabo la evaluación serán necesarios instrumentos que nos permitan adquirir datos, identificar factores clave de estudio y lógicamente obtener resultados. Estos instrumentos son los siguientes:

- **Encuestas:** son estudios retrospectivos cuya intención es documentar relaciones y resultados, siendo una de las herramientas más útiles para recabar un buen número de datos que posteriormente serán analizados.

- **Casos de estudio:** son empleados para identificar y documentar factores clave que pueden afectar los resultados de una actividad.
- **Experimentos formales:** se emplean para, de forma controlada y rigurosa, investigar aquellos factores que afectan a las actividades a realizar.

Los tres instrumentos enumerados, encuestas, casos de estudio y experimentos formales, se utilizan tanto para estudios cualitativos como cuantitativos. Los estudios cualitativos buscan la interpretación de un fenómeno en su entorno natural, recabando información entre las personas involucradas. Los estudios cuantitativos, por su parte, buscan medir la influencia de una variable en un fenómeno, es decir la relación causa-efecto entre ambos (por ejemplo la relación entre la profundidad en la jerarquía de clases en un diseño orientado a objetos y el número de errores en el código de una clase).

En la Ingeniería del Software, los tres tipos de evaluación empírica son igualmente valiosos ya que las encuestas cualitativas sirven como base para la formulación de hipótesis, los experimentos formales confirman o rechazan las hipótesis y los casos de estudio sirven como contraste para determinar si las hipótesis pueden o no generalizarse. Todas estas técnicas se explican brevemente a continuación con el objeto de tratar de identificar cuál de ellas resultará más apropiada en cada momento, pero tenga en cuenta que la elección de la técnica apropiada es siempre dependiente no sólo del propósito, sino también de los posibles participantes y del tipo de análisis de resultados a realizar.

3.8.1 Encuestas

Las encuestas se emplean para recabar datos que están en la memoria de los entrevistados. Por ejemplo, tras introducirse un nuevo método o herramienta en una organización, puede evaluarse su efectividad a través de un cuestionario que deberán cumplimentar los empleados que la utilizan. Las encuestas forman parte de lo que se conoce como investigación a gran escala (*research in-the-large*), pues sirven para recopilar un gran número de datos acerca de diferentes personas y proyectos. Este hecho permite que las conclusiones puedan generalizarse, siempre y cuando la selección de los entrevistados haya sido realmente aleatoria y significativa.

Una vez establecido el objeto (u objetos) de interés, y dependiendo tanto de la población seleccionada como del porcentaje de respuestas esperadas, las entrevistas pueden diseñarse para ser realizadas cara a cara, por correo o a través de internet (por web o email). Dependiendo del método utilizado existen tres maneras de llevar a cabo las entrevistas:

- Entrevistas *estructuradas*, en las que las preguntas corren a cargo del entrevistador. Dichas preguntas están fijadas de antemano, son las mismas para todos los entrevistados y no se modifican durante la entrevista.
- Entrevistas *no estructuradas*, donde el entrevistado puede ser fuente tanto de respuestas como de preguntas con el objetivo de obtener de él la mayor información posible.

- Entrevistas *semi-estructuradas*, formadas por preguntas abiertas, lo que facilita que el entrevistado pueda ofrecer información no prevista por el entrevistador.

Aunque generalmente las encuestas son cualitativas, el tipo de estudio (cuantitativo o cualitativo) depende de cómo se haya diseñado la encuesta. Además, dependiendo del tipo de encuesta, el análisis de los datos extraídos de las mismas puede utilizarse tanto para generar teorías como para confirmarlas. La generación de teorías se realiza extrayendo aquellas notas de las entrevistas que permitan afirmaciones avaladas por los datos, es decir, son estudios descriptivos o explicativos. Por otro lado, las entrevistas ayudan a la formulación de hipótesis cuando se emplean como estudios exploratorios que ayudan a buscar futuras líneas de investigación (por ejemplo mediante entrevistas semi-estructuradas).

Existen guías tanto para la correcta planificación y realización de los cuestionarios como para su análisis. En éste último caso, pueden utilizarse técnicas estadísticas si lo que se desea evaluar son los resultados de cuestionarios cuantitativos. Para datos cualitativos y generación de teorías pueden utilizarse por ejemplo el *método de comparación constante*, que consiste en añadir códigos a los resultados, agrupar la información según los códigos y sintetizar las conclusiones.

Un ejemplo clásico de estudio mediante encuesta, fue el realizado por Beck y Perkins en 1983. En dicho estudio se analizaron las prácticas de la Ingeniería del Software en la industria y se buscaron correlaciones entre dichas prácticas y determinados problemas que aparecían en las instalaciones de usuario de los sistemas una vez en producción.

3.8.2 Casos de estudio

Los **casos de estudio** constituyen una técnica de investigación que se basa en la observación de entornos reales para contrastar o evaluar datos, encontrar relaciones y descubrir tendencias. En la Ingeniería del Software se usan mucho para comparar métodos o herramientas, pudiendo ser tanto cuantitativos como cualitativos. Así, por ejemplo, si se desea introducir una nueva herramienta en una organización pero no se está seguro de si el incremento en la productividad merecerá la pena el esfuerzo en inversión, podría diseñarse un caso de estudio como parte de un proyecto piloto cuyo único objeto fuese comparar la productividad de la nueva herramienta con la de la herramienta actual.

Los casos de estudio se clasifican en lo que se ha denominado «investigación en lo típico» (*research in the typical*) ya que reflejan la información de un proyecto representativo en lugar de toda la casuística con la que nos podríamos encontrar. La diferencia entre los casos de estudio y los experimentos formales es que en los experimentos las variables son manipuladas, mientras que en los casos de estudio simplemente se analizan dentro de su contexto. Los casos de estudio tienen la ventaja de que son más fáciles de diseñar que los experimentos formales, aunque son difíciles de replicar pues se realizan generalmente en entornos industriales concretos. Además, al no tener los investigadores ningún control sobre las variables, el establecimiento de relaciones causales entre ellas es más difícil y por tanto la generalización de las conclusiones más complicada.

3.8.3 Experimentación formal

La **experimentación formal** busca medir relaciones causales con la mayor exactitud posible, o lo que es lo mismo, establecer el grado de influencia de unas variables en otras. Para ello es necesario configurar entornos en los que se tenga un alto nivel de control sobre las variables para, modificándolas, medir sus efectos.

Fenton y Pfleeger clasifican la experimentación formal como investigación a pequeña escala (*research in the small*) debido a la dificultad de controlar todos los posibles factores. Se suelen realizar en entornos académicos (por ejemplo con estudiantes en un laboratorio) o en organizaciones con un grupo reducido de trabajadores, precisamente por la dificultad mencionada, lo que ha hecho que a estos estudios también se los conozca como experimentos *in vitro*. Los experimentos formales son por tanto más difíciles de diseñar y más costosos que las encuestas o los casos de estudio.

La experimentación formal en la Ingeniería del Software, al igual que en otras disciplinas como la Física o la Medicina, ayuda a confirmar teorías, explorar relaciones causa-efecto entre variables, corroborar o rechazar creencias sobre métodos, procesos y tecnologías, evaluar métricas, etc. Dado que el factor humano es especialmente importante dentro la Ingeniería del Software, los tipos de experimentos muchas veces se asemejan a los realizados en las ciencias sociales.

Un conocido ejemplo de experimentación formal en la Ingeniería del Software es el que Scanlan llevó a cabo para descubrir si sus estudiantes tenían diferentes niveles de comprensión de la lógica de un programa en función de la herramienta de representación utilizada. Su estudio demostró que los estudiantes preferían los diagramas de flujo, pues les ayudaban a entender mejor los programas que el pseudocódigo.

Ingeniería del Software basada en la evidencia

La Ingeniería del Software basada en la evidencia es un novedoso modelo de experimentación que intenta acercarse a la forma de proceder de otras áreas, particularmente de la Medicina, a través de la determinación de cuáles son las cosas que funcionan, en qué casos y cuándo. El modelo se apoya en la recolección y análisis sistemáticos de todos los datos empíricos disponibles sobre un determinado fenómeno. El objetivo último es conocer dicho fenómeno con mayor profundidad y perspectiva de los que los estudios individuales podrían proporcionar, ya que éstos a menudo están sesgados por factores como el contexto en que se llevan a cabo o las personas que participan.

El elemento central de la Ingeniería del Software basada en la evidencia es la revisión sistemática de la literatura, una forma de estudio muy utilizada en Medicina. Según este método, los resultados de estudios basados en un único ensayo no pueden considerarse ni lo suficientemente seguros ni lo suficientemente fiables como para ser generalizados. Este principio se aplica a la Ingeniería del Software, en palabras de Kitchenham, mediante «la provisión de medios a través de los cuales se integren las mejores evidencias de la investigación, la experiencia práctica y los valores humanos en el proceso de toma de decisiones que se lleva a cabo durante el desarrollo y mantenimiento de software».

La experimentación formal sigue un proceso definido a lo largo de 5 pasos, el cual se muestra detallado a continuación y de manera resumida en la Figura 3.9:

1. Definición: en primer lugar se debe declarar la intención del experimento, es decir, cuáles son sus objetivos, en qué contexto se llevará a cabo, cuál es su propósito, etc. Es frecuente el uso del método GQM (ver Sección 3.7.1) para sistematizar, ordenar y considerar todos los aspectos del experimento.
2. Planificación: en esta fase se diseña el experimento mediante el enunciado de la hipótesis, la selección de variables de estudio (las cuales determinarán los tipos de análisis estadísticos que se pueden aplicar), la selección de los sujetos y finalmente el diseño del experimento propiamente dicho. Es especialmente relevante la elección de la instrumentación y la determinación del método de evaluación.
3. Operación: es la fase de preparación del experimento, que consiste en realizar la selección de participantes, formularios y guías para que los participantes entiendan el proceso, entrenar a los participantes si es necesario, y en definitiva realizar todas las acciones necesarias para poner en marcha el experimento. En esta fase suele realizarse un estudio piloto para encontrar errores en el diseño y subsanarlos para que el experimento pueda llevarse a cabo.

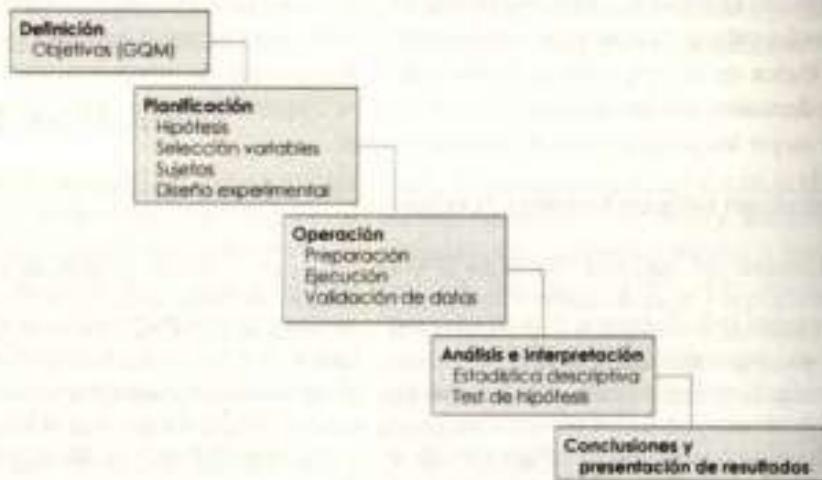


Figura 3.9: Proceso que sigue la experimentación formal

4. Interpretación: a la hora de analizar e interpretar los resultados, el tipo de análisis está condicionado por la hipótesis del experimento, el tipo de variables seleccionadas según su escala y el diseño experimental. El primer paso es caracterizar los datos utilizando estadística descriptiva, para posteriormente realizar el *test de hipótesis* con el fin de confirmar o rechazar las hipótesis enunciadas.

5. Conclusiones y presentación de resultados: al final del experimento se escribe un informe comentado del proceso seguido, donde se debe enfatizar por qué es relevante la hipótesis del estudio y cómo se han solventado o minimizado las amenazas a la validez. Dado que a partir de las conclusiones del experimento otros podrían querer poner en marcha nuevos estudios para confirmar o rebatir los hallazgos, es necesario incluir toda la información que permita crear posibles réplicas del experimento.

3.9 Resumen

En este capítulo hemos estudiado las bases de la medición en la Ingeniería del Software, elementos fundamentales que son utilizados en la mayoría de los capítulos para describir métricas específicas del tema tratado en cada uno. Hemos visto pues la medición desde un punto de vista global de la disciplina, donde las entidades a medir son los productos, los procesos y los recursos. Para cada entidad hemos distinguido entre métricas directas (las que podemos medir directamente) e indirectas (aquellas que dependen de factores externos).



Figura 3.10: Principales conceptos tratados en el capítulo.

Una parte importante del capítulo se ha dedicado a proporcionar una visión general de los estándares, métodos y metodologías relacionados con la medición, modelos que ayudan a determinar qué medir de entre las múltiples magnitudes que podrían ser objeto de medida.

Se ha introducido además la experimentación, un área joven dentro de la Ingeniería del Software cuya importancia está creciendo rápidamente como mecanismo de justificación y evaluación de muchas de las prácticas de la disciplina. Y por qué no decirlo, también como forma de rechazar otras prácticas que no cuentan con sustento experimental suficiente. No nos cabe duda de que la medición y la experimentación marcarán el camino por el que la Ingeniería del Software transitará en los próximos años.

3.10 Notas bibliográficas

Software Metrics: A Rigorous and Practical Approach (Fenton y Pfleeger, 1998) es la obra fundamental sobre métricas, habiendo influido a la mayoría de los libros posteriores que han tratado la medición, incluido el presente. Otra obra esencial, esta vez sobre experimentación, es *A Framework for Software Measurement* (Zuse, 1998), donde se realiza un estudio en gran profundidad y detalle sobre numerosas métricas de software. Los interesados en ampliar sus conocimientos sobre la medición en este campo deberían consultarla.

En español existe un trabajo muy completo sobre la medición y experimentación en la Ingeniería del Software titulado «*Medición para la gestión en la Ingeniería del Software*» (Dolado y Fernández, 2000). Muchos de los contenidos del capítulo se tratan en dicho libro con mayor amplitud, por lo que resulta un excelente complemento.

Sobre evaluación y validación de métricas, el artículo «*Towards a framework for software measurement validation*» (Kitchenham, Pfleeger y Fenton, 1995) resulta aún hoy día la referencia más citada. De hecho, el apartado que hemos dedicado a la evaluación sigue su estela. Otras referencias interesantes son *Software Engineering Measurement* (Munson, 2002) que toca aspectos de la teoría de la medición, y las propiedades de validación de métricas publicadas por Weyuker (1988).

A pesar del tiempo transcurrido desde su publicación, y de estar pensadas para la programación estructurada, las métricas de complejidad de McCabe (1976) y Halstead (1977) se encuentran implementadas en multitud de herramientas y por tanto, ampliamente disponibles en la literatura.

Para profundizar en el método objetivo-pregunta-métrica (GQM), la referencia principal es Basili y Rombach (1988), aunque existe también un libro muy interesante titulado *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development* (van Solingen y Berghout, 1999).

Sobre medición y experimentación existen muchas referencias. Resaltamos los artículos introductorios que enfatizan la importancia de los estudios empíricos, en particular el de Glass (1999). El artículo de Zelkowitz y Wallace (1997) incluye el estudio mencionado en el apartado de experimentación –aquel en que se reporta un estudio sobre 600 artículos científicos dentro del área de la informática-. Como ya dijimos, los resultados obtenidos mostraron que aproximadamente el 30% de los artículos no incluían experimentación aunque ésta era necesaria y que sólo el 10% de los artículos incluían algún tipo de experimentación controlada. Tichy *et al.* (Tichy, Lukowicz, Prechelt y Heinz, 1995) llegan a conclusiones parecidas con un estudio sobre 400 artículos.

En relación a la experimentación en la Ingeniería del Software, Basili y sus colaboradores (Basili, Selby y Hutchens, 1986) fueron los primeros en iniciar el área y aunque no es un artículo introductorio fácil de leer, es el origen y cita referenciada en los libros sobre experimentación en la Ingeniería del Software. Para una estudio más detallado se referencia al lector a la obra ya mencionada de Fenton y Pfleeger (1998). Otro libro introductorio que trata la experimentación es el de Wholin *et al.* (2000).

Relacionados con experimentación, los trabajos de B. Kitchenham son especialmente relevantes, siendo de destacar una serie de artículos sobre experimentación donde se detallan, entre otros, guías para la selección de las técnicas más apropiadas en la evaluación empírica, *Evaluating Software Engineering Methods and Tools* (publicado en 6 volúmenes).

Finalmente, varios autores comúnmente nombrados en el área han publicado una guía para llevar a cabo la experimentación en la Ingeniería del Software evitando los típicos errores (Kitchenham, Pfleeger, Pickard, P.W.Jones, Hoaglin, Emam y Rosenberg, 2002).

3.11 Cuestiones de autoevaluación

3.1 En el texto del capítulo se empleó el concepto de «condición de representación», como algo que han de satisfacer las medidas pero ¿qué es la condición de representación?

R. Es una regla que establece que las relaciones empíricas deben preservar las relaciones numéricas y viceversa. La función Programadores de un proyecto, definida como «número de personas que trabajan como programadores en el proyecto», cumple esta condición pues para todo par de proyectos P_1 y P_2 , si en P_1 hay menos programadores contratados participando en el proyecto, entonces se verifica que $\text{Programadores}(P_1) < \text{Programadores}(P_2)$.

3.2 Razone sobre la afirmación: «La escala intervalo no tiene un valor inicial o cero absoluto».

R. Es una afirmación cierta, pues se trata de un tipo de escala donde la distancia entre intervalos es fija aunque no existe un valor inicial de referencia. El final de una actividad en Ingeniería del Software, por ejemplo, es un atributo que se mide según una escala intervalo. Así, podemos decir que el proyecto P_1 terminó 100 días después que el proyecto P_2 y 50 días después que P_3 pues aunque P_1 acumuló más retraso comparado con P_2 que comparado con P_3 , el tipo de escala hace que carezca de sentido decir algo como «Comparado con P_2 , P_1 terminó el doble de tarde que comparado con P_3 ».

3.3 ¿Cuál es la diferencia entre medida y métrica?

R. Ninguna realmente, si hacemos caso a lo dicho en el capítulo. Se ha comentado que algunos autores dan un significado específico y distinto a cada uno de estos términos, aunque mayoritariamente medida y métrica se utilizan indistintamente.

3.4 Sobre qué entidades fundamentales se realizan mediciones en Ingeniería del Software?

R. Los procesos, los productos y los recursos son las entidades fundamentales cuyos atributos se definen y sobre los que finalmente se llevan cabo las mediciones.

3.5 Comente la siguiente afirmación: «La cuenta de las líneas de código de un programa de computadora es una actividad tan trivial que el resultado de la misma debería ser idéntico, lo lleve a cabo quien lo lleve a cabo».

R. Falso. Como hemos visto en el capítulo, esto no es así en absoluto. Mientras una persona podría considerar las líneas en blanco, o los comentarios, como líneas válidas, otra podría no tenerlos en cuenta. Igualmente, las declaraciones de variables y otras partes del código no «directamente transformables en instrucciones al microprocesador» podrían ser discutibles.

integrantes de la cuenta de instrucciones. De hecho, el problema es tan patente que existen multitud de variantes de la métrica tales como el número de líneas de código sin comentarios, el número de líneas de código con comentarios, o el número de instrucciones.

3.6 ¿Qué es la complejidad esencial?

R. *Se trata de una métrica de calidad del software definida por McCabe que proporciona un indicador de la «pureza» del código. Al igual que la complejidad ciclomática, se calcula utilizando un grafo de representación del código. No obstante, se trata de un gráfico simplificado donde se eliminan las estructuras básicas de la programación estructurada (secuencia, selección e iteración).*

3.7 Indique brevemente en qué consiste el método GQM.

R. *El método GQM (Goal-Question-Metric) es una metodología para la medición cuyo objeto fundamental es ayudar a identificar, centrar, documentar y analizar un número reducido de métricas con la intención de mejorar progresivamente un conjunto de objetivos de negocio (G-Goals) que se establecen al inicio.*

3.8 De acuerdo con lo estudiado en el capítulo, confirme o desmienta la siguiente afirmación: «La importancia de la experimentación dentro la Ingeniería del Software ha sido siempre, y seguirá siendo, algo marginal».

R. *No es verdad, la importancia de la experimentación dentro nuestra disciplina gana terreno rápidamente. De hecho ya hay toda una comunidad científica y de interés que publica libros sobre el tema, celebra conferencias dedicadas, etc. Todo ello teniendo en cuenta que una de las definiciones más aceptadas del término Ingeniería del Software incluye la experimentación como algo esencial de la disciplina.*

3.9 Dentro del campo de la Ingeniería del Software ¿puede indicar cuáles son los tres instrumentos que existen para la evaluación empírica?

R. *Las encuestas, los casos de estudio y los experimentos formales.*

3.10 ¿Por qué se dice que los casos de estudio son «investigación en lo típico»?

R. *Los casos de uso se clasifican como «investigación en lo típico» (research in the typical) porque reflejan preferentemente la información del proyecto más habitual, representativo, típico hasta cierto modo, en lugar de intentar abarcar todas las posibles variantes con que nos podríamos llegar a encontrar.*

3.12 Ejercicios y actividades propuestas

3.12.1 Ejercicios resueltos

3.1 Suponga que se ha decidido implantar un plan de medición en su compañía. Explique brevemente qué información debería contener un plan de métricas.

Solución propuesta: Analizando aspectos comunes de las metodologías de métricas, podríamos empezar con los objetivos de la organización, para posteriormente responder a las preguntas ¿porqué?, ¿qué?, ¿quién?, ¿cómo?, ¿dónde? ¿cuándo?. Por ejemplo:

- ¿Por qué las medición ayuda a alcanzar esos objetivos?
- ¿Qué medidas debemos recabar? Aquí habremos de especificar su definición, escala, a quién van dirigidas y cómo se van a analizar.
- ¿Quién se encarga de realizar las mediciones de las métricas seleccionadas?
- ¿Cada cuánto tiempo medir?
- ¿Dónde y cómo se almacenan los datos medidos para futuros usos?

3.2 Identifique la escala correcta para cada una de las siguientes medidas:

- *LoC*
- Lenguaje de programación utilizado, siendo las posibles opciones: C, C++, Java y Ruby.
- Complejidad ciclomática de McCabe.
- Nivel o profundidad de anidamiento.

Solución propuesta

- *LoC* - escala absoluta: la diferencia entre valores es constante y existe un valor de referencia o cero absoluto que es el valor 0 líneas de código. Además, el número de líneas en un código es algo que sólo se puede medir contando.
- Lenguaje de programación utilizado - escala nominal: en la clasificación de lenguajes de programación es obvio que existen diferentes categorías, si bien en este caso no existe una relación de orden «es menor que» entre ellas que pudiera determinar una escala ordinal. Se trata por tanto de una escala de categorías nominal.
- Complejidad ciclomática de McCabe - escala intervalo: dado que el valor obtenido para la complejidad ciclomática se calcula como uno más el número de decisiones, la diferencia entre 5 y 6 es la misma que entre 9 y 10. Esto determina que la escala será bien de ratio o bien de intervalo, si bien la inexistencia de un cero absoluto descarta la escala de ratio pues la complejidad ciclomática no puede ser cero.
- Nivel o profundidad de anidamiento - escala ordinal: es posible clasificar los niveles de anidamiento en nivel 1, nivel 2, etc. cada uno de los cuales está relacionado con el anterior y el posterior mediante una relación «es menor que». Así, diremos que una sentencia en un nivel 3 de anidamiento está «menos anidada» que una en un nivel 4 de anidamiento.

3.3 Si en un cierto proyecto un equipo de desarrollo crea 100.000 líneas de código nuevas y reutiliza 20.000 tomadas de bibliotecas de funciones. ¿Podría usted decir cuál será su porcentaje de reutilización?

Solución propuesta: El cálculo del porcentaje de reutilización es muy sencillo en este caso, pues será $20.000 / (20.000 + 100.000)$, es decir un 16.7% del código total.

- 3.4 La siguiente función retorna el máximo valor entero de los almacenados en un array de números enteros positivos:

```
public int maximo(int[] lista, int longLista) {
    int i=0, max=-1;
    while (i<longLista) {
        if (lista[i]>max)
            max=lista[i];
        i++;
    }
    return max;
}
```

Calcule las siguientes métricas de Halstead: longitud, vocabulario, volumen, dificultad y esfuerzo. Calcule también el número de sentencias y su complejidad ciclomática.

Solución propuesta: La siguiente tabla muestra los cálculos correctos:

Métrica	Resultado
Número de sentencias	7
Complejidad ciclomática	3
Halstead - longitud de Halstead	30
Halstead - vocabulario	15
Halstead - volumen	117
Halstead - dificultad	14
Halstead - esfuerzo	1640

- 3.5 Cierta organización ha decidido utilizar metodologías ágiles en sus proyectos. ¿Qué posibilidades de evaluación existen para medir si la productividad de los proyectos realizados con la nueva metodología mejora o no la actual productividad?

Solución propuesta: Tenemos a priori 3 posibilidades: utilizar encuestas, casos de estudio o experimentos formales. No podemos realizar encuestas al no ajustarse la situación a la de un estudio retrospectivo en el que ya disponemos de toda la información. Si podemos, no obstante, realizar casos de estudio para analizar la ejecución de nuevos proyectos y compare sus resultados con el de otros proyectos base.

3.12.2 Actividades propuestas

- 3.1 Aprenda más sobre algunos de los ejemplos clásicos de estudios experimentales mencionados en el capítulo, como el realizado por Beck y Perkins (1983) para analizar la práctica de la

ingeniería del Software en la industria, o el de Scanlan sobre la utilidad de los diagramas de flujo en la clase de estructuras de datos (Scanlan, 1989).

- 3.2 A partir del código de la siguiente función, que busca un elemento dado dentro de una lista ordenada de números enteros, calcule las siguientes métricas: número de sentencias, complejidad ciclomática, longitud, vocabulario, volumen, dificultad y esfuerzo:

```
public int busquedaBinaria(int[] lista, int longLista, int buscado) {
    int primero=medio=0;
    int ultimo = longLista - 1;
    boolean encontrado = false;
    while (primero <= ultimo && !encontrado) {
        medio = (primero + ultimo) /2;
        if (lista[medio] == buscado)
            encontrado = true;
        else {
            if(lista[medio] > buscado)
                ultimo = medio -1;
            else
                primero = medio + 1;
        }
    }
    if(encontrado)
        return medio;
    else
        return(-1);
}
```

- 3.3 Utilice un programa de métricas, como por ejemplo *javaNCSS*, para medir un proyecto de código abierto, analizando algún módulo, clase o función cuyas métricas se salgan de los valores medios o recomendados. Compruebe lo que tenga de particular dicho módulo. Si se dispone de varias versiones del mismo programa, sería interesante comprobar si los módulos, clases o funciones con valores «extraños» en las métricas han sido modificados en versiones posteriores, lo que probaría de algún modo su propensión al error.

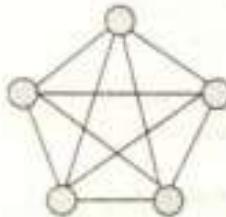
- 3.4 A partir del siguiente fragmento de programa, calcule los valores de $n1$, $n2$, $N1$ y $N2$ según la Ciencia del Software de Halstead:

```
while (i < 10){
    if (i > j){
        x += j;
        j++;
    }
    y = i + j*2;
    System.out.print(i,j,x);
}
```

- 3.5 Uno de los libros más ampliamente reconocidos sobre gestión de proyectos de Ingeniería del Software, «El mítico hombre mes» (Brooks, 1995), presenta un conjunto de hechos sobre

las tareas de gestión y lanza opiniones provocadoras al respecto. En su clásico libro, Bennis argumenta en particular que al añadir recursos humanos a un proyecto que ya se encuentra en una situación de incumplimiento de plazos, no se conseguirá más que hacer que dicho proyecto finalice aún más tarde. Discuta esta afirmación a la luz de lo dicho en el capítulo sobre las medidas de recursos y la productividad.

- 3.6 En un proyecto A, 14 trabajadores han desarrollado 50.000 *LoC* en un año. En otro proyecto B, 3 miembros del equipo de desarrollo han creado 12.000 *LoC* en 9 meses. ¿En qué proyecto se ha conseguido mayor productividad medida en *LoC/día*? **Nota:** Asuma que un hombre/mes son 150 horas de trabajo.
- 3.7 La siguiente figura muestra un ejemplo de grafo de comunicaciones de un equipo. Refiérase a la tabla de métricas adjunta.



Métrica	Resultado
Tamaño del equipo	
Densidad de comunicación	
Nivel de comunicación	
Nivel medio de comunicaciones	

- 3.8 Existen en la actualidad numerosos modelos de crecimiento de la fiabilidad, que pueden dividirse en modelos de cuenta de fallos y modelos de tiempo entre fallos. Lleve a cabo un pequeño trabajo de investigación donde estudie estas métricas de pruebas, elaborando como producto final una tabla con los modelos de crecimiento de la fiabilidad más relevantes y sus características distintivas.
- 3.9 Seleccione dos documentos de texto distintos, uno un artículo de una enciclopedia y otro una noticia de un periódico. Calcule a partir de ellos su legibilidad según los índices de Gunning-Fog y Flesch pero recuerde que ambos deberán estar escritos en inglés para que las métricas tengan validez. **Nota:** Puede contrastar sus resultados con los proporcionados automáticamente por Google Docs o Microsoft Word activando las opciones correspondientes.
- 3.10 Imagine que trabaja usted como responsable del mantenimiento de un sistema de facturación de una gran compañía telefónica. El sistema que usted mantiene saca nuevas versiones cada 6 meses aproximadamente, pero las últimas versiones introducidas en producción están generando más problemas de los esperados. Detalle un plan de métricas para medir y subsanar la situación en futuras versiones.

Parte II

Procesos fundamentales de la Ingeniería del Software

I

Ex
en
cap
sol

up
na
cie
cor

esp
de
econ
con
que
má

La
de
pro
ma
los
req
hab
las
de

Introducción a la Segunda Parte

Esta Segunda Parte es un paseo por los diferentes parajes de la Ingeniería del Software, diferenciados en el tipo de actividad que se realiza. Las categorías de actividades que se exponen en los respectivos capítulos de esta segunda parte rara vez se llevan a cabo en estricta secuencia: lo habitual es que se solapen en el tiempo, a través de iteraciones que ejercitan «un poco» de cada tipo de actividad.

Comenzaremos con el problema de especificar qué debe hacer el software. Se describen así los tipos fundamentales de requisitos, y las actividades necesarias para conseguir una especificación de requisitos que sea fidedigna con las necesidades de los usuarios. Veremos cómo las actividades relacionadas con los requisitos llegan hasta el análisis del dominio de la aplicación para, acto seguido, comenzar con las actividades de diseño, pasando del espacio del problema al espacio de la solución.

El diseño de la solución se abordará desde el nivel más general (arquitectura) hasta un nivel específico (diseño detallado), teniendo en cuenta que, como en todo proceso de ingeniería, no se trata de escoger la mejor solución, sino la mejor dentro de las posibles teniendo en cuenta las restricciones económicas, temporales y organizativas en las que se encuadra el proceso. El siguiente paso es la construcción del software en sí. Es decir, la programación y la depuración de los programas. Dado que se presupone del lector conocimientos al menos básicos de programación, el capítulo se centra más en aspectos relacionados con la calidad y la medición.

El capítulo de pruebas introduce los conceptos y técnicas clásicas para la prueba y su tipología. La prueba de cada pieza o módulo de código –la prueba unitaria–, está muy cerca de las actividades de programación, y en ocasiones se realiza intercalada con la propia codificación; otros tipos de prueba son más globales. El último capítulo aborda las actividades de cambio del software: su mantenimiento y evolución. Las actividades de mantenimiento no son esencialmente diferentes de las que se han visto en los otros tres capítulos pues la mayoría de los cambios implican tomar nuevos requisitos, rediseñar o adaptar el diseño y recodificar o escribir partes nuevas de código. Después habrá que probar para asegurarse de que no se han introducido errores que antes no existían. Aunque las actividades son las mismas, la forma de gestionarlas no lo es, pues se deben hacer bajo un proceso de control de cambios.

Scutellaria *Barbata* *Glauca* *Glauca*

Scutellaria *Barbata* *Glauca* *Glauca*

Requisitos

Lo más complicado de construir un sistema software es decidir exactamente qué construir [...] Ninguna otra parte del trabajo afecta tan negativamente al resultado si se hace mal. Ninguna otra es tan difícil de rectificar.

—F. P. Brooks

4.1 La difícil tarea de determinar qué debe hacerse

En un importante hospital español ocurrió hace unos años un suceso del que pueden extraerse no sólo sarcásticos comentarios sobre la capacidad de determinadas compañías para llevar a cabo un desarrollo de software a nivel profesional, sino también interesantes lecciones sobre la dificultad de las actividades de obtención de requisitos.

Todo comenzó cuando un bisoño consultor recién titulado fue enviado por su empresa, una compañía de desarrollo de software a medida, a la unidad de cuidados intensivos del hospital en cuestión. Su encargo era estudiar la forma de trabajo del personal, como paso previo al desarrollo de un sistema informático integral que mejoraría la eficacia en la gestión de la unidad. Es posible que otros consultores (o él mismo) fueran enviados a otras unidades simultáneamente, pero los autores sólo cuentan con información de lo que ocurrió en la unidad de cuidados intensivos.

Tras un primer contacto optimista, donde la mayor parte del personal se ofreció colaborar, el consultor (que acudía todos los días puntualmente a tomar notas y preguntar a los médicos y enfermeras sobre cómo desarrollaban su trabajo) expresó públicamente su certeza en que el trabajo concluiría «como mucho en un par de meses». Algo sorprendente, pues cualquier persona medianamente familiarizada con la obtención de los requisitos de un sistema software de tal complejidad es consciente de que esto no suele ser tarea de «un

par de meses». Así, y contrariamente a su vaticinio, el trabajo se prolongó durante más de seis meses, tiempo en el que se emplearon cientos de horas de trabajo a la postre inútil, pues el proyecto fue finalmente considerado inviable y desestimado, motivo por el cual el consultor un buen día dejó de ir al hospital y nunca más apareció por allí.

Este caso muestra uno de los problemas recurrentes a los que se enfrenta la ingeniería del software: la dificultad para determinar exactamente cuáles son los requisitos de un sistema. Es decir, las funcionalidades que el sistema a construir debe incluir así como todas las consideraciones adicionales sobre seguridad, rendimiento, fiabilidad, cuestiones legales, etc. Y lo que resulta aún más complicado: establecer claramente qué es lo que no debe contemplarse como parte de los requisitos del sistema, bien porque no ha sido incluido en el contrato de desarrollo, o bien porque simplemente queda fuera del alcance del sistema a construir.

La dificultad de la tarea proviene de la dificultad inherente a enunciar, de modo claro y preciso, cualquier problema complejo. Incluso en aquellos casos en que la experiencia de las personas que realizan la tarea de obtener y especificar los requisitos permite enfrentarse a ella con cierto optimismo, el éxito nunca está garantizado *a priori*. Muchos factores afectan negativamente a esta tarea, como por ejemplo el hecho de que los usuarios del futuro sistema pueden no colaborar en la especificación de los requisitos (o lo hacen deficientemente), que los requisitos cambian a lo largo del desarrollo, o que aparecen otros nuevos. En resumen, se trata de una tarea de tal envergadura que algunos autores han acuñado el término «ingeniería de requisitos» para referirse al conjunto de actividades a realizar en esta etapa del desarrollo.

4.2 Objetivos

El objetivo general de este capítulo es presentar los conceptos, conocimientos y técnicas asociados con la obtención, especificación, análisis y validación de los requisitos de un sistema software, estableciendo el papel de estas actividades dentro de la disciplina de la ingeniería del software. Más en detalle, tras la lectura de este capítulo el lector deberá:

- Comprender la importancia de obtener y gestionar los requisitos de un sistema software, en qué consisten estas actividades, y su influencia en el éxito de un proyecto.
- Conocer y distinguir los conceptos fundamentales relacionados con los requisitos de un sistema software.
- Diferenciar los distintos tipos de requisitos y ser capaz de clasificarlos.
- Conocer las técnicas más relevantes en la captura y especificación de los requisitos de un sistema, y ser capaz de elaborar un documento de especificación de requisitos haciendo uso de ellas.
- Reconocer las notaciones de las diferentes actividades de requisitos.

4.3 Introducción

Dentro de las actividades de la ingeniería del software, las relacionadas con la obtención y gestión de los requisitos resultan especialmente críticas. Este conjunto de actividades consiste en obtener y documentar los requisitos para desarrollar el sistema y posteriormente analizarlos con el objetivo de responder a la pregunta «¿Qué debe hacerse?».

Un proyecto software, simplificado al máximo, es básicamente la transformación de un conjunto de requisitos en un sistema informático. En consecuencia, si se parte de diferentes requisitos se obtendrán diferentes sistemas como resultado. He aquí uno de los componentes más importantes de la gestión de requisitos, puesto que si la descripción de los mismos no es adecuada, o se desvía de lo que el cliente o los usuarios finales desean, entonces tal vez obtendremos un sistema perfecto (o tal vez no) pero es evidente que dicho sistema no satisfará las expectativas generadas. Diremos entonces que el proyecto en cuestión ha fracasado (véase la Figura 4.1). Establecer con exactitud los requisitos de un sistema es, por tanto, un principio esencial para llevar a cabo con éxito un desarrollo de software. Como veremos, se trata de una de las más complicadas tareas a que se enfrentan los desarrolladores.



Figura 4.1: Las dificultades de comunicación en la descripción de los requisitos.

La obtención de requisitos es, en definitiva, un proceso muy complejo en el que intervienen diferentes personas, las cuales tienen distinta formación y conocimiento del sistema (desarrolladores, clientes, usuarios, etc.). En dicho proceso, es necesario tener en cuenta la influencia de diversos factores, que a veces de forma no evidente, tienen un impacto en el desarrollo del proceso en sí:

- En primer lugar hay que tener en cuenta la complejidad del problema a resolver. Como se apuntó en el primer capítulo, la complejidad es un factor inherente al software, y uno de los motivos fundamentales por los que ésto es así radica en el hecho de que con software se resuelven, generalmente, problemas complejos. La determinación exacta de los requisitos de un sistema que se adivina complejo no es, en la mayoría de los casos, tarea sencilla.

- Otros factores están relacionados con la forma de identificar los requisitos por parte del cliente. Muchas veces ni siquiera sabe (en términos concretos) lo que quiere.
- Dificultades de comunicación entre desarrolladores y usuarios, y entre los propios miembros del equipo de desarrolladores, como la viñeta de la Figura 4.1 muestra con humor. A menudo resulta complicado obtener información de ciertos usuarios, pues muchos no saben (o no desean) revelar todos los detalles sobre el funcionamiento de un sistema existente. En otras ocasiones, la persona que obtiene los requisitos no tiene la formación o la experiencia necesaria para ello, desconoce el dominio del problema al que se enfrenta, o no comprende la jerga en que se expresan los usuarios.
- Un buen número de requisitos del futuro sistema no pueden obtenerse a través del cliente y los usuarios, pues dependen de detalles internos al proceso de construcción del software. A menudo denominados «requisitos ocultos», deben ser llevados a cabo aunque no son visibles al usuario.
- Aspectos relacionados con la naturaleza cambiante de los requisitos, tales como la aparición de nuevos requisitos durante el proceso de desarrollo, o el hecho de que muchos de ellos se vean alterados a lo largo del mismo. En el capítulo dedicado al mantenimiento se estudian las denominadas leyes de la evolución del software, algunas de las cuales (la de cambio continuo o la de crecimiento continuo, por ejemplo) explican esta naturaleza de los requisitos.

Las dificultades intrínsecas de las actividades de requisitos se pueden ilustrar perfectamente con una metáfora que ya es clásica, la del iceberg. Aproximadamente, sólo un 10% de la masa de un iceberg se encuentra por encima de la superficie del mar y es por tanto visible. Enfrentarse a las tareas de requisitos sin la adecuada preparación lleva a pensar que la masa total del «iceberg de requisitos» es sólo la masa visible, lo cual constituye un error importante de previsión en la cantidad de esfuerzo necesario para llevarlas a cabo satisfactoriamente (ver Figura 4.2).

Este ejemplo pretende transmitir la complejidad de las actividades relacionadas con los requisitos. Complejidad que proviene de una definición ambigua, incorrecta o incompleta del problema, pero también del hecho de que la comprensión de cualquier problema y sus implicaciones es una actividad particularmente difícil de sistematizar. En este capítulo se proporciona una visión del conjunto de actividades relacionadas con la gestión sistemática y organizada de los requisitos, pero no se utilizará para denominarlas el término *ingeniería de requisitos* –empleado por muchos autores– porque presuponemos a toda ingeniería un enfoque sistemático, disciplinado y cuantificable del que carecen estas actividades. Usaremos en consecuencia el término *actividades de requisitos* para referenciarlas. Y si bien otros libros de ingeniería del software incluyen algunas de las actividades que aquí se tratarán como parte de otras etapas tales como el análisis del sistema, en este texto se sigue la estructura de la guía SWEBOK y por tanto todas las actividades relacionadas con los requisitos se estudian en el presente capítulo.

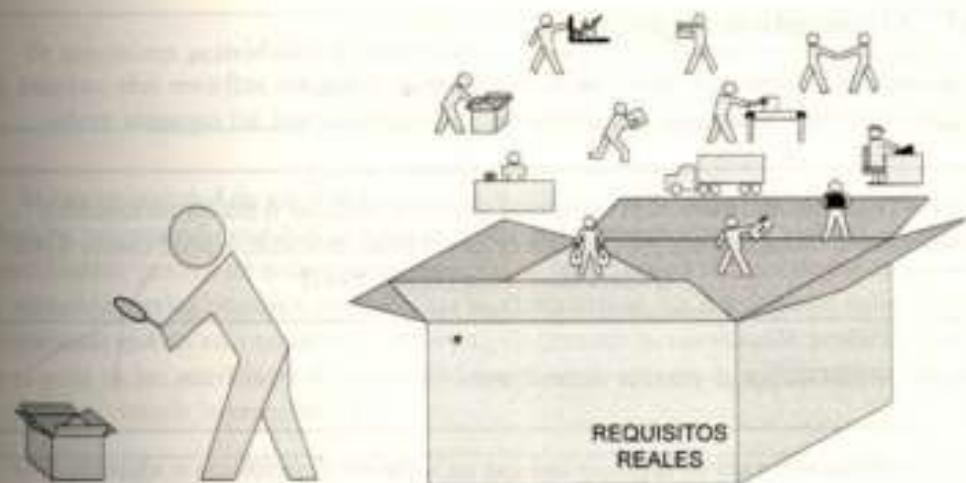


Figura 4.2: Lo que se ve dista mucho de los requisitos reales del sistema

Por otra parte, y como reza la cita de Brooks que encabeza el capítulo, ninguna otra parte del desarrollo afecta tanto al sistema resultante si se lleva a cabo de manera incorrecta. Ninguna, de hecho, es más difícil de modificar *a posteriori* si se hizo mal en un principio. Así, se estima que entre un 75% y un 85% de todos los errores que aparecen en un software «terminado» se deben a errores cometidos bien en las actividades de requisitos o bien durante el diseño del sistema.

En el resto del capítulo se pretende ayudar a conseguir una comprensión de las actividades relacionadas con los requisitos, para lo cual se definen los conceptos esenciales dentro de las actividades de requisitos, se estudia la clasificación de los diversos tipos de requisitos, se analizan las técnicas más usuales de obtención y especificación, así como las notaciones más utilizadas para el modelado conceptual y finalmente se estudian otros temas relacionados con la iteración, gestión de cambios y seguimiento de los requisitos.

4.4 Definiciones preliminares y características

Antes de profundizar en los temas que se tratan en este capítulo, es preciso definir formalmente lo que se entiende por «requisito software». A partir de dicha definición, se definirán las actividades que conlleva la gestión de los requisitos y los actores que participan en las mismas, para finalmente enumerar las características propias de los requisitos software y establecer el significado de uno de los elementos centrales de las actividades de requisitos: el documento de especificación de requisitos software.

4.4.1 El concepto de requisito

A continuación se muestran dos de las definiciones de requisitos software más comunes. El Glosario IEEE de Términos de Ingeniería del Software lo define del siguiente modo:

Un requisito software es la capacidad que debe alcanzar o poseer un sistema o componente de un sistema para satisfacer un contrato, estándar, especificación u otro documento formal (IEEE, 1990)

La Guía SWEBOK por su parte lo define como:

Un requisito software es la propiedad que un software desarrollado o adaptado debe tener para resolver un problema concreto

De acuerdo con la última definición, un requisito software puede consistir en dar soporte a procesos de negocio de la organización que ha solicitado el software (como calcular e imprimir las nóminas de los empleados a fin de mes), automatizar ciertas tareas de las que lleva a cabo alguna persona que utilizará el software (por ejemplo la introducción de datos de un pedido a un proveedor mediante un terminal con un lector óptico), realizar funciones que no realiza el software actual de una organización (cuando se implantó la moneda única en la Unión Europea, hubo que modificar los sistemas que manejaban unidades monetarias para que soportaran precios en euros), etc.

Cuando se especifican los requisitos, se almacena información sobre los mismos que permite gestionarlos e interpretarlos, información que se conoce como *atributos*.

Se denomina **atributo** de un requisito a cualquier información complementaria que se utiliza para su gestión y que se incluye en su especificación

Esta información a menudo consiste en una descripción general del requisito, su tipo según diferentes sistemas de clasificación (funcional o no funcional, del sistema, de usuario, etc.), la fuente del requisito, la historia de cambios sufrida, u otros que pudieran definirse. La mayoría de herramientas software para la gestión de requisitos permiten al usuario asociar nuevos tipos de atributos.

4.4.2 Actividades de requisitos

Según la guía SWEBOK, el conjunto de tareas a realizar en el área de los requisitos software se engloba bajo la denominación genérica de «actividades de requisitos»:

Se denominan **actividades de requisitos** a la obtención, análisis, especificación y validación de requisitos software

Se trata en realidad de un proceso del que se obtienen, en primer lugar, un conjunto de objetivos preguntando qué hay que hacer y por qué. Posteriormente, se lleva a cabo una especificación precisa de necesidades, que será la base para el análisis de los requisitos y la validación con las personas involucradas en el desarrollo. En este proceso debe definirse exactamente qué se va a construir, con objeto de permitir la verificación posterior (que ya no es parte de las actividades de requisitos) donde se comprueba que el sistema construido y entregado cumple lo especificado.

La obtención de requisitos consiste en capturar el propósito y funcionalidades del sistema desde la perspectiva del usuario

La tarea de obtención consiste en elaborar un catálogo de necesidades del cliente y de los usuarios del futuro sistema, lo cual incluye obtener información sobre las necesidades generales del mismo, su funcionamiento, su rendimiento, el diseño y restricciones de la interfaz, y muchas otras. Sin embargo, este catálogo de necesidades «en bruto» debe estudiarse y refinarse mediante un proceso de análisis que permita obtener un conjunto detallado de requisitos del sistema.

El análisis de requisitos es el proceso de estudiar las necesidades del usuario para obtener una definición detallada de los requisitos

La tarea de análisis consiste por tanto en comprender cuáles son los comportamientos que se desea tenga el software a desarrollar, delimitando y determinando detalladamente todos los requisitos del producto a partir de lo solicitado por el cliente y los usuarios del futuro sistema. Estos requisitos deberán ser documentados por los analistas en un proceso denominado especificación de requisitos, donde se registrará el comportamiento del sistema software que se va a construir.

Se denomina **especificación de requisitos** al proceso de documentar el comportamiento requerido de un sistema software, a menudo utilizando una notación de modelado u otro lenguaje de especificación.

Al tener que estudiar de modo detallado los requisitos, el proceso de especificación sirve a su vez para refinarlos. El resultado de la especificación es un conjunto de modelos

donde se representan los requisitos desde varias perspectivas (del usuario, estructural, de comportamiento, de flujo de información, etc.)

Finalmente, la validación de requisitos software consiste en confirmar que los requisitos obtenidos, tras su análisis y especificación, concuerdan con las necesidades que los clientes y usuarios expresaron.

La validación de requisitos consiste en examinar los requisitos para asegurarse de que definen el sistema que el cliente y los usuarios desean

No debe confundirse validación de requisitos con verificación de los requisitos, pues esta última es una actividad posterior, que no forma parte de las actividades propias de los requisitos, y que consiste en comprobar el correcto funcionamiento de un requisito en el sistema desarrollado.

4.4.3 Actores

El proceso de requisitos se desarrolla en buena medida en un escenario pluridisciplinar, donde participan especialistas diversos, tanto en el dominio del problema como en ingeniería del software. Los roles de las personas involucradas en el proceso de requisitos no son siempre los mismos, y en consecuencia varían de un proyecto a otro. A cada uno de los diferentes roles que participan en el proceso de requisitos, y que pueden ser desempeñados por la misma o por distintas personas según el punto del desarrollo en que nos encontremos, lo denominaremos *actor*.

Se denomina **actor** a un rol perfectamente definido que una persona puede desempeñar en el proceso de requisitos

Partiendo pues de esta definición, es posible identificar los siguientes actores, comunes a la mayoría de proyectos de desarrollo de software:

- *Usuarios*: se trata de un grupo heterogéneo que comprende a todos aquellos que operan con el software.
- *Clientes*: todos aquellos que tienen interés en adquirir el software o representan de algún modo al mercado potencial al que se dirige el software.
- *Analistas de mercado*: en los productos software destinados a un mercado mayor (Microsoft Office, por ejemplo) no existe un cliente definido, se ha de contar con personas especializadas en recabar las posibles necesidades del mercado obteniendo requisitos a través de los potenciales clientes.

- *Reguladores*: muchos dominios de aplicación, como la administración pública o la industria aeroespacial, tienen normativas estrictas o requisitos legales regulados por autoridades específicas. El software desarrollado en estos dominios debe ajustarse a lo establecido por las entidades reguladoras.
- *Ingenieros de software*: personas que tienen interés, por ejemplo, en la reutilización de ciertos componentes software para otros proyectos (por ejemplo una compañía de desarrollo de juegos de computadora que desea poder reutilizar en futuros juegos las capacidades de representación de personajes en 3 dimensiones implementadas en el juego en desarrollo). Así, cuando un involucrado tiene unos requisitos específicos que pueden comprometer la capacidad de reutilización de un componente software, habrá de valorarse la posibilidad de llevar a cabo el desarrollo en los términos que plantea este actor en cuestión.

Puede imaginarse lo difícil que resulta cumplir con todos y cada uno de los requisitos expresados por los actores participantes. Será tarea de los ingenieros de software plantear soluciones de compromiso que satisfagan suficientemente a todos los involucrados.

4.4.4 Características de los requisitos

Una vez definido el concepto de requisito, el conjunto de tareas que constituyen las actividades de requisitos, y los actores que participan en ellas, es necesario estudiar las características de los requisitos software.

La característica esencial de todo requisito software es que sea *verificable*, propiedad intrínseca que permitirá comprobar más adelante que el sistema software en su conjunto, o alguno de sus componentes, da cumplimiento al requisito tal y como fue especificado. Desde un punto de vista «contractual», se puede decir que gracias a esta propiedad los clientes pueden comprobar que los desarrolladores «han cumplido su parte del acuerdo».

La verificabilidad es una propiedad intrínseca de un requisito, por la cual es posible comprobar que el sistema software en su conjunto, o alguno de sus componentes, satisfacen el requisito tal y como fue especificado

De acuerdo con esta propiedad, los requisitos deben establecerse, cuando sea posible, de manera *cuantificable*. Es muy importante evitar especificaciones ambiguas tales como «la interfaz de usuario será amigable y fácil de utilizar», de muy difícil verificación. Lo deseable es, por el contrario, que la mayor parte de los requisitos estén formulados de tal modo que su verificabilidad sea sencilla. Una definición cuantificable sería por ejemplo: «el tiempo máximo en que el usuario obtiene respuesta a una consulta en la base de datos nunca debe ser superior a 2 segundos». Esta capacidad de cuantificar los requisitos facilita la medición de su cumplimiento, otorgando valor a un sistema software.

Además de la verificabilidad, existen otras propiedades no esenciales, externas a los requisitos, las más importantes de las cuales son:

- *Priorización*: al desarrollar el sistema, será necesario abordar los requisitos en un orden que en muchos casos dependerá del propio requisito. En circunstancias especiales, por ejemplo cuando el presupuesto se desvíe demasiado o cuando el tiempo estimado de desarrollo se haya superado, algunos requisitos de baja prioridad se pospondrán para ciclos de desarrollo futuros, o incluso se desestimarán. Estas situaciones justifican la necesidad de clasificar los requisitos según su prioridad (e incluso de marcar algunos como «prescindibles»).
- *Seguimiento*: debe ser posible hacer un seguimiento del requisito que permita conocer su estado (especificado, verificado, analizado, etc.) en cada momento del desarrollo.
- *Identificación única*: cada requisito debe tener un identificador único que lo distinga y que permita a cualquier involucrado en el desarrollo hacer referencia al mismo en cualquier punto del ciclo de vida del software sin ambigüedad.
- *Cuantificabilidad*: es muy deseable que se pueda medir el grado de cumplimiento de un requisito en términos precisos. El tráfico telefónico, que se caracteriza por la necesidad de garantías de alta calidad, tiene requisitos cuantificables y su valor depende de que estos requisitos se cumplan con precisión. No obstante, no todos los requisitos pueden cuantificarse de forma precisa.

4.4.5 El documento de especificación de requisitos

Como puede verse en la Figura 4.3, el resultado principal de las actividades de requisitos es la obtención de un catálogo de necesidades detallado, modelado en un lenguaje de especificación y validado, cuya razón de ser fundamental es que se pueda comprobar que los productos generados se corresponden con los deseos del cliente. A este catálogo se le conoce con el nombre de «documento de especificación de requisitos del software». En dicho documento, que como se verá más adelante sirve de base para el contrato entre los desarrolladores y el cliente, se establece qué hará el software, pero no cómo lo hará.

El documento de especificación de requisitos del software contiene un conjunto exhaustivo y preciso de requisitos modelados en un lenguaje de especificación y validados, los cuales sirven como contrato entre lo que desea el cliente y lo que los desarrolladores se comprometen a construir

Dado que será la base de trabajo de los desarrolladores y los ingenieros de prueba a la hora de implementar la aplicación y verificar que cumple los términos del contrato con el cliente, es importante que el documento de especificación de requisitos incluya una catalogación exhaustiva y correcta de los requisitos.

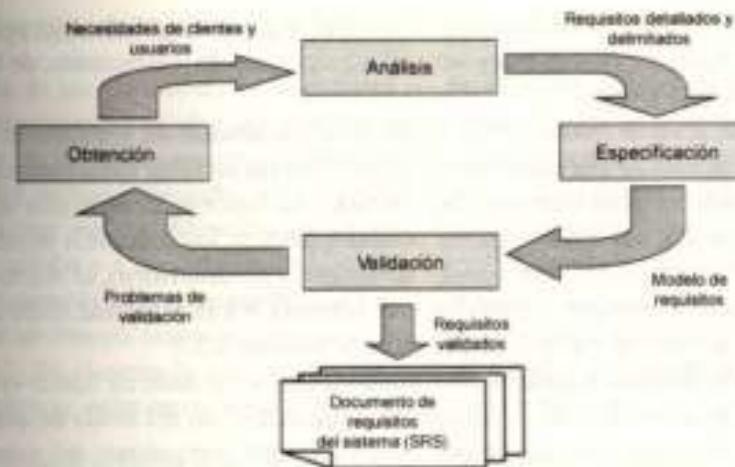


Figura 4.3: El proceso de requisitos

4.5 Tipos de requisitos

Tradicionalmente se han identificado dos tipos de requisitos atendiendo a criterios de funcionalidad: *requisitos funcionales* y *requisitos no funcionales*. Los primeros definen aquellas funciones que el sistema o alguno de sus componentes debería ser capaz de hacer. Los segundos, por el contrario, describen restricciones y exigencias de calidad del sistema o de los procesos realizados por el mismo. El glosario IEEE de ingeniería del software (IEEE, 1990) define el primero de estos términos del siguiente modo:

Un **requisito funcional** especifica una función que un sistema o componente de un sistema debe ser capaz de llevar a cabo

Por otra parte, es posible definir requisito no funcional de la siguiente manera:

Los **requisitos no funcionales** son aquellos que especifican aspectos técnicos que debe incluir el sistema, y que pueden clasificarse en restricciones y calidades

Dentro de las *restricciones* se encuentra cualquier limitación a que se enfrenten los desarrolladores del sistema, como por ejemplo el sistema operativo del entorno del usuario, la plataforma-hardware a utilizar o el perfil de los desarrolladores disponibles dentro de la organización de desarrollo. Dentro de las denominadas *calidades* se encuentran todas aquellas características de un sistema que importan a los clientes y usuarios del mismo, y

que por tanto serán relevantes a la hora de establecer su grado de satisfacción con el sistema final. Algunos ejemplos de calidades serían las exigencias de rendimiento, de fiabilidad o de disponibilidad del sistema.

No obstante, a veces resulta difícil situar inequívocamente un requisito en una u otra categoría, pues se trata de una clasificación cuyas fronteras no están claramente delimitadas. Y si bien la clasificación en requisitos funcionales y no funcionales es la más ampliamente utilizada, existen muchas otras. Así, es posible clasificar los requisitos atendiendo a su nivel de detalle y formalización (requisitos detallados y no detallados), al objeto final sobre el que se hace el requerimiento (el proceso o el sistema), o a su prioridad, entre otros. Estas clasificaciones se estudian con mayor detalle en la Sección 4.5.3.

Por otra parte, algunos requisitos representan lo que se ha dado en llamar «propiedades emergentes», o lo que es lo mismo, propiedades que se derivan del modo en que el sistema en su conjunto funciona, pero que no pertenecen a ningún componente del sistema en particular. La velocidad de ejecución, la complejidad o la facilidad de mantenimiento de un sistema son algunos ejemplos de propiedades emergentes.

4.5.1 Requisitos funcionales

Los **requisitos funcionales** definen el comportamiento del sistema que se va a desarrollar, incluyendo los procesos fundamentales que el software llevará a cabo. La mayoría de los requisitos funcionales proviene directamente de un requisito de usuario, es decir, de alguna descripción en lenguaje natural que el usuario ha hecho llegar a los desarrolladores a través de las entrevistas mantenidas durante la obtención de requisitos. Así, si tomamos como ejemplo un pequeño negocio de venta y alquiler de herramientas, algunos requisitos funcionales podrían ser los siguientes:

1. Imprimir los contratos de alquiler.
2. Almacenar la información relativa a los contratos de alquiler en vigor.
3. Guardar información de ventas y pagos.
4. Realizar un seguimiento, por cliente, del estado de los pagos.
5. Gestionar el inventario de productos en venta y en alquiler.

Es importante, durante las actividades de requisitos, obtener un catálogo completo de requisitos funcionales, aunque esto no es algo que se consiga en un primer intento. Es necesario que los desarrolladores se familiaricen con el sistema y comiencen a analizar los requisitos más obvios para poco a poco ir descubriendo aquellos menos evidentes.

Durante la determinación de los requisitos funcionales deben tenerse muy en cuenta los riesgos derivados de una imprecisa especificación. Resulta imprescindible expresar los requisitos de manera inequívoca para así evitar ambigüedades que podrían llevar a que desarrolladores y usuarios interpreten un mismo requisito de manera distinta. Para ello, y como se verá en la Sección 4.6.3, la especificación deberá ser completa (incluyendo todos los servicios solicitados por el usuario) y consistente (sin contradicciones).

4.5.2 Requisitos no funcionales

A diferencia de los anteriores, los **requisitos no funcionales** tienen que ver con restricciones y exigencias de calidad del sistema, entre las que se encuentran los requisitos de rendimiento, de interfaces externas (tales como el tiempo de respuesta), requisitos relativos a la facilidad de mantenimiento, de seguridad, de fiabilidad, relativos a la capacidad de almacenamiento, etc. A diferencia de los requisitos funcionales, los no funcionales son difíciles de validar y a menudo su evaluación se lleva a cabo de manera subjetiva: ¿cómo evaluar, por ejemplo, si un sistema de banca *online* realiza una transferencia de dinero entre cuentas «en un tiempo razonable para el usuario»? o ¿cómo cuantificar si un sistema es «amigable»? En el ejemplo anterior del negocio de alquiler y venta de herramientas, se pueden identificar, entre otros, los siguientes requisitos no funcionales:

1. Aunque el cliente actual utiliza un sistema operativo determinado, se ha de desarrollar el software de manera que pueda ejecutarse sobre diferentes plataformas, con vistas a ofrecerlo en el futuro a establecimientos similares que utilizan sistemas operativos de otros fabricantes.
2. El sistema debe estar disponible al menos el 95% de cada periodo de 24 horas.
3. En el desarrollo debe regirse obligatoriamente por los procesos y actividades definidos por la metodología estándar de planificación y desarrollo de sistemas de información de la Administración Pública Española, Métrica.
4. El registro de datos personales debe cumplir la Ley Orgánica española 15/1999 de Protección de Datos de Carácter Personal.

Una clasificación amplia de los requisitos no funcionales permite identificar 3 categorías:

- Requisitos del producto: aquellos que detallan limitaciones o comportamientos exigidos al producto resultante del desarrollo. Por ejemplo la cantidad de memoria requerida o la velocidad de respuesta en operaciones interactivas.
- Requisitos de la organización: aquellos relacionados con las normativas de funcionamiento de la organización que lleva a cabo el desarrollo, sus procedimientos y políticas. Por ejemplo los estándares de desarrollo, la documentación a entregar junto con el producto, los plazos de entrega, etc.
- Requisitos externos: cubren aspectos externos al sistema y a su proceso de desarrollo. Por ejemplo interoperabilidad con otros sistemas, requisitos legales, etc.

De acuerdo con esta clasificación, los dos primeros ejemplos citados anteriormente (la capacidad de que el software pueda ejecutarse en diversas plataformas y los requisitos de disponibilidad) serían requisitos del producto. Por otra parte, la obligatoriedad de ajustarse a una metodología de desarrollo determinada es un requisito de la organización. Finalmente, la necesidad de cumplir con la Ley de Protección de Datos de Carácter Personal se entiende como un requisito externo.

Es posible, no obstante, hacer una taxonomía más detallada de los requisitos no funcionales. Así, podemos decir que los requisitos no funcionales de un sistema se clasifican en alguna de las siguientes categorías (la lista exhaustiva de se muestra en la Tabla 4.1):

- Requisitos de ejecución, donde se agrupan todos aquellos que tienen que ver con el nivel de satisfacción (medible) de alguna propiedad del comportamiento del sistema, pero no de su implementación interna. En otras palabras, definen cómo de bien se comporta el sistema en un aspecto determinado, y este «cómo de bien» puede ser evaluado externamente. Estos requisitos proporcionan valor al usuario e introducen elementos diferenciales con otros sistemas similares en términos competitivos a corto plazo. Dentro de esta categoría se encuentran los siguientes tipos de requisitos:
 - Usabilidad: facilidad de uso, facilidad con que se aprende a utilizar la aplicación, facilidad para recordar cómo proceder a la hora de realizar una cierta función (*memorability*), y eficiencia.
 - Facilidad de configuración y de soporte.
 - Corrección, confiabilidad y disponibilidad.
 - Requisitos del entorno de ejecución (del sistema operativo, de compatibilidad entre distintas plataformas hardware de trabajo, servidores en los que se ejecutará, y otros).
 - Requisitos relacionados con la calidad de servicio, tales como el rendimiento, tasa de transferencia, tiempo de respuesta, latencia, etc.
 - Propiedades relacionadas con la seguridad: tolerancia a fallos, protección de información, etc.
 - Escalabilidad operativa, que incluye soporte para usuarios o instalaciones adicionales, para mayores volúmenes de transacción, etc.
 - Restricciones de negocio: reglas de negocio que se deben definir en la aplicación para su correcto funcionamiento interno. Ejemplo: «no se pueden servir pedidos cuyo pago no se haya formalizado».
- Requisitos de desarrollo: a diferencia de los requisitos de ejecución, que proporcionan valor directo a los usuarios, los requisitos de desarrollo proporcionan valor de negocio, y por tanto influyen en la competitividad a largo plazo del producto software desarrollado.

Tabla 4.1: Requisitos no funcionales

Apariencia	Dependencia de terceros	Fiabilidad
Aspectos culturales y políticos	Disponibilidad	Gestión de fallos
Aspectos económicos	Documentación	Integridad
Aspectos legales y de licencias	Eficacia	Limitación de recursos
Auditoria y control	Eficiencia	Rendimiento
Calidad	Entorno de desarrollo	Resistencia
Certificación	Entorno físico	Robustez
Compatibilidad	Escalabilidad	Seguridad
Comunicaciones	Estabilidad	Soporte
Confidencialidad	Extensibilidad	Tiempo de respuesta
Custodia	Facilidad de mantenimiento	Usabilidad

El test TPC-C

El denominado «TPC benchmark C» es un banco de pruebas para el procesamiento de transacciones en línea. Este test constituye un ejemplo interesante de herramientas al servicio de la verificación de los requisitos no funcionales.

Creado en 1992 por el *Transaction processing Performance Council* (TPC), es una herramienta de aplicación general que puede ser utilizada en cualquier entorno de negocio que deba gestionar, vender o distribuir un producto o servicio. TPC-C simula un entorno completo de ejecución donde un conjunto de usuarios ficticios ejecuta transacciones sobre una base de datos. La herramienta tiene soporte para diferentes tipos de transacciones, así como para la gestión de bases de datos complejas.

Su uso más habitual es la medición del rendimiento del tiempo de respuesta en un sistema, cuyos resultados se establecen en transacciones por minuto. Sus conclusiones pueden ayudar a refinar el diseño o la construcción para permitir ajustar los resultados obtenidos a los esperados, si bien algunos autores estiman que la utilización de este tipo de herramientas tiene también efectos negativos, como la preocupación excesiva por la afinación del código con el objeto de mejorar obsesivamente el rendimiento.

Por último, sólo referir que con objeto de eliminar la evaluación subjetiva de los requisitos no funcionales, algunos autores proponen expresarlos mediante métricas. Se trata, en suma, de proporcionar un mecanismo para cuantificar ciertas propiedades utilizando escalas públicas y objetivas, como por ejemplo evaluar la facilidad de uso de un software a partir del tiempo que un usuario medio tarda en obtener un cierto nivel de capacitación.

4.5.3 Otras clasificaciones de los requisitos

Algunos autores señalan la incorrecta separación entre los distintos niveles de detalle de los requisitos como causa de muchos de los problemas que surgen durante las actividades de

requisitos. Utilizando el nivel de detalle como criterio de clasificación, es posible identificar requisitos con un alto nivel de abstracción, también denominados «requisitos de usuario», y requisitos detallados, que algunos autores denominan «requisitos del sistema». Por otra parte, es posible también clasificarlos por el objetivo de los mismos en requisitos de proceso y de producto. A continuación se analizan con mayor detalle estas clasificaciones.

Requisitos según su nivel de detalle

Los *requisitos del usuario* son especificaciones abstractas, generalmente en lenguaje natural, de las funciones que se espera que lleve a cabo el sistema (requisitos funcionales de alto nivel) o de las restricciones aplicables al mismo (requisitos no funcionales de alto nivel). Así, para el negocio de alquiler de herramientas que se puso como ejemplo en la Sección 4.5.3, se pueden identificar como requisitos del usuario los siguientes:

1. Debe ser posible imprimir un informe mensual por cliente que incluya información detallada de todos los alquileres realizados.
2. El número de unidades disponibles de una herramienta se actualizará tras cada operación de compra o alquiler de manera instantánea.

Por otra parte, los *requisitos del sistema* son descripciones más detalladas, que establecen con precisión qué debe hacer el sistema. Estos requisitos tienen carácter «contractual» entre los desarrolladores de software y el cliente. Si bien es posible utilizar descripciones en lenguaje natural, lo habitual será modelar estos requisitos en algún lenguaje o notación específica *ad hoc*. Siguiendo con el ejemplo del negocio de alquiler de maquinaria, un requisito detallado que se deriva de uno de alto nivel anterior podría ser el siguiente:

3. El informe mensual por cliente debe incluir, junto con todos los datos fiscales de la tienda, el identificativo fiscal del cliente, el subtotal para cada servicio facturado (entendiendo por ello cada alquiler o compra unitaria realizada) así como el importe total antes y después de impuestos.

Según la guía SWEBOK no se debe hablar de requisitos de usuario, pues éstos pueden haber sido establecidos por clientes u otras personas que no son usuarios. Así, debe utilizarse el término solamente para aquellos requisitos que provienen directamente bien de los clientes o bien de los usuarios finales del sistema. Por el contrario, los requisitos del sistema son una categoría más general que engloba los requisitos de usuario pero también los que provienen de otros actores involucrados (autoridades reguladoras, por ejemplo) así como todos aquellos que no tengan un origen «humano» claramente identificado.

Requisitos según su objetivo

Otra forma de clasificar los requisitos es la identificación del objetivo de cada requisito en sí. Según este criterio, existen dos tipos de requisitos:

- *Requisitos sobre el producto* (software) que se va a desarrollar. Por ejemplo: «Para que un cliente pueda realizar una nueva operación de alquiler, no debe tener ningún pago pendiente de operaciones anteriores».
- *Requisitos sobre el proceso*, que se materializan en restricciones sobre el propio proceso de desarrollo en sí. Por ejemplo: «El lenguaje de programación a utilizar será Java». Estos requisitos pueden venir impuestos, entre otros factores, por entidades externas de verificación de la calidad, por normas vigentes en el territorio donde se lleva a cabo el desarrollo o por la propia estructura de la organización de desarrollo.

En lo relativo a la relación entre ambos, es importante reseñar que algunos requisitos sobre el producto generan requisitos sobre el sistema de forma implícita. Así por ejemplo, la verificación de algunos requisitos sobre el software suele requerir ciertas técnicas específicas de verificación, lo que constituye una restricción sobre el proceso de desarrollo (de la fase de pruebas, concretamente).

4.6 Las actividades de requisitos

Las actividades a llevar a cabo para la obtención, análisis, especificación y validación de requisitos deben llevarse a cabo de manera disciplinada y de acuerdo a un proceso claramente definido (ver Figura 4.4). En dicho proceso intervienen *actores*, cada uno de los cuales desempeña un papel que viene determinado por las actividades a realizar.



Figura 4.4: Actividades de requisitos

En las siguientes secciones se detallan las actividades más relevantes de requisitos: la identificación y documentación de las necesidades del usuario, el análisis y especificación

de dichas necesidades, la creación de un documento formal de especificación que incluya el catálogo completo de requisitos y, finalmente, la validación de dicho documento con objeto de evitar inconsistencias a lo largo del desarrollo. Se trata de un proceso iterativo de análisis del problema, en el que desarrolladores, clientes, usuarios finales del sistema, y otros actores involucrados trabajan en equipo para llegar a comprender en profundidad lo que se desea que haga el software. Dicho proceso se inicia al principio de cada proyecto y va siendo refinado durante todo su ciclo de vida.

4.6.1 Obtención de requisitos

Esta actividad recibe diferentes nombres, tales como «obtención de requisitos», «adquisición de requisitos» o «licitación¹ de requisitos», entre otros. Se trata de la primera de las actividades a realizar, donde el objetivo fundamental de los ingenieros de software consiste en determinar cuáles son los requisitos del sistema a desarrollar para llegar a un conocimiento suficiente del problema a resolver. Se trata, en pocas palabras, de establecer las fronteras del futuro sistema.

Gestión de requisitos en proyectos *open source*

Los proyectos desarrollados en comunidades *open source* llevan a cabo las actividades de requisitos de un modo diferente, pues las actividades de requisitos están definidas de modo poco rígido, y articuladas en torno a la utilización de la Web como medio de expresión y representación. En cuanto a las fuentes para la obtención de requisitos identificamos:

1. Propuestas espontáneas en forma de mensaje de correo electrónico o en un foro de discusión a tal efecto. Estas propuestas a menudo se denominan peticiones de características del software (*feature requests*).
2. Un documento de referencia (a menudo referido como «la visión») que establece el estado actual del software y cómo sería idealmente, y donde se invita a formar parte de la comunidad de desarrolladores. Este documento, que describe someramente aquellos elementos que permitirían obtener el software ideal descrito por el documento de referencia, constituye en realidad un catálogo de requisitos no funcionales que a menudo incluye cuestiones de rendimiento, mejora de la calidad, etc.
3. Listas de funcionalidades (o requisitos técnicos) por implementar publicadas por los responsables del desarrollo de una clase, paquete o componente.

Una vez ha sido remitido por los usuarios en forma de mensaje, los miembros más expertos de la comunidad y las personas clave en el desarrollo estudian la adecuación y oportunidad del requisito, aprobándolo o rechazándolo mediante argumentaciones que a menudo se suceden como mensajes de respuesta al mensaje original. El registro de mensajes permite *a posteriori* conocer quién solicitó el requisito, por qué, cuándo, etc., sin necesidad de un registro formal o de un proceso más elaborado de análisis, especificación y validación.

¹El verbo «licitar» no existe en castellano, si bien este término es de uso habitual en la jerga de inglés como traducción del inglés «requirements elicitation».

Durante esta actividad, el equipo de desarrolladores trabaja codo con codo con clientes, usuarios y otros actores involucrados, por lo que la fluidez en la comunicación es importante para que el proyecto llegue a buen término. A menudo como parte de esta política de buena comunicación se elabora un glosario de términos que permite a cualquiera involucrado en las tareas de requisitos conocer la definición aceptada para un determinado concepto.

En primer lugar, deben determinarse las *fuentes de información* de las que se obtendrán los requisitos. Al ser una actividad donde la relación entre las personas participantes es especialmente relevante, la información se obtiene de aquellos actores involucrados que tienen un conocimiento profundo del dominio del problema, investigando en las siguientes fuentes de información:

- Los *objetivos generales o de alto nivel del software*, que constituyen el motivo fundamental por el que el desarrollo se lleva a cabo.
- El *dominio del problema*, del que el ingeniero de software tiene un conocimiento limitado, pero del que otros actores conocen información muy valiosa.
- Los *propios involucrados o actores del proceso* y sus diferentes puntos de vista, tanto de la organización como del software en desarrollo.
- El *entorno de operación* en el que se ejecutará el software, que permitirá establecer las restricciones del proyecto y los costes que éstas comportarán.
- El *entorno de la organización*, al que debe adaptarse el software.

En segundo lugar, deben establecerse las *técnicas de obtención de requisitos* a utilizar. Los miembros del equipo de desarrollo deben obtener la información sobre los requisitos de los otros actores, pero no deben pensar en este proceso como en algo estático. Lamentablemente las cosas no son tan sencillas como sentarse a preguntar y anotar lo que los clientes, usuarios y otros interesados digan. El ingeniero de software debe saber hacer las preguntas precisas, investigar puntos oscuros, etc., en una labor dinámica de extracción de información donde la experiencia proporciona una capacitación difícil de obtener de otro modo. Para ello usará varias técnicas, siendo las más utilizadas las siguientes:

- Entrevistas: conjunto de técnicas, dentro de las que se incluyen visitas al cliente o los usuarios, cuestionarios, encuestas y entrevistas más o menos estructuradas de carácter formal o informal. Pueden ser cerradas –el cliente responde a un cuestionario con preguntas concretas y respuestas delimitadas–, o abiertas.
- Escenarios: constituyen una interesante herramienta para contextualizar los requisitos que permiten responder a preguntas del estilo «qué pasaría si» o «cómo hacer ésto». Aunque existen diferentes enfoques, el más ampliamente utilizado son los casos de uso de UML (ver la Sección 4.7.1).

- Prototipos: una buena forma de probar cualquier producto (incluido el software) y realizar cambios cuando aún se está a tiempo, consiste desarrollar un prototipo. Durante la obtención de los requisitos puede servir para clarificar requisitos confusos o obtener algunos que se hayan pasado por alto. Existen diferentes técnicas de prototipado, desde las más simples, como esbozos en papel con el diseño de las pantallas, hasta las más sofisticadas, como las versiones beta del producto software.
- Reuniones de grupo: se programan con el propósito de aunar esfuerzos y conseguir entre varios lo que resulta difícil de alcanzar individualmente, pues permiten detectar puntos de vista contrapuestos. Durante la obtención de requisitos, sirven para detectar requisitos entre los que existen conflictos, proporcionando el medio idóneo para que los diferentes interesados aporten sus visiones, contrasten sus necesidades y faciliten una solución de compromiso. Pueden concebirse como «tormentas de ideas» al principio de la fase de obtención de requisitos, si bien más adelante tendrán un carácter más aclaratorio y de análisis de los diferentes puntos de vista.
- Observación: se trata de un conjunto de técnicas que permiten a los ingenieros de software aprender cómo se llevan a cabo las tareas de usuario con el objeto de obtener los requisitos del sistema. A menudo consisten en introducirse en el ambiente en que trabajan los usuarios, observar cómo trabajan e interaccionan entre ellos y documentar dichos comportamientos.
- Otros: tales como el estudio de documentos y formularios actualmente en uso, visitas a otras instalaciones similares, presentaciones comerciales, estudio de productos, asistencia a jornadas profesionales, etc.

De entre todas las técnicas anteriores, los ingenieros de software pueden hacer el uso que cada caso particular requiera. Aunque cada proyecto tiene sus propias especificidades, lo habitual será utilizar una combinación de todas ellas.

Las entrevistas

Las entrevistas son una de las técnicas más utilizadas en la obtención de requisitos. Sin embargo, para que sean productivas y útiles deben ser cuidadosamente preparadas y planificadas, tanto su desarrollo como las actividades de inicio y finalización de las mismas. A continuación se establecen las fases de una entrevista:

1. Planificación: durante esta fase debe clarificarse fundamentalmente cuáles son los datos que se desean obtener. Para ello, habrá que determinar qué personas deben ser entrevistadas, cuándo y en qué lugar.
2. Preparación: en esta fase, los entrevistadores deben recabar información externa útil de cara a la entrevista. Así, prepararán un conjunto de preguntas como guía de la entrevista, se informarán de las funciones, personalidad y cargo de la persona que va

a ser entrevistada, fijarán el día, hora y lugar exactos de la entrevista y comunicarán por anticipado a dicha persona el objeto de la entrevista y la agenda de la misma.

Desarrollo conjunto de aplicaciones (JAD)

JAD (*Joint Application Development*), es una técnica desarrollada en los años 1970 por IBM, y que constituye una interesante alternativa a las entrevistas. Las cuatro ideas básicas en que se basa JAD resumen la filosofía de esta técnica:

- (a) Quien mejor conoce un trabajo es la gente que actualmente lo desempeña.
- (b) Quien mejor conoce las posibilidades de las tecnologías de la información son los miembros del equipo de desarrollo con formación técnica específica.
- (c) Los sistemas de información y los procesos de negocio no están aislados, por lo que frecuentemente transcienden las fronteras de un único departamento o sección. Los que trabajan en dichas áreas conocen bien el funcionamiento del sistema en una comunidad concreta.
- (d) Los mejores sistemas de información son los diseñados por grupos donde todos los actores anteriores trabajan de igual a igual.

En la práctica, JAD se basa en la realización de reuniones breves en grupo cuya duración aproximada es 2 horas. En estas reuniones, analistas y desarrolladores trabajan con los representantes de los clientes y usuarios con el fin de ayudarles a formular problemas y explorar posibles soluciones, involucrarse y en definitiva hacerles sentirse partícipes del desarrollo. Durante las sesiones, los participantes exponen sus ideas, las cuales se discuten, analizan y refinan hasta alcanzar un acuerdo. Para ayudar a que el trabajo en estas sesiones proporcione resultados provechosos, se hace especial énfasis en las dinámicas de grupo, en el uso de ayudas visuales que permitan mejorar la comunicación, en el mantenimiento de un proceso organizado y racional, y en un tratamiento de la documentación basado en trabajar directamente sobre los documentos a generar. Algunas de las ventajas que aporta, en comparación con las entrevistas individuales, son:

- Ahorra tiempo: las opiniones de los clientes no se contrastan por separado.
- Todo el grupo, incluyendo los clientes y los futuros usuarios, revisa la documentación generada, no sólo los ingenieros de requisitos.
- Implica más a los clientes y usuarios en el desarrollo.

3. Inicio: sirve de preámbulo a la entrevista propiamente dicha. Comienza con una exposición general de cómo se desarrollará, sus objetivos y duración estimada. A continuación se explica brevemente al entrevistado la utilidad de la entrevista y se pide su colaboración. Finalmente se solicita su autorización para tomar notas.
4. Desarrollo: esta es la fase central de la entrevista, que consiste básicamente en la realización de preguntas por parte del entrevistador y la contestación de las mismas por parte del entrevistado. Las preguntas deben apuntar a la obtención de la información deseada, pero permitiendo que el entrevistado siga su línea de pensamiento y

exposición sin interrupciones. Un buen entrevistador sabe guiar la conversación, invitando a entrar en detalles cuando es necesario, e introduciendo momentos de síntesis con recapitulaciones. Deben evitarse controversias o críticas, así como sugerencias precipitadas. El objetivo es recabar toda la información que permita posteriormente entender el problema.

5. Cierre: se trata de la fase donde se resume la información anotada y se comprueba que contamos con toda la información solicitada. Sin embargo, debe dejarse abierta la posibilidad de entrevistas posteriores que permitan aclarar o detallar algún aspecto relevante que no ha sido tratado. La entrevista finaliza agradeciendo al entrevistado su colaboración y, según las circunstancias, ofreciéndole la posibilidad de entregársela una copia del documento resumen de la entrevista.
6. Conclusiones: tras la entrevista propiamente dicha debe elaborarse un resumen formal de la misma, incluyendo conclusiones que, en caso de duda, pueden ser confirmadas en entrevistas posteriores.

Las preguntas serán diversas, pero clasificadas dentro de los siguientes tipos:

- Preguntas sobre detalles específicos.
- Preguntas sobre la visión de futuro que el entrevistado tiene sobre algo.
- Preguntas sobre ideas alternativas.
- Preguntas sobre una solución mínimamente aceptable.
- Preguntas acerca de otras fuentes de información.

Las entrevistas proporcionan información de primera mano que será de gran ayuda en el resto del desarrollo. Además, puede solicitarse al entrevistado tanto nivel de detalle en la información como sea necesario. Tienen además la ventaja de que si la persona entrevistada no tiene más que decir, se pueden dar por terminadas en cualquier momento. Sin embargo, podemos contar entre sus desventajas que se trata de una técnica que requiere mucho tiempo, tanto en la propia entrevista en sí como en el análisis posterior de la información recabada. Además, la opinión de las personas entrevistadas puede estar sesgada y no proporcionar información tan útil como sería deseable. Finalmente, deberá tenerse en cuenta que los conflictos entre la información aportada por diferentes entrevistados, cuando es contradictoria, son difíciles de resolver.

4.6.2 Análisis de requisitos

El análisis de requisitos describe las tareas que deben llevarse a cabo para examinar los requisitos con el objeto de delimitarlos y definir exactamente cada uno de ellos. Según la guía SWEBOK, se trata fundamentalmente de:

- Detectar y resolver conflictos entre requisitos.

- Delimitar el software y establecer con qué elementos externos interacciona.
- Elaborar los requisitos del sistema para obtener, a partir de ellos, los requisitos del software a desarrollar.

El análisis de los requisitos enlaza la definición de alto nivel del sistema desde la perspectiva del usuario –perspectiva necesariamente externa–, con el diseño del software que permitirá implementar dicho sistema –una visión interna de cómo se comportará el software-. Es absolutamente imprescindible entender exactamente qué tienen en mente los clientes y usuarios para, basándonos en la especificación de dichas necesidades, construir el sistema que los clientes desean y no otro.

Una visión simplista del análisis de requisitos, aunque muy extendida, establece que este proceso consiste exclusivamente en realizar un modelo conceptual de los requisitos. Sin embargo, y aunque el modelado es importante, veremos más adelante que no es la única tarea a realizar. La clasificación de los requisitos, el diseño de la arquitectura del sistema y la situación de los requisitos en la misma, así como la toma de decisiones de compromiso en los casos en que surgen conflictos entre varios requisitos, son tareas igualmente relevantes.

Clasificación de los requisitos

La *clasificación de los requisitos* consiste en establecer un conjunto de categorías y situar cada requisito en ellas. Como se estableció en el Apartado 4.5 de este mismo capítulo, la forma de clasificar los requisitos no es siempre la misma, existiendo diversos criterios con diferente nivel de aceptación. Así, pueden clasificarse en funcionales o no funcionales, del proceso o del producto, por prioridad, derivados (por ejemplo, de otro requisito) o impuestos (por ejemplo, por el sistema existente), por su ámbito, por su volatilidad, etc.

En cuanto a la priorización, debe determinarse la importancia relativa de cada requisito en relación con los demás. Para computar este valor de importancia, habitualmente se utiliza el método de preguntar a clientes y usuarios acerca del grado de satisfacción que les proporcionaría la implementación de las funcionalidades, restricciones o calidades del requisito, junto con el grado de insatisfacción que les produciría el que dicho requisito no fuera implementado. Utilizando estos datos como entrada del cálculo, se obtendría un valor de prioridad para el requisito.

Modelado conceptual

El modelado conceptual tiene como objetivo fundamental facilitar la comprensión de los requisitos mediante su representación en un lenguaje o notación que «comprendan» quienes van a tratar con los requisitos. El modelado no es parte de la elaboración de una solución: se trata simplemente de una representación que permite comprender el problema. Sin embargo, comunicar nuestro entendimiento a otros para mejorar su comprensión del sistema a través de un modelo útil constituye una difícil tarea para la que se requiere experiencia.

La elaboración de un modelo conceptual puede enfocarse desde muy diferentes perspectivas, cada una de ellas aportará información de un aspecto diferente, pero complementario con los demás, del sistema. Así, será posible elaborar modelos conceptuales de datos (que muestran las entidades del dominio del problema, sus relaciones y dependencias), de flujo de control (donde se describen las interacciones de procesos) y de datos, modelos de estados (que muestran comportamientos dentro del sistema mediante estados y transiciones), modelos de interacción, modelos de traza de sucesos, o modelos de objetos, entre otros.

Modelado de requisitos en métodos ágiles

Los métodos ágiles llevan a cabo un modelado de requisitos adaptado a las singularidades de este tipo de desarrollos. Al principio del proyecto se realiza el modelado de requisitos estrictamente necesario para conocer los requisitos del sistema a alto nivel. A partir de este punto, se propone ir obteniendo detalles sobre los requisitos a medida que van siendo necesarios. Tampoco existe ninguna planificación para aquellos requisitos que pudieran aparecer en el futuro, ni se diseña pensando en requisitos futuros. Renunciando a la documentación de los requisitos al estilo «tradicional», los métodos ágiles encapsulan los requisitos como casos de prueba a los que habrá que someter al código una vez construido. Todo ello pretende promover el seguimiento y establecer correspondencias entre los requisitos y el código que implementa la funcionalidad o calidad requeridas.

Al principio del proyecto se dedican varios días a entender tanto los requisitos de alto nivel del sistema como el alcance de la entrega actual (los métodos ágiles conciben el desarrollo como una iteración continua en torno al concepto de entregas sucesivamente más completas). Con ello se pretende conocer qué debe hacer el sistema. En este esfuerzo, no se dedica tiempo documentar detalladamente todo aquello que deberá hacer el sistema para evitar retrasar el momento en que comienzan las tareas de construcción, pero esto no significa que se renuncie completamente a la documentación: si es necesario más adelante, puede realizarse *a posteriori*.

En cuanto al modelado, se recomienda utilizar uno o más de los siguientes modelos:

- Un modelo de uso que permita conocer cómo se lleva a cabo la interacción de los usuarios con el sistema. Se plasmará en diagramas de casos de uso.
- Un modelo inicial del dominio, que identifique las entidades fundamentales y sus relaciones con un nivel de detalle justo para entender los principales conceptos. Se plasmará en uno o más diagramas de clases.
- Modelos de interfaz de usuario, que consistirían en modelos iniciales de las pantallas o en un prototipo de interfaz.

El número de modelos a crear y el nivel de detalle dependerá en gran medida de la complejidad del proyecto. La regla de oro es utilizar aquello que es estrictamente necesario para entender los requisitos y nada más que esto. El objetivo es, como ya se ha dicho antes, de comenzar la construcción cuanto antes.

Para el modelado conceptual de los requisitos pueden utilizarse diferentes notaciones, cualquiera a tal efecto pues no se ha demostrado que unas sean mejores que otras. No obstante, la existencia de un lenguaje común de modelado facilita enormemente la comuni-

cación y evita esfuerzos de formación pues elimina la necesidad de conocer cada notación particular de cada proyecto en el que se participa. A este respecto, es importante resaltar el papel de UML (*Unified Modeling Language*), verdadera *lingua franca* en el modelado asociado al desarrollo de software. El uso de un lenguaje de modelado como UML proporciona la ventaja de no admitir ambigüedades, eliminando así la existencia de diferentes interpretaciones, si bien el uso de otras notaciones formales (por ejemplo en algunos dominios muy especializados, han tenido cierto éxito notaciones derivadas de la matemática discreta) es aún muy extendido. No obstante, algunos autores abogan por el uso de métodos «formales ligeros» (*lightweight formal methods*) para el modelado, aduciendo razones tales como que los métodos formales son difíciles de comprender o que su aplicación implica costes y retrasos innecesarios. La Sección 4.7 aborda con mayor detalle las notaciones más comúnmente utilizadas para el modelado conceptual de los requisitos de un sistema.

Situación de los requisitos en la arquitectura del sistema

En un cierto momento del desarrollo debe establecerse qué elementos del sistema software van a ser los responsables de satisfacer las demandas planteadas por los distintos requisitos. A estas tareas es a lo que se denomina *situación de los requisitos en la arquitectura del sistema*. Así, por ejemplo, será necesario establecer qué componente de un sistema de control domótico integrado va a tratar con los requisitos relativos a la climatización. Una vez hecho esto, habrá que analizar cada requisito específico (medición de las temperaturas, secuencia de puesta en marcha de los aparatos de calefacción o aire acondicionado, determinación por habitaciones de qué aparatos deben ponerse en marcha y a qué temperaturas, etc.) para establecer qué parte del sistema de climatización debe satisfacer la necesidad que el requisito representa. A menudo esta tarea de situación de requisitos en la arquitectura permite descubrir nuevos requisitos derivados de la implementación de un conjunto concreto de requisitos por parte de un componente específico del sistema, por lo que resultan de especial importancia.

Negociación de los requisitos

Muchos autores consideran que la etapa de análisis de requisitos debería en realidad denominarse etapa de «Análisis y negociación de requisitos». Esto es así porque durante el análisis de los requisitos surgen a menudo conflictos de intereses (o de otro tipo) entre las diferentes partes involucradas. En la negociación de los requisitos, se deberán detectar y resolver posibles conflictos entre requisitos, definir de manera precisa cuáles son los límites del sistema y cómo éste debe interaccionar con su entorno. La resolución de conflictos tiene que ver con la eliminación de los problemas entre las diferentes necesidades (plasmadas en requisitos) de los actores implicados en el proyecto: solicitud de características incompatibles, conflictos entre requisitos funcionales y no funcionales, y otras. La solución más habitual es tomar una decisión de consenso que satisfaga suficientemente a todos los interesados, aunque todos ellos deban ceder, en cierta medida, en sus pretensiones iniciales.

Técnicas de detección y resolución de conflictos

Existen diferentes técnicas para resolver los posibles conflictos en los requisitos, muchas de las cuales tienen en cuenta diferentes aspectos sociales y psicológicos útiles en el proceso de negociación. Algunos autores afirman incluso que es posible beneficiarse de los conceptos de la inteligencia emocional –que podría definirse como la capacidad para entender las emociones y manejarlas adecuadamente– durante este proceso de negociación. Lo cierto es que cualquier técnica que ayude a descubrir problemas con los requisitos y a facilitar el acuerdo entre todos los involucrados será de ayuda en esta etapa.

El modelo de siete capas del papel del facilitador

Uno de los modelos más interesantes acerca de la resolución de conflictos durante las actividades de requisitos es el denominado *modelo de siete capas del papel del facilitador* (Macaulay, 1999). Se trata de un modelo en el que una persona, el facilitador, es elemento central de la negociación y resolución de conflictos entre las partes interesadas. En este modelo, un facilitador es una persona que:

- Resuelve problemas de diferencias (en la información, en las creencias, en los intereses o en los deseos), escasez de recursos (como dinero, tiempo, espacio, etc.), de rivalidad entre personas o grupos, de dificultades de las tareas a realizar, de presión para evitar fallos, etc.
- Controla las actividades, observando y controlando la progresión de cada grupo, monitorizando los problemas y buscando patrones que permitan aplicar una solución ya conocida.
- Coordina el proyecto, conociendo a todos los interesados, analizando sus intereses y encargándose de aspectos prácticos importantes no estrictamente relacionados con la obtención y análisis de requisitos.
- Hace más sencilla la comunicación entre personas, prepara las reuniones y el entorno en que se desarrollarán, y en último término facilita los acuerdos.

Un ejemplo del tipo de situaciones en que la figura del facilitador resulta útil lo constituyen las sesiones de desarrollo conjunto de aplicaciones JAD, donde representantes de los distintos grupos involucrados en el desarrollo participan en discusiones para obtener requisitos, analizar sus detalles y descubrir implicaciones transversales a varias funcionalidades.

El objetivo final de la negociación de los requisitos es eliminar redundancias (un requisito semánticamente equivalente especificado desde diferentes perfiles o dominios), inconsistencias y puntos de desencuentro (requisitos que son entendidos de forma diferente por clientes, usuarios y desarrolladores). Al final de esta actividad se obtiene un conjunto de requisitos aceptados, que deberían especificarse con detalle y documentarse. Tras la especificación y documentación de los requisitos aceptados se genera un borrador del documento de requisitos que deberá ser validado por todas las partes involucradas antes de que pueda tratarse de documento de requisitos validados. Dicho documento se supone lo suficiente

Tabla 4.2: Tabla para detectar conflictos de requisitos

Requisito	R1	R2	R3	R4	R5
R1	0	1	0	1.000	0
R2	1	0	1.000	1	0
R3	0	1.000	0	0	0
R4	1.000	1	0	0	0
R5	0	0	0	0	0

temente maduro como para poder ser utilizado como base del proceso de desarrollo. Sin embargo, este documento no permanecerá inalterable a lo largo del desarrollo; es más que probable que evolucione durante el resto del ciclo de vida con los nuevos requisitos que se vayan incorporando o con modificaciones a algunos ya incluidos.

Como parte importante durante la detección de conflictos dentro del análisis de los requisitos, la técnica de las matrices o tablas de doble entrada es una de las más ampliamente utilizadas. Este método consiste en ordenar los identificadores de los requisitos en filas y columnas, comparando cada requisito con cada uno de los restantes y poniendo en la celda correspondiente un valor entero según lo siguiente:

- Si hay conflicto poner un 1.
- Si hay solapamiento poner 1.000.
- Si hay independencia poner un 0.

Una vez completada la tabla, se suman los valores de filas y columnas y se divide cada suma entre 1.000. El resto de dicha división indica el número de conflictos para el requisito en cuestión, mientras que el cociente nos da el número de solapamientos de dicho requisito.

Obsérvese la Tabla 4.2 para la detección de conflictos y solapamientos en un conjunto de requisitos (cuyos códigos son R1, R2, ..., R5), cuyo objeto es ilustrar la utilización de la técnica descrita. Como resultado, se observa lo siguiente:

- R1 y R4 se solapan.
- R2 y R3 se solapan.
- R2 presenta conflictos con R1 y con R4.
- R5 es independiente.

Esta técnica se considera útil si el número de requisitos es inferior a doscientos. Si no es así, resulta preferible dividir el total de requerimientos en grupos funcionalmente homogéneos y aplicar la técnica separadamente a cada grupo. Para la confección de los grupos se aplican criterios de homogeneidad, creando diferentes categorías como entradas de datos, procesamiento, salidas de datos o subsistemas.

4.6.3 Especificación de requisitos

La especificación de los requisitos consiste en la completa descripción de los requisitos del sistema a desarrollar. En dicha especificación se incluye, para cada requisito, informa-

ción complementaria que permite gestionarlo e interpretarlo, información que a menudo se denomina «atributos de los requisitos».

La especificación de requisitos implica con frecuencia la construcción de un modelo (o varios) del sistema a construir desde el punto de vista de los usuarios del sistema, que incluya todos y cada uno de los requisitos obtenidos. Es esencial su realización, por ejemplo, cuando se elabora un anteproyecto (que suele incluir el documento de requisitos, uno o varios modelos, opcionalmente un prototipo, una estimación de esfuerzo y plazo y un presupuesto), pues facilita la elaboración una estimación de costes y tiempos.

Según el estándar 830-1998 de IEEE para la especificación de requisitos del software, una especificación debe ser:

- Completa.
- Verificable.
- Consistente.
- Modificable.
- Susceptible de permitir seguimientos.
- Utilizable durante las fases de operación y mantenimiento.
- Y además, no debe contener ambigüedades.

Otras propiedades que a menudo se mencionan son la consistencia interna y externa, el tamaño reducido, o la no inclusión de redundancias.

De manera generalizada, los requisitos se plasman en un documento denominado especificación de requisitos del software, comúnmente conocido por sus siglas en inglés SRS (*Software Requirements Specification*). Para sistemas sencillos la elaboración de un único documento puede ser suficiente, pero en sistemas complejos es habitual la elaboración de tres documentos distintos como parte de esta actividad: el documento de definición del sistema, el documento de requisitos del sistema y finalmente la especificación de requisitos del software (SRS). A continuación se estudian con algo más de detalle estos tres documentos.

Documento de definición del sistema

A veces denominado también documento de requisitos del usuario, establece los requisitos de alto nivel del sistema desde el punto de vista del dominio del problema. Dirigido a los usuarios y clientes del sistema, detalla los requisitos generales según los objetivos que se intentan alcanzar e incluye generalmente una introducción al entorno y objetivos, la lista de participantes, la definición de los principales conceptos del sistema, los requisitos funcionales y no funcionales de alto nivel, las limitaciones que deben observarse, los roles y las principales actividades de cada rol identificado. Puede incluir también modelos conceptuales que permitan comprender el entorno del sistema o escenarios de utilización.

Documento de requisitos del sistema

Constituye una especificación formal de los requisitos del sistema, de los que se derivarán los requisitos del software a desarrollar. Sus principales objetivos consisten en proporcionar una aproximación al contexto, alcance y capacidades del sistema, especificar los interfaces del mismo, así como sus requisitos (incluyendo las restricciones aplicables), documentando cualquier mejora futura y los posibles puntos aún no cerrados o incompletos acerca del mismo. Sirve como marco para realizar, entre otras actividades, estimaciones de coste y complejidad del software a desarrollar, o planificaciones para el sistema descrito. Para comprender mejor la diferencia entre un documento de requisitos del sistema y un documento de especificación de requisitos del software, deben tenerse en cuenta algunos de los puntos a incluir en el primero:

- Funcionalidades del sistema.
- Especificaciones de rendimiento.
- Requisitos de seguridad.
- Definición de la interfaz de usuario.
- En algunos sistemas específicos, definición de qué tareas serán llevadas a cabo por el software y cuáles por el hardware.

Documento de especificación de requisitos del software (SRS)

Define explícitamente todo aquello que el software debe realizar, así como las restricciones que debe contemplar el futuro sistema. También debería incluir todo aquello que el software no debe realizar, sirviendo como acuerdo formal (a menudo con implicaciones contractuales) entre los clientes y los proveedores del software. Los requisitos del software, que a menudo se detallan en lenguaje natural, son descritos de manera más precisa en el SRS utilizando alguna notación de modelado u otro lenguaje de especificación. Las buenas prácticas de la ingeniería del software recomiendan por tanto completar este documento con un modelo (utilizando notaciones como UML, por ejemplo).

Es importante observar que un documento de especificación de requisitos software contiene sólo requisitos funcionales y no funcionales, pero no sugerencias sobre cómo diseñar el futuro sistema, posibles soluciones a cuestiones de negocio o tecnológicas, o cualquier otra información fuera de la que implica la comprensión por parte del equipo de desarrollo de cuáles son los requisitos del cliente para el sistema. Esto constituye además la diferencia esencial con el documento anterior. Además, es importante tener en cuenta que según la organización de desarrollo, el contenido del SRS variará ligeramente. No obstante, es posible identificar un conjunto de contenidos comunes que deben tenerse en cuenta siempre que se diseña y escribe un SRS. Como referencia a la hora de enfrentarse a esta tarea, el estándar IEEE 830-1998 para la elaboración de documentos de especificación de requisitos recomienda tratar los siguientes puntos:

- Capacidad funcional: ¿qué deberá hacer el software?

- Interfaces: ¿cómo interacciona el software con los usuarios, con el hardware sobre el que funcionará y con otros elementos hardware y software externos?
- Niveles de rendimiento: ¿qué se requiere en términos de velocidad, disponibilidad, tiempo de respuesta, recuperación ante fallos, etc.?
- Atributos: ¿Qué consideraciones de seguridad, fiabilidad, protección de datos y privacidad, o calidad deben tenerse en cuenta?
- Restricciones y limitaciones de diseño: ¿existen estándares u otras especificaciones que deban seguirse durante el desarrollo?, ¿qué políticas sobre integridad de la base de datos, lenguajes de implementación, entornos de operación, etc. existen?

Según este estándar, un documento de especificación de requisitos software debe permitir el seguimiento de los requisitos, ser correcto, evitar ambigüedades, ser completo y consistente, estar ordenado por importancia y/o estabilidad, ser verificable y también modificable. Sin embargo, a la hora de plasmar estas recomendaciones generales en un documento de especificación de requisitos software en concreto, surgen bastantes dudas. ¿Por dónde empezar? ¿Qué incluir específicamente? ¿Cómo estructurar la información? Para abordar con confianza el trabajo detallado de elaboración del SRS, es muy recomendable utilizar una plantilla, conjuntamente con un método para enlazar los requisitos con las fuentes que los originaron y una matriz de seguimiento.

Una de las tareas más importantes es elegir una plantilla adecuada que permita organizar los contenidos y ajustarlos a las necesidades del desarrollo en curso. Téngase en cuenta que no existe una única plantilla para completar el SRS, pues los requisitos que lo componen son únicos, no sólo de una organización a otra, sino incluso de un proyecto a otro dentro de la misma organización. La clave es elegir una plantilla para comenzar e irla adaptando según sea necesario. La Tabla 4.3 muestra cómo sería una plantilla de SRS según el estándar 830-1998 de IEEE. En dicha figura está poco desarrollado el apartado 3, *requisitos específicos*, pues el estándar permite una ordenación de los requisitos a medida de las necesidades de cada proyecto. Para aquellos interesados en el detalle de contenidos en dicho apartado, se recomienda acudir al estándar, donde se analizan cuidadosamente los diferentes puntos que habría que tratar en esta parte.

Algunos autores se refieren al SRS como al «documento padre», puesto que el resto de documentos de gestión del proyecto, esto es, las especificaciones de diseño, la planificación del trabajo, la especificación de la arquitectura, los planes de pruebas y validación y otros, están relacionados con (y en cierto modo dirigidos por) su contenido. No obstante, en algunos enfoques de desarrollo modernos como los métodos ágiles, la concepción de lo que debe ser un documento de requisitos es ligeramente diferente al enfoque «tradicional». Según esta filosofía, la posibilidad de observar anticipadamente cómo se comportan las funcionalidades en un prototipo ofrece una información muy interesante que es fuente de ideas imposibles de concebir antes de comenzar ningún desarrollo, y que por tanto difícilmente podrían haberse incluido en un documento de requisitos detallados antes de comenzar el

Tabla 4.3: Plantilla para un SRS según el estándar IEEE 830-1998**Tabla de contenidos**

- 1. Introducción**
 - 1.1. Propósito
 - 1.2. Alcance
 - 1.3. Definiciones, siglas y abreviaturas
 - 1.4. Referencias
 - 1.5. Visión general
- 2. Descripción de conjunto**
 - 2.1. Perspectiva del producto
 - 2.2. Funcionalidades
 - 2.3. Características de usuario
 - 2.4. Restricciones
 - 2.5. Asunciones y dependencias
- 3. Requisitos específicos**
- Apéndices (opcional)**

proyecto. Estos métodos de desarrollo no afirman que no haga falta un documento de requisitos, pero se resalta que son menos importantes que los productos que funcionan. De este modo, abogan por reducir al mínimo indispensable el uso de documentación, considerando que el generarla implica realizar un trabajo que no aporta valor directo al producto final.

La importancia de una correcta especificación

Para comprender la importancia de la especificación de los requisitos, téngase en cuenta que el documento final de especificación de requisitos del software sirve como contrato entre el cliente que solicita el software y los desarrolladores del mismo. Puesto que representa un acuerdo contractual entre las diferentes partes involucradas, los contratos de desarrollo incluyen condiciones, acuerdos y otras cláusulas que se basan en lo incluido en este documento, como por ejemplo el detalle sobre los productos a entregar, los hitos del desarrollo y las fechas de entrega. Cualquier desviación, es decir, cualquier cosa que el sistema final no contemple y que haya sido catalogada dentro del conjunto de funciones solicitadas, podrá ser denunciada por el cliente como un incumplimiento del contrato entre ambos.

Las causas principales de conflictos legales entre desarrolladores y clientes son la entrega de un producto inadecuado o incompleto, la inclusión de errores en el software, la falta de cobertura de los requisitos de usuario, los retrasos en la entrega y los sobrecostes. Por ello, y dado que el documento de especificación de requisitos (SRS) debería incluir solamente aspectos relativos al producto, y no al proceso de producción, se recomienda no incluir en el SRS detalles sobre costes, plazos de entrega, procedimientos de generación y

publicación de informes, métodos de desarrollo de software, aseguramiento de la calidad, criterios de validación y verificación, y procedimientos de aceptación. Estos datos deberían por el contrario incluirse en otro tipo de documentos (en el plan de desarrollo, en el de aseguramiento de la calidad o en la descripción de trabajos), donde se hicieran explícitos aquellos acuerdos entre clientes y desarrolladores que pudieran tener implicaciones legales.

¿Es siempre necesario un documento de especificación de requisitos?

Esta pregunta se plantea a menudo en el ámbito académico cuando los alumnos de las titulaciones de informática comienzan a estudiar ingeniería del software, pero también en otros ámbitos. Acostumbrados a programar aplicaciones pequeñas, meros ejemplos didácticos deliberadamente simplificados y pensados para ser desarrollados por una sola persona, los alumnos no alcanzan a ver cómo puede ser necesario un documento de requisitos para el tipo de desarrollo al que están acostumbrados. Y en cierto modo tienen razón, en casos tan simples no es necesario elaborar un documento formal de requisitos. Si la aplicación a desarrollar es lo suficientemente pequeña como para ser desarrollada por una sola persona, y si la población de usuarios es también pequeña, tal como unas pocas personas o un único grupo de trabajo, es posible llegar a un acuerdo común sobre qué deberá hacer la aplicación sin necesidad de un documento formal escrito. En todos los demás casos, es necesario un documento de requisitos escrito. En pocas palabras, habrá que desarrollar un documento de requisitos siempre y cuando alguna de las siguientes situaciones sea cierta para su aplicación:

- Transcurrirá más de un mes desde la concepción del proyecto hasta su puesta en producción.
- Trabajará más de una persona en la aplicación.
- Será utilizada por más de un grupo de usuarios.

Como resumen, puede decirse que la estructura y nivel de detalle del documento de requisitos debe adaptarse a la naturaleza del software a desarrollar. Si éste es lo suficientemente sencillo, puede pasarse sin él, pero en la mayoría de los casos será necesario.

Sin embargo, existe un importante problema en este enfoque: los requisitos cambian. Así por ejemplo, el cliente tiende a entender que quería una cosa cuando en realidad pidió otra distinta porque realmente no entendió lo que se estableció en el contrato o porque di por sentado algo que los ingenieros de software ignoraron. O se detectan necesidades a lo largo del proyecto que no fueron reflejadas en el contrato pero que resulta obvio que deben ser incluidas y, para bien de los desarrolladores, presupuestadas. Por ejemplo, la especificación de requisitos se podría enriquecer durante el diseño con requisitos relacionados con el entorno tecnológico o el manual de usuario difíciles de identificar en etapas anteriores del desarrollo. O simplemente el propio desarrollo aconseja al cliente modificar los requisitos establecidos para mejorar el contrato, incluyendo elementos no contemplados, por lo que tras el visto bueno del cliente se procede a una modificación formal del contrato. Por ejemplo la inclusión de un medio de pago novedoso para un sistema de venta por internet surgido

en los últimos tiempos y que no fue incluido en la especificación inicial por inmaduro.

Para ilustrar lo anterior, comparemos la construcción de software y la construcción de un edificio. Durante la edificación, lo que se pidió al constructor (y lo que éste presupuestó y fue firmado por ambas partes) necesita ser modificado más a menudo de lo que el cliente quería. Así por ejemplo, si las tareas de excavación previas a la cimentación del edificio descubren una zona de inestabilidad que obliga a excavar más profundo, esto implicará un coste que deberá asumir el cliente. Sin embargo, no sólo la construcción de una vivienda, sino la reforma de una ya construida (lo que podría tener un paralelismo con los requisitos para la realización de una tarea de mantenimiento dentro del desarrollo de software) se encuentra a menudo con situaciones similares. Un pintor ha presupuestado un trabajo de pintura a partir del número de metros cuadrados de pared. No obstante, si cuando se encuentra a mitad de realizar su trabajo el yeso está demasiado viejo, se resquebraja y para finalizar su trabajo con garantías debe picar la pared, sanearla y enyesar de nuevo, dichas tareas deberán acometerse y presupuestarse aparte, lo cual constituye un buen ejemplo de requisito oculto no presupuestado inicialmente.

4.6.4 Validación de requisitos

La validación de los requisitos consiste en examinar si los documentos de requisitos definen el software que los usuarios esperan y no otro. Al igual que cualesquiera otros productos del proceso de desarrollo, los documentos de requisitos están sujetos a procesos de validación y verificación para comprobar, entre otras cuestiones, que el ingeniero de software ha comprendido los requisitos, que los documentos son acordes a los estándares establecidos o que dichos documentos son consistentes, comprensibles y completos.

Todo ello puede realizarse, de manera más sencilla, si se emplean lenguajes de especificación o notaciones de modelado para llevar a cabo su elaboración. Los métodos más comúnmente empleados para la validación de requisitos son los siguientes:

- *Revisión de los requisitos*: un grupo de personas, designado especialmente para tal fin (y que a menudo incluye algún representante del cliente), revisa los documentos de requisitos en busca de inconsistencias, malentendidos, puntos poco claros, conflictos entre requisitos y otros problemas similares. Como resultado de este proceso, se elabora y publica una lista de problemas y posibles soluciones.
- *Prototipado*: el desarrollo de un prototipo constituye una inmejorable forma de probar cualquier producto. Aplicado a la validación de requisitos, el prototipado permite mostrar el funcionamiento de los requisitos, sirviendo de gran ayuda a la hora de descubrir problemas en los mismos y de clarificar a otros interesados algunas de las asunciones realizadas por los ingenieros de software.
- *Validación del modelo*: se lleva a cabo para verificar si el modelo es consistente y si refleja adecuadamente los requisitos reales del sistema. Cuando en la elaboración de los documentos de requisitos se han utilizado notaciones de modelado o cualquier

otro lenguaje de representación, es posible introducir cierto grado de automatización a la hora de validar el modelo, lo que apoya la utilización de este tipo de notaciones.

- *Pruebas de aceptación:* en secciones anteriores se enunció como propiedad esencial de los requisitos la verificabilidad, es decir, su capacidad para permitir comprobar si el producto terminado satisface o no el requisito. Las pruebas de aceptación de requisitos consisten en la elaboración de un plan que establece cómo deben ser verificados los diferentes requisitos. Este procedimiento resulta útil para detectar problemas ya que por ejemplo, para aquellos requisitos que estén descritos de manera ambigua, es difícil elaborar un plan de verificación de los mismos. A su vez, este procedimiento proporciona ideas para la resolución de los problemas encontrados.

Revisión de los requisitos

La validación de requisitos mediante la técnica de revisión se lleva a cabo gracias a un grupo de revisores que habrá de constituirse de modo que cuente con representación de todos los actores (o al menos de los más significativos), cada uno de los cuales cumplimenta una tabla con sus análisis acerca de los requisitos. La composición ideal del grupo de revisores debería contar con uno o varios usuarios, un responsable del cliente, desarrolladores, algún analista de requisitos y varios expertos funcionales en el problema. Además de todos los anteriores, es conveniente la participación –y suele sugerirse su inclusión– de algún experto externo que actúe como «abogado del diablo».

El objetivo principal de este grupo es buscar errores y contradicciones en los requisitos, descripciones poco claras o ambiguas y desviaciones de las prácticas estándar. Para ello, como se ha dicho, llenarán una tabla como la que se muestra en la Tabla 4.4.

Tabla 4.4: Tabla para la validación de requisitos

	Completo	Consistente	Comprendible	Ambiguo	Estructurado	Trazable
Req ₁		•				•
Req ₂			•			•
Req ₃	•			•		•
...						
Req _n		•	•		•	

La valoración de los requisitos generalmente se realiza de acuerdo a un cuestionario estandarizado que comparten todos los miembros del equipo y que les sirve de apoyo a la hora de validar los requisitos. Un ejemplo de cuestionario podría ser el de la Tabla 4.5.

Uso de prototipos

El prototipado consiste en la elaboración de un modelo del sistema que se construye (o de una parte del mismo) para evaluar mejor sus requisitos. Estos modelos pueden ser desde un conjunto de esbozos en papel que muestran no más que una vista estática de la interfaz

Tabla 4.5: Cuestionario de validación de requisitos

Conformidad con estándares	La especificación de requisitos en su conjunto, ¿es conforme a los estándares definidos? ¿es conforme a los estándares definidos cada uno de los requisitos individualmente?
Seguimiento	¿Pueden identificarse unívocamente los requisitos? ¿Incluyen referencias y/o enlaces a otros requisitos relacionados así como las razones para incluirlos?
Estructuración	¿Está estructurado el documento de requisitos? ¿Están agrupados los requisitos que están relacionados entre sí? ¿Sería más fácil de entender si tuviera otra estructura?
Ambigüedad	¿Son ambiguos algunos requisitos? ¿Pueden darse distintas interpretaciones de los mismos?
Comprensión	¿Son comprensibles los requisitos? ¿Puede un lector entender lo que significa cada uno de ellos? ¿Están libres de detalles de diseño e implementación?
Consistencia	¿Son consistentes los requisitos? ¿Hay alguna contradicción entre los distintos requisitos?
Complejidad	¿Es completo el conjunto de requisitos? ¿Falta algún requisito? ¿Es completo cada uno de los requisitos individualmente? ¿Falta alguna información?
Relevancia	¿Es cada requisito, individualmente, pertinente para el problema y su solución?
Gestionable	¿Están expresados los requisitos de modo que un cambio no afecte demasiado a los demás? En aquellos casos en que los cambios pueden afectar a otros requisitos, ¿se han identificado las dependencias?
Vialidad	¿Es posible implementar todos los requisitos con las técnicas, herramientas y recursos disponibles? ¿son viables dadas las restricciones de coste y plazos?

del sistema que se planea construir (conocidos como «prototipos de baja fidelidad»), hasta complejos modelos dinámicos formados por conjuntos de pantallas con ciertas funcionalidades restringidas del sistema (prototipos «de alta fidelidad»). Una definición formal del término prototipo es la siguiente:

Un prototipo es un modelo fácilmente ampliable y modificable de un sistema software donde se muestran su interfaz y las funcionalidades de entrada-salida más relevantes

Para la elaboración de prototipos pueden utilizarse herramientas software específicas. En la Figura 4.5 se muestra un prototipo de diseño de baja fidelidad de un diario en formato web realizado con una herramienta de este tipo: Denim. La utilización de herramientas software permite esbozar un prototipo de baja fidelidad con ciertas ventajas sobre la elaboración tradicional en papel, como la posibilidad de guardar en formato digital el prototipo para su distribución, copia, etc. o la facilidad de exportar el resultado a código HTML.

Otras clasificaciones dividen los prototipos según la cobertura del sistema, en globales y locales; según el objeto con el que han sido creados, en exploratorios, operacionales y experimentales; y según el nivel de detalle mostrado, en verticales y horizontales.

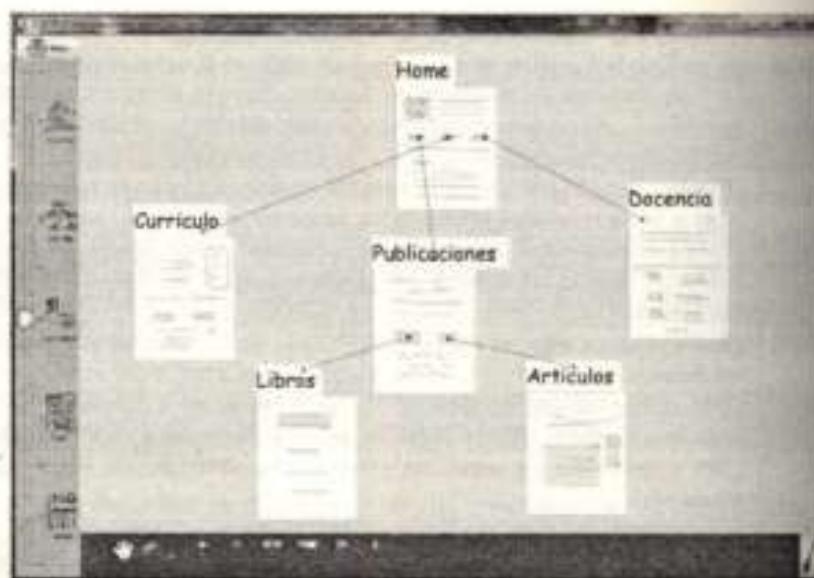


Figura 4.5: Prototipo de baja fidelidad realizado con la herramienta Denim

Las dimensiones del prototipado son:

1. **Capacidad de ejecución.** Indica si el prototipo es ejecutable o no. En caso afirmativo, habrá que establecer de qué tipo de prototipo ejecutable se trata:
 - **Prototipo guiado.** Versión en papel de la interfaz que permite mostrar al usuario cómo respondería el sistema a la interacción, más o menos de la siguiente forma «usted hace clic aquí y aparece esta pantalla».
 - **Animación.** Es ejecutable en el sentido de que lo hace diapositiva a diapositiva como una animación en una computadora.
 - **Prototipo «Mago de Oz».** Alguien entre bambalinas dirige la ejecución y proporciona los resultados a conveniencia. El usuario no sabe que las respuestas que genera la aplicación como consecuencia de su interacción con la interfaz están siendo elaboradas por un humano en lugar de por el sistema informático.
 - **Interactivo.** Prototipo que se ejecuta en una computadora y responde a las entradas del usuario en tiempo real.
 - **Funcional.** Prototipo interactivo que realiza algunos de los cálculos esperados.
2. **Madurez.** En aquellos prototipos pensados para evolucionar, indica en qué estado de desarrollo con respecto al sistema final se encuentra.
3. **Representación.** Se refiere a la fidelidad que alcanzará el prototipo con respecto al componente o sistema a construir.

4. Alcance. Determina las áreas de funcionalidad a las que se limita el prototipo. Si alcanza todas, se dice que es un prototipo del sistema completo.

El uso de prototipos durante las actividades de requisitos es útil tanto para descubrir de nuevos requisitos para el sistema, como para validar la interpretación que los ingenieros de software han hecho de los requisitos obtenidos. En función del tipo de ciclo de vida en que se haga uso de los prototipos, existen diferentes técnicas:

- En los modelos de ciclo de vida en cascada se utilizan los prototipos indistintamente en cualquiera de las fases, no sólo en la etapa de requisitos.
- En el modelo de prototipado rápido se busca mostrar funcionalidades lo antes posible a través del uso de prototipos. El prototipo es una primera versión del sistema con funcionalidad limitada.
- En el modelo de prototipado evolutivo se elaboran prototipos como desarrollo de pequeñas partes del sistema completo cuya funcionalidad ha sido completada. El objetivo es que el usuario proporcione información que permita refinar el prototipo e ir añadiendo funcionalidades al mismo progresivamente.

4.7 Notaciones para el modelado conceptual

En un modelo conceptual cada entidad se corresponde de manera unívoca con un objeto en el mundo real. Son modelos formales que, a menudo, se plasman en modelos visuales. En ellos se emplea una notación establecida que permite comunicar fácilmente el modelo a otras personas distintas a los autores del mismo. Para crear estas representaciones es necesario un lenguaje y un conjunto de reglas que establezcan cómo se han de usar los elementos del lenguaje a la hora de construir modelos.

El modelado conceptual de los requisitos mediante una notación formalizada y estándar es una práctica habitual que tiene un buen número de beneficios. Entre otros, ayuda a comprender los requisitos, revela inconsistencias, y permite la discusión entre diferentes actores del proceso de requisitos a la vista de los diagramas que conforman el modelo. Actualmente existen infinidad de notaciones y especificaciones de diseño, algunas de las cuales están específicamente orientadas al modelado de requisitos mientras que otras simplemente pueden adaptarse para ello aunque no hubieran sido originalmente concebidas para tal fin. En las siguientes secciones se exponen brevemente algunas de las más utilizadas.

4.7.1 Casos de uso

Dentro de las notaciones disponibles, posiblemente la notación de casos de uso de UML es la más ampliamente utilizada para el modelado y captura de requisitos funcionales. Hoy en día es parte esencial del lenguaje de modelado unificado (UML) y su utilización se enfoca a muchos aspectos de la ingeniería del software, no sólo a aquellos directamente relacionados

con los requisitos. No obstante, sus aplicaciones más comunes son el modelado del contexto de un sistema y el modelado y validación de requisitos.

Esta técnica permite modelar una vista externa de un sistema y servir de base de comunicación entre diferentes actores involucrados en el desarrollo. Su utilidad principal es definir el comportamiento de una entidad (un sistema o subsistema) sin entrar en la especificación de su estructura interna, siendo sus elementos centrales los casos de uso y los actores.

Actor

Los actores representan entidades externas al sistema. De modo similar a lo visto en la Sección 4.4 sobre los actores de requisitos, aquí cada actor modela un conjunto coherente de roles que los usuarios (en esta ocasión de los casos de uso) desempeñan al interactuar con éstos. Representan por tanto categorías de usuarios o sistemas con los que se interacciona, si bien no tienen por qué ser humanos: otros sistemas o hardware externo se definen también como actores. Los actores interactúan con el sistema mediante mensajes: se dice que «se conectan» a los casos de uso mediante asociaciones que permiten la comunicación. Su representación gráfica es deliberadamente simple –un monigote– para facilitar la comprensión de personas como los clientes o los usuarios, que no tienen por qué conocer las notaciones para la representación de requisitos.

Caso de uso

Un caso de uso es la descripción de un conjunto de secuencias que representan la interacción de elementos externos con el sistema. Indican «qué» hace el sistema y no «cómo» lo hace, y se inicián por la interacción de un actor. Téngase en cuenta que el término «caso de uso», en sentido estricto, hace referencia a un tipo de caso, al que se ajustan las diferentes instancias (sucesos y comportamientos) de dichos casos de uso. Gráficamente, un caso de uso se denota mediante una elipse con un nombre representativo de la funcionalidad que proporciona.

Para completar la descripción de un caso de uso se deben tener en cuenta las circunstancias bajo las que se lleva a cabo (precondiciones y postcondiciones), los actores con los que interacciona, y finalmente los flujos de eventos principal y alternativos, que representan los escenarios que forman parte del caso de uso.

Escenarios

Cada caso de uso tiene al menos un escenario principal, que representa la secuencia de acciones más habitual. Es frecuente que existan además uno o más escenarios secundarios que representen secuencias de eventos alternativas. Estas secuencias de eventos se detallan mediante un conjunto de pasos numerados. Así por ejemplo, el caso de uso «Pagar con tarjeta de crédito el importe de un tramo de autopista de peaje», tendría la siguiente secuencia de pasos en su escenario principal:

1. El sistema solicita la introducción de la tarjeta de crédito
2. El usuario introduce la tarjeta
3. El sistema solicita el cargo del importe al consorcio bancario
4. El consorcio bancario confirma el cargo en la tarjeta
5. El sistema acepta la operación y expulsa la tarjeta
6. La interfaz indica al usuario que puede continuar su viaje

Un posible escenario secundario vendría definido por un error de saldo insuficiente en la tarjeta de crédito, lo que no permite hacer frente al pago:

1. El sistema solicita la introducción de la tarjeta de crédito
2. El usuario introduce la tarjeta
3. El sistema solicita el cargo del importe al consorcio bancario
4. El consorcio bancario deniega la operación por saldo insuficiente
5. El sistema muestra un mensaje de error y expulsa la tarjeta

Relaciones

Otro elemento importante en la notación de casos de uso son las relaciones. Las relaciones permiten representar tanto las asociaciones de comunicación entre un actor y un caso de uso, como otras asociaciones entre casos de uso. Gráficamente se muestran como líneas que unen los dos elementos relacionados, si bien la representación varía según el tipo de relación (ver Figura 4.6):

- Conexión o comunicación: la asociación entre un actor y un caso de uso, que representa la invocación o empleo de una funcionalidad del sistema.
- Generalización: un elemento hijo –caso de uso o actor– hereda el comportamiento de otro –caso de uso base o actor base– al que se denomina padre. Simplifica el trabajo y potencia la reutilización.

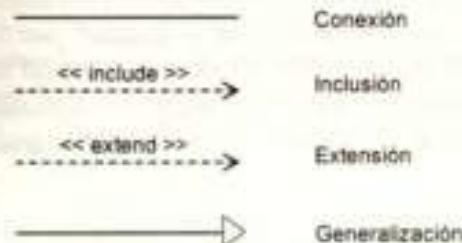


Figura 4.6: Representación de las relaciones en los diagramas de casos de uso

- Inclusión (*include*): un caso de uso base incorpora explícitamente otro caso de uso en un lugar indicado en el caso de uso base. Representa un comportamiento obligado.
- Extensión (*extend*): un caso de uso base incorpora implícitamente otro caso de uso en un lugar indicado en el caso de uso base. Representa un comportamiento opcional.

Representación detallada

Frecuentemente, la información relativa a un caso de uso particular se representa en una tabla donde se incluye el código del caso de uso, su nombre, los actores que intervienen, una descripción de sus escenarios y otras informaciones relacionadas, como los puntos de extensión. La Tabla 4.6 muestra en detalle la descripción de un caso de uso para la reclamación a una compañía de seguros de la prestación de un servicio cubierto por una póliza de seguro de hogar.

Ejemplo resumen

El siguiente ejemplo muestra el modelado de algunos requisitos de alto nivel de un sistema de compra *online* mediante casos de uso, el cual se representa en la Figura 4.7. En ese sistema, cualquier usuario puede consultar los precios de los artículos en venta o proceder a registrarse para acceder al resto de funcionalidades. Un usuario validado ante el sistema (el cliente) puede además acceder, en cualquier momento, a los datos de su carro de compra para –por ejemplo– modificar la cantidad elegida de un determinado artículo, eliminar un producto o simplemente listar por pantalla el contenido actual del carro.

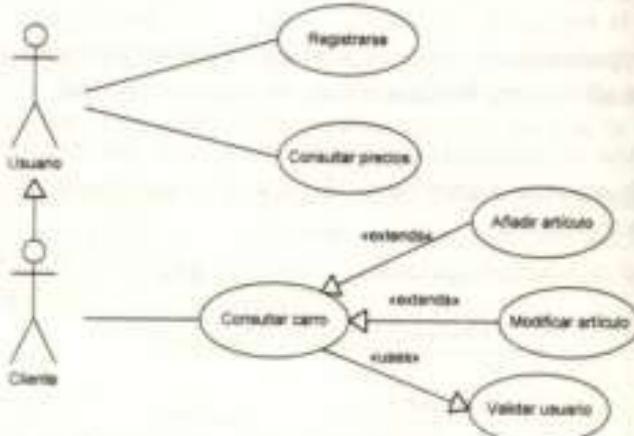


Figura 4.7: Modelado de requisitos mediante casos de uso

Los casos de uso son una notación muy utilizada para el modelado y captura de requisitos funcionales. Sin embargo, algunos autores han criticado ciertos aspectos de la misma.

com
usu
bas
men
com
plem

4.7.2

El m
mund
Desar

Tabla 4.6: Representación tabular de un caso de uso

Código: CU0015

Nombre: Reclamación de prestación

Actor(es): Asegurado

Descripción: Un cliente de una compañía de seguros (asegurado) comunica un desperfecto en su vivienda (siniestro) para que aquella se haga cargo de su reparación.

Precondiciones:

El asegurado tiene su póliza al corriente de pago

Se ha producido un siniestro en la vivienda asociada a la póliza

La póliza cubre el tipo de siniestro

Postcondiciones:

La compañía se hace cargo de la reparación del siniestro

Escenario principal:

1. El asegurado cumple con un parte de siniestro informando del siniestro
2. La compañía verifica la compleción de la documentación remitida
3. La compañía asigna un agente para examinar el caso
4. El agente de seguros comprueba que el siniestro efectivamente está cubierto por la póliza suscrita por el asegurado
5. La compañía procede a la prestación asegurada en la póliza

Escenario alternativo:

[Si tras el paso 2 la documentación remitida es insuficiente]

3. La compañía solicita al asegurado la información necesaria para tramitar el parte
4. Volver al paso 2 del escenario principal.

como que no proporciona soporte para recibir información de los desarrolladores o los usuarios, o que su especificación no incluye una guía acerca de cómo realizar el desarrollo basándose en los casos de uso. También se ha dicho de esta notación que enfoca deliberadamente el desarrollo desde un punto de vista funcional –lo cual es inapropiado en enfoques como la programación orientada a objetos– puesto que describe las funcionalidades a implementar pero no qué clase o clases serán las responsables de dicha implementación.

4.7.2 Modelos entidad-relación

El modelo de entidad-relación es una forma de modelado conceptual de datos donde el mundo se ve como un conjunto de objetos básicos, denominados *entidades*, y sus relaciones. Desarrollado en 1976 por Peter Chen, y desde entonces ligeramente refinado y ampliado.

es una de las formas más comunes de expresar los resultados del análisis de un sistema en etapas tempranas del desarrollo.

Aunque principalmente orientado al diseño de esquemas de bases de datos, permite representar la estructura lógica de cualquier sistema de datos. Los elementos fundamentales de un diagrama entidad-relación son por tanto las entidades, sus atributos y sus relaciones.

- Una **entidad** es un objeto que existe y que es distinto de otros objetos. Así por ejemplo, *Mr. Increíble* es un superhéroe, una entidad con su propia identidad y distinto por tanto a cualquier otra entidad existente. Las entidades pueden ser concretas (superhéroes, edificios, etc.) o abstractas (alarma, evento, etc.). Un conjunto de entidades comprende todas las entidades del mismo tipo, como por ejemplo el conjunto de todos los superhéroes. La representación de una entidad en un diagrama es un rectángulo etiquetado con el nombre de la entidad.
- Un **atributo** es una propiedad de una entidad, como por ejemplo el *nombre del superhéroe*, o si tiene o no la *capacidad para volar*. Todo atributo tiene un dominio que define el conjunto de valores que puede tomar la propiedad en diferentes entidades. El dominio del atributo nombre es «cadenas de caracteres», mientras que el de la capacidad de volar es un valor lógico: verdadero o falso. La representación de un atributo en un diagrama entidad-relación es una elipse que se une a la entidad a la que califica con una línea continua.
- Una **relación** es una asociación entre dos o más entidades, aunque formalmente se define como una asociación entre conjuntos de entidades. Ejemplos de relaciones son las que asocian a una persona con un vehículo del que es *propietario*, o la que une a los alumnos de una universidad con la universidad en la que *cursan* sus estudios. La relación más frecuente es la binaria, que se establece entre dos entidades. Las relaciones se representan mediante rombos etiquetados con el nombre de la relación.

La Figura 4.8 muestra, a modo de ejemplo, un diagrama entidad-relación simplificado para un sistema donde se almacena información sobre investigadores, proyectos y entidades financieras. En dicho diagrama no se muestran muchas de las informaciones susceptibles de reflejarse en este tipo de diagramas, pues su propósito es meramente ilustrativo.

4.7.3 Diagramas de clases UML

El lenguaje unificado de modelado (UML) es un conjunto de notaciones de propósito general de especificación y representación de sistemas. Especialmente utilizado para sistemas de software, UML incluye diagramas de casos de uso, estructurales, de comportamiento, y de implementación.

Dentro de la variedad de representaciones de UML, los diagramas más adecuados para el modelado conceptual estático son los diagramas de clases. Aunque en dichos diagramas es posible incluir detalles de implementación, cuando se utilizan para modelado conceptual

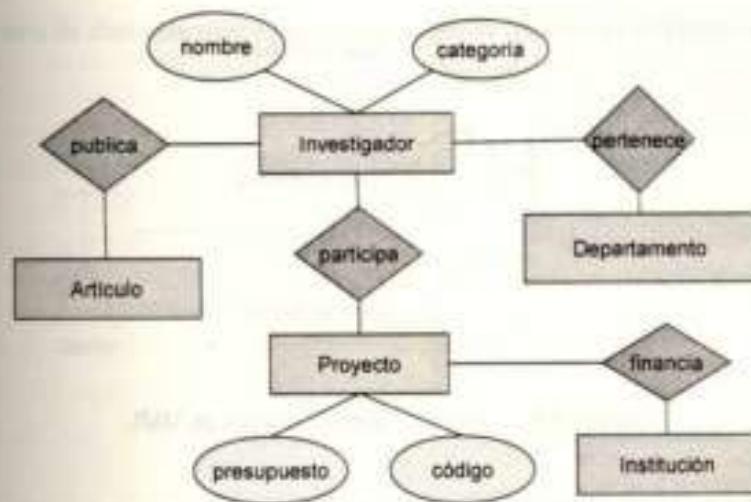


Figura 4.8: Diagrama entidad-relación

de requisitos no se incluyen, obteniéndose como resultado diagramas similares (conceptualmente) a los modelos entidad-relación.

Un diagrama de clases representa un conjunto de clases y las relaciones entre ellas. Las clases permiten modelar cualquier entidad mediante la enumeración de sus características, que pueden ser estáticas (atributos) o dinámicas (métodos):

- Los atributos representan propiedades de un objeto de la clase, tales como el nombre de una persona o la fecha de una factura.
- Los métodos representan el comportamiento de los objetos, y son por tanto funciones que pueden invocarse sobre el objeto, tales como el cálculo del área en una figura geométrica plana o el método acelerar en un vehículo que permite incrementar su velocidad actual.

La realización de un caso de uso puede representarse en un diagrama de clases UML mediante una colaboración en un modelo de análisis, lo que se denomina «realización de caso de uso-análisis». La Figura 4.11 representa la realización de la Figura 4.10. Estos diagramas incluyen abstracciones especiales denominadas genéricamente «clases de análisis», las cuales se clasifican en tres categorías (ver Figura 4.9):

- Clases de análisis: se centra en los requisitos funcionales y pospone los no funcionales. Su comportamiento se define mediante responsabilidades en un nivel alto y poco formal. Sus atributos son de nivel alto, reconocibles en el dominio del problema. Participa en relaciones, aunque muy conceptuales.

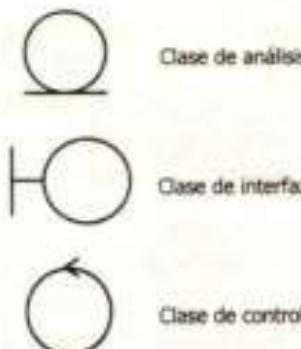


Figura 4.9: Tipos de clases de análisis en UML

- **Clase de interfaz:** modela la interacción entre el sistema y sus actores, representando a menudo abstracciones de alto nivel de ventanas, formularios, paneles, etc. Cada una se relaciona al menos con un actor.
- **Clase de control:** representa coordinación, secuencia, transacciones, y control de otras clases, y se utilizan para encapsular el control de un caso de uso. Modelan los aspectos dinámicos del sistema, coordinando las acciones principales y los flujos de control y delegando el resto de trabajos en las clases de interfaz y de entidad.

El siguiente ejemplo muestra el modelado de algunos requisitos de alto nivel de un sistema mediante el refinamiento sucesivo desde el diagrama de casos de uso hasta un diagrama de clases en UML relativamente detallado.

Se muestra parte de un sistema de compra de entradas de cine en línea, concretamente la consulta de las películas que se proyectan en una determinada sesión. Esta interacción funciona de modo que cualquier cliente solicita la visualización los datos de una sesión en la interfaz. Para ello, el sistema se sirve de dos entidades controladoras: el gestor de visualización, que lleva el control «maestro» del proceso, y el Selector de Sesión, elemento que se encarga de seleccionar una sesión a partir de la fecha y hora de la misma. La selección de la sesión adecuada se lleva a cabo mediante una búsqueda en un almacén de datos donde se guarda toda la información acerca de las sesiones programadas.

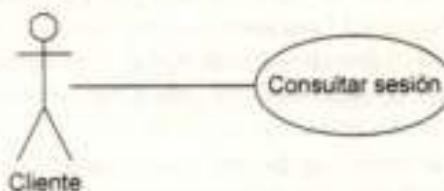


Figura 4.10: Consultar datos de sesión: caso de uso

El diagrama de clases de análisis para el ejemplo se muestra en la Figura 4.11.

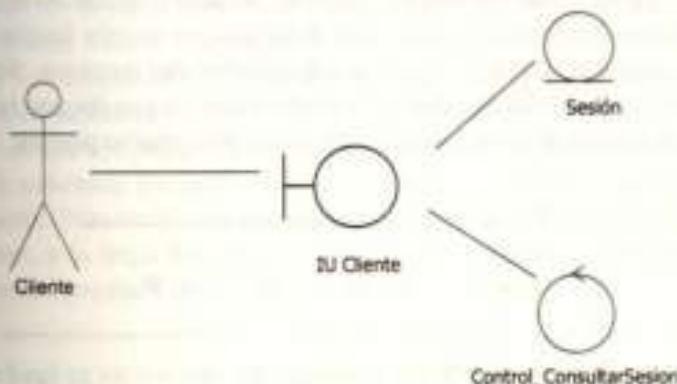


Figura 4.11: Consultar datos de sesión: diagrama de clases de análisis

Finalmente, un diagrama estructural detallado permite mostrar los atributos, métodos y relaciones identificados entre las clases que conforman la parte estructural del modelo de análisis. No es sino un diagrama de clases más detallado, como se aprecia en la Figura 4.12.

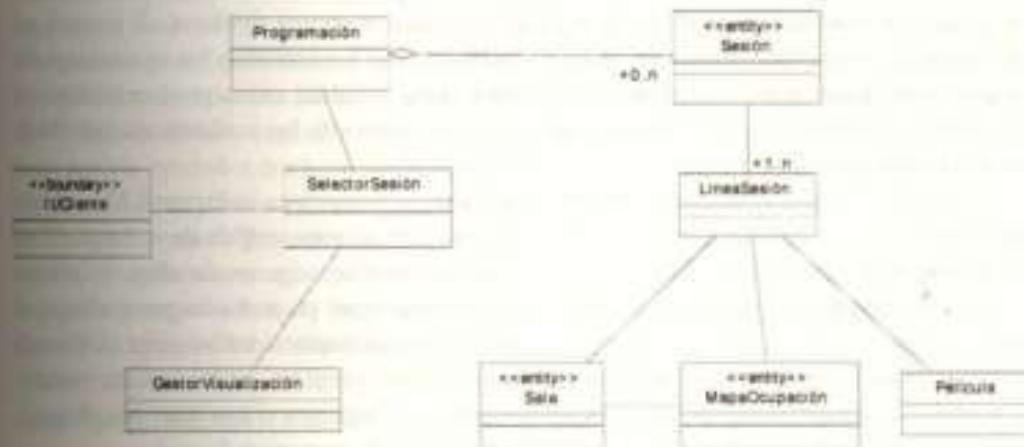


Figura 4.12: Consultar datos de sesión: diagrama de clases detallado

4.7.4 Notaciones formales

Ya hemos comentado ampliamente la dificultad para extraer los requisitos de un sistema a desarrollar, así como el impacto en la calidad final del producto que la correcta (o incorrecta) descripción de los requisitos tendrá.

Partiendo de la importancia de una descripción precisa y sin ambigüedades, las notaciones formales aspiran a lograr especificaciones de requisitos donde no existan fallos que luego serían difíciles de reparar, así como a no dejar escapar ningún detalle aparentemente irrelevante que luego se revelará vital para la comprensión del requisito. Se trata de métodos basados en las matemáticas que aspiran a la obtención de una documentación donde se especifiquen las funcionalidades del sistema del modo más preciso posible.

$\boxed{\begin{array}{l} [W] \\ \hline - \underline{\text{select}} _ \\ - \underline{\text{remove}} _ : \text{seq } W \times \mathbb{P} W \rightarrow \text{seq } W \\ \hline \forall s : \text{seq } W; w : \mathbb{P} W \bullet \\ \quad s \text{ select } w = \text{squash}(s \triangleright w) \wedge \\ \quad s \text{ remove } w = \text{squash}(s \triangleright w) \end{array}}$
--

Figura 4.13: Ejemplo de especificación formal de requisitos con Z

Para ilustrar la sección, pondremos como ejemplo la notación en lenguaje Z. Este lenguaje está basado en la lógica y se emplea en ingeniería del software para expresar en general requisitos, propiedades de algoritmos o de estructuras de datos. Z permite así definir estructuras –denominadas *esquemas*– para modelar los estados y las operaciones del sistema. En dichos esquemas se pueden declarar tanto variables como predicados lógicos, los cuales sirven para especificar cómo cambian los valores de las variables en función de ciertos condicionantes.

El pequeño fragmento de especificación en Z que se muestra en la Figura 4.13 –tomado del libro de Bowen (1996)– indica que las ventanas en un sistema gráfico de ventanas deben renombrarse después de una operación en la que se eliminan algunas de ellas, de manera que la pila de ventanas continúe siendo una secuencia. Este pequeño fragmento requiere para su comprensión el conocer la compleja notación matemática del lenguaje Z. Cuando se está diseñando un sistema de ventanas para un sistema operativo, podría resultar rentable la especificación detallada de todas las operaciones en lenguajes como éste. No obstante, en la mayoría de las aplicaciones no se llega a este nivel de estructura formal.

4.8 Gestión del proceso de requisitos

La industria de desarrollo de software está sometida a presiones cada vez mayores, ciclos de desarrollo más cortos y mercados extremadamente competitivos. Todo ello influye de manera importante en la forma en que se gestionan los proyectos en su conjunto, desde el inicio hasta la entrega. El impacto de esta presión sobre la gestión de los requisitos es especialmente fuerte.

En la práctica, resulta imposible tratar los requisitos como algo cerrado que se lleva a cabo a lo largo de un proceso lineal y no sujeto a cambios. En realidad, los requisitos no se obtienen, analizan, validan y documentan para que el equipo encargado del desarrollo implemente las funcionalidades que les dan soporte, sino que evolucionan siguiendo un proceso iterativo característico.

Lo habitual es que los requisitos evolucionen iterativamente hasta alcanzar un nivel de calidad y detalle suficiente que permita iniciar los trabajos de diseño de las funcionalidades que les dan soporte. Para facilitar la comprensión completa de todos los aspectos asociados con los requisitos, a lo largo del proceso iterativo descrito deben definirse *líneas base de requisitos* dentro del proyecto.

Una línea base es un conjunto de requisitos que deberá contener una entrega del producto

La existencia de líneas base permite almacenar la historia de cambios en el desarrollo de un proyecto y volver, si se considera necesario, a un estado anterior que viene determinado por un hito previo en la línea base.

En prácticamente todos los proyectos, la comprensión de los requisitos es continua y evoluciona a lo largo de todo el proceso de diseño y construcción. Esta característica de los requisitos a veces obliga a revisar una especificación en ciclos tardíos del desarrollo, con el consiguiente riesgo de introducir errores y desviaciones en el coste y esfuerzo del proyecto. lamentablemente, hemos de reconocer la naturaleza cambiante de los requisitos, por lo que la única solución aceptable consiste en realizar una correcta gestión de los cambios, si fuera posible ayudados por herramientas automatizadas para la gestión de las actividades de requisitos. La utilización de métricas para evaluar el tamaño de un cambio en los requisitos resulta igualmente de gran ayuda.

En las siguientes secciones se trata un poco más en detalle el uso de herramientas para la gestión y seguimiento de los requisitos, así como las métricas que pueden emplearse durante las actividades de requisitos.

4.8.1 Seguimiento

Poder realizar un seguimiento a los requisitos resulta esencial a la hora de evaluar y comprender el impacto de las propuestas de cambios en el software en construcción. Si es posible realizar un seguimiento de los requisitos, será más improbable que el sistema final tenga inconsistencias por culpa de cambios realizados sin un control exhaustivo.

Como ya se dijo, la capacidad para realizar un seguimiento de los requisitos software es una importante característica de los mismos. Así, debe ser posible hacer un seguimiento que permita conocer su estado (especificado, verificado, analizado, etc.) en cada momento del desarrollo. Para facilitar el seguimiento puede hacerse uso de una útil herramienta denominada *matriz de seguimiento*.

Tabla 4.7: Matriz de seguimiento de requisitos

	Requisito 1	Requisito 2	Requisito 3	...	Requisito n
Caso prueba 1		*	*		
Caso prueba 2	*	*			*
Caso prueba 3	*				
Caso prueba 4			*		
Caso prueba 5		*			*
Caso prueba 6	*	*			

Una matriz de seguimiento es una tabla donde se relacionan dos documentos, los cuales pertenecen probablemente a etapas distintas del desarrollo. Su utilización más frecuente es seguir la pista de los requisitos a lo largo de todo el desarrollo, fundamentalmente en el diseño (tanto de alto nivel como detallado) plan de pruebas y casos de prueba, en cuyo caso recibe el nombre de *matriz de seguimiento de requisitos*.

Una matriz de seguimiento hace corresponder los requisitos con los componentes software que los implementarán y especialmente con los casos de prueba desarrollados para comprobar que dichas funcionalidades son correctamente soportadas por el componente. Para ser completa, la matriz de requisitos debería relacionar cada requisito incluido en la especificación con los apartados de los documentos de especificación de los usuarios. Se trata de una herramienta de particular utilidad por ejemplo para las actividades de mantenimiento, pues al almacenar la correspondencia entre requisitos y componentes, facilita tremadamente la realización de cambios en una funcionalidad.

La creación de una matriz de seguimiento de requisitos es sencilla. Para ello, debes ponerse los identificadores de los requisitos en la primera fila y los identificadores de elementos del documento con el que se quiera realizar el seguimiento en la columna izquierda (en la Tabla 4.7 por ejemplo, se relacionan con los casos de prueba). Cuando un elemento de la columna izquierda está relacionado con uno de los requisitos, en la celda que representa su intersección se pone una marca. En caso contrario, se deja en blanco. El número de relaciones para cada columna indica el seguimiento del requisito. Cuando un requisito no tiene ninguna relación, lo cual indica que no está cubierto por ninguno de los elementos listados, se informa al responsable de esa etapa del desarrollo de la incidencia, pues debe crearse alguna. Por el contrario, valores muy altos de relación indican que el requisito es demasiado complejo y que por tanto debería simplificarse.

Para facilitar la creación de matrices de seguimiento, todos los documentos han de incluir información que permiten crearla e información bidireccional que facilite hacer un seguimiento de los requisitos en los dos sentidos del desarrollo (progresivo y regresivo).

4.8.2 Métricas de los requisitos

La existencia de medidas es importante en la gestión de un proyecto, así como en muchos procesos de mejora o cambio en la organización. Así por ejemplo, muchos de los métodos

de mejora, tales como Six Sigma o el modelo de madurez CMMI, utilizan técnicas que se basan en la existencia de medidas exactas, por lo que la existencia de estas últimas resulta esencial para la correcta implementación del método.

Es posible tomar medidas de los requisitos software que indiquen el alcance del proyecto, su crecimiento potencial, su estabilidad y progreso. Las medidas de requisitos son únicas, pues permitirán caracterizar el espacio del problema, a diferencia de otras medidas del desarrollo que constituyen una caracterización del espacio de la solución. Se trata además de datos que se encuentran disponibles en etapas tempranas del desarrollo, por lo que resultan especialmente útiles en la realización de planes de proyecto, así como en la predicción de alternativas, riesgos y resultados a obtener.

Algunas de las medidas del proceso de requisitos que resultan más relevantes son:

- Número de requisitos nuevos (o modificados) por mes. Se calcula como el cociente entre el número de cambios realizados y el número total de requisitos, a menudo en términos porcentuales, por lo que para el cálculo de este dato es necesaria la descomposición y cómputo previo del número total de requisitos.

$$\text{Porcentaje de requisitos nuevos} = \frac{\text{Cambios en los requisitos}}{\text{Requisitos totales}}$$

El porcentaje de nuevos requisitos es una medida muy útil pues refleja la estabilidad o inestabilidad del desarrollo, ya que los cambios en los requisitos tienen un fuerte impacto en la estabilidad del producto final.

- El indicador de cobertura mide el porcentaje de requisitos actualmente incluidos en alguna línea base que han sido soportados por un componente incluido en el diseño detallado del sistema. Para computar este valor, es necesario comprobar, por ejemplo mediante una matriz de seguimiento, que existe una correlación entre el requisito y el componente y que dicha correlación está actualizada.

$$\text{Cobertura de requisitos} = \frac{\text{Requisitos soportados}}{\text{Requisitos en la línea base}}$$

- El número de requisitos derivados a partir de cada requisito inicial es un indicador de la explosión de requisitos que se produce como consecuencia de una incorrecta especificación o de su descripción a un nivel de detalle incorrecto. En el caso medio, cada requisito inicial produce un cierto número de requisitos derivados. Sin embargo, la existencia de requisitos incorrectamente descritos conlleva una expansión e interpretación a menudo excesivas. Del mismo modo, la obtención de una tasa de requisitos derivados por requisito individual demasiado baja indicará que los requisitos pueden estar a un nivel de detalle incorrecto (demasiado bajo, en este caso), por lo que deberá evaluarse el proceso de análisis de requisitos.

$$\text{Tasa de requisitos derivados} = \frac{\text{Requisitos derivados}}{\text{Requisitos iniciales}}$$

- La compleción de los requisitos mide cuántos requisitos han sido completamente definidos en un determinado punto del desarrollo. Algunos requisitos no pueden ser completamente definidos hasta etapas tardías del ciclo de desarrollo, sin embargo, posponer innecesariamente la definición de ciertos requisitos puede causar efectos negativos, especialmente si estos requisitos influyen en el diseño del sistema. A menudo los requisitos no descritos completamente han sido identificados como TBD (*to be determined*), que en español podría traducirse como AD (a determinar).

$$\text{Tasa de requisitos incompletos} = \frac{\text{Requisitos TDB}}{\text{Requisitos totales}}$$

Dado que el estado de los requisitos cambia constantemente durante el desarrollo, es posible obtener medidas detalladas sobre el estado actual de los requisitos no completados, computando un valor según su estado actual (en codificación, codificación completada, validado mediante pruebas, etc.) Así por ejemplo tendríamos:

$$\text{Tasa de requisitos probados} = \frac{\text{Requisitos probados}}{\text{Requisitos totales}}$$

- Otras medidas interesantes en el proceso de requisitos son el número de requisitos asignados a una persona (útil para medir las cargas de trabajo en el proyecto), la frecuencia de cambios en el conjunto completo de requisitos con respecto al tiempo, el número de cambios con respecto a una línea base, el porcentaje de defectos cuya causa es un error en la especificación de requisitos o el número de solicitudes de cambio en los requisitos.

La utilización de estas medidas dentro de una adecuada política de medición, resulta beneficiosa para la gestión del proceso de desarrollo en su conjunto, ya que permite reducir el número de defectos, minimiza costes imprevistos y aumenta la calidad del producto final.

4.8.3 Herramientas para la gestión de requisitos

El uso de herramientas software como soporte para la gestión de requisitos es un aspecto cada vez más importante. Considerando el tamaño y complejidad de los desarrollos, el uso de herramientas se ha convertido poco a poco en algo esencial, pues no sólo facilita el mantenimiento de los requisitos, sino que aporta beneficios significativos como la reducción del número de errores.

Ya hemos visto cómo los requisitos, una vez determinados y documentados, no permanecen inalterables. Por el contrario, están sujetos a cambios continuos. Las herramientas de gestión de requisitos proporcionan un medio adecuado de seguimiento de los cambios que se producen durante el ciclo de vida del proyecto, permitiendo automatizar algunas de las relacionadas con los cambios y el seguimiento de requisitos, y mejorando en definitiva la productividad y la calidad en el desarrollo.

Los motivos para utilizar herramientas en la gestión de los requisitos son cada vez más sumerios. En realidad, hoy en día habría que buscar un motivo de peso para justificar su uso. Las ventajas fundamentales que aporta el uso de herramientas en la gestión de requisitos son las siguientes:

- Ahorro significativo en los costes de especificación y de desarrollo, minimizando el impacto de los errores.
- Mejora de la calidad mediante un adecuado análisis y gestión de los requisitos.
- Aumento de la productividad, facilitando la reutilización real desde la especificación.
- Facilidad para administrar adecuadamente la especificación.
- Apoyo para cumplir con los estándares de calidad.

Si bien cada herramienta tiene un conjunto de características que la diferencian del resto, bien puede decirse que hay un buen número de funcionalidades comunes a la mayoría de ellas. A continuación se enumeran algunas de las características más relevantes que suelen implementar este tipo de herramientas:

- Permiten gestionar versiones y cambios mediante la definición de líneas base. Como ya se ha dicho, las líneas base permiten almacenar la historia de cambios asociada a un requisito y revertir en cualquier momento el estado actual al de versiones anteriores si se considera necesario.
- Almacenan todo tipo de información acerca de los requisitos, lo que frecuentemente se conoce como «atributos de los requisitos». Estos atributos pueden ser consultados con diferentes niveles de acceso (modificación, consulta, etc.) por cualquier persona que participe en el desarrollo.
- Permiten enlazar los requisitos con otros elementos, lo que facilita su seguimiento y garantiza que el software final no obviará ninguno por descuido o error. Además, simplifican el estudio del impacto de un cierto requisito al permitir determinar fácil y rápidamente qué componentes se ven afectados por un cambio en el mismo.
- Gestionan el estado de cada uno de los requisitos durante el desarrollo, lo que permite un seguimiento minucioso de todo el proyecto. Permiten conocer, por ejemplo, el porcentaje de requisitos que han sido implementados y verificados, cuántos han sido implementados pero aún no han sido verificados, etc.
- Permiten crear subconjuntos de requisitos, agrupándolos por valores específicos de alguno de sus atributos y realizar búsquedas, filtrados o cualquier otra consulta que pudiera necesitarse.

- Tienen en cuenta el control de accesos, permitiendo la clasificación de usuarios y la definición de perfiles y permisos de acceso asociados a los mismos.
- Incluyen herramientas para la verificación y aceptación de los requisitos, como las estudiadas durante el capítulo. Así, permiten definir criterios de aceptación y escenarios de prueba, y también asignar dichos escenarios a los requisitos.
- Facilitan la comunicación con el resto de actores, pues la mayoría de las herramientas hoy en día disponibles permiten realizar discusiones acerca de los requisitos sobre la propia plataforma. Así por ejemplo, puede seleccionarse que se envíe un correo electrónico para notificar a los afectados por una decisión o un cambio que va a abrirse un nuevo foro de discusión al respecto.
- Incluyen facilidades para la elaboración de informes, y como la mayoría pueden funcionar integradas con las herramientas de edición de texto más populares, las tareas de documentación se simplifican considerablemente.

Dentro de las opciones comerciales existentes en el mercado, algunas de las más utilizadas son REquisitePRO, DOORS, IrqA, MKS Integrity suite, OSRMT, RTM Workshop, o Caliber RM. Entre todas ellas destaca especialmente OSRMT (*Open Source Requirements Management Tool*) pues se trata de una herramienta de fuente abierto que puede utilizarse bajo los términos de la licencia GNU GPL. La mayoría utilizan sistemas centralizados de gestión de bases de datos para almacenar la información de los requisitos. Todas permite estructurar los requisitos jerárquicamente, extraer texto de archivos generados por procesadores de texto comerciales para incorporarlos (o para transformar directamente dichos archivos en requisitos) y realizar consultas sobre los requisitos en función de determinados valores de sus atributos.

Otras herramientas de menor difusión son DRES, Active!Focus, Analyst Pro, CARE, SteelTrace, XTie-RT, Prosareq, Focal Point, Easy-RM, RDT, PROJECTRICITY, Reconcile, PACE, SoftREQ y Qualica.

4.9 Resumen

La correcta realización de las actividades de requisitos es esencial para el éxito de un proyecto de desarrollo de software. En este capítulo hemos tratado, en primer lugar, los conceptos fundamentales relacionados con los requisitos de un sistema software, como atributo, actor, etc. para a continuación analizar con mayor detalle las principales actividades de requisitos: obtención, especificación y validación, comentando las técnicas más utilizadas en cada una de estas actividades. Vimos además cuáles son las diferentes clasificaciones de los requisitos, haciendo especial énfasis en la clasificación tradicional en requisitos funcionales y no funcionales, y cómo no siempre resulta sencillo clasificar en una u otra categoría un cierto requisito.

Se dedicó un apartado específico a la descripción de notaciones para el modelado de requisitos, pues de la buena comprensión de todas las partes involucradas depende el futuro éxito del proyecto y, para ello, nada más útil que un lenguaje sencillo que permita llevar a cabo discusiones sobre las funcionalidades y calidades que serán requeridas. La importancia de una correcta especificación de requisitos como parte esencial del contrato entre desarrolladores y clientes, y base de la posterior validación de los mismos, fueron detenidamente examinadas y estudiadas con detalle. El análisis de las diferentes partes de la validación (revisión, validación del modelo, elaboración de prototipos y aceptación) fue también objeto de nuestro estudio.

En la última parte del capítulo hemos tratado la gestión del proceso de requisitos. En primer lugar, el modo en que el seguimiento de los requisitos resulta esencial en entornos de desarrollo tan variables como los actuales. En segundo lugar las métricas específicas de las actividades de requisitos, y finalmente las herramientas que el ingeniero del software tiene a su disposición para la mejor gestión de esta parte del desarrollo.

En la siguiente nube de palabras se muestran los conceptos más importantes tratados en el capítulo, donde un tipo de letra con mayor tamaño indica que el concepto tiene una mayor importancia relativa.



Figura 4.14: Principales conceptos tratados en el capítulo

4.10 Notas bibliográficas

En la segunda edición del libro «*Software requirements*» (Wiegers, 2003) hace un estudio detallado y profundo de los temas tratados en el capítulo, por lo que resulta una excelente obra de referencia. Otro texto interesante es «*An introduction to requirements engineering*» (Bray, 2002). Especialmente orientado a aquellos que estudian las actividades de requisitos por vez primera, dedica la parte 3 completa al estudio de casos prácticos donde se muestra de manera sencilla cómo aplicar muchos los conceptos teóricos analizados en el capítulo.

Algunas de las prácticas recomendadas en el capítulo sobre la especificación de requisitos se tratan con más detalle en el libro «*System requirements engineering*» (Locopoulis y Karakostas, 1995). Es un libro algo antiguo pero su lectura aún resulta de gran interés.

Si se quiere profundizar en los beneficios del uso de herramientas en la gestión de requisitos, «*Automating requirements management*» (Wieggers, 1999) sigue siendo una interesante lectura, donde además se puede encontrar una completa comparativa de varias herramientas.

La referencia original sobre casos de uso es el ya clásico libro de Jacobson «*Object-Oriented Software Engineering: A Use Case Driven Approach*» (Jacobson, 1992). Otros libros muy interesantes sobre el tema son «*Managing Software Requirements: A Use Case Approach*» (Leffingwell y Widrig, 2003) y «*Writing Effective Use Cases*» (Cockburn, 2000) donde se trata en profundidad la técnica y se muestran ejemplos suficientes de su uso tanto en el modelado de requisitos como en otras actividades del desarrollo. Debe tenerse en cuenta que la notación está en continua evolución, por lo que se recomienda acudir a la última especificación del lenguaje siempre disponible en línea en <http://www.uml.org>.

El estándar 830-1998 de IEEE para la especificación de requisitos del software (IEEE, 1998a) describe el contenido y cualidades que ha de tener un buen documento de especificación de requisitos. Incluye varias plantillas y ejemplos de SRS que merece la pena consultar y tener en cuenta si se va a ver envuelto en la elaboración de uno.

Aquellos lectores interesados en saber más sobre las particularidades de las actividades de requisitos en proyectos *open source*, encontrarán sin duda interesante el artículo «*Understanding the Requirements for Developing Open Source Software Systems*», donde el autor explica los requisitos en este tipo de proyectos (Scacchi, 2002), y «*Where Do Open Source Requirements Come From (And What Should We Do About It)?*», donde Massey analiza las diferencias entre el modelo de actividades de requisitos «tradicional» y el de los desarrollos *open source* (Massey, 2002).

4.11 Cuestiones de autoevaluación

- 4.1 Identifique los actores involucrados en las actividades de requisitos. ¿Cuál es la diferencia entre un cliente y un usuario?
R: *Usuario es una definición abstracta que engloba a todas aquellas personas que operan con el software, mientras que un cliente es todo aquel que tienen interés en adquirir el software, pero no necesariamente hará uso del mismo una vez lo haya adquirido. A parte de éstos otros actores son los analistas de mercado, los reguladores y los ingenieros de software.*
- 4.2 Según lo visto en este capítulo, existen varios criterios para la clasificación de requisitos. ¿Cuál es el más utilizado? ¿Por qué a veces resulta difícil determinar si un requisito pertenece a una u otra categoría?
R: *La clasificación más usual divide los requisitos en funcionales y no funcionales. Las dificultades para determinar si un requisito es funcional o no, radican en el hecho de que ésta es una clasificación que ha sido ideada con el único propósito de simplificar algunas actividades de requisitos, por lo que la frontera entre ambos tipos es difusa.*

4.3 ¿En qué consiste el modelado conceptual?

R: En elaborar una representación que permita comprender el problema planteado, siendo uno de sus objetivos fundamentales el comunicar nuestro entendimiento a otros para mejorar su comprensión del sistema. En ningún caso este modelado implica elaborar la solución al problema planteado, ni aun un esbozo de la misma.

4.4 Describa algunas ventajas de la utilización de notaciones para la descripción de

R: La ventaja fundamental es facilitar la comprensión de los requisitos a aquellos que van a tratar con ellos. Si además se utiliza una notación estandarizada y ampliamente difundida y conocida, el beneficio es mayor ya que la especificación formalizada podrá ser comprendida por cualquiera que conozca dicha notación. Por ejemplo, por personas que no participaron inicialmente en el desarrollo y que puedan necesitar analizar los requisitos como parte de una tarea de mantenimiento, con el consiguiente beneficio en términos de esfuerzo.

4.5 Según lo visto en el capítulo, ¿Cuáles son las fases de una entrevista para la obtención de requisitos?

R: Planificación, preparación, inicio, desarrollo, cierre y conclusiones.

4.6 ¿Cuáles son las propiedades que debe tener un requisito de software?

R: La propiedad fundamental es que sea verificable, pero además es deseable que se pueda hacer un seguimiento del mismo y que sea priorizable, identificable y cuantificable.

4.7 En la revisión de requisitos se conforma un grupo de personas para validar los requisitos obtenidos, ¿quiénes deben integrar dicho grupo?

R: La composición ideal incluirá al menos un representante de los siguientes grupos: usuarios, clientes, desarrolladores, analistas de requisitos y expertos funcionales en el problema. Es deseable además la inclusión de un experto externo que actúe como «abogado del diablo».

4.8 Enumere algunas de las funcionalidades que implementan las herramientas software para la gestión de requisitos.

R: Si bien cada herramienta implementa sus propias características, algunas de las más interesantes (y que incluyen prácticamente todas las herramientas actualmente disponibles) son la gestión automatizada de versiones y cambios, el almacenamiento de atributos, las facilidades para el seguimiento, la inclusión de mecanismos de validación y aceptación de requisitos o el control del estado de cada uno de los requisitos durante el desarrollo.

4.9 Indique razonadamente si la siguiente afirmación es verdadera o falsa: «Los casos de uso son una notación exclusivamente orientada al modelado de requisitos».

R: Falso. Si bien fue inicialmente ideada para la captura de requisitos funcionales de un sistema, hoy en día se aplica en muchos otros ámbitos de la ingeniería del software.

4.10 ¿Qué es una matriz de seguimiento?

R: Una matriz de seguimiento es una tabla donde se relacionan dos documentos, probablemente de etapas distintas del desarrollo. Su utilización más frecuente es seguir la pista de los requisitos a lo largo de todo el desarrollo, fundamentalmente en el diseño, plan de pruebas y casos de prueba, en cuyo caso recibe el nombre de matriz de seguimiento de requisitos.

4.12 Ejercicios y actividades propuestas

4.12.1 Ejercicios resueltos

- 4.1 Utilizando la notación de casos de uso, modele la siguiente especificación de requisitos: «Un sencillo juego de computadora permite a un usuario guiar un tanque por un escenario en el que existe cierto número de obstáculos que deben destruirse. Para mover el tanque por el escenario el jugador utiliza un joystick o el teclado. El tanque tiene una cantidad de munición limitada, de forma que cada vez que dispara disminuye la cantidad de munición disponibles».

Solución propuesta: La aplicación tiene dos casos de uso principales: mover el tanque y disparar. Cada vez que se dispara, se comprueba la munición disponible para ver si puede hacerse efectivo el disparo (*include*) y en caso afirmativo se decremente la munición (*include*) y se procede al disparo. Sólo en algunas ocasiones el disparo impacta en un obstáculo, por lo que el comportamiento se amplía (*extend*) con una animación que muestra la destrucción del obstáculo alcanzado.

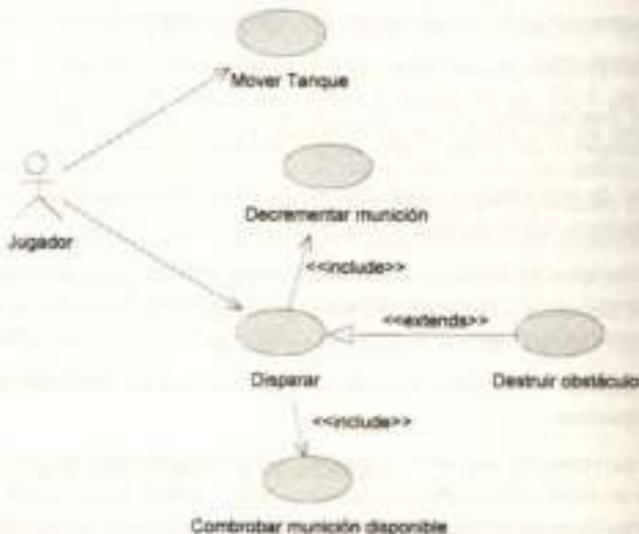


Figura 4.15: Solución propuesta al ejercicio 4.1

- 4.2 Clasifique los siguientes requisitos software en funcionales o no funcionales. En los requisitos no funcionales, diga a qué categoría (de producto, de la organización o externo) pertenece.

1. La interfaz debe seguir la normativa de colores e imagen corporativa de la empresa.
2. La web debe seguir la normativa internacional en las áreas de protocolos, contenidos, herramientas, formatos y lenguajes relacionados con la accesibilidad a internet.
3. El área de trabajo debe poder ampliarse o reducirse mediante los correspondientes botones de acceso directo en la barra de herramientas.

4. La aplicación permitirá consultar las facturas por diversos criterios, tales como facturas de pronto vencimiento, por cliente, por importe, etc.
5. El acceso a la información personal sobre clientes no podrá ser consultada por los usuarios, excepto aquellos a los que les hayan sido concedido privilegios especiales para ello.
6. El programa cliente que realiza las búsquedas en la base de datos debe poder estar seguro de que la respuesta del sistema no ha sido falsificada o alterada.
7. El lenguaje de programación a utilizar será obligatoriamente Java.
8. La aplicación en desarrollo sigue la filosofía del software libre y debe por tanto ponerse a disposición de la comunidad su código fuente.
9. La información de enrutamiento debe protegerse contra modificaciones no autorizadas.
10. Deben almacenarse todas las facturas emitidas por el sistema para su posterior consulta.
11. El sistema incluirá algún tipo de comprobación de la identidad para las operaciones sensibles.
12. El sistema debe permitir la realización periódica de copias de seguridad para toda la información de clientes y ventas.

Solución propuesta: Requisitos no funcionales: 1 (de la organización), 2 (externo), 5 (del producto), 6 (del producto), 7 (del producto), 8 (externo), 9 (del producto) y 12 (del producto). Requisitos funcionales: 3, 4, 10 y 11.

- 4.3 En la figura 4.16 se muestra la representación, mediante la notación de casos de uso, de parte de una aplicación para la consulta en línea de horarios de trenes. Utilice la notación tabular descrita en la Sección 4.7.1 para representar el caso de uso «Venta de un billete de cercanías». Incluya al menos dos escenarios, el principal y uno alternativo relevante.

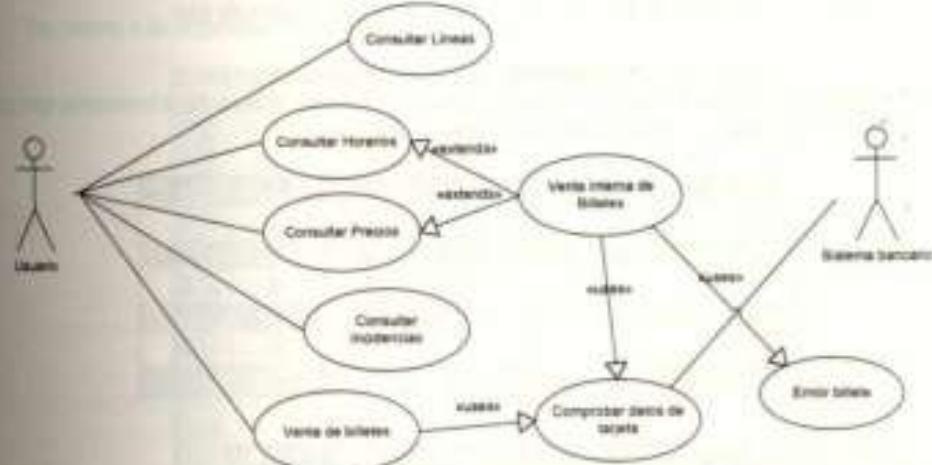


Figura 4.16: Casos de uso para un sistema de información ferroviario

Solución propuesta: A continuación se muestra una posible solución:

Código: CU009

Nombre: Venta de un billete de cercanías

Actor(es): Usuario

Descripción: Venta de un billete de cercanías entre dos estaciones (origen y destino)

Precondiciones: No hay

Postcondiciones: El sistema registra la venta de un billete

Escenario principal:

1. *El usuario accede a la interfaz de compra de billetes*
2. *El usuario especifica la estación destino (el origen se establece por defecto a la estación actual)*
3. *El sistema calcula la ruta*
4. *El sistema calcula el precio según zonas de tarificación*
5. *El sistema ofrece comprar un billete según la descripción*
6. *El usuario introduce su tarjeta de crédito para proceder a la compra*
7. *Se realiza el caso de uso 008 (comprobación de datos tarjeta)*
8. *Se expende el billete*
9. *El usuario retira su tarjeta y el billete adquirido*
10. *La interfaz del sistema muestra el mensaje inicial*

Escenarios alternativos:

2(ALT). *Si tras cierto tiempo el usuario no selecciona un destino, la interfaz muestra un mensaje de aviso*

5(ALT). *El usuario cancela los datos con la opción CANCELAR porque no desea realizar la operación o porque los datos no son correctos*

7(ALT). *Si la realización del caso de uso 008 es errónea (por abortar la operación o el rechazo de la tarjeta) no se continua con el caso de uso.*

4.4 A partir de la siguiente matriz de conflictos entre requisitos, indique la información que podemos obtener y qué acciones deberían llevarse a cabo.

Requisito	R1	R2	R3	R4	R5	R6
R1	0	1.000	0	0	1	1
R2	1.000	0	0	0	0	1.000
R3	0	0	0	0	0	0
R4	0	0	0	0	1.000	1
R5	1	0	0	1.000	0	0
R6	1	1.000	0	1	0	0

Solución propuesta: Tal y como muestra la figura, R2 se solapa con R1 y con R6; y también R5 y R4 se solapan. R1 presenta conflictos con R5 y R6, y R4 también los presenta con R6. R3 es independiente de los demás. Las acciones a tomar en un caso así serían evitar tanto los solapamientos como los conflictos detectados.

- 4.5** Realice un prototipo de web para un investigador usando una herramienta para diseño de prototipos de baja fidelidad. La página debe contener algunas secciones básicas como las dedicadas al currículo del investigador, a las publicaciones realizadas o a sus ocupaciones docentes. Las publicaciones se dividirán en dos grupos: artículos y libros. La página principal incluirá una foto del investigador, una breve biografía suya y enlaces al resto de secciones.

Solución propuesta: Se muestra un prototipo de la página principal realizado con la herramienta Denim, donde se ve la estructura de la misma:

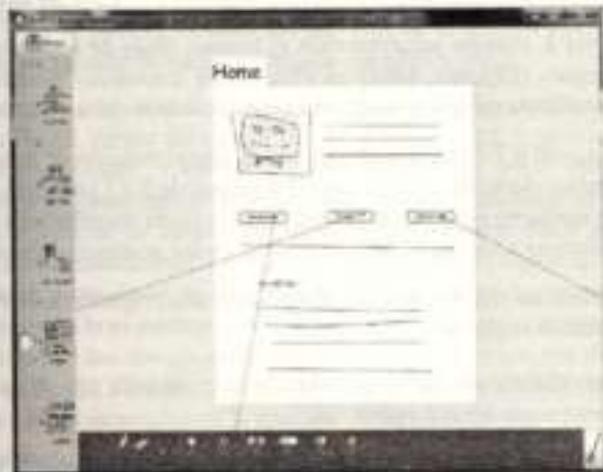


Figura 4.17: Estructura de la página principal

En cuanto a la organización general de la web y los enlaces entre las diferentes páginas:

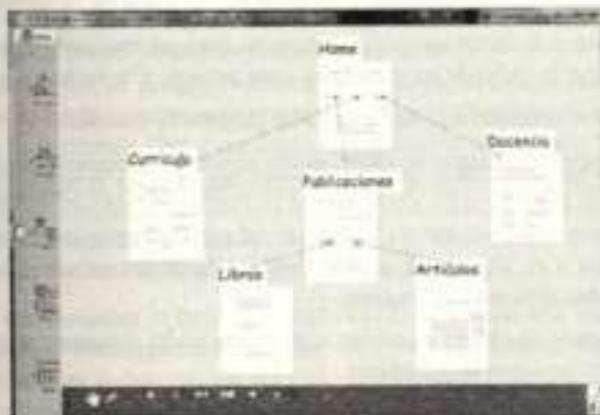


Figura 4.18: Organización general de la web

4.12.2 Actividades propuestas

- 4.1 Según lo visto en el capítulo, una de las características más importantes de los requisitos es que deben ser *priorizables*. Un estudio realizado en 1998 por el *Standish Group* da un manifiesto que el 45% de las funcionalidades de los sistemas más populares nunca se utilizan, el 19% se utiliza rara vez, el 16% se utiliza de vez en cuando, el 13% se utiliza a menudo y sólo un 7% se utiliza siempre. Discutir en grupo las consecuencias de este estudio en la clasificación de los requisitos según su prioridad, especialmente la posibilidad de etiquetar algunos requisitos como «*prescindibles*».
- 4.2 Los casos de uso también tienen detractores. A este respecto, resulta interesante la discusión que hace Anthony J. Simons parafraseando el famoso título de Dijkstra «*GO TO statements considered harmful*» (Dijkstra, 1968) en «*Use cases considered harmful*» (Simons, 1999). Lea el artículo y discuta en grupo las afirmaciones en contra del uso de esta conocida técnica.
- 4.3 Al final de la Sección 4.7.1 aparece un ejemplo de representación de requisitos de un sistema de compra online mediante casos de uso. En grupos de 2 o 3 personas, analizar los requisitos del ejemplo mediante otras notaciones diferentes (cada miembro del grupo debe emplear una notación diferente a los casos de uso) y compare los resultados obtenidos.
- 4.4 Se desea desarrollar un sistema de contabilidad para una empresa de televisión por cable que atienda los siguientes requisitos:
 1. El sistema debe almacenar los datos personales de cada uno de sus clientes, así como los descuentos que les son aplicables.
 2. Un cliente puede tener más de una cuenta con la compañía (puede tener contratado el servicio en distintos domicilios de su propiedad).
 3. El sistema debe almacenar la información relativa a los servicios estándar que ofrece la empresa y los precios de dichos servicios.
 4. Una cuenta activa implica el abono del cliente al paquete básico de servicios.
 5. Sólo el cliente puede modificar directamente sus datos de facturación.
 6. El cliente puede elegir servicios adicionales para sus cuentas, lo que obligará al sistema a almacenar la fecha de suscripción a cada servicio y la fecha de baja en el mismo.
 7. Un servicio generalmente se refiere a un paquete de canales.
 8. El sistema debe almacenar información sobre las peticiones de eventos de pago por visión (*pay per view*), así como los eventos disponibles en cada momento.
 9. La empresa puede cargar en la cuenta del cliente conceptos extraordinarios, como la instalación o el alquiler de algún elemento hardware propiedad de la empresa tales como el decodificador o similares.
 10. Una vez dado de alta en un servicio, el cliente podrá comenzar a utilizarlo transcurridos un período no superior a 24 horas.
 11. Deberá ser posible conocer en tiempo real el saldo de cada cuenta de los clientes de la empresa, y realizar un listado que refleje la situación actual de cada una.

Clasifique los requisitos anteriores en funcionales y no funcionales. Valídelos utilizando el cuestionario que aparece en la Sección 4.6.4.

- 4.5 La gestión de requisitos mediante un Wiki es un campo innovador y que puede reportar interesantes beneficios dada su flexibilidad y bajo coste. Lea el artículo «*Wiki-Based Stakeholder Participation in Requirements Engineering*» (Decker, Ras, Rech, Jaubert y Rieth, 2007) y discuta en grupo las diferencias entre el uso de un wiki y una herramienta tradicional de gestión de requisitos, así como los beneficios e inconvenientes de ambos enfoques.
- 4.6 Utilizando una herramienta para diseño de prototipos de baja fidelidad realice un prototipo para el proceso de gestión del «carrito de la compra» de una librería virtual tomando como ejemplo Amazon.com. No es necesario realizar el prototipo de la fase de registro, que requiere crear una cuenta de usuario en la librería, ni otras funcionalidades tales como búsquedas en el sitio web. Sólo se solicita modelar los aspectos de la interfaz que tengan que ver con la gestión del carrito (añadir, modificar, eliminar elementos, etc.).
- 4.7 Utilizando casos de uso, modele el siguiente supuesto: «Un software para dibujar permite crear nuevos dibujos, cargar dibujos existentes para editarlos, y guardar los dibujos actuales o modificar los existentes. Cada dibujo puede constar de una o más hojas, dentro de las cuales habrá figuras geométricas simples, líneas, imágenes incrustadas, cuadros de texto, etc. La aplicación consta de un menú, una barra de herramientas desde donde se selecciona la figura o elemento a representar, una zona de edición, que es donde se muestra la hoja activa del dibujo actual, y una barra de selección de dibujos y hojas que permite cambiar el dibujo actual, seleccionar la hoja a editar dentro del dibujo actual, etc. La aplicación deberá permitir al usuario almacenar los dibujos creados en un archivo en disco con diferentes formatos, así como cargar un dibujo existente para editarlo. Esto último implica mantener unos filtros de conversión que serán invocados en la apertura de un archivo».
- 4.8 Dentro de las fuentes para la obtención de requisitos, los denominados «objetivos generales del software» generalmente están expresados de manera vaga o imprecisa, por lo que los ingenieros de software a menudo realizan un estudio sobre la factibilidad de desarrollar algunos de ellos. En un grupo de 3 a 5 personas, determinar los objetivos generales de una aplicación de contabilidad para una empresa familiar y realizar un pequeño estudio de factibilidad de dichos objetivos estableciendo las prioridades de los mismos según su importancia en el desarrollo.
- 4.9 La utilización de métodos de representación en el análisis de requisitos, y en general durante el desarrollo de software, resulta importante pero también tiene desventajas que deben ser tenidas en cuenta. Así, es posible enumerar, entre otros, la incapacidad de algunos usuarios para comprender los formalismos de representación o la necesidad de una formación específica en dichos formalismos, con el consiguiente incremento del tiempo y coste del proceso. El artículo «*Application of Lightweight Formal Methods in Requirement Engineering*» de Vinu George y Rayford Vaughn, presenta una interesante discusión a este respecto (puede descargarse en línea en la web de «Crosstalk: The Journal of Defense Software Engineering», en el número de Enero de 2003 en <http://www.crosstalkonline.org/>).
- 4.10 Instale en su computadora la herramienta OSRMT y cree con ella un escenario a partir de un caso de uso que contenga un flujo principal y otro secundario. Por ejemplo, el caso de uso podría basarse en la descripción del escenario de sacar dinero de un cajero y contar con dos flujos secundarios: el cliente no tiene saldo suficiente y el cajero sólo tiene billetes de 20 euros, lo que no permite extraer el importe originalmente solicitado por el usuario y por lo que se indica que se introduzca un nuevo importe que sea múltiplo de 20 euros. OSRMT puede descargarse gratuitamente de <http://sourceforge.net/projects/osrmt/>

En es
tieri
men
el tur
cuant
tal po
corta
mos e
bosq
tablu
comp
comp
tar el
const
blem
ción
desar
del p

Diseño

Es difícil hojear un libro de magia y no mirar de vez en cuando a la portada para asegurarse de que no se trata de un libro de diseño de software.

— Bruce Tognazzini

5.1 No es posible construir sin diseñar

En este capítulo más que en ningún otro viene al caso la vieja comparación entre la Ingeniería del Software y la arquitectura civil. En todo caso, la frase que sirve de encabezamiento a esta sección —no es posible construir sin diseñar— es más cierta cuanto mayor es el tamaño del sistema en construcción. Una caseta para el perro se puede hacer con unas cuantas tablas, quizás reciclando alguna cosilla por ahí, un martillo (herramienta fundamental para toda obra en casa) y clavos, muchos clavos. Si somos un poco ordenados, incluso cortaremos las tablas a medida y la pintaremos al final. Pero por muy orgullosos que estemos una vez concluida nuestra obra, casi seguro que no habremos necesitado hacer ningún bosquejo previo a la construcción: basta con conseguir el material para ponernos a cortar tablas y a clavar. Incluso para una caseta más grande y sofisticada, lo más probable es que compremos una estructura prefabricada (estilo Ikea) y en este caso —segundo en orden de complejidad en la escala del aprendiz de constructor— seguramente sólo hará falta consultar el plano a la hora de montarla si la cosa va terriblemente mal. Los equivalentes en la construcción de proyectos informáticos serían pequeños trozos de código hechos muy posiblemente para ser desechados posteriormente (prototipos, programas sencillos de transformación entre formatos, las prácticas de nuestros primeros años en la universidad, etc.), todos desarrollados por un solo individuo y donde lo que prima es el bajo coste económico y salir del paso lo más honradamente posible.

En toda obra civil los arquitectos plasman los elementos componentes en distintos tipos de planos y otras especificaciones. Además, una vez terminados los trabajos hay que superar procesos de garantía de habitabilidad que garanticen los niveles de seguridad y confort requeridos por la ley. Así, a la hora de construir un edificio o vivienda, se detallarán en forma de planos las plantas, habitaciones, localización, etc. En cada plano se superpondrán distintas vistas o aspectos de interés, mostrando detalladamente las conducciones de agua en el edificio, las acometidas de gas –si las hubiera–, la disposición del cableado para la electricidad y las comunicaciones (telefónicas, TV, radio y redes informáticas), el aire acondicionado, etc. Otras especificaciones incluirán los materiales utilizados, que se plasmarán en una memoria de calidades, un plan de mantenimiento... En todo ello se ven involucradas personas de distintos gremios, cada uno de los cuales no sólo lleva a cabo la construcción de aquello en que es especialista, sino que además produce documentación sobre su trabajo: entregables, por decirlo así. En proyectos informáticos de equivalente envergadura se muestra imprescindible la Ingeniería del Software, que no lo es en aquellos que por su simplicidad –como los mencionados anteriormente– puedan llevarse a cabo sin un proceso organizado y controlado. Es en estos proyectos donde nos encontraremos con licitaciones, participación de distintos equipos y roles, procesos definidos a seguir donde se especifican la documentación necesaria, distintos tipos de pruebas y aceptación por parte del cliente que hay que superar... Se especificarán también unas garantías y un mantenimiento que hay que garantizar. En este capítulo se proporciona una visión general de los conceptos fundamentales del diseño de sistemas de software, de los diferentes tipos de diseño existentes y de sus diferentes vistas.

5.2 Objetivos

Los nuevos métodos de construcción de sistemas han dado al diseño un papel esencial dentro del ciclo de vida de desarrollo del software. Este capítulo tiene como objetivo fundamental introducir el concepto de diseño y resaltar su importancia en la construcción y el mantenimiento de un software de calidad. En cuanto a los objetivos específicos, podemos mencionar los siguientes:

- Introducir al lector en los conceptos fundamentales del diseño del software, reflejando su importancia.
- Describir el proceso de la fase de diseño, y los documentos y modelos que se generan durante en esta fase para su documentación y evaluación.
- Mostrar los tipos básicos de diseño que en la actualidad se aplican al software.
- Introducir y describir el propósito de la arquitectura del software.
- Introducir los fundamentos de la calidad en el diseño del software y sus propiedades.

5.3 Introducción

En la fase de diseño, tomando como punto de partida los requisitos (funcionales y no funcionales), se pretende obtener una descripción de la *mejor* solución software/hardware que dé soporte a dichos requisitos, teniendo no solamente en cuenta aspectos técnicos, sino también aspectos de calidad, coste y plazos de desarrollo. Idealmente, se deberían plantear varios diseños alternativos que cumplan con los requisitos, para posteriormente hacer una elección de acuerdo a criterios de coste, esfuerzo de desarrollo o de calidad tales como la facilidad de mantenimiento. Es importante resaltar que en esta fase se pasa del *qué* (obtenido en la fase de requisitos) al *cómo* (que es el objetivo de la fase de diseño).

Veamos un ejemplo muy sencillo de lo anterior, plasmado en un sistema de facturación. En este sistema, el *qué* (un requisito funcional), podría ser que las facturas se impriman ordenadas por código postal para que la empresa se beneficie de la rebaja que por ello hace la oficina de correos. En cuanto al *cómo*, habría que pensar en el diseño del módulo de ordenación y la comunicación con el proceso general de facturación, y a más bajo nivel, en la decisión del algoritmo de ordenación más adecuado de entre los que cumplen los requisitos funcionales y no funcionales.

Además de todo ello, en la fase de diseño se pasa a un ámbito más técnico donde se trabaja fundamentalmente el lenguaje de los desarrolladores ya que los productos del diseño son documentos y artefactos orientados a ingenieros del software, y poco o nada el de los clientes, pues la comunicación con los clientes o usuarios es limitada, si bien puede ser necesario generar documentación de diseño para los clientes llegado el caso. Con todo lo anterior en mente, es momento de definir formalmente el diseño. Concretamente, el Glosario IEEE de Términos de Ingeniería del Software (IEEE, 1990) lo define indistintamente con las dos acepciones siguientes:

El **diseño** puede definirse como (1) el proceso para definir la arquitectura, los componentes, los interfaces, y otras características de un sistema o un componente, y (2) como el resultado de este proceso

En esta definición se mencionan varios elementos importantes en los diseños software como por ejemplo la arquitectura, que tiene que ver —como estudiaremos más adelante— con la descomposición de un sistema en sus componentes e interfaces. No obstante, no aclara un hecho importante sobre la estructura del propio proceso de diseño, y que resulta esencial introducir ya. Nos referimos a que el diseño se descompone claramente en dos subprocesos que son los siguientes:

- Diseño de la arquitectura o de alto nivel, en el cual se describe cómo descomponer el sistema y organizarlo en los diferentes componentes (lo que se conoce como «la arquitectura del software»).

- Diseño detallado, en el que se describe el comportamiento específico de cada uno de los componentes de software identificados.

Así, y en términos aún muy generales, podemos decir que la arquitectura es la visión de alto nivel que posteriormente se detalla hasta el nivel de componentes, elementos modulares que representan la solución final.

5.4 Conceptos fundamentales de diseño

En esta sección se cubren los conceptos fundamentales de diseño, tales como abstracción y modularidad, que la guía SWEBOk denomina *principios de diseño de software* y que son los siguientes:

- Abstracción.
- Acoplamiento y cohesión.
- Descomposición y modularización.
- Encapsulamiento y ocultación de información.
- Separación de interfaz e implementación.
- Suficiencia, complejidad y sencillez.

Hablaremos de todos ellos con mayor profundidad, pero además estudiaremos aquellos aspectos que resultan clave para que las propiedades que definen un buen diseño –de acuerdo con los principios enunciados– se verifiquen.

5.4.1 Abstracción

Como en el resto de los problemas de ingeniería, en el desarrollo de una solución de software, el resultado se representará de forma abstracta con diferentes grados de detalle. Partiendo desde un nivel de abstracción alto, y refinando dicha solución, se llega hasta un nivel de detalle próximo a la implementación. Así, en el desarrollo de un sistema se diferencian tres tipos fundamentales de abstracciones:

- Abstracción de datos. Define un objeto compuesto por un conjunto de datos. La abstracción *cliente*, por ejemplo, incluirá todos los datos de un cliente tal y como se entiende en el contexto de la aplicación en desarrollo, que podrían ser *nombre*, *dirección*, *teléfono*, etc.
- Abstracción de control. Define un sistema de control de un software sin describir información sobre su funcionamiento interno. Una abstracción de control típica es el *semáforo* para describir la coordinación en el funcionamiento de un sistema operativo.
- Abstracción procedimental. Aquella que se refiere a la secuencia de pasos que conforman un proceso determinado, por ejemplo un algoritmo de ordenación.

Estos tipos de abstracciones se utilizarán a lo largo de todo el diseño, si bien el nivel de detalle irá aumentando a medida que el diseño avanza. Así, los primeros esquemas o bocetos del diseño contendrán abstracciones de muy alto nivel, donde tal vez se oculten un buen número de detalles que en este punto no sean aún necesarios. En los últimos momentos del diseño detallado, los diagramas tendrán un nivel de abstracción mucho menor pues la proximidad a la fase de construcción obliga a proporcionar detalles que permitan comprender sin ambigüedades lo expresado.

5.4.2 Componentes e interfaces

Uno de los objetivos primordiales del diseño es la especificación de los componentes, módulos o fragmentos software del sistema, y del modo en que éstos se comunican, pero sin describir sus detalles internos. Estas comunicaciones se expresan mediante la especificación de las operaciones que los componentes exponen para que otros puedan usarlos, lo que comúnmente se conoce como *interfaces*.

Un componente es una parte funcional de un sistema que oculta su implementación proveyendo su realización a través de un conjunto de interfaces

Por tanto, un componente es generalmente un elemento reemplazable y autocontenido dentro de una arquitectura bien definida que se comunicará con otros componentes a través de interfaces.

Una **interfaz** describe la frontera de comunicación entre dos entidades software, definiendo explícitamente el modo en que un componente interacciona con otros

Nótese que en otros contextos el término *interfaz* se utiliza con diversos significados que nada tienen que ver con el que aquí le damos, como en el caso de la interfaz de usuario.

5.4.3 Descomposición y modularización

La descomposición y modularidad son consecuencia de la complejidad de los problemas y de la necesidad de simplificar la solución de los mismos. Así, para abarcar el desarrollo de un sistema complejo, el problema se divide en subproblemas más fácilmente manejables, que integrados formarán la solución al sistema completo. Más formalmente, Meyer (1999) definió las propiedades para evaluar la modularidad:

- **Descomposición:** esta propiedad permite definir componentes de alto nivel en otros de bajo nivel. Esta descomposición a menudo es recursiva, lo que implica aplicar una y otra vez la máxima «divide y vencerás» hasta alcanzar el punto en que cada módulo puede ser desarrollado individualmente.

- **Composición:** es el problema inverso a la descomposición. Un módulo de programación preserva la composición modular si facilita el diseño de elementos de programación que se pueden ensamblar entre sí para desarrollar aplicaciones. Un ejemplo de composición modular son los componentes ya desarrollados que se describen en la Sección 5.6.2. La composición modular está directamente vinculada con la reutilización, pues se busca diseñar elementos que respondan a funcionalidades bien definidas y reutilizables en diversos de contextos.
- **Comprendión:** un método de programación preserva la comprensión modular si facilita el diseño de elementos de programación que se pueden interpretar fácilmente sin tener que conocer el resto de los módulos. Un aspecto fundamental de la comprensión es la documentación, y en el caso particular de los componentes ya desarrollados, la gestión y clasificación de los mismos para facilitar su reutilización.
- **Continuidad:** un pequeño cambio en la especificación debe implicar un cambio igualmente pequeño en la implementación. Una de las leyes de Lehman (1996) sobre la evolución del software (ver Sección 8.5.2), afirma que si un proyecto está vivo inevitablemente evolucionará y cambiará. Por tanto, es importante que los cambios en los requisitos repercutan en un número limitado y localizado de módulos.
- **Protección:** Según esta propiedad, los efectos de las anomalías de ejecución han de quedar confinados al módulo donde se produjo el error, o a un número limitado de módulos con los que éste interacciona directamente. En la validación de los datos introducidos por el usuario, por ejemplo, la validación ha de llevarse a cabo en los módulos que tratan la entrada, no permitiendo que los datos incorrectos se propaguen a los módulos de proceso.

5.4.4 Medición de la modularidad

Las propiedades y reglas mencionadas anteriormente buscan que las dependencias entre módulos sean mínimas. Dichas dependencias se suelen medir utilizando los conceptos de acoplamiento y cohesión que se definen a continuación.

Acoplamiento

El acoplamiento mide el grado de interconexión existente entre los módulos en los que se ha dividido el diseño de la arquitectura de un sistema software.

El objetivo es conseguir un acoplamiento bajo entre módulos –acoplamiento débil– pues genera sistemas más fáciles de entender, mantener y modificar. Con un acoplamiento débil entre componentes los cambios en la interfaz de un componente afectarían un número reducido de cambios en otros. En consecuencia, debemos acercarnos al mínimo número de relaciones posibles entre todos los módulos, que será, si n es el número de módulos de

en sistema, aquel en que un módulo sólo se comunicaría con otro módulo: $(n - 1)$ comunicaciones entre módulos (Figura 5.1-a). Por el contrario debemos alejarnos del máximo número de conexiones, $(n \cdot (n - 1)) / 2$ –acoplamiento fuerte– (Figura 5.1-b).

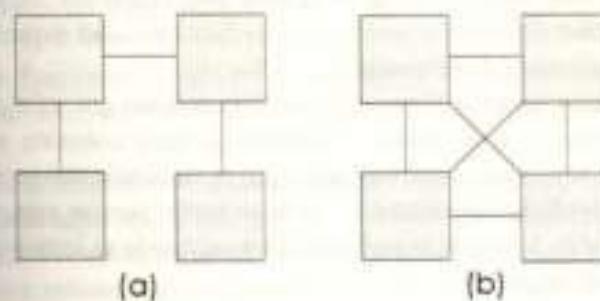


Figura 5.1: (a) Acoplamiento mínimo (b) Acoplamiento máximo

Además del número de conexiones, el grado de acoplamiento puede depender de la complejidad de las mismas, los lugares donde se realicen las referencias a los módulos o el volumen y tipo de datos que intercambian los módulos.

Reglas de modularidad de Meyer

Meyer (1999) propuso 5 reglas que hacen referencia a la conexión entre programas y a la interacción entre módulos, que se deben preservar para garantizar la modularidad.

- **Correspondencia directa.** Debe existir una relación coherente (correspondencia) entre el dominio del problema y la solución.
- **Limitación en el número de interfaces.** Se debe reducir al máximo el número de comunicaciones entre módulos.
- **Limitación del tamaño de las interfaces.** Además del número de comunicaciones entre módulos (regla anterior), se debe limitar al mínimo el tamaño de la información intercambiada entre los mismos.
- **Facilidad de reconocimiento de las interfaces.** La comunicación entre módulos tiene que ser pública y reconocible, es decir, las interfaces deben ser explícitas.
- **Ocultación de la información.** Se debe obtener una modularización que aísle los cambios al menor número posible de módulos. En el diseño de un módulo se especificarán qué propiedades del módulo constituirán la información a la cual tendrán acceso los otros módulos, lo que compondrá la interfaz del módulo (Parnas, 1972).

Cohesión

Otro aspecto fundamental del diseño, también derivado de una concepción modular del mismo, es la cohesión. Un subsistema o módulo tiene un alto grado de cohesión si todos sus

elementos mantienen una funcionalidad común. Por ejemplo, una clase dedicada al manejo de fechas, tiene sólo operaciones relacionadas con las fechas y no otras funcionalidades.

El objetivo es diseñar módulos robustos y altamente cohesionados cuyos elementos estén fuerte relacionados entre sí buscando la **cohesión funcional**, en la que un módulo realiza operaciones bien definidas y suscritas a una funcionalidad requerida, facilitando la comprensión y potenciando su reutilización.

Cuando los módulos agrupan elementos por otros motivos que no sean de estricta funcionalidad, la cohesión no será óptima. Ejemplos de mala cohesión serían la *cohesión secuencial*, en la que la funcionalidad se agrupa por la secuencia de ejecución, la *cohesión por comunicación*, donde los elementos se agrupan juntos porque comparten los mismos datos, o la peor de todas, la *cohesión por coincidencia*, donde se incluyen funcionalidades sin ningún orden (cajón de sastre).

5.4.5 Arquitectura de sistemas

Con los criterios comentados anteriormente, podemos definir más formalmente el concepto de arquitectura. Aunque es un concepto muy amplio, el estándar IEEE de Prácticas Recomendadas para la Descripción de Arquitecturas Software (ISO/IEC 42010:2007) define la arquitectura del software como un acuerdo de mínimos:

La **arquitectura** de un sistema software es la organización fundamental de dicho sistema plasmada en sus componentes, las relaciones entre éstos y con el entorno, y los principios que guían su diseño e implementación.

El estándar mencionado sólo define las prácticas recomendadas, por lo que su adopción e interpretación son responsabilidad de las organizaciones que las implementan. Se trata de un marco general en el que se tratan los siguientes puntos:

- Identificar todas las personas interesadas en el proyecto y sus intereses.
- Seleccionar e identificar los puntos de vista para los distintos intereses.
- Documentar las vistas de la arquitectura que satisfacen los puntos de vista.
- Documentar las inconsistencias entre vistas.
- Establecer una base para las decisiones sobre la arquitectura.

Ya hemos comentado que la complejidad de especificar, diseñar y construir un sistema software requiere diferentes perspectivas. El diseño arquitectónico nos permite organizar estos diferentes puntos de vista y controlar el desarrollo del sistema mediante la organización, evaluación y selección tanto de los aspectos estructurales (para cumplir con los

requisitos funcionales), como de los aspectos no estructurales (para cumplir con los requisitos no funcionales). Las diferentes aproximaciones al diseño describirán los sistemas con distintas técnicas, por lo que dependiendo del tipo de proyecto, seleccionaremos la técnica más apropiada de entre las siguientes:

- Arquitectura funcional, basada en la descripción de las distintas funciones del sistema. En este enfoque, el criterio de división en subsistemas puede ser:
 - Vertical o funcional de usuario, donde la descomposición se basa en agrupar funcionalidades que provienen de –y están reflejadas en– los requisitos funcionales. Por ejemplo, en una aplicación para la enseñanza virtual, podrían considerarse subsistemas la gestión de contenidos, la gestión de comunicaciones con los alumnos, y el subsistema de gestión y administración académica (grupos, notas, etc.).
 - Transversal a las funcionalidades, incluyendo todos los aspectos que no son específicos de ciertos grupos funcionales. Siguiendo el ejemplo, un subsistema de seguridad (políticas de control y derechos de acceso) o un subsistema de persistencia que agrupa todos los servicios relacionados con el almacenamiento.
- Estilos arquitectónicos que implementan la arquitectura funcional, describiendo por ejemplo las capas que lo componen, sus componentes y el modo en que los distintos elementos interaccionan entre ellos.
- Arquitectura de la base de datos, tal y como se describe en la Sección 5.5.2.
- Arquitectura hardware y de red que componen el sistema, por ejemplo en un sistema de telefonía influyen la elección del tipo de red –SMDS, ATM, etc.– y de sistemas operativos para tiempo real.

En resumen, la arquitectura definirá la estructura interna del software, pero deteniéndose en el nivel de los componentes e interfaces que afectan a la interrelación entre componentes. Así, incluirá información sobre los protocolos de comunicación, así como todo lo relacionado con la sincronización y distribución física del sistema, pero no tratará ni el diseño detallado ni los algoritmos a implementar.

Estilos arquitectónicos

La expresión «estilo arquitectónico» hace referencia a cada uno de los diferentes modelos de representación conceptual en que se pueden organizar los componentes dentro de una arquitectura de software. A continuación se clasifican los estilos arquitectónicos más básicos de estructuras de software, siendo posible articular formas complejas mediante la composición de estos estilos fundamentales:

- Filtro-tubería o procesamiento por lotes. Modelo basado en los flujos de datos, donde un componente transforma una entrada en una salida que a su vez es la entrada para otro componente (Figura 5.2). Aunque se trata de sistemas conceptualmente sencillos y fáciles de mantener, el estilo filtro-tubería no resulta adecuado para sistemas que necesitan interactividad, siendo su principal desventaja el rendimiento, pues los datos se transmiten en forma completa entre módulos.

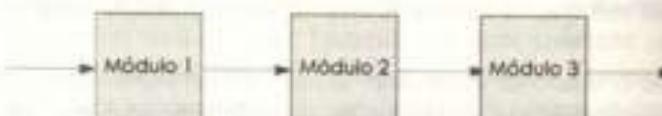


Figura 5.2: Estilo arquitectónico de procesamiento por lotes

- Orientación a objetos. En este estilo, la arquitectura se representa principalmente mediante un diagrama de clases, que se podría definir como un grafo cuyos nodos son objetos y los arcos son los conectores que comunican los objetos mediante los métodos. Lo estudiaremos con más detalle más adelante.
- Arquitectura basada en eventos (invocación implícita). En lugar de producirse invocaciones de métodos explícitas, en este modelo los componentes anuncian sus eventos y otros registran su interés en ser notificados cuando se produzcan ciertos eventos. Así, la ocurrencia de un evento causa la invocación *implícita* de métodos en otros módulos cada vez que se produce. Por ejemplo, en la interfaz de usuario, el clic del ratón dispara la ejecución de ciertas funciones previamente asociadas a dicho evento.
- Arquitecturas basadas en capas. Los componentes están organizados jerárquicamente por capas, donde cada capa provee servicios a la capa inmediatamente superior recibiendo servicios de la capa inmediatamente inferior. Las capas o componentes del sistema se conectan entre sí mediante los protocolos definidos de interacción entre las mismas. Un ejemplo típico es el diseño de un sistema operativo (Figura 5.3).

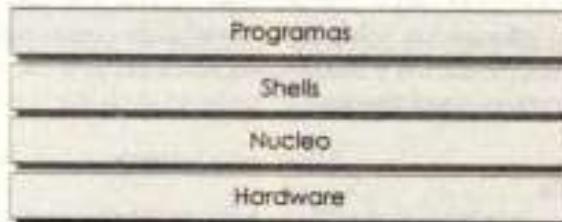


Figura 5.3: Arquitectura en capas de un sistema operativo

- Sistemas basados en repositorios o de datos compartidos. En este estilo, existe una estructura central de datos, independiente de los componentes, a la cual acceden los

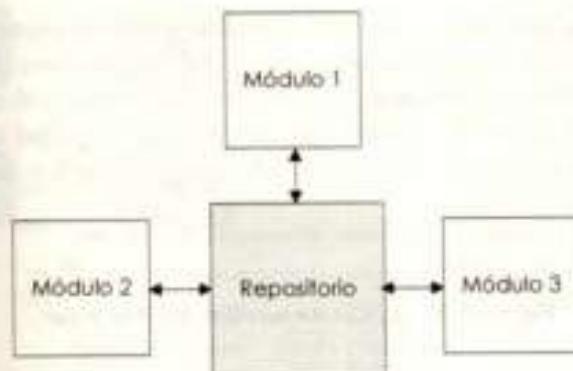


Figura 5.4: Ejemplo de arquitectura basada en repositorios

distintos componentes, los cuales operan en función de los datos del mismo. Es un modelo que permite la integración de distintos agentes, siendo el estado de los datos del repositorio el criterio de selección para el siguiente proceso a ejecutar.

- Control de procesos. Conceptualmente basados en los sistemas de control de los procesos industriales, este estilo asocia componentes de control –un bucle de control (*feedback loop*) en el modelo más simple– bien a la entrada o bien a la salida de un determinado proceso (Figura 5.5) para supervisar su comportamiento y asegurar así que el comportamiento de dicho componente es el esperado.

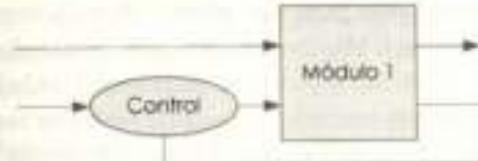


Figura 5.5: Ejemplo de arquitectura de control

- Procesos distribuidos. La funcionalidad global del sistema se divide en diferentes procesos generalmente distribuidos en diferentes máquinas. A su vez se pueden diferenciar arquitecturas distribuidas *cliente-servidor*, como la Web, y arquitecturas *par a par* (P2P – *Peer-to-Peer*) por ejemplo con sistemas de distribución de ficheros TV IP o voz sobre IP (VoIP – *Voice over IP*), esquemas mostrados en la Figura 5.6.

Lenguajes arquitectónicos

Los lenguajes arquitectónicos (*Architecture Description Languages*, ADL) describen arquitecturas software mediante sus componentes, conectores y enlaces de comunicación. Cuando una arquitectura se describe mediante estos lenguajes, es posible tener herramientas para su verificación y prototipado rápido. Existen lenguajes arquitectónicos de propósito

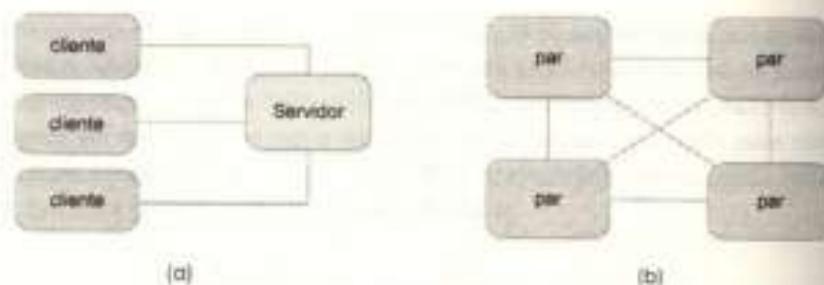


Figura 5.6: (a) Cliente-servidor; (b) Par a par

general y otros de dominio específico (DSSA – *Domain-Specific Software Architectures*). Un ejemplo de lenguaje arquitectónico es ACME (*Architecture Based Languages and Environments*), aunque también puede considerarse UML (de uso mucho más extendido hoy día) para representar arquitecturas. Otros lenguajes como MODAF/DoDAF *Architectural Model* (MAP) y SysML conforman un abanico de diferentes posibilidades cuyo empleo depende de las necesidades de modelado y las especificidades de cada uno.

5.4.6 Notaciones de diseño

Existen multitud de notaciones de diseño para describir la arquitectura de un sistema y su diseño. Algunas de las técnicas son exclusivas de los métodos estructurados, otras de los métodos orientados a objetos, pero es bastante habitual que un mismo proyecto haga uso de notaciones diferentes en las distintas partes del mismo. Hoy día lo más común es basarse en la orientación a objetos y utilizar UML, como lenguaje de modelado, o que un proyecto que emplea UML modele su base de datos mediante diagramas entidad/relación, use tablas de decisión para describir algoritmos concretos y describa mediante *storyboards* la navegación en las interfaces de usuario. Lo cierto es que muchas notaciones se utilizan para ciertas tareas, sin tener muy en cuenta el método empleado. La siguiente sección dedica una parte importante a describir las notaciones más relevantes.

5.5 Métodos de diseño

En esta sección se mostrará cómo han ido evolucionando los métodos y técnicas de diseño desde los métodos estructurados, a la representación de los datos y la orientación a objetos.

5.5.1 Métodos estructurados

La aparición de los primeros lenguajes de programación permitió desarrollar una gran cantidad de código. Sin embargo, se trataba de código muy difícil de mantener por lo que posteriormente, y de modo despectivo, fue denominado *código spaghetti*. A finales de los años 1960, Dijkstra estableció las bases de la programación estructurada demostrando

que todo programa podía escribirse utilizando únicamente bloques secuenciales de instrucciones, instrucciones condicionales y bucles. Con estos lenguajes estructurados aparecieron numerosos métodos de diseño y análisis para sistematizar el desarrollo del software, métodos que se conocen bajo el nombre de *métodos estructurados*. La Figura 5.7 muestra la evolución del análisis y diseño estructurados, enumerando los métodos más importantes.

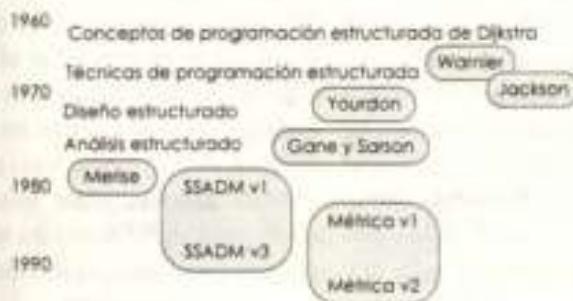


Figura 5.7: Evolución de los métodos estructurados

Los métodos estructurados se basan en una aproximación descendente (*top-down*) que aboga por descomponer el sistema completo en niveles funcionales desde la perspectiva global completa hasta el nivel de detalle necesario para su implementación. Las características más importantes de estos métodos son la descomposición funcional, el modelado de los datos y la representación del flujo de información. Estos tres aspectos conforman las distintas vistas del sistema: la especificación de datos, la especificación de los procesos y la especificación del control –ver Figura 5.8, adaptada de (Pressman, 2001)–.

Entre las técnicas más comunes para describir estas vistas en los métodos estructurados se incluyen las siguientes:

- **Diagramas de flujo de datos (DFD):** los diagramas de flujo de datos se asemejan a un grafo que representa los flujos de datos y las transformaciones que se aplican sobre ellos. Los nodos representan procesos y los vértices las entradas y salidas a los mismos. Las entradas y salidas pueden ser externas al sistema y puede haber almacenes de datos entre los nodos. Una característica distintiva de los DFD es que pueden descomponerse en otros sub-diagramas hasta llegar al nivel de granularidad adecuado para el diseño, siguiendo una aproximación descendente. La Figura 5.9 muestra un ejemplo de descomposición de DFD en posibles niveles. El nivel superior se denomina *nivel de contexto*, y a los procesos que no se descomponen se les denomina *procesos primitivos*. Los componentes de los DFD son:

- *Procesos*, describen las funcionalidades del sistema.
- *Almacenes*, representan los datos utilizados por el sistema.
- *Entidades externas*, fuentes o destinos de la información del sistema.
- *Flujos de datos*, muestran el trasiego de datos entre las funciones.

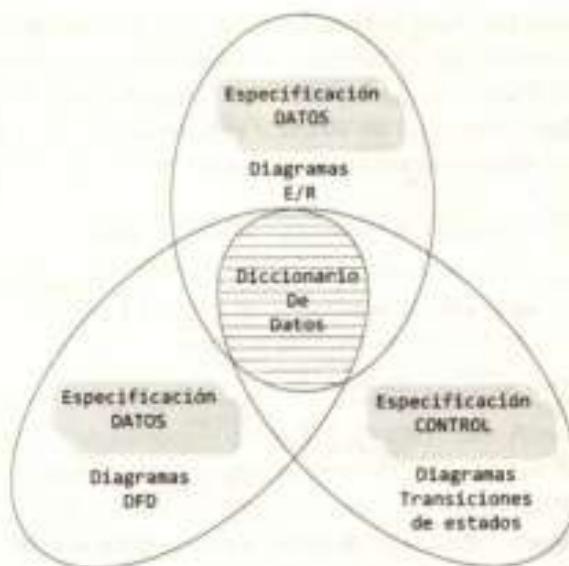


Figura 5.8: Vistas de los métodos estructurados

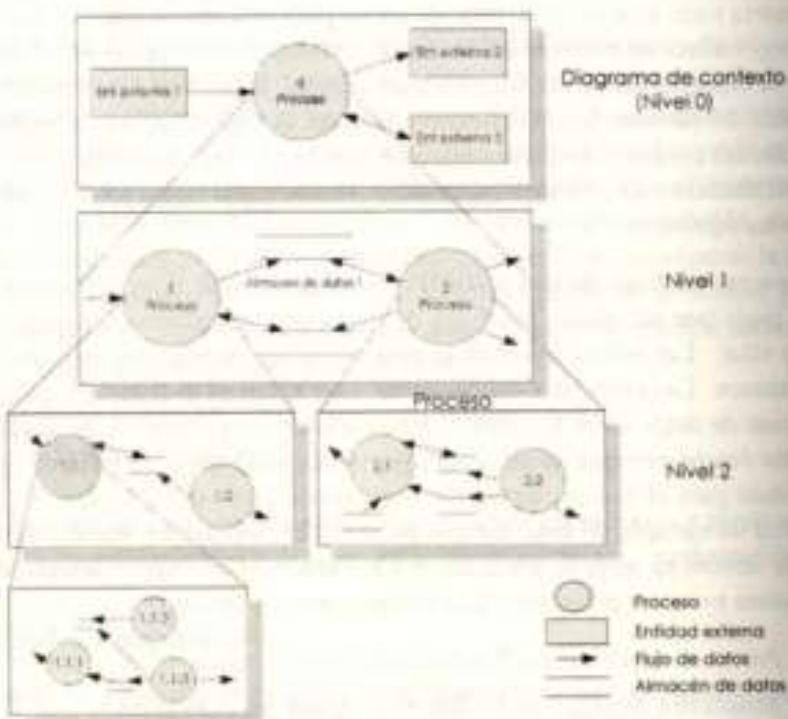


Figura 5.9: Niveles en los diagramas de flujo de datos

- **Diagramas entidad-relación (E/R):** propuesto por Chen en 1976 para describir las entidades existentes en el dominio del problema y sus relaciones. Emplea tres elementos básicos: entidades, relaciones y atributos, y dada su importancia, se trató con detalle en el Capítulo 4.
- **Diccionarios de datos:** contienen los datos utilizados en el sistema, para que todos los participantes del proyecto tengan la misma visión de la información manejada. Representan la información entre los flujos de datos y los almacenes del sistema. Más abajo se muestra un ejemplo de diccionario de datos para un registro de clientes y direcciones en una aplicación de gestión. En este ejemplo, la información del cliente consta de una clave única (DNI), nombre, dirección y múltiples números de teléfono, y las direcciones incluyen calle, número y provincia o bien un apartado de correos. En la notación utilizada, el símbolo @ precede a todo identificador, las llaves indican iteración, el símbolo + representa composición y los corchetes selección:

```

CLIENTE = @DNI + NOMBRE_CLIENTE + DIRECCION + {NUM_TFNO}
DIRECCION = [CALLE + NUM + PROV | APT_CORREOS ]

```

- **Diagramas de estructura:** permiten representar gráficamente la estructura modular en un sistema estructurado, es decir, la jerarquía de módulos junto con las llamadas entre los mismos. Tal y como muestra la Figura 5.10, se obtienen transformando los DFD en árboles con las funcionalidades básicas del sistema. Pero además de la descomposición del sistema en sus módulos, los diagramas de estructura muestran también información sobre la secuencia de ejecución (secuencial, repetitiva y alternativa), control y datos enviados o recibidos.

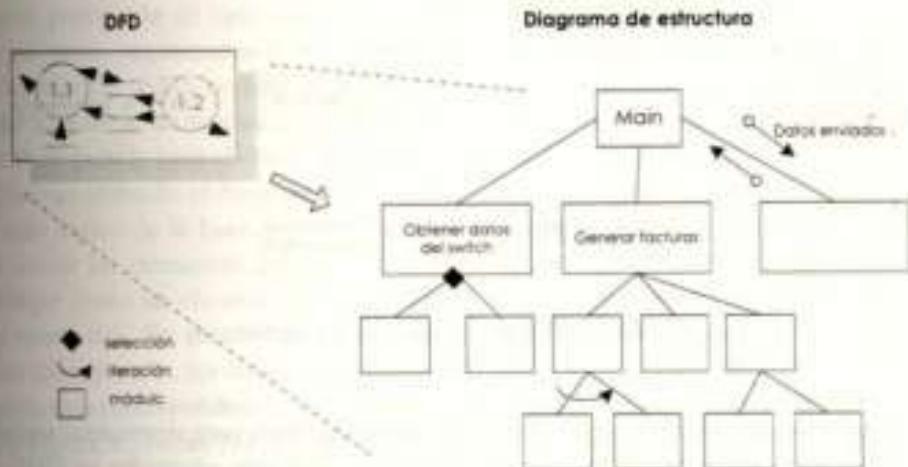


Figura 5.10: Transformación de DFD a diagramas de estructura

En cuanto a la descripción de los módulos individuales, ésta puede hacerse mediante numerosas técnicas, que incluyen:

- Las *tablas de transiciones*, también llamadas tablas de decisión, definen en forma matricial reglas con acciones a realizar dadas ciertas condiciones, tal y como muestra el ejemplo de la Figura 5.11.

	Regla		
	1	2	3
Condiciones			
Edad > 60	Sí	No	Sí
Estadio I	No	Sí	No
Estadio II	No	No	Sí
Acción			
Tratamiento A	X	X	-
Tratamiento B	-	X	X
Tratamiento C	-	-	X

Figura 5.11: Ejemplo de tabla de transiciones

- *Árboles de decisión*, similares a las tablas de decisión pero en forma de árbol, tal y como muestra el ejemplo de la Figura 5.12.



Figura 5.12: Ejemplo de árbol de decisión

- *Diagramas de estados*, técnica originalmente desarrollada para representar autómatas finitos pero que se utiliza en muchas otras áreas. Han sido adoptados en UML y se describen en más detalle en la Sección 5.5.3.
- *Otros modos de descripción*: pseudocódigo, diagramas de flujo, etc.

Otros métodos clásicos de diseño

Dentro de los métodos clásicos, también existen otras técnicas que por la importancia que tuvieron en el pasado se enumeran y describen brevemente:

- *Lenguajes de diseño de programar*: conocidos por sus siglas en inglés (PDL, *Program Design Languages*), expresan la lógica de los programas de manera similar al pseudocódigo. Surgieron como sustitutos de los diagramas de flujo y tenían la ventaja de poder ser validados mediante herramientas software (Budgen, 2003).
- *Métodos estructurados orientados a datos*: muchos de los programas en lenguajes procedimentales como COBOL, estaban orientados a la transformación y manipulación de datos en registros, siendo los datos el verdadero núcleo del sistema. Métodos representativos de este grupo fueron el método Warnier (1981) y JDM (*Jackson Development Method*). Este último método (Jackson, 1975) fue especialmente relevante al ser la base de otros métodos posteriores, inicialmente basado en las tres estructuras básicas para modelar programas del teorema de Dijkstra (secuenciación, iteración y selección), por lo que fue después extendido a otras fases del ciclo de vida y automatización en la generación de código.
- *JAD (Joint Application Development)*: esta técnica, ya descrita en el Capítulo 4 fue muy popular, por lo que se extendió a la fase de diseño. Siendo una técnica exploratoria para involucrar a los usuarios en el proceso de desarrollo, se aplicó al diseño a través de estudios sobre cómo abordar las reuniones en la fase de diseño en una misma localización, si era importante realizar reuniones virtuales, etc.

5.5.2 Métodos orientados a datos

Un gran porcentaje de aplicaciones y prácticamente las aplicaciones de gestión manejan datos, organizados bien en archivos o bien mediante bases de datos. El modelo entidad/relación (E/R), no sólo es una de las formas más comunes de expresar los resultados del análisis de un sistema en etapas tempranas del desarrollo, sino que además suele ser el punto de partida para el diseño de las bases de datos en los sistemas.

Una vez obtenido el diagrama E/R (modelo conceptual), éste se normaliza para obtener el diseño lógico de la base de datos. Ello se consigue aplicando un conjunto de reglas a cada uno de los elementos del modelo conceptual para que, conservando la semántica, se identifique como un elemento del modelo lógico, que a menudo es el modelo de base de datos relacional. En muchos casos la transformación es directa porque el concepto es el mismo (por ejemplo, las entidades pasan a ser tablas en la base de datos), pero otras veces no existe esta correspondencia y es necesario un proceso de transformación más elaborado.

La Figura 5.13 muestra –simplificada– la transformación desde un modelo conceptual en E/R a tablas en una base de datos. En este ejemplo, la relación *participa N : M* –muchos a muchos– se transforma en una tabla intermedia, conservando así la semántica de la relación. En cuanto al modelo físico, reflejará aspectos de implementación que no se deducen del modelo lógico de manera evidente, tales como índices, tipos de estructuras, etc.

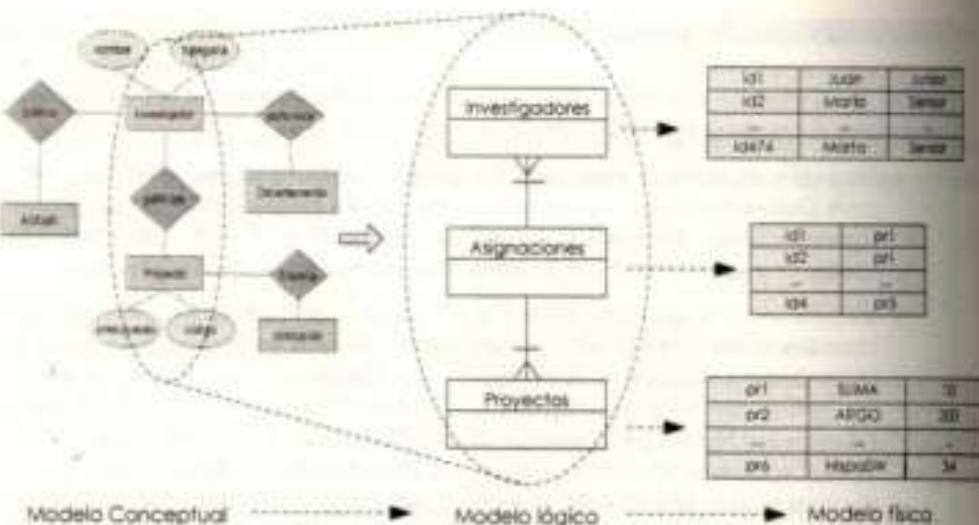


Figura 5.13: Transición desde el modelo E/R al diseño físico de bases de datos

5.5.3 Diseño orientado a objetos

El paradigma de la programación orientada a objetos se remonta a 1967, año en que los investigadores noruegos O.J. Dahl y K. Nygaard desarrollaron los conceptos básicos de este modelo en un lenguaje llamado SIMULA. Hoy en día, éste es el paradigma de programación más utilizado y los lenguajes orientados a objetos –la mayoría influenciados por SIMULA tales como Smalltalk, C++, Eiffel, Java o C#— cumplen ciertas propiedades fundamentales de la orientación a objetos (ver cuadro dedicado) encaminadas a mejorar la calidad del software producido.

Al igual que con los métodos estructurados, también con la aparición de los lenguajes orientados a objetos surgieron numerosos métodos de diseño orientados a objetos, con distintas notaciones y herramientas, dando lugar a lo que algunos denominaron «la guerra de las metodologías». Entre las metodologías orientadas a objetos destacaron la de Booch, OMT (*Object Modeling Technique*) y OOSE/*Objectory* cuyos principales autores, Booch, Rumbaugh y Jacobson, se unieron dando lugar al lenguaje unificado de modelado (UML - *Unified Modeling Language*) y al *Proceso Unificado*.

Con el tiempo, UML ha ido aglutinando las notaciones más importantes de distintas metodologías, convirtiéndose en el lenguaje estándar y de referencia para el diseño orientado a objetos. Se trata, como afirman sus autores, de un lenguaje para visualizar, especificar, construir y documentar sistemas en general pero particularmente adaptado a sistemas software construidos mediante el paradigma de la orientación a objetos.

Propiedades de la orientación a objetos

- **Abstracción:** se reduce la complejidad del dominio abstrayendo hasta el nivel adecuado. En la orientación a objetos la abstracción se representa mediante el concepto de clase, que representa un conjunto de objetos concretos, llamados *instancias*, que tienen propiedades y operaciones comunes.
- **Herencia:** esta propiedad permite definir una clase a partir de otras –una o más– clases existentes, de modo tal que la nueva clase hereda las características de la(s) superclase(s), a las que se añadirán ciertas características propias.
- **Encapsulamiento:** los datos y operaciones de una clase están organizados para que los clientes de la clase sólo necesiten conocer la interfaz pública de la misma. Así, sabrán cómo invocarlas porque conocerán cómo pueden hacerlo, pero no cómo están implementadas.
- **Polimorfismo:** Propiedad que permite que un mismo nombre de método esté asociado a distintos comportamientos, pudiendo ser de dos tipos: estático o dinámico. El *polimorfismo* estático consiste en que operaciones con el mismo nombre pueden realizarse sobre distintos tipos de parámetros, lo que se consigue mediante plantillas, elementos genéricos o sobrecarga de operadores. El *polimorfismo* dinámico está intimamente relacionado con la herencia y acarrea la asignación tardía o dinámica de memoria (*dynamic binding*), propiedad mediante la cual se determina el método concreto a invocar –de entre un abanico de posibles métodos con la misma interfaz dentro de la jerarquía de clases– en tiempo de ejecución. Por ejemplo, cuando una referencia declarada como de tipo *Persona* –superclase de las clases *Alumno* y *Profesor*– está asociada a un objeto de la clase *Alumno*, una llamada al método *getCreditos()* sobre dicha referencia devuelve el número de créditos de los que se ha matriculado un estudiante; por otro lado, si estuviera referenciando a un profesor, retornaría la suma de los créditos de las asignaturas que imparte.

Aunque UML es independiente del proceso que se siga, está generalmente ligado a procesos iterativos e incrementales como el Proceso Unificado y Open UP, descritos en el Capítulo 2. En estos procesos, y también en otros, los sistemas orientados a objetos se describen utilizando las 4+1 vistas (ver cuadro) de Kruchten (1995) mediante modelos y diagramas. Un **modelo** captura una vista de un sistema abstrayendo y describiéndolo un apropiado nivel de detalle. Los modelos a su vez se representan mediante **diagramas**, que son representaciones gráficas de un conjunto de elementos de modelado y sus relaciones.

En UML, los diagramas están clasificados en dos grupos:

- **Diagramas de estructura**, que reflejan la estructura física (estática) del sistema por medio de sus clases, métodos, atributos, interfaces, paquetes, etc. y sus relaciones.
- **Diagramas de comportamiento**, que muestran la forma en los distintos elementos del sistema interaccionan, colaboran y cambian de estado durante la ejecución del sistema para proveer la funcionalidad requerida.

En UML versión 2 existen trece tipos de diagramas, que permiten elaborar los distintos modelos para cada una de las vistas. A continuación se proporciona una visión general de los diagramas más importantes, de los que se puede consultar un resumen en la Tabla 5.1, si bien los diagramas se describen de manera muy general y sin entrar en completo detalle, ya que queda fuera del alcance de este libro el describir toda la notación, muy amplia por otra parte. Además, es importante resaltar que no es necesario utilizar todos los tipos de diagramas en los proyectos, al igual que generalmente no se especifica completamente la diagramas al 100%, es decir, hay un balance entre complejidad y eficiencia desde el punto de vista del diseñador.

Las 4+1 vistas de Kruchten

En procesos como el proceso unificado (*Unified Process*, UP) y OpenUP (ver Capítulo 2), la arquitectura de un sistema se describe mediante cuatro vistas complementarias más una vista adicional de casos de uso que complementa la información de las otras cuatro:

- La *vista de diseño* muestra cómo se llevan a cabo los requisitos funcionales mediante la descomposición del sistema en sus elementos componentes (clases y relaciones).
- La *vista de implementación* describe la organización del sistema en módulos, componentes y paquetes cubriendo el ensamblado del sistema y la gestión de configuración, reutilización y portabilidad.
- La *vista de procesos* describe no sólo los procesos y sus comunicaciones, sino además los hilos de control en las clases (conurrencia y distribución de procesos) cuando el lenguaje lo permite y la implementación lo contempla.
- La *vista de despliegue* se utiliza para representar el conjunto de nodos físicos que forman la topología hardware del sistema.
- La *vista de los casos de uso* describe los requisitos funcionales del sistema utilizados para complementar las otras vistas. Se trata de una vista transversal que se emplea durante todo el ciclo de vida, por ejemplo y entre otros cometidos, para crear los casos de prueba.



Figura 5.14: Las 4 + 1 vistas de Kruchten

Tabla 5.1: Diagramas de UML 2

Diagramas de estructura	
Clases	Es el diagrama más importante. Muestra un conjunto de clases, interfaces y colaboraciones con sus relaciones.
Objetos	Muestra un conjunto de objetos y sus relaciones en un cierto estado. Generalmente, representan la instanciación de un diagrama de clases en un determinado punto en el tiempo.
Componentes	Describe los componentes que conforman una aplicación, sus interrelaciones e interfaces públicas.
Estructura compuesta	Permite visualizar de manera gráfica las partes que definen la estructura interna de un clasificador, incluyendo sus puntos de interacción con otras partes del sistema.
Paquetes	Muestran la organización en paquetes de los diferentes elementos que conforman el sistema, permitiendo especificar de manera visual el nombre de los espacios de nombres.
Despliegue	Muestra la arquitectura física de un sistema, los nodos en sus entornos de ejecución y cómo se conectan.
Diagramas dinámicos	
Actividad	Muestra los procesos de negocio y flujos de datos.
Comunicación	Muestra la organización estructural de los objetos y el paso de mensajes entre ellos.
Interacción	Variante del diagrama de actividades para mostrar el flujo de control de un sistema o proceso de negocio.
Secuencia	Modela la secuencia temporal lógica de mensajes entre participantes (generalmente clases).
Estados	Describe los distintos estados en que puede encontrarse un objeto, junto con las transiciones entre los mismos. Muy útil para representar el comportamiento de clases complejas.
Tiempos	Muestra los cambios de estado o condición producidos en los objetos por eventos externos.
Casos de Uso	Representa funcionalidades del sistema mediante casos de uso, actores y relaciones entre ellos.

Diagrama de casos de uso: como se ha visto en la Sección 4.7.1, los casos de uso son una técnica para la captura y especificación de requisitos, principalmente requisitos funcionales, modelados como funcionalidades que un sistema proporciona en un entorno. Aunque estrechamente hablando la técnica de los casos de uso no pertenece al enfoque orientado a objetos, ha sido adoptada por UML y por ende, al diseño orientado a objetos. Una vez especificados los requisitos, las distintas vistas de un sistema –cinco si se emplean por ejemplo las 4+1 vistas de Kruchten– se pueden describir con el resto de diagramas de UML.



Figura 5.15: Ejemplo de diagrama de casos de uso

Diagrama de clases: es el modelo estático más importante. Muestra las clases, las interfaces y sus relaciones. Las *clases* abstraen las características comunes a un conjunto de objetos, por ejemplo la clase *Persona* representa una abstracción de todas las características comunes al conjunto de individuos que tienen nombre, fecha de nacimiento, sexo, etc. (sin características que los objetos de la clase *Edificio* no tienen). Los objetos son instancias concretas de las clases, por ejemplo *Ángela* o *Íñigo* serían objetos de la clase *Persona*. Los objetos necesitan colaborar entre ellos para llevar a cabo alguna funcionalidad requerida, para lo que resulta esencial que las clases a las que pertenecen estén *asociadas*.

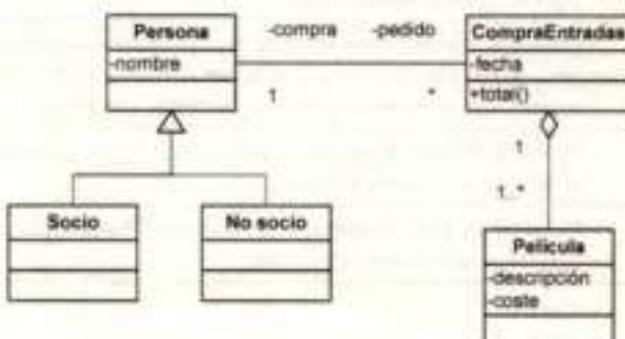


Figura 5.16: Diagrama de clases

Las clases se representan mediante un rectángulo dividido en 3 secciones, la primera tiene el *nombre* de la clase, la segunda sus *atributos* (propiedades) y la tercera las *operaciones* (métodos) de la clase indicando los servicios que proporciona a otras clases. La figura 5.16 muestra un ejemplo de diagrama de clases, donde la clase CompraEntradas almacena las órdenes de compra de entradas de un cine, la fecha sería el atributo y total() uno de los métodos de la clase. La clase tiene conexiones con Persona y Película.

A una instanciación de un diagrama de clases que muestra los objetos y sus relaciones en un instante concreto, es decir, a una «fotografía del sistema» en un determinado punto en el tiempo, es lo que conocemos como **diagrama de objetos**.

Diagrama de componentes: estos diagramas muestran los elementos físicos del sistema tales como librerías, APIs, archivos, etc. y sus relaciones. En un componente se pueden agrupar, por ejemplo, varias clases e interfaces representando cierta funcionalidad agrupada en una biblioteca dinámica o en un programa ejecutable. La Figura 5.17 muestra un ejemplo de diagrama de componentes. La asociación (mostrada mediante una flecha discontinua) muestra la dependencia de un paquete sobre otro, representando el círculo una interfaz.

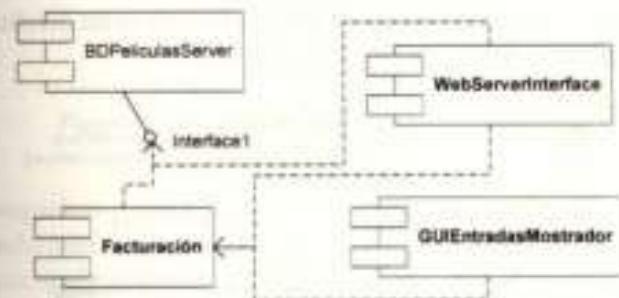


Figura 5.17: Diagrama de componentes

Diagramas de interacción: estos diagramas muestran como interactúan (mediante el paso de mensajes) los objetos o clases entre sí. Pueden ser de dos tipos:

- **Diagramas de secuencia:** resaltan el paso de mensajes en el tiempo, es decir, su ordenación temporal.
- **Diagramas de comunicación:** (anteriormente llamados diagramas de colaboración). Son equivalentes a los diagramas de secuencia, pero destacan la organización en la comunicación entre los objetos sobre la ordenación en el tiempo.

La Figura 5.18 muestra un ejemplo muy simplificado de diagrama de secuencia (a) y su diagrama de comunicación equivalente (b), resaltando el paso de mensajes en el tiempo y la secuencia de mensajes respectivamente.

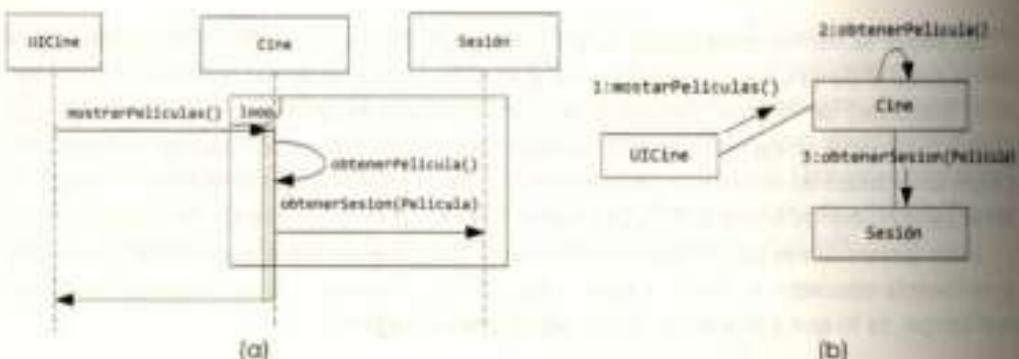


Figura 5.18: Ejemplos de diagrama de secuencia (a) y de comunicación (b)

Diagramas de estados: originalmente creados por Harel (1987), representan autómatas de estados finitos que muestran los estados por los cuales puede pasar un objeto, así como los cambios que permiten pasar de un estado a otro. En la Figura 5.19 los nodos representan estados de la clase y los arcos eventos que realizan la transición entre los estados; la barra que aparece bajo los eventos indica acciones y el texto entre corchetes representa las condiciones para realizar la transición entre estados.

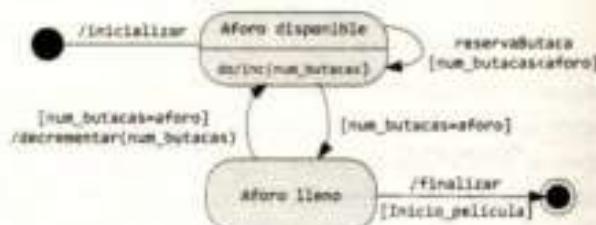


Figura 5.19: Diagrama de estados

Diagramas de despliegue: muestran la arquitectura del hardware del sistema y la distribución física de los componentes software en el mismo (un ejemplo en la Figura 5.20).

Diagramas de actividad: un diagrama de actividad muestra paso a paso la estructura de un proceso o algoritmo en forma de flujo de control. Así, representa cómo se hacen y qué ocurre durante las acciones, así como su secuencia, pudiendo además mostrar actividades paralelas, tomas de decisión, etc. Si bien la notación es similar a la de los diagramas de estados, los nodos de los diagramas de actividad muestran actividades, mientras que en los diagramas de estados los nodos muestran estados «fijos». La Figura 5.21 muestra un ejemplo de diagrama de actividad.

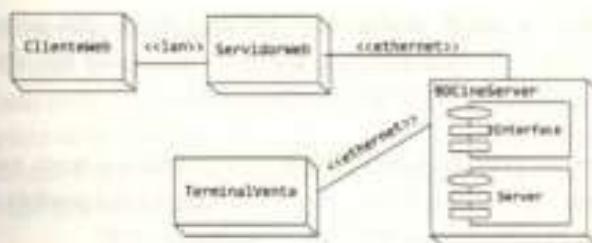


Figura 5.20: Ejemplo de diagrama de despliegue

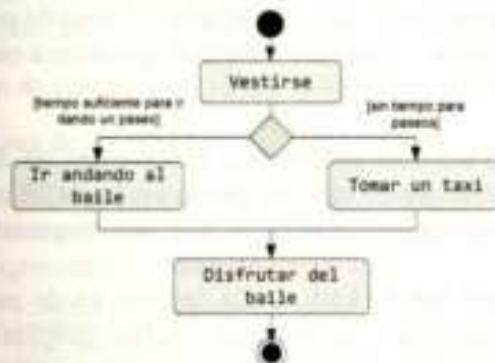


Figura 5.21: Ejemplo de diagrama de actividad

Diagramas de paquetes: muestra la organización lógica en paquetes de los diferentes elementos que conforman el sistema, de forma que se puede especificar de manera visual el nombre de los espacios de nombres. La Figura 5.22 muestra un ejemplo.

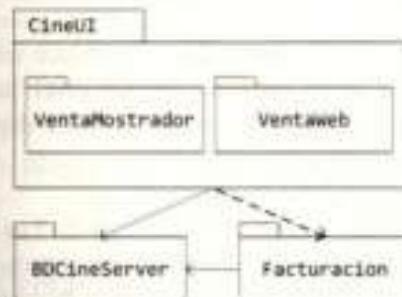


Figura 5.22: Ejemplo de diagrama de paquetes

En lo anterior se ha proporcionado una visión general de los diagramas UML, pero sin entrar en profundidad en las notaciones, o en la forma correcta de relacionar y unir componentes, clases, etc. en un diagrama, puesto que, como ya hemos dicho, esto queda fuera del alcance de este libro y su aplicación depende mucho de la experiencia. Se proporcionan, para terminar, unos principios básicos.

Principios del diseño orientado a objetos: en un buen diseño orientado a objetos se aspira a alcanzar una serie de principios que permiten conseguir los objetivos de un bajo acoplamiento y una alta cohesión. Estos principios son:

- **Principio abierto-cerrado** (Meyer, 1999). Un módulo debería estar a la vez abierto (que siempre sea posible ampliarlo) y cerrado (que no sea posible editar ni modificar funcionalidades del mismo que estén en funcionamiento).
- **Principio de responsabilidad única.** Introducido por DeMarco (1979) en el diseño estructurado, está relacionado con el concepto de cohesión pues aboga por que una clase debe tener sólo una razón para cambiar lo que lleva a que cada clase tenga una única responsabilidad.
- **Principio de separación de la interfaz.** Los clientes no deberían ser forzados a depender de interfaces que no utilizan. En otras palabras, es preferible tener muchas interfaces específicas que una sola interfaz de propósito general.
- **Principio de sustitución de Liskov.** La herencia ha de garantizar que cualquier propiedad que sea cierta para los objetos de la clase, también lo sea para los objetos de sus subclases. En otras palabras, las subclases deben poder sustituirse por la clase base.
- **Ley de Demeter.** Establece que cada unidad debería tener solamente un conocimiento limitado sobre otras unidades –sólo conocer las unidades con quienes tiene que relacionarse una unidad–. Informalmente descrita como «*no hables con extraños*», esta ley busca mejorar el acoplamiento entre clases.
- **Principio de inversión de dependencias.** Establece que (1) los módulos de alto nivel no deben depender de los módulos de bajo nivel, pues ambos deben depender de las abstracciones; y (2) las abstracciones no deben depender de los detalles, sino los detalles de las abstracciones.
- **Principio de dependencias estables.** (Martin, 1995) Las dependencias entre paquetes en un diseño deberían ir encaminadas a la estabilidad de los paquetes, midiéndose la estabilidad por el número de dependencias de entrada y salida de un paquete. Cuantas más dependencias de entrada más estable necesitará un paquete ser, ya que cualquier cambio afectará a todos los paquetes que dependen de él.

Finalmente, el paradigma de la orientación a objetos sigue evolucionando con técnicas como la *programación orientada a aspectos*, la publicación de nuevos estándares y notaciones, o la aparición de herramientas como el lenguaje de restricción de objetos (OCL). Este último se describe brevemente a continuación.

El Lenguaje de restricción de objetos (OCL): UML incorpora OCL (*Object Constraint Language*) para extender la semántica de los elementos de UML. Se trata de un lenguaje que se ha incorporado en herramientas capaces de traducir restricciones escritas en OCL a asecciones del lenguaje de programación, demostrando su utilidad no sólo en el diseño sino también en la verificación y prueba de programas. OCL es un lenguaje perteneciente a la lógica de primer orden pensado para:

- Especificar *invariantes* en las clases, tipos o interfaces.
- Describir *precondiciones* y *postcondiciones* en los métodos, guardas de los diagramas de estados, actividad, etc.
- Definir esquemas de navegación y reglas de formación en grafos.
- Definir restricciones de operaciones.

El ejemplo de la Figura 5.23 muestra un diagrama de clases donde el número de pasajeros de un autobús está limitado por el número de asientos disponibles, una restricción de cardinalidad que no puede representarse utilizando sólo UML.

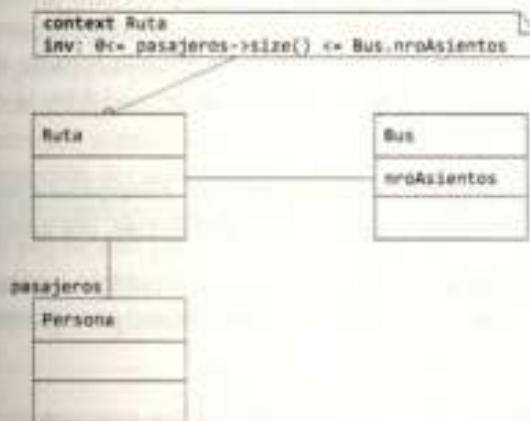


Figura 5.23: Diagrama donde se incluyen restricciones en OCL

5.6 Otras técnicas relacionadas con el diseño

5.6.1 Los patrones de diseño software

Los patrones tienen su origen en el trabajo de C. Alexander en 1979 en un libro titulado «*The Timeless Way of Building*» —que podría traducirse como *una forma intemporal de construir*—, obra en la que se describen los fundamentos de un lenguaje de patrones para que individuos o grupos de individuos construyan sus propias viviendas sin necesidad de arquitectos. En este texto, ya clásico, el autor afirma lo siguiente:

«Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podremos volver a utilizar esta solución un millón de veces en el futuro sin tener que volver a pensarla de nuevo».

En el campo de la Ingeniería del Software, Cunningham y Beck propusieron algunos patrones de interfaz de usuario a finales de la década de 1980, pero no fue hasta la publicación de la obra de Gamma *et al.* (1995) cuando el estudio y uso de patrones se convirtió en algo común. En la actualidad podemos atestiguar que se han especificado multitud de patrones en diferentes dominios y niveles de abstracción, de tal modo que en la industria del software los patrones se suelen clasificar de acuerdo con el ámbito que abarcan, algo a lo que se ha dado a llamar *lenguajes de patrones*. Así, existen los siguientes lenguajes de patrones:

- Patrones de interfaces de usuario, interacción hombre-computador. El más conocido es quizás el MVC (modelo-vista-controlador).
- Patrones de diseño, conjuntos de clases para modelos de orientación a objetos.
- Modismos (*programming idioms*), «trucos» en lenguajes de programación concretos (C++, Java, etc.) que a veces son simples reglas a la hora de escribir código, como por ejemplo prefijos para identificar el tipo de las variables (a este respecto véase la Sección 6.6.1).
- Patrones para la integración de sistemas (EAI - *Enterprise Application Integration*) para la intercomunicación y coordinación de sistemas, por ejemplo para Java EE.
- Patrones de organización y flujos de trabajo, *workflows*, para la gestión de flujos de trabajo y procesos con sistemas empresariales, por ejemplo cómo organizar al personal y su forma de trabajar.

En particular, en este capítulo nos interesan aquellos patrones de diseño que describen soluciones a problemas al nivel de las clases, sus relaciones e interfaces.

Un patrón de diseño es una solución a un problema recurrente de carácter general en términos de clases, interfaces, objetos e interacciones entre estos elementos

Para que una solución sea considerada un patrón debe poseer ciertas características que lo validan. Una de ellas es que debe haberse comprobado su efectividad resolviendo problemas similares anteriormente, lo que se ha dado en llamar *la regla de tres*. Esta regla afirma que algo no es un patrón a menos que se haya usado en tres ocasiones independientes. Otras son la comparación con otras posibles soluciones mostrando sus debilidades, que no haya sido escritas por un único autor y que hayan sido criticadas por la comunidad.

Aunque, lógicamente, el empleo de patrones también tiene sus inconvenientes –por ejemplo la empinada curva de aprendizaje inicial–, una vez introducidos en los entornos de desarrollo proporcionan múltiples ventajas:

- La reutilización de arquitecturas completas, yendo un paso más allá de la reutilización de código. Además, representan conocimiento sobre decisiones de diseño.
- Formalizan un vocabulario común entre diseñadores.
- Facilitan el aprendizaje, condensando conocimiento ya existente.

El libro de Gamma *et al* (1995), referencia esencial en el área, describe 23 patrones concretos mediante una plantilla común con los campos principales: nombre del patrón, propósito, motivación, aplicabilidad, desarrollo del patrón, usos y ejemplos. Los autores además clasifican los patrones en tres categorías: de creación (de ayuda a la hora de crear objetos especialmente cuando se trata de tomar decisiones durante el proceso de creación), estructurales (de ayuda para modelar el modo en que las clases y los objetos se combinan para dar lugar a estructuras más complejas) y de comportamiento (de ayuda en el modelado de la comunicación e interacción entre los objetos de un sistema).

Como ejemplo de patrón, el patrón de comportamiento *Observer* define una dependencia de *uno a muchos* entre objetos, de modo que cuando un objeto cambia de estado todos los objetos dependientes son notificados y actualizados automáticamente. La motivación de este patrón sería la reducción del acoplamiento entre clases de manera que un objeto puede notificar a otros sin tener conocimiento de cuáles son aquellos objetos que requieren la notificación. Un posible uso de este patrón son las interfaces de usuario, donde unos mismos datos pueden tener distintas vistas, cada una de las cuales recibe una notificación cuando los datos cambian. La estructura se puede observar en el diagrama de clases de la Figura 5.24.

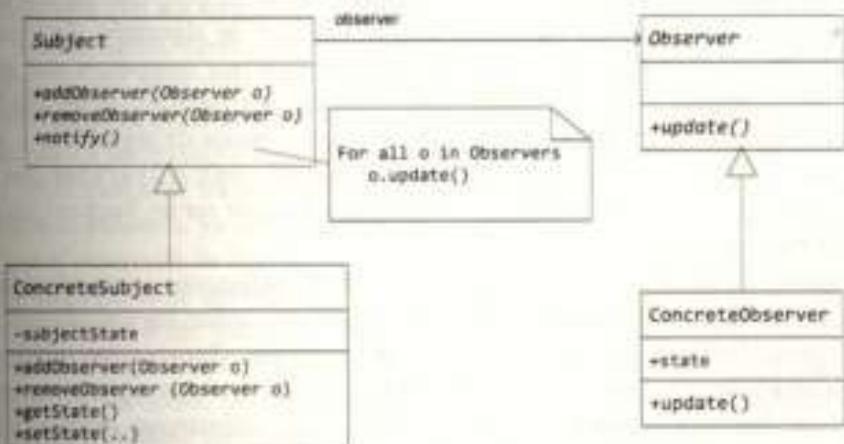


Figura 5.24: Patrón Observer

En algunos lenguajes como Java ya viene incluso implementada la clase `Observer` y la interfaz `Observable`. La clase `java.util.Observable` es la clase abstracta `Subject` del patrón, que proporciona métodos para añadir, borrar y modificar observadores que son guardados en un tipo `Vector`. Cualquier clase que quiera ser observada debe extender esta clase. Las subclases sólo son responsables de la llamada de notificación cuando el estado cambia, generalmente dentro de una método `<set...(...)>`. Un bosquejo de código que implementa el patrón con las clases proporcionadas en Java sería algo así:

```
interface Subject {
    public void addObserver(Observer o);
    public void removeObserver(Observer o);
    public String getState();
    public void setState(String state);
}

interface Observer {
    public void update(Subject o);
}

class ObserverConcreto implements Observer {
    private String state = "";

    public void update(Subject o) {
        estado = o.getState();
        System.out.println("Nuevo estado" + estado);
    }
}
```

Otros ejemplos de patrones muy conocidos incluyen el patrón `Strategy`, utilizado para implementar distintos algoritmos que comparten interfaz pero que pueden utilizarse indistintamente, con la ventaja de que se puede cambiar de algoritmo dinámicamente. `Composite`, utilizado para construir jerarquías de objetos que comparten una interfaz y en la que unos elementos pueden formar parte de otros (por ejemplo, en las interfaces de usuario las ventanas se componen de paneles, botones campos de texto, etc. y a su vez por ejemplo los botones de una etiqueta de texto y una imagen); o `Iterator` que permite recorrer los elementos de una clase contenadora (como puede ser un `Vector`) sin conocer su implementación.

Finalmente, ha de hacerse una reseña al concepto de *antipatrón*, a menudo relacionado con los patrones de diseño. Aunque se describen de manera similar, los antipatrones intentan prevenir contra errores comunes en el diseño del software. Representan problemas que ciertos diseños acarrean, diseños que a primera vista pueden parecer correctos, pero donde a la larga las desventajas superan a los beneficios. Los antipatrones pueden definirse a distintos niveles, por lo que incluso se han detallado antipatrones de gestión. En diseño, los antipatrones puede clasificarse como *refactorizaciones* (ver más detalles en el Capítulo 8).

5.6.2 Software frameworks, plug-ins y componentes

Parece evidente que resulta mucho mejor la reutilización de arquitecturas y diseños completos a la reutilización a nivel de clases. Los patrones de diseño nos permiten reutilizar un pequeño número de clases en conjunto, pero los *frameworks*, *plug-ins* y componentes van más allá, permitiéndonos reutilizar diseños y arquitecturas enteras en distintos dominios.

Marcos de trabajo

Un *framework*¹ abstrae las entidades, estados y comportamientos en un dominio, proporcionando puntos de extensión, denominados *hot-spots* en inglés. La idea es proporcionar toda la funcionalidad genérica que se pueda y mostrar a los desarrolladores las interfaces (API – Application Program Interface) a extender para que sólo tengan que centrarse en lo específico de la aplicación que están desarrollando. La mejor prueba de su aceptación y utilidad es que hoy en día muchos lenguajes de programación van acompañados de marcos de trabajo estándar que se proporcionan como parte del lenguaje.

Un marco de trabajo es un conjunto de clases, interfaces y sus relaciones que proporcionan un diseño reusable para un sistema software o parte de un sistema diseñado específicamente para ser extendido y no como aplicación final

El ejemplo más típico, y origen de muchos de los marcos de trabajo que hoy en día son tan populares, es el de las interfaces de usuario (GUI), donde el usuario se centra en desarrollo de la interfaz abstrandose de cómo se implementan sus distintos componentes (botones, ventanas, adornos, etc.). Otro ejemplo muy popular es el *framework* para pruebas unitarias JUnit que se explica en el Capítulo 7, o los *frameworks* para el desarrollo de aplicaciones web con la típica estructura de bases de datos y operaciones sobre ella, por ejemplo Apache Struts, Spring, etc. Generalmente, estos últimos implementan un patrón muy general de alto nivel conocido como *Modelo-Vista-Controlador*.

La motivación de desarrollar un *framework* es el que un conjunto de clases, interfaces y relaciones vaya a ser reutilizado en múltiples aplicaciones. En estos casos, el desarrollo del conjunto como marco de trabajo ofrece una importante reducción de costes a largo plazo, que justifica el esfuerzo, ya que el desarrollo del *framework* en sí mismo es mucho más costoso que el desarrollo de aplicaciones «normales». Además, en general, los *frameworks* tienen una larga curva de aprendizaje, por lo que en su diseño hay que llegar a un balance entre su simplicidad y las funcionalidades que proporcionan. Es decir, el *framework* debe ser lo suficientemente simple como para que pueda ser aprendido por los desarrolladores extendiendo sus *hot-spots*, pero lógicamente debe suministrar la funcionalidad suficiente para ser útil en el dominio para el que se ha diseñado.

¹ El término *marco de trabajo* es muy poco utilizado, empleándose coñecientemente su denominación en inglés.

Arquitectura dirigida por modelos (MDA)

La MDA (*Model Driven Architecture*) es un *marco de trabajo* ideado para desarrollo mediante modelos. MDA está basado en UML, y propone un desarrollo que se realiza en los siguientes tres niveles:

- Primero, se define un modelo con un alto grado de abstracción, independiente de la tecnología sobre la que se va a desarrollar (PIM, *Platform Independent Model*).
- Segundo, el modelo PIM se transforma en modelos específicos de la plataforma, llamados PSM (*Platform Specific Model*).
- Finalmente, se transforman los PSM en código ejecutable.

La transformación entre los niveles de abstracción, de PIM a PSM y de PSM a código, se realiza mediante herramientas, automatizando el proceso de transformación. En consecuencia, el desarrollo se limita al modelado de alto nivel mediante UML y estándares relacionados como OCL.

Aunque ya empiezan a aparecer algunas herramientas que lo hacen realidad, el principal inconveniente para la adopción de MDA es el gran esfuerzo necesario para especificar los modelos en un nivel de detalle que permita su «compilación», la dificultad de proponer modelos independientes y la transformación entre modelos.

Plug-ins

Además de los *frameworks*, las tendencias actuales para la mejora de la reutilización se encuentran en la creación de *plug-ins*.

Un *plug-in* es una aplicación completa, módulo o componente que interacciona con una aplicación anfitriona extendiéndola o adaptando su comportamiento pero sin modificarla

Ejemplos de utilización cotidiana de *plug-ins* se encuentran en los navegadores web, donde es habitual extender su funcionalidad básica de mostrar páginas en HTML o XML con terceras aplicaciones para ver documentos en pdf, videos, música, etc. En el desarrollo del software con plataformas genéricas –como *Eclipse* y *Netbeans* por mencionar sólo dos de las más conocidas– los *plug-ins* permiten que terceras partes desarrollen aplicaciones con mucha de la funcionalidad ya proporcionada por estas plataformas y sin necesidad de introducir modificaciones en las mismas. Aparte de los aspectos técnicos, la no modificación de la plataforma anfitriona puede ser importante desde el punto de vista legal para evitar problemas por motivos de licencias.

La forma de implementar *plug-ins* se lleva a cabo a menudo mediante un patrón denominado *inversión de control*, que se suele describir con el lema de Hollywood: «*No nos llame, nosotros le llamaremos*». La idea es que la aplicación anfitriona no tenga que ser modificada, esto es, no tenga que conocer los detalles de implementación del *plug-in*.

Componentes

La idea de los componentes de software se asemeja a la idea de componentes hardware con elementos intercambiables. Los componentes de software deben adherirse a una especificación que definen sus interfaces de tal manera que las aplicaciones puedan construirse mediante la composición de interfaces o que terceras partes puedan desarrollar componentes sin tener que mostrar el código fuente. Algunos ejemplos de componentes software son JavaBeans, Microsoft COM, CORBA o los componentes SOAP.

Los componentes son elementos de funcionalidad vendidos como una unidad e incorporados en múltiples usos

En la más pura forma de composición, los sistemas son construidos enteramente de componentes adquiridos (no implementados localmente) que son ensamblados sin modificación. El valor de los componentes cobra mayor sentido cuando se ensamblan unos con otros para formar un sistema como solución a medida. Esta forma de desarrollar sistemas ha dado lugar a la *Ingeniería del Software basada en componentes* que implica realizar una serie de tareas adicionales como la búsqueda, evaluación, selección, compra y reemplazo de componentes, y adaptar otras como las pruebas. También ha dado lugar a la adaptación de metodologías no específicamente creadas para el desarrollo mediante componentes, lo que se ha hecho considerando 3 actividades particulares:

- *Cualificación de componentes*: actividades de análisis para evaluar cada componente.
- *Adaptación de componentes*: procedimientos para que los componentes se adapten a la arquitectura requerida.
- *Composición de componentes*: consideraciones sobre los mecanismos de conexión y coordinación.
- *Actualización de componentes*: actividades para reemplazar componentes, donde deben considerarse especialmente la no disposición del código fuente, la compra a terceros y la dependencia de éstos.

5.6.3 Diseño por contrato

Meyer (1999) enunció el diseño por contrato en el contexto de la construcción de software orientado a objetos. Según este modelo, cada método tiene una precondición y una postcondición que deben cumplirse en la entrada y en la salida del método respectivamente. Para comprobar dichas condiciones pueden usarse aserciones, estableciendo en suma un acuerdo formal (contrato) entre una clase y sus clientes: «*si tú, cliente, me garantizas ciertas precondiciones, entonces yo, proveedor, generaré resultados correctos. Pero si de lo contrario, violas las precondiciones, no te prometo nada*».

Las aserciones se convierten así en la herramienta perfecta para expresar en los contratos los derechos y obligaciones de cada una de las partes. Las obligaciones del cliente mediante precondiciones, y las del proveedor –que vistas desde el otro lado son los derechos del cliente– con postcondiciones. Rosenblum (1995) las define así:

Las aserciones son restricciones formales en el comportamiento de un sistema software comúnmente escritas como anotaciones en el código fuente

Las *aserciones* fueron creadas por Hoare como un sistema de axiomas para demostrar la corrección de programas Algol (Hoare, 1983). Además, al permitir expresar suposiciones del estado del programa durante la ejecución, las se convierten en un modo más de documentación, con la ventaja de que seremos notificados en el caso que nuestras suposiciones sobre el código no fueran ciertas. Siguiendo esta idea, es posible definir las especificaciones de un software orientado a objetos incluso antes de tener el código implementado, expresando las precondiciones y postcondiciones de sus métodos. Y dado que las aserciones son un modo de verificar nuestras suposiciones en cualquier punto del código, también nos facilitan la detección de errores de implementación, lo que permite ahorrar una gran parte del tiempo de pruebas que se gastaría en buscando un error, ya que un fallo en la aserción nos devuelve información detallada sobre dicho error. Esta verificación tomaría la forma {Precondiciones} Código {Postcondiciones}, siendo un ejemplo muy trivial donde se prueba la corrección de un trozo de código el siguiente:

$$\{a \leq 0\} \quad a++; \quad \{a \leq 1\}$$

Diferencias entre bibliotecas de funciones, frameworks, plug-ins y componentes

- Las *bibliotecas de funciones* son almacenes de clases o funciones que son llamadas para realizar una funcionalidad específica sin flujo de control incorporado. Un ejemplo típico es el de una biblioteca de funciones matemáticas.
- Los *frameworks* por su parte incluyen tanto una arquitectura como flujo de control. Están pensadas como aplicaciones incompletas diseñadas para ser extendidas.
- Los patrones de diseño se asemejan a los *frameworks*, pero desde el punto de vista lógico, y no de implementación como los *frameworks*.
- Los *componentes* están diseñados para ser piezas reemplazables dentro de un entorno donde se asume que las aplicaciones se desarrollarán mediante composición.
- Los *plug-ins* se asemejan a los componentes, pero están diseñados para ser llamados desde una aplicación anfitriona.

5.7 Diseño de sistemas distribuidos

Los sistemas distribuidos se encuentran en todos los ámbitos de la vida: redes de computadoras, sistemas de telefonía móvil, robots industriales, coches, aviones, etc. En esta sección se comentan brevemente las características de estos sistemas, y cómo éstas impactan en el diseño de sistemas y sus implementaciones. Como veremos, los sistemas distribuidos tienen conceptos y técnicas particulares (Coulouris, Dollimore y Kindberg, 2001):

Un sistema distribuido es aquel cuyos componentes de hardware o software se encuentran localizados en diferentes puntos dentro de una red de computadoras, comunicándose y coordinando sus acciones únicamente mediante el paso de mensajes

La diferencia entre las redes de computadoras y los sistemas distribuidos es el nivel en el que se encuentran. Mientras que las redes de computadoras se centran en la interconexión y el envío de información, los sistemas distribuidos muestran al usuario de forma transparente un conjunto de computadoras trabajando de forma autónoma en una misma aplicación. Las motivaciones al crear sistemas distribuidos son varias:

- Razones funcionales, donde la distribución de la información es inherente al sistema (web, correo, sistemas de compra-venta cliente/servidor, etc.).
- Aspectos relacionados con la distribución y el balanceo de carga en los servidores.
- Motivos de fiabilidad, por ejemplo tener sistemas redundantes y copias de respaldo en diferentes localizaciones.
- Razones económicas, ya que por ejemplo es mucho más barata la construcción de computadoras potentes a base de agrupaciones (*clústeres*) de computadoras más asequibles. También es una motivación económica frecuente la compartición de recursos tales como impresoras, sistemas de respaldo, etc.

Un sistema distribuido puede ser desde un *clúster* de computadoras homogéneas en una red local situadas todas ellas en una misma oficina, hasta un sistema de computadoras completamente heterogéneo –en cuanto a capacidades, sistemas operativos, etc.– que se encuentra distribuido por distintos países y comunicándose a través de múltiples redes (wireless, Ethernet, ATM, etc.). Como consecuencia derivada de su propia naturaleza, los sistemas distribuidos deben tener en cuenta un conjunto de cuestiones que incluyen:

- **Concurrencia:** proporcionar y gestionar los accesos concurrentes a los recursos compartidos. En los sistemas distribuidos se gestionan computadoras autónomas que trabajan en paralelo, por lo que la coordinación de tareas ha de tener en cuenta la no existencia de un «reloj» global en la comunicación. Otra cuestión importante es la

preservación de las dependencias, evitando puntos muertos y proporcionando un acceso justo a los recursos.

- **Heterogeneidad:** la existencia de múltiples tipos de computadoras, redes y sistemas operativos obliga a la creación de protocolos abiertos para su permitir su comunicación por paso de mensajes.
- **Transparencia:** la distribución debería ocultarse a usuarios y desarrolladores. Hay diferentes tipos:
 - *Transparencia en el acceso:* el acceso a ficheros locales o remotos debería ser idéntico (como por ejemplo sucede con los sistemas de ficheros en red).
 - *Transparencia en la localización:* la aplicación distribuida debería permitir el acceso a los recursos independientemente de donde se encuentren (por ejemplo, los servicios web o CORBA pueden dinámicamente descubrir e invocar servicios independientemente de su localización).
- **Tolerancia a fallos:** los sistemas distribuidos necesitan recuperarse de forma transparente de los fallos en cualquiera de sus componentes. Para ello deben implementar mecanismos de detección de fallos, enmascaramiento de los mismos, tratamiento de excepciones, recuperación mediante mecanismos de *rollback*, etc.
- **Escalabilidad:** los sistemas distribuidos deberían funcionar eficientemente cuando se incrementa el número de usuarios, o escalar la eficiencia en consonancia con el número de recursos nuevos que se ponen en la red.
- **Seguridad:** un sistema distribuido debe tener muy en cuenta la confidencialidad de los datos, impidiendo que individuos no autorizados accedan a información no permitida. Igualmente, debe velar por el mantenimiento de la integridad, protegerse contra la alteración o corrupción de información, estar preparado para ataques malintencionados y el no-repudio mediante pruebas de envío y recepción de la información. Todo esto se consigue mediante técnicas específicas de encriptación, autenticación y autorización.

Dentro de los estilos arquitectónicos que hemos comentado, los que se pueden englobar dentro de los sistemas distribuidos incluyen cliente-servidor, *peer-to-peer* y *grid*. Existe una variedad de tecnologías para implementar dichos estilos, entre las que se encuentran las siguientes:

- **Servicios web.** Un servicio web es un sistema software identificado por una URI (*Uniform Resource Identifier*), cuyas interfaces públicas y puntos de enlace (conocidos por el término inglés *bindings*) están definidos y descritos en XML. Su definición puede ser descubierta por sistemas software, los cuales pueden entonces interaccionar

con el servicio web de la forma prescrita en su definición, utilizando mensajes basados en XML basados en protocolos de internet.

- **CORBA** (*Common Request Broker Architecture*) es un *middleware* para la programación concurrente de sistemas distribuidos mediante la orientación a objetos. Proporciona una plataforma para invocar objetos en servidores remotos, que pueden encontrarse en la misma computadora o en otra conectada a través de redes. Utilizando CORBA, una clase puede transparentemente invocar métodos de otras clases sin conocer su localización (en qué máquina se encuentra dicha clase).
- Otras tecnologías que permiten implementar sistemas distribuidos específicas de lenguajes de programación o plataformas. Entre éstas se encuentran **RMI** (*Remote Method Invocation*) en Java o **DCOM** (*Distributed Component Object Model*) en MS-Windows.

5.8 Evaluación y métricas en el diseño

Los diseños deben ser evaluados para medir distintos criterios de calidad y comparar alternativas, ya que –y esto es un principio esencial del diseño– dada una especificación pueden existir varios diseños perfectamente válidos pero con distintos criterios de calidad. Entre los criterios más importantes y que ya hemos visto en la Sección 5.4.4 están los de cohesión y acoplamiento, como medida de la modularidad de un sistema. Otros criterios más generales están relacionados con los modelos de calidad tal y como se introdujeron en el Capítulo 3, modelos que se describen en más detalle en el Capítulo 9. Estos modelos de calidad pueden generarse siguiendo GQM (ver Sección 3.7.1), y suelen mencionar o incluir factores de los cuales muchos resultan relevantes para el diseño:

- **Extensibilidad.** Debe ser posible añadir nueva funcionalidad sin necesidad de hacer cambios significativos en la arquitectura.
- **Solidez.** Poder operar bajo presión y tolerar entradas inválidas o impredecibles.
- **Fiabilidad.** La función requerida debe llevarse a cabo en las condiciones deseadas durante un tiempo especificado.
- **Tolerancia a fallos.** El sistema debe ser robusto y ser capaz de recuperarse ante fallos.
- **Compatibilidad.** Capacidad para operar con otros productos, a menudo dependiente de su adherencia a estándares de interoperabilidad.
- **Reusabilidad.** Los componentes deberían capturar la funcionalidad esperada, ni más ni menos. Esta única finalidad, hace que los componentes sean reutilizables en otros diseños con idénticas necesidades.

Un tipo de validación, aunque menos común, es la validación matemática. Los diseños se pueden validar matemáticamente si se ha seguido una metodología formal en la especificación de requisitos. Si el diseño de la arquitectura se ha especificado utilizando algún lenguaje formal de definición de arquitectura, pueden validarse y compararse distintas alternativas mediante herramientas específicas antes de su implementación definitiva.

Otras técnicas más comunes son las revisiones e inspecciones. Al igual que en otras fases del ciclo de vida, se pueden realizar unas reuniones llamadas revisiones o inspecciones de diseño donde se analiza los artefactos que son generados en esta fase. Las revisiones de diseño se suelen clasificar según el nivel de detalle en:

- Preliminares, donde se examina el diseño conceptual con los clientes y usuarios.
- Críticas, donde el diseño se presenta a los desarrolladores.
- Revisiones del diseño del programa, donde los programadores obtienen *feedback* antes de la implementación. A estas reuniones también se las conoce como inspecciones (o *walkthroughs* si el moderador de la reunión es la persona que ha generado el diseño, siendo generalmente más informales).

A menudo resulta interesante la medición de la complejidad del diseño, para lo que se emplean métricas de su complejidad estructural. El objetivo último es identificar partes del diseño que puedan simplificarse en aras de la facilidad de mantenimiento, por ejemplo. Esta medición de la complejidad estructural se realiza midiendo las llamadas entre módulos con los conceptos de *fan-in* y *fan-out* (ver Figura 5.25):

- *Fan-in*, o grado de dependencia de un módulo, es el número de módulos que llaman a dicho módulo.
- *Fan-out*, o grado de responsabilidad de coordinación de un módulo, es el número de módulos que son llamados por dicho módulo.

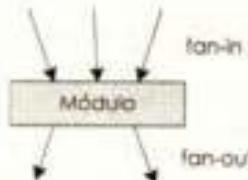


Figura 5.25: Conceptos de *fan-in* y *fan-out*

Un valor alto de *fan-in* indica que el módulo está fuertemente acoplado, por lo que los cambios en el mismo afectarán al resto del sistema. Valores altos de *fan-out* pueden indicar módulos complejos debido a la complejidad de la lógica de control necesaria para coordinar las llamadas al módulo.

Basándose en estos conceptos de flujo de información, Henry y Kafura (1981) enuncian una conocida métrica de la **complejidad estructural**:

$$HK_m = C_m \cdot (fan - in_m \cdot fan - out_m)^2$$

donde C_m es la complejidad del módulo generalmente medida en *LoC*, aunque pueden considerarse otras, como la complejidad ciclomática (ver Sección 3.4).

Basándose en la teoría de grafos, Fenton (1997) propone medidas de morfología de la estructura jerárquica de módulos del sistema:

- Tamaño = número de nodos + número de aristas.
- Profundidad de anidamiento.
- Anchura.
- Proporción de arcos y nodos

Dentro de la orientación a objetos, las métricas más ampliamente difundidas son las conocidas como MOOD (*Metrics for Object Oriented Design*) definidas por Brito-Abreu y Melo (1996) y las de Chidamber y Kemerer (1994). En líneas generales, ambos grupos de métricas identifican clases mal diseñadas mediante mediciones de mecanismos estructurales básicos tales como la encapsulación, la herencia, el polimorfismo y el paso de mensajes. Las métricas del sistema, y por ende el conocer si el sistema en general está bien diseñado, pueden derivarse de las métricas de las clases usando la media u otros instrumentos estadísticos. Entre las métricas MOOD podemos citar como ejemplos:

- **Proporción de métodos ocultos (MHF – Method Hiding Factor).** Proporción del número de métodos definidos como protegidos o privados entre el total de métodos. Esta métrica mide la encapsulación.
- **Proporción de atributos ocultos (AHF – Attribute Hiding Factor).** Es la proporción entre los atributos definidos como protegidos o privados y el número total de atributos. Aunque a veces por mejorar el rendimiento se acceden o modifican los atributos directamente, idealmente esta métrica debería ser 100%. Esta métrica también mide la encapsulación.
- **Proporción de métodos heredados (MIF – Method Inheritance Factor).** Se define como la proporción de la suma de todos los métodos heredados en todas las clases entre el número total de métodos (localmente definidos más los heredados) en todas las clases.
- **Proporción de atributos heredados (AIF – Attribute Inheritance Factor).** Definido como la proporción del número de atributos heredados entre el total de atributos. AIF se considera un medio para expresar el nivel de reusabilidad en un sistema.

- **Proporción de polimorfismo** (PF – *Polymorphism Factor*). Proporción entre el número real de posibles diferentes situaciones polimórficas para una clase C_i y el máximo número posible de situaciones polifórficas en C_i . En otras palabras, número de métodos heredados redefinidos dividido entre las posibles situaciones polimórficas distintas. Es una medida indirecta de la asociación dinámica en un sistema.

Al igual que las métricas MOOD, Chidamber y Kemerer (1994) definieron una serie de métricas que han sido ampliamente adoptadas para medir características como encapsulamiento, ocultación de información o herencia. Entre estas métricas tenemos:

- **Acoplamiento entre objetos** (CBO – *Coupling Between Objects*). El CBO de una clase es el número de clases a las cuales está ligada porque usa métodos o variables de la misma (medida del *fan-out*). Cuando una clase tiene un alto CBO y todas las demás tienen valores próximos a cero, el sistema no posee una estructura orientada a objetos pues existe «una clase principal dirigente». Por el contrario, la existencia de muchas clases con un alto valor de CBO indica que el diseñador ha afinado excesivamente la granularidad del sistema. Esta métrica además puede utilizarse para medir el esfuerzo en el mantenimiento y las pruebas. A mayor acoplamiento, mayor dificultad de comprensión y reutilización, mayor dificultad de comprensión más difícil el mantenimiento.
- **Respuesta para una clase** (RFC – *Response For a Class*). Esta métrica cuenta las ocurrencias de llamadas a otras clases desde una clase particular y mide tanto la comunicación interna como la externa. RFC es en suma una medida de la complejidad de una clase a través del número de métodos y de su comunicación.
- **Profundidad del árbol de herencia** (DIT – *Depth of Inheritance Tree*). Mide el máximo nivel en la jerarquía de clases, siendo la cuenta directa de los niveles de profundidad en la misma. En el nivel cero de la jerarquía se encuentra la clase raíz. A medida que crece su valor, es más probable que las clases de niveles inferiores hereden muchos métodos y esto da lugar a posibles dificultades cuando se intenta predecir el comportamiento de una clase y, por lo tanto, su mantenimiento. Una jerarquía profunda conlleva también una mayor complejidad de diseño. Por otro lado, los valores grandes de DIT implican que se pueden reutilizar muchos métodos, lo que debe ser considerado como un elemento a favor de la facilidad de mantenimiento.
- **Número de descendientes** (NOC – *Number of Children*). Cuenta el número de clases subordinadas a otra clase de la jerarquía, es decir, el número de subclases de la misma. A medida que crece el número de descendientes se incrementa la reutilización, pero esto también implica que la abstracción representada por la clase predecesora se ve diluida. Esto dificulta el mantenimiento, ya que existe la posibilidad de que algunos de los descendientes no sean realmente miembros propios de la clase.

- **Métodos ponderados por clase** (*WMC – Weighted Methods per Class*). Mide la complejidad de una clase, dando un peso a cada método y obteniendo la suma de todos ellos. Si todos los métodos son considerados igualmente complejos (mismo peso), WMS es poco más que el número de métodos definidos en la clase.
 - **Falta de cohesión en los métodos** (*LCOM – Lack of Cohesion in Methods*). Es una medida de la bondad de la abstracción de una clase (esto es, si representa realmente una abstracción o más de una). Para ello, compara el número de métodos que usan atributos comunes con el número de métodos que no comparten atributos con otros.

5.9 Resumen

En este capítulo hemos definido los conceptos fundamentales del diseño y proporcionado una visión global de los mecanismos y pasos para su realización. Hemos visto cómo, mediante la abstracción, descomposición y modularización, se lleva a cabo el *diseño de alto nivel*, y cómo después se lleva a cabo un refinamiento o *diseño detallado* de los componentes e interfaces que componen un sistema. Al igual que en otros capítulos, la siguiente sube de palabras muestra la importancia relativa de los conceptos vistos.



Figura 5.26: Principales conceptos tratados en el capítulo.

Hemos comentado cómo las notaciones y técnicas de diseño han ido evolucionando a medida que lo hacían los lenguajes de programación, y cómo si bien hoy día es común que un mismo proyecto utilice distintas notaciones, algunos diagramas son más propios de un tipo de método que de otros (por ejemplo, los diagramas de flujo de datos son más típicos de los métodos estructurados). Por sus peculiaridades, se ha comentado aparte el diseño de los sistemas distribuidos; además, hemos mencionado brevemente otras técnicas relacionadas con el diseño, como los patrones de diseño, la *refactorización*, el uso de *frameworks* y *plugins* y el diseño por contrato. Finalmente, se han descrito los criterios de calidad y algunas técnicas de evaluación del diseño.

5.10 Notas bibliográficas

Existen literalmente cientos de libros que cubren todos los aspectos de diseño tratados en este capítulo, por lo que inevitablemente dejaremos excelentes obras sin mencionar. Entre los libros que cubren exclusivamente la fase de diseño, Budgen (2003) destaca la importancia del diseño en el contexto global del desarrollo del software y cubre tanto las metodologías estructuradas como las orientadas a objetos (aunque resalta más los métodos estructurados y las metodologías tradicionales).

Sobre el diseño con métodos estructurados, las mejores referencias son las de los autores originales de los métodos, como (Gane y Sarson, 1977), (Jackson, 1975) o (Yourdon, 1993), entre otros. Estas referencias, algunas comentadas en el texto, son resumidas en el libro de Budgen que hemos mencionado anteriormente.

En cuanto al diseño orientado a objetos, existen literalmente cientos de excelentes referencias, siendo de obligada lectura la de B. Meyer (1999). Otra de más reciente publicación y que cubre la Ingeniería del Software desde el punto de vista de la orientación a objetos es la de Bruegge y Dutoit (2003). Además, dentro de la orientación a objetos pero centrado especialmente en UML y el proceso unificado, son de referencia obligatoria las obras de sus autores originales (conocidos como «los tres amigos»): G. Booch (2005), I. Rumbaugh (2004) e I. Jacobson (1999). Entre ellos tres han escrito tres volúmenes: una guía del lenguaje UML, una referencia de UML y la descripción del proceso unificado. Otra referencia importante por su sencillez y rápida lectura como introducción a UML es la de Fowler y Kendall (2004). Sobre OCL, la referencia más interesante y recomendada es la de Warmer y Kleppe (1999). También de los últimos autores, es muy recomendable su trabajo sobre introducción a la arquitectura dirigida por modelos, MDA (Warmer y Kleppe, 2003).

El libro de patrones por excelencia es el de Gamma *et al* (1995). Otro más recientemente publicado por Larman (2004) también es muy referenciado. Además de las referencias bibliográficas, en este campo existen multitud de sitios web que explican los patrones principales en diferentes lenguajes de programación. Relacionado con patrones y refactorización la referencia es el libro de Fowler (2004). Los trabajos originales sobre la descripción y elaboración de frameworks son los de Johnson y Foote (1988), aunque existe un libro más reciente igualmente imprescindible (Fayad, Schmidt y Johnson, 1999).

Dos libros de obligada referencia sobre los sistemas distribuidos son los de Coulouris *et al* (2001) y el de Tanenbaum y van Steen (2006). Ambos cubren todos los aspectos relacionados con los sistemas distribuidos y no solamente el aspecto de diseño. Son libros de texto, lo que facilita su lectura y comprensión por parte de estudiantes. Finalmente, en cuanto a las métricas y evaluación, nos remitimos a la referencia principal del capítulo de medición (Fenton y Pfleeger, 1997), y a los artículos citados en el texto (Chidamber y Kemerer, 1994) y (Brito e Abreu y Melo, 1996).

5.11 Cuestiones de autoevaluación

5.1 ¿Qué dos conceptos se utilizan en el diseño de todo sistema, primero a alto nivel y luego se van refinando a bajo nivel?

R. *Componentes e interfaces. Los componentes, como partes funcionales del sistema, interactúan con otros componentes por medio de sus interfaces.*

5.2 ¿Con qué dos conceptos generales se puede medir la modularidad de sistemas software?

R. *Con los conceptos de acoplamiento, que mide el grado de interconexión entre componentes, y cohesión, que mide el grado en que las operaciones de un módulo están ligadas a una funcionalidad bien definida.*

5.3 Brevemente... ¿Cuál es la diferencia entre análisis y diseño en la orientación a objetos?

R. *El análisis consiste en encontrar los conceptos del dominio de la aplicación, es decir, qué se necesita construir (requisitos). En el diseño se definen los objetos y cómo estos colaboran, en otras palabras, cómo se soluciona el problema.*

5.4 ¿Qué diagrama refleja la descomposición modular en métodos estructurados?

R. *Los diagramas de flujo de datos (DFD) reflejan la descomposición modular desde el nivel superior (nivel de contexto) hasta el nivel de detalle próximo a la implementación.*

5.5 En su clásico libro sobre patrones de diseño, Gamma clasifica los patrones presentados dentro de tres categorías ¿Cuáles son dichas categorías?

R. *Gamma identifica patrones de creación (útiles para crear objetos), estructurales (útiles para modelar combinaciones de clases y objetos) y de comportamiento (de ayuda para modelar la comunicación entre objetos).*

5.6 ¿Qué es un lenguaje arquitectónico?

R. *Se trata de lenguajes que describen arquitecturas software, mediante la representación de sus componentes, conectores y enlaces de comunicación en un lenguaje formal, lo que resulta muy útil de cara principalmente a la verificación y prototipado de la arquitectura.*

5.7 Indique si la siguiente afirmación es cierta o falsa y razoné su respuesta: «En el modelado de un sistema se ha de optar por una de las diferentes vistas disponibles en el lenguaje de representación utilizado (por ejemplo la vista de casos de uso de UML)».

R. *Falso. El modelado y diseño de un sistema, independientemente del método empleado (estructurado, orientado a objetos, etc.) puede –y debe– hacerse tomando datos desde distintas vistas, siendo este modo de proceder el único que permitirá capturar toda la riqueza de detalles del sistema. Así, en los métodos estructurados, tendríamos que elaborar 3 vistas del sistema: la especificación de datos, la de procesos y la del control.*

5.8 Indique cómo se llevan a cabo las revisiones e inspecciones durante el diseño de un sistema.

R. *Al igual que en otras fases del ciclo de vida, durante el diseño pueden analizarse los artefactos generados. Estas revisiones generalmente se llevan a cabo mediante inspecciones a diferentes niveles de detalle: Revisiones de diseño preliminares, revisiones de diseño críticas, y revisiones del diseño del programa.*

5.9 ¿Por qué es necesario un lenguaje como OCL?

R. *Para llevar a cabo especificaciones más precisas y sin ambigüedades, pues los modelos gráficos, como los diagramas de clases, no son suficientes para lograrlo.*

5.10 Queremos ampliar y añadir nuevas funcionalidades a una clase ya definida pero sin tener que modificarla. ¿Es posible conseguir lo que se pide con la programación orientada a objetos?

R. *Sí, es posible. lo garantiza el principio abierto-cerrado. Para ello, el mecanismo a utilizar sería la herencia.*

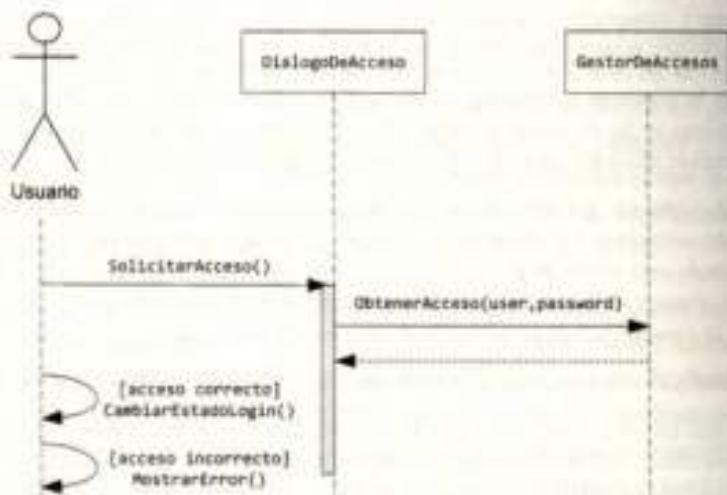
5.12 Ejercicios y actividades propuestas

5.12.1 Ejercicios resueltos

5.1 Se desea modelar una parte de un sistema de bases de datos con acceso restringido a las consultas mediante un diagrama de secuencia de UML. El caso de uso de inicio de sesión viene especificado por la siguiente secuencia de pasos:

1. Se muestra el diálogo de inicio de sesión
2. El usuario introduce el nombre de usuario y contraseña
3. El usuario hace clic en el botón «Aceptar»
4. El nombre de usuario y la contraseña son revisadas y aprobadas por el sistema
5. Se permite el acceso al sistema [Alternativa: si el nombre de usuario y contraseña no son correctas, se solicita al usuario que vuelva a intentarlo]

Solución propuesta: un posible diagrama que represente la interacción descrita en el enunciado podría ser el que se muestra en la figura.



- 5.2 En una determinada aplicación se han definido dos clases, Rectangulo y Cuadrado. La clase Rectangulo se ha definido así :

```
public class Rectangulo {
    private double anchura;
    private double altura;
    public Rectangulo(){}
    public double getAnchura() {return anchura;}
    public double getAltura() {return altura;}
    public void setAnchura(double a) {anchura = a;}
    public void setAltura(double b) {altura = b;}
    public double area() {return (anchura * altura);}
}
```

y la clase Cuadrado se ha definido como subclase de la anterior:

```
public class Cuadrado extends Rectangulo {
    public Cuadrado(){}
    public void setAnchura(double a) {
        super.setAnchura(a);
        super.setAltura(a);
    }
    public void setAltura(double b) {
        super.setAltura(b);
        super.setAnchura(b);
    }
}
```

que son usadas en este fragmento de código:

```
public class Main {
    public static void inicializar(Rectangulo r) {
        r.setAnchura(7);
        r.setAltura(5);
        System.out.println("Área: " + r.area());
    }

    public static void main(String[] args) {
        Rectangulo r = new Rectangulo ();
        Cuadrado c = new Cuadrado ();
        inicializar(r);
        inicializar(c);
    }
}
```

el cual al ser ejecutado produce la siguiente salida:

```
Área: 35.0
Área: 25.0
```

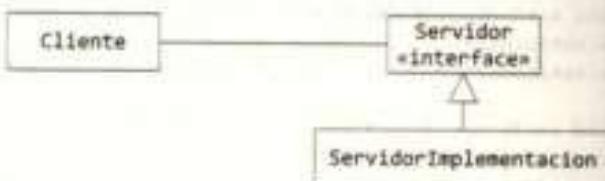
¿Hay algo incorrecto en este diseño? Si es así, ¿Qué principio se ha violado?

Solución propuesta: el método inicializar(Rectangulo r) recibe como referencia un objeto de la clase Rectangulo. Sin embargo, es posible pasar a este método objetos de la clase Cuadrado al ser ésta una subclase de Rectangulo. Ello acarrea que, en el ejemplo, Rectangulo.setAltura() invoque Cuadrado.setAltura(), lo que finalmente asigna el mismo valor a los atributos anchura y altura. En consecuencia, este código viola el principio de sustitución de Liskov, pues Cuadrado no es una subclase correcta de Rectangulo.

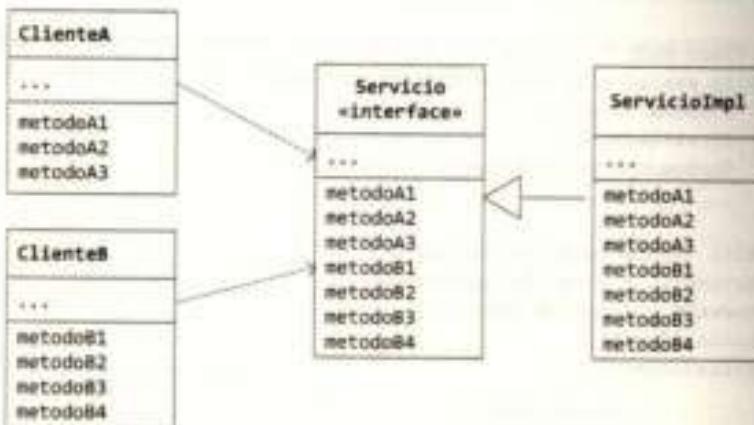
5.3 ¿Qué problema tiene un diseño tan simple como el de la siguiente figura?



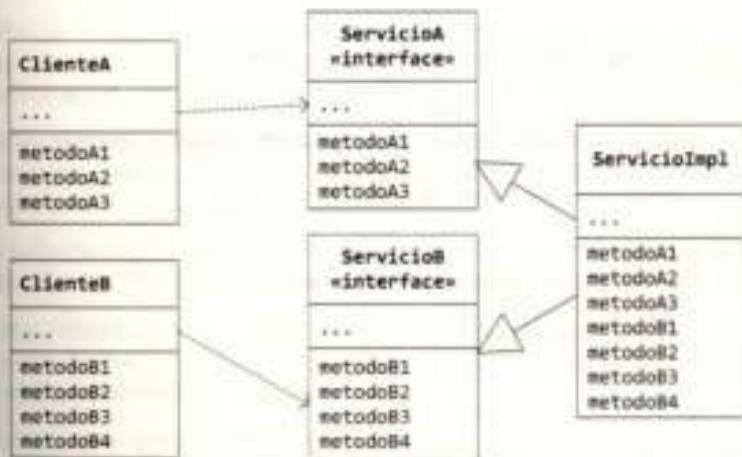
Solución propuesta: este diseño rompe el principio de abierto-cerrado. Si se cambia la clase que implementa el servidor, necesitamos cambiar el cliente. Un mejor diseño utilizará clases para flexibilizarlo, tal vez por medio de interfaces como muestra la siguiente figura.



5.4 Comente el diseño y el código de la nota de la siguiente figura. ¿Contra qué principio arroja? ¿Cómo se podría mejorar el diseño?



Solución propuesta: rompe el principio de separación de la interfaz, pues los diferentes clientes que hay utilizan sólo subconjuntos de la interfaz definida. Es mejor descomponer una interfaz tan compleja en otras cuyos métodos sean utilizados por los clientes de dicha interfaz, tal y como muestra la siguiente figura.



5.12.2 Actividades propuestas

- 5.1 El siguiente diagrama representa un diagrama de barras (Histograma) de una serie de valores (*VectorValores*). En la aplicación gráfica que lo muestra se descubren añadir otro tipo de visualizaciones. ¿Hay algo incorrecto en este diseño para añadir la nueva funcionalidad? Si es así, ¿qué principio se ha violado? Proponga un mejor diseño.



- 5.2 Complemente con aserciones el siguiente código que implementa una cola.

```

class Nodo{
    public int valor;
    public Nodo sig;

    public Nodo(int v){
        valor = v;
        sig = null;
    }
}

class Cola {
    // Referencias internas para la gestión de la cola
    private Nodo frente, fin;

    // Constructor
    public Cola(){
        frente = fin = null;
    }
}
  
```

```

// Método para añadir un elemento a la cola
public void push(int n){
    Nodo nuevoNodo = new Nodo(n);
    // Si la cola está vacía ponemos frente y fin a apuntar al nuevo nodo
    if (frente == null){
        frente = fin = nuevoNodo;
    }
    else{
        // Se añade siempre por el final
        fin.sig = nuevoNodo;
        fin = nuevoNodo;
    }
}

// Método para sacar un elemento de la cola
public int pop(){
    int n = 0;
    // Si la cola no está vacía, sacar por el frente
    if (frente != null){
        n = frente.valor;
        frente = frente.sig;
        // Si sólo había un elemento, poner fin a null
        if (frente == null) fin = null;
    }
    else{
        // Si la cola está vacía emitir un mensaje y retornar 0
        System.out.println("La cola está vacía");
    }
    return n;
}

public void mostrar(){
    if(frente == null){
        System.out.println("La cola está vacía");
    }else{
        Nodo tmp = frente;
        while(tmp != null){
            System.out.print("[ "+tmp.valor+" ]-> ");
            tmp = tmp.sig;
        }
        System.out.print("\n");
    }
}
}

```

- 5.3 En relación con los componentes ya desarrollados, siga las especificaciones y ejemplos en <http://java.sun.com/docs/books/tutorial/javabeans/> y cree un nuevo componente extendiendo un componente visual que pueda ser integrado en la plataforma Netbeans.
- 5.4 Busque y lea un artículo sobre cómo UML se puede extender a dominios específicos de su interés, por ejemplo, dominios de tiempo real.

- 5.5 Los fallos de Therac-25 están principalmente relacionados con el diseño. Lea y saque conclusiones del artículo de Levenson y Turner «An Investigation of the Therac-25 Incidents», en la revista *IEEE Computer*, volumen 26(7), páginas 18–41.
- 5.6 Analice los mecanismos que ofrecen algunos lenguajes populares como Java, C++ o Phyton, para soportar el diseño por contrato (invariantes, pre- y post-condiciones). Compárelos con los mecanismos originales ofrecidos por Eiffel.
- 5.7 Utilice la herramienta Dependency Finder para analizar las dependencias de un paquete Java de código abierto. Puede descargar este software en <http://depfind.sourceforge.net/>
- 5.8 El paquete JHotDraw, <http://www.jhotdraw.org/> está realizado teniendo muy en cuenta los patrones de diseño. Se propone leer el material que los autores suministran en la página web y analizar el código de JHotDraw.
- 5.9 Hay herramientas que generan aserciones a partir del lenguaje OCL. Encuentre alguna y ejecute algunos de los ejemplos que vienen con ellas.
- 5.10 Busque varios componentes ya desarrollados para un lenguaje de su elección y analice si con la información proporcionada son fácilmente comparables.

C

6.1

Cu
dic
de r
kra
coo
das
tico
mio
da i
con
pob
com

un
no
axi
los
ser
ma

6

Construcción

Hay dos formas de construir un diseño software. Una es hacerlo tan simple que obviamente no haya deficiencias. La otra es hacerlo tan complicado que no haya deficiencias obvias.

— C.A.R. Hoare

6.1 No da igual cómo esté construido

Cuando se enseña a programar, debería tenerse muy presente la influencia que en los aprendices tienen los malos hábitos –como programadores– de sus profesores, evitando el uso de recursos didácticos que trivializan el proceso de construcción. Cuando se estudia la utilización de variables, por ejemplo, se describe su importancia, se establecen las diferencias con las constantes (ya que éstas generalmente se estudian antes y por tanto, son conocidas por los alumnos), se hace hincapié en la necesidad de establecer de manera precisa su tipo, se detalla cómo declararlas y dónde es más conveniente hacerlo, y como no, se transmite la importancia de inicializarlas antes de hacer uso de ellas. Sin embargo, apenas se da importancia al nombre que las identifica. De hecho, es frecuente ver cómo se utiliza un conjunto bastante limitado de nombres, a menudo los mismos, haciendo gala de una muy pobre imaginación y una aún menor capacidad para inculcar a nuestros alumnos algo muy simple: es importante cómo vamos a llamar a las variables.

Algunos de los nombres que más a menudo se aplican a las variables numéricas reales son `x`, `y` o `z`, mientras que para variables de las que nos servimos en algún cómputo pero que no consideramos dentro del elenco de «actores principales de un programa» solemos utilizar `aux` (auxiliar), `cont` (contador), `acu` (acumulador), `tmp`, etc. Se da la misma situación con los nombres de funciones, pues frecuentemente vemos nombres tan poco imaginativos –por ser generosos con el calificativo– como `getInfo()`, `f()` o `checkData()`. Todo ello buena muestra de la falta de esfuerzo dedicado a elaborar un código más sencillo de mantener.

Los autores de este libro recuerdan a un profesor que les acompañó durante sus primeros pasos en la programación que, posiblemente preocupado por no utilizar en los ejemplos que ponía en sus clases los mismos identificadores que todo el mundo usaba (`temp, x, y, etc.`), decidió hacer uso de sus —a partir de entonces— famosos «pepes». Este profesor utilizaba invariablemente `pepe1, pepe2, pepe3, etc.` como nombres para sus variables, independientemente, por supuesto, del tipo de las mismas o de la función que desempeñaran dentro del programa. No podemos decir que aquella decisión le sirviera para hacernos más llevaderas las clases de iniciación a la programación pero sí, al menos, para que todos los que pasamos por su curso llegáramos a pensar que sería capaz de llamar `pepe1, pepe2` y `pepe3` a sus hijos con tal de simplificar la tarea de idear un nombre correcto según las circunstancias. Al menos esta anécdota ha servido para que le recordásemos con cariño.

En la construcción de software no es irrelevante el cómo llamemos a las variables. El software se escribe una vez, pero se lee decenas, cientos de veces. Escoger nombres de identificador que no permitan deducir fácilmente para qué sirve un procedimiento, o qué contiene una determinada variable o constante, añadirá tiempo innecesario a la ya de por sí difícil tarea de mantener los programas. Cuando se está intentando entender un código que no hemos escrito nosotros, no da igual que las variables se llamen `q1, q2 y q3`, pues de haberlas llamado `tasa_de_interes, precio_final y num_horas_trabajadas` el software habría sido mucho más fácil de entender. Lo importante es facilitarse a uno mismo y a los demás el comprender qué se quiso decir al escribir un código, pues en ocasiones, ni el mismo creador es capaz de entenderlo. Si la forma en que está escrito el software no ayuda a comprenderlo (y si además de todo ello quien debe modificar el código no es el autor del código original) la frustración está prácticamente asegurada.

Como veremos más adelante, la deficiente legibilidad del código es un problema a tomar muy en serio. Pero cuando se construye software, hay muchas otras cosas que han de tenerse en cuenta. Un buen software no es sólo aquél que «funciona» (cosa que se debe dar por supuesta) sino también aquel que es fácil de mantener, aquel cuyo código es legible y conforme con ciertas reglas de estilo, aquel que sigue ciertos estándares o aquel que no tiene problemas de rendimiento ni escapes de memoria, por citar sólo algunos ejemplos en los que existe consenso casi unánime.

En este capítulo se muestra un buen número de prácticas efectivas de construcción de software. Se trata de prácticas que permiten no sólo producir un código más fácil de entender y mantener, sino también acortar el tiempo de desarrollo, reducir errores y facilitar la depuración. Ya veremos que todas estas cosas son también importantes. No es aceptable construir y entregar software bajo la filosofía de «armario cerrado», según la cual todos los armarios son iguales en tanto en cuanto la puerta esté cerrada, independientemente de si su contenido está ordenado o desordenado. Nosotros pensamos que el software, como el armario de una persona pulcra, no sólo debe aparecer «desde fuera» que todo esté correcto, al abrir la puerta tampoco queremos encontrarnos con la desagradable sorpresa de que la ropa está arrugada, o que los cajones contienen cualquier cosa en cualquier orden.

6.2 Objetivos

El propósito fundamental de este capítulo es estudiar los principios en los que debe basarse la construcción de software de calidad, describiendo algunas de las técnicas de Ingeniería del Software que deben tenerse en cuenta cuando se construyen programas.

Los objetivos específicos son los siguientes:

- Reconocer la importancia de la construcción de software como parte esencial en el proceso de desarrollo.
- Comprender el concepto de construcción como la creación de software en un lenguaje de programación, junto con las microtareas de diseño y pruebas asociadas.
- Estudiar los principios que gobiernan la construcción de software de calidad.
- Conocer técnicas de implementación que resultan de utilidad desde el punto de vista de la Ingeniería del Software.
- Comprender el papel de la reutilización en la construcción de software, conociendo sus beneficios y teniendo en cuenta sus riesgos.

6.3 Introducción

El término *construcción de software* define un conjunto de actividades que engloban fundamentalmente la codificación, pero también la verificación del código, su depuración y ciertos tipos de pruebas. Estas actividades parten de una especificación de requisitos detallados que fueron elaborados durante las actividades de diseño, y dan como resultado un software libre de errores que cumple con dicha especificación. No obstante, la comprobación exhaustiva de la corrección del software no puede hacerse en esta etapa, ya que necesita someterse a un proceso minucioso de pruebas que la determine. Este proceso no forma parte de las actividades de construcción en sentido estricto.

El producto resultante de la construcción de software, el software propiamente dicho, es en último término el entregable más importante de un proyecto de desarrollo. Ciertamente es el motivo principal por el que se está llevando a cabo. Sin embargo, no todo el mundo tiene la misma concepción de la construcción de software. En los modelos de ciclo de vida más tradicionales, este término es sinónimo de codificación. Estos enfoques estudian la construcción como una actividad claramente separada del diseño y las pruebas, que consiste en implementar, utilizando un cierto lenguaje de programación, las soluciones elaboradas durante el diseño y plasmadas en diversos diagramas, teniendo en cuenta las restricciones y la documentación generada durante dicha etapa. Otros modelos más modernos, particularmente los métodos ágiles –que se caracterizan por iterar sobre cortos ciclos de desarrollo– y los métodos basados en la evolución de prototipos, tienen un concepto distinto de la construcción, pues incluyen como parte de la misma la codificación, pero también

otras actividades como las pruebas o el diseño detallado. En la programación extrema, por ejemplo, la prueba unitaria se considera simultánea a la codificación.

Con el propósito de fijar las diferencias entre construcción y codificación, digamos que esta última puede definirse de la siguiente manera:

Codificar es el proceso de expresar un programa de computadora en un lenguaje de programación (IEEE, 1990)

En cuanto al término *construcción*, la guía SWEBOK lo define del siguiente modo:

Construir consiste en crear software mediante una combinación de codificación, verificación, pruebas unitarias y depuración

Existen numerosos libros (algunos de ellos excelentes) que explican profusamente cómo debe construirse un programa en un lenguaje determinado. Sin embargo, no analizaremos en este libro la forma en que debe construirse un programa utilizando un lenguaje en particular. Nuestro ánimo es, por el contrario, estudiar la construcción de software desde diferentes perspectivas adaptables a cualquier lenguaje, para lo cual dividiremos nuestro esfuerzo en el estudio de dos bloques diferenciados:

- Un análisis de los fundamentos de la construcción de programas, donde se abordan los principios esenciales que gobiernan la construcción de software de calidad.
- Una descripción de la gestión del proceso de construcción, donde se estudian tanto los modelos que pueden aplicarse a la construcción de software, como la planificación y medida en esta actividad.

Por su propia naturaleza, la construcción de software se encuentra íntimamente relacionada con todas las demás actividades del desarrollo, pero especialmente con el diseño, las pruebas y el mantenimiento. Esto es así porque la propia construcción no es una actividad aislada que pueda separarse de las demás. Para construir cualquier cosa (y el software aquí no es una excepción) se necesita un esfuerzo previo en su diseño y otro posterior para comprobar que funciona adecuadamente. Además, la manera en la que se cambia o mejora estará íntimamente relacionada con la manera en la que se construyó. En un proceso iterativo de construcción, la detección de errores como consecuencia de la realización de pruebas puede llevar a un replanteo del diseño y a modificar o reconstruir parte de lo ya construido. La frontera que marca cuándo se está diseñando, cuándo probando y cuándo construyendo es a menudo difusa. Para el desarrollo del software también sería deseable poder abordar la construcción con la metodología que suele emplearse en los procesos de fabricación industrial: primero se diseña, luego se construye y finalmente, se prueba lo construido. Sin

embargo, las características específicas del software como producto no permiten este enfoque, por lo que la mayoría del trabajo de diseño se lleva a cabo simultáneamente con la construcción, y lo mismo puede decirse de muchas de las actividades de pruebas.

En cuanto a su relación con el diseño, sólo a medida que se programa se llegan a entender completamente algunos de los requisitos, se identifican restricciones que no se tuvieron en cuenta durante el diseño o se detectan errores en el mismo que impiden llevar a cabo la programación tal y como fue pensada en un primer momento. También se detectan durante la construcción numerosos problemas derivados de que los diseñadores no han tenido en cuenta todas las especificidades del entorno sobre el que se implantará el software.

Por otra parte, y como se estudia detenidamente en el Capítulo 7 dedicado a las pruebas, los desarrolladores no esperan a completar la construcción de todos los elementos que conforman un sistema software para posteriormente probarlo, sino que muchas de las actividades de prueba, tales como las pruebas de unidad o las de integración, se llevan a cabo a medida que el software se va construyendo.

Otras actividades, como el mantenimiento o la gestión de la configuración también tienen una fuerte relación con la construcción. En el caso del mantenimiento (Capítulo 8), la firma en que se lleve a cabo la construcción es fundamental: un software bien construido no sólo permite ahorrar tiempo y esfuerzo de mantenimiento, sino que permitirá abordar dicho mantenimiento con mayor seguridad y menor propensión a introducir errores. En cuanto a la gestión de la configuración (Capítulo 11), la influencia de la construcción se comprende porque es en la etapa de construcción cuando se producen la mayor parte de los elementos de configuración que será necesario gestionar como parte del proyecto.

6.4 Lenguajes de construcción

El Glosario IEEE de Términos de Ingeniería del Software define lenguaje de computadora como «*un lenguaje diseñado para permitir a los humanos comunicarse con las computadoras*». Dentro de esta categoría tan genérica, existen diferentes subcategorías, tales como los lenguajes de consulta, los lenguajes de especificación o los lenguajes de construcción. En un capítulo dedicado a la construcción de software, hemos de centrar deliberadamente nuestro estudio en los lenguajes de construcción, que la guía SWEBOK define así:

Los lenguajes de construcción incluyen todas las posibles formas de comunicación mediante las cuales un humano puede especificar a una computadora una solución ejecutable a un problema

La misma guía SWEBOK propone una clasificación de los lenguajes de construcción en tres tipos principales:

- Lenguajes de configuración: son lenguajes que, a partir de un conjunto de opciones, permiten especificar cómo se configurará una determinada instalación de un software.

Es habitual que las especificaciones de entrada se proporcionen en forma de ficheros de texto, legible por tanto con cualquier editor. Un ejemplo de esto lo constituyen el conjunto de ficheros de configuración de los sistemas Linux: /etc/profile, /etc/passwd, /proc/devices, etc.

- Lenguajes de las cajas de herramientas (*toolkits*): más complejos que los anteriores, se emplean para construir aplicaciones a partir de bloques predefinidos en las cajas de herramientas, bien utilizando una interfaz específica para ello (caso más habitual) o bien a través de un lenguaje de programación simplificado incorporado en la propia caja de herramientas. APL, un lenguaje con su propio conjunto de símbolos creado por Kenneth E. Iverson para describir de manera más exacta y breve el procesamiento de datos, es quizás el ejemplo más conocido de este tipo de lenguajes.
- Lenguajes de programación: son los más flexibles, potentes y ampliamente utilizados. Los lenguajes de programación no son sólo la forma en que los programadores implementan prácticamente el diseño detallado realizado por los diseñadores. La elección de un determinado lenguaje determina inequívocamente el modo en que el proceso de construcción se llevará a cabo. Como una vez dijo Alan Perlis, pionero de los lenguajes de programación y primera persona en la historia en recibir el premio Turing: «*un lenguaje de programación que no afecte a la forma en que uno aborda la programación es un lenguaje que no merece la pena*».

Un lenguaje de programación está formado por un vocabulario y por un conjunto de reglas gramaticales que permiten escribir órdenes que una computadora llevará a cabo. Cada lenguaje tiene sus reglas y su vocabulario propio, si bien existe un conjunto de elementos fundamentales que la mayoría de ellos comparte y que permiten a un programador de cierta experiencia ser capaz de escribir programas en muchos lenguajes diferentes.

Cuando hacemos uso del término *lenguajes de programación* generalmente nos estamos refiriendo a los lenguajes de programación de alto nivel, si bien existen otros lenguajes de programación que pueden clasificarse del siguiente modo y que esquemáticamente se muestran en la Figura 6.1:

- Los lenguajes de primera generación (1GL) son conocidos como *lenguajes máquina*. Se trata de un conjunto de códigos numéricos binarios que se introducían directamente en la máquina para que ésta, sin mediar traducción, ejecutase las órdenes del programador. Cada CPU, cada máquina, tiene su propio lenguaje 1GL, lo que obligaba a los antiguos programadores a conocer tantos de estos lenguajes como máquinas quisieran programar:

00000 00000 000010 00000 00100 000000

- Los lenguajes de segunda generación (2GL) están a medio camino entre el lenguaje máquina y los lenguajes más cercanos al programador, denominados 3GL. A menudo



Figura 6.1: Los diferentes niveles de lenguajes de programación

conocidos como *lenguajes ensamblador*, pueden ser leídos por un programador, aunque su estructura sintáctica aún dista bastante del lenguaje humano. Para que un programa escrito en un lenguaje de este tipo pueda ejecutarse, primero debe traducirse a código máquina, lo cual sucede también con el resto de lenguajes de esta clasificación:

```
PUSH EAX
MOV AX, 18H
```

- Los lenguajes de tercera generación (3GL) aparecieron a finales de la década de 1950 por la necesidad de acercar la forma de comunicación de los humanos a la comunicación con las computadoras. Lenguajes como Fortran, Cobol o Algol fueron los primeros en desarrollar una sintaxis similar a la de los lenguajes humanos, incorporando por vez primera la posibilidad de declarar tipos, variables, emplear estructuras de control, etc. La mayoría de los lenguajes de programación más populares, como C, C++, Java, Ada o Basic, son lenguajes de tercera generación.

```
for (i=0; i<10; i++) System.out.println(i);
```

- Los lenguajes de cuarta generación (4GL) son los más cercanos a la forma de expresarse y comunicar de los humanos. En estos lenguajes el programador no expresa directamente cómo debe hacer las cosas la computadora, sino sólo qué es lo que se debe hacer. Constituyen un escalón por encima de los lenguajes de tercera generación y, aunque no hay un consenso absoluto sobre lo que es o no es un lenguaje de este tipo, suele catalogarse como tal todo lenguaje que ha sido diseñado y creado con un propósito específico, a menudo relacionado con las consultas y accesos a bases de datos. SQL, NATURAL, PowerBuilder o R son algunos ejemplos concretos de 4GL.

```
FIND ALL FLIGHTS WHERE DESTINATION IS "MADRID"
```

- Se denomina lenguajes de quinta generación (5GL) a aquellos orientados a especificar a una computadora el problema a resolver, pero no cómo resolverlo. A menudo relacionados con la inteligencia artificial, su diseño permite especificar el problema a resolver mediante un conjunto de restricciones, sin necesidad de escribir un algoritmo que guíe su resolución. Aunque no tan populares como los 3GL y 4GL, se utilizan mucho en la investigación (en áreas como el procesamiento de lenguaje natural o los sistemas expertos) y en el ámbito académico. Haskell, Prolog, Lisp o Mercury, son algunos lenguajes de este tipo:

```
suspende(X)  <= noEstudia(X) and haceExamen(X)
```

Existen muchas otras clasificaciones de los lenguajes de programación, como las que los dividen en *procedimentales* y *no procedimentales*, las que los clasifican de acuerdo a su complejidad, o las que los dividen en lenguajes específicos de un dominio y lenguajes de propósito general. En este libro no es posible detenerse a estudiar todas las posibles clasificaciones, pero los amantes de los lenguajes de programación sin duda encontrarán de utilidad la web «99 botellas de cerveza» (<http://99-bottles-of-beer.net/>) donde se muestran ejemplos prácticos de más de 1.300 lenguajes de programación.

6.5 Reutilización del código

Desde principios de la década de 1970, una de las preocupaciones de la Ingeniería del Software ha sido la reutilización del código. Los programadores siempre han reutilizado el código, bien del modo más simple (copiando y pegando secciones de un código ya escrito), o bien utilizando mecanismos específicos para reutilización tales como plantillas, procedimientos o funciones. Sin embargo, no fue hasta 1968, año en que Douglas McIlroy propuso basar la industria del software en la creación y empleo de componentes reutilizables, cuando la reutilización pasó a ser una prioridad. Evitar resolver una y otra vez el mismo problema permitiría, según McIlroy, avanzar únicamente en la resolución de nuevos retos y facilitaría los desarrollos, al incrementar la productividad de los equipos de trabajo. Así, surgieron nuevos paradigmas como la programación orientada a objetos o el diseño orientado a componentes, paradigmas cuya base teórica esencial es el diseño para la reutilización. Esta filosofía se basa en la idea de que el software, de modo similar a lo que sucede en otras ramas de la ingeniería, debería construirse haciendo uso de componentes «prefabricados», y aboga no sólo por reutilizar componentes creados con anterioridad siempre que sea posible, sino también por crear los nuevos componentes pensando en que, muy probablemente, serán a su vez reutilizados en el futuro.

Se dice que un componente o módulo software es **reutilizable** cuando puede utilizarse en más de un programa de computadora o sistema de software (IEEE, 1990).

El diseño para la reutilización consiste, por tanto, en programar aplicaciones teniendo en cuenta que en el futuro otras aplicaciones pueden necesitar dar respuesta a los mismos problemas. De este modo, será no sólo adecuada sino recomendable la elaboración de un repositorio de componentes software en sentido general (que pueden ser funciones, clases, procedimientos, etc.) que permita hacer uso de dichos componentes tal y como están –sin modificación alguna– en nuevas aplicaciones. En el caso más habitual, y definitivamente en los lenguajes de programación modernos, dicho repositorio se concreta en bibliotecas.

Una biblioteca de software es una colección controlada de software y documentación relacionada que ha sido diseñada para facilitar el desarrollo, uso o mantenimiento de software (IEEE, 1990)

Una biblioteca de software no es otra cosa que un producto software resultado de un desarrollo. Sin embargo, y a diferencia de las aplicaciones propiamente dichas, las bibliotecas raramente son ejecutables por sí mismas. Consisten, por el contrario, en un conjunto de subprogramas tales como clases, funciones independientes, etc., que una vez completado y adecuadamente probado puede utilizarse potencialmente en cualquier aplicación. Los desarrolladores que programan una biblioteca de software ignoran, por supuesto, en qué contextos y aplicaciones será utilizado su código, por lo que el diseño de las mismas debe ser deliberadamente independiente de cualquier aplicación particular. Puede decirse que el código incluido en una biblioteca está pensado para dar un servicio específico a cualquier otra aplicación que lo requiera. Así, existen bibliotecas de software para interacción con los dispositivos de entrada-salida, para la creación de interfaces de usuario, para el acceso o interacción con una aplicación ya existente (las denominadas API), bibliotecas de funciones estadísticas o que proporcionan capacidades para la persistencia en un lenguaje que carece de ella, por sólo citar unas cuantas de las muchas posibles.

Entre las ventajas del uso de bibliotecas de software para la construcción de aplicaciones se pueden citar las siguientes:

- Se trata de código ya probado y que, por tanto, salvo excepciones, está libre de fallos.
- Las bibliotecas contienen código optimizado cuyo rendimiento es, en general, superior al código equivalente que crearía un programador durante la construcción de la aplicación donde usara la biblioteca.
- Las bibliotecas cubren todos (o casi todos) los casos y situaciones posibles, incluso aquellos menos habituales.
- Es más económico, más rápido y más fiable reutilizar código en forma de bibliotecas de software que desarrollar código nuevo.

En cuanto a sus desventajas:

- Es difícil modificar o ajustar algunos detalles que pueden afectar al rendimiento o a la salida esperada.
- Es necesario un tiempo de aprendizaje para comprender cómo funcionan sus componentes y poder sacarle el máximo partido.
- La utilización de código externo puede crear riesgos ocultos de seguridad en el software en desarrollo.
- Pueden tener un coste de adquisición o uso.
- La utilización de bibliotecas externas puede acarrear problemas relacionados con la titularidad de los derechos de propiedad intelectual.

A pesar de las desventajas apuntadas, hoy en día, es muy frecuente reutilizar componentes de código de terceros en forma de bibliotecas, tanto de código fuente abierto como propietario. De hecho, el modo de crear aplicaciones ha cambiado sustancialmente en los últimos años y en buena medida esto es debido a la amplia oferta de bibliotecas de componentes reutilizables que existen a disposición de los desarrolladores. Puede decirse que, por el momento, las ventajas de la reutilización superan a las posibles desventajas que el empleo de bibliotecas podría suponer.

6.6 Principios fundamentales de la construcción de software

Según la guía SWEBOK, los principios fundamentales de la construcción de software son los siguientes:

1. Minimizar la complejidad,
2. Anticipar los cambios,
3. Construir para verificar,
4. Utilizar estándares.

Estos cuatro principios deben guiar la construcción de software de calidad. Sin embargo, los tres primeros se aplican no sólo a la construcción, sino también al diseño. A continuación estudiaremos detalladamente cada uno de ellos.

6.6.1 Minimizar la complejidad

En 1972, Dijkstra señaló que «*no existe una mente lo suficientemente capaz como para memorizar completo un programa de computadora moderno*». Esta afirmación hace hincapié en la extensión y complejidad de los programas, demasiado grandes como para que una sola persona, por las limitaciones inherentes a la mente humana, sea capaz de retener

con precisión todos los elementos que lo conforman. Asumiendo esta limitación, los desarrolladores deben organizar el código de forma que éste permita centrarse en diferentes partes en cada momento, pues es imposible pensar en todo el programa a la vez. Y si todo esto era ya cierto en 1972, décadas después la mente humana –que no ha evolucionado al ritmo que lo ha hecho la complejidad del software– está aún más lejos de poder contener los enormes programas que se producen hoy en día.

La complejidad expresa el grado en que un sistema o componente tiene un diseño o implementación difícil de entender o verificar (IEEE, 1990)

Se trata, por tanto, de obtener un código simple (o al menos sin complejidades innecesarias), que sea más fácil de probar y mantener. Para alcanzar este objetivo, se han de aplicar técnicas que faciliten la legibilidad y simplicidad del código que producen. Pero también hacer uso de estándares especialmente orientados a guiar la construcción. Por su importancia en la construcción de software, este punto será tratado en una sección aparte.

A continuación se exponen nueve técnicas específicas que pueden emplearse para la simplificación del código:

- Técnicas de legibilidad, orientadas a crear código más fácilmente comprensible, como las técnicas de unificación de nombres o las de estructuración del texto del código.
- Uso de elementos de estructuración, como clases, variables, constantes con nombre, tipos enumerados, etc.
- Uso de estructuras de control.
- Utilización de estructuras de gestión de las condiciones de error, tanto para los errores previsibles como para las excepciones.
- Prevención de errores de seguridad propiciados por una inadecuada codificación (desbordamiento de memoria, acceso fuera de los límites correctos en un *array*, etc.)
- Utilización de recursos a través de mecanismos de exclusión y control cuando se hace acceso simultáneo o compartido.
- División jerárquica del código fuente, que permita ir descendiendo en el nivel de complejidad. Así, se recomienda dividir un programa completo en paquetes, éstos en clases, las cuales a su vez están formadas por métodos, sentencias, y así sucesivamente. Esta técnica está muy relacionada con el *refactoring*, conjunto de técnicas de reestructuración del código que se estudia en la Sección 8.7.3.
- Documentación del código.
- Afinación del código.

De las técnicas enumeradas, el uso de elementos de estructuración, tales como clases, tipos enumerados, variables, constantes con nombre, etc., su organización jerarquizada y el uso de estructuras de control, es hoy tan común y extendido que no nos detendremos a explicar en qué consisten ni por qué resulta aconsejable su uso. No obstante, puede resultar ilustrativo en este punto un ejemplo de una de estas técnicas: el uso de constantes con nombre. A menudo se denomina «*evitar los números mágicos*» a una práctica que consiste en no escribir un código como el siguiente:

```
if (i > 120) {  
    System.out.print("El vehículo supera el límite legal de velocidad");  
}
```

eliminando el *número mágico* 120 y sustituyéndolo por una constante con nombre. Así:

```
private static final int LIMITE_VELOCIDAD = 120;  
...  
if (i > LIMITE_VELOCIDAD) {  
    System.out.print("El vehículo supera el límite legal de velocidad");  
}
```

La utilización de mecanismos de exclusión y control para evitar problemas en el acceso simultáneo o compartido a datos es igualmente de uso generalizado, por lo que tampoco parece oportuno dedicar un apartado específico a su estudio. Resulta impensable hoy en día la existencia de un sistema gestor de bases de datos que no implemente mecanismos de bloqueo que eviten, por ejemplo, la consulta de un dato que está siendo modificado, lo que podría provocar inconsistencias o pérdidas en la información almacenada. Del mismo modo, el uso de mecanismos de exclusión tales como semáforos, monitores o regiones críticas es tan básico y esencial en la programación concurrente que no es preciso detenerse en los beneficios de su uso.

Sí merece la pena, por contra, detenernos y comentar las técnicas relacionadas con ciertos aspectos que impactan particularmente en la calidad del producto construido: la legibilidad, la gestión de condiciones de error mediante excepciones, la documentación y la afinación del código.

Técnicas de legibilidad

Una de las mejores formas de reducir la complejidad, si no del código, al menos de la capacidad de comprensión del mismo, es hacer el código más legible. Esto resulta especialmente importante cuando se considera que el software que se está construyendo deberá ser posteriormente probado y mantenido, y que además, dichas tareas serán llevadas a cabo, probablemente, por personas diferentes a las que lo escribieron originalmente.

La sentencia GOTO vs. la programación estructurada

Desde que en 1968 Dijkstra publicara su ya clásico artículo «*GOTO statement considered harmful*», en el que hacia hincapié en el uso de estructuras de control para sustituir el uso generalizado de sentencias goto de la época, la evolución de la forma de programar y los propios lenguajes modernos permiten afirmar que, si bien no completamente desterrado, el goto es hoy día una reliquia del pasado. Al menos como recurso de programación generalizado.

No obstante, hay veces en que su uso está ciertamente recomendado. De hecho, existen ejemplos de código donde la codificación sin goto es tan complicada o tan difícil de mantener *a posteriori* que resulta recomendable elegir la opción con goto. Es posible que más de un lector se encuentre en este momento frotándose los ojos con asombro, y pensando algo como: «Toda la vida escuchando que la instrucción goto jamás debe usarse, opinión respaldada por los escritos de eminentes autores y ¿ahora resulta que el goto está recomendado?» Pues sí, ha leído bien. Tanto goto como otras sentencias que rompen el normal flujo de control y que no han sido tan criticadas (tales como *continue* o *break*) pueden y deben ser utilizadas. Pero eso sí, recuerde hacerlo sólo cuando su uso sea estrictamente necesario (por ejemplo, cuando las soluciones alternativas sean aún peores).

En la primera sección de este capítulo se utilizó deliberadamente el problema del código descuidado para ilustrar la necesidad de abordar la construcción de software de acuerdo a ciertos criterios de calidad. Se comentó entonces que se emplea más tiempo en mantener un programa que en crearlo por primera vez, que el mantenimiento (corrección de errores y ampliación de funcionalidades) obliga a un profundo estudio previo del código fuente, y que es muy difícil que un solo programador mantenga una aplicación a lo largo de toda su vida útil. Por tanto, hay que procurar que el código esté escrito de tal modo que cualquiera sea capaz de leerlo y entenderlo sin necesidad de un esfuerzo extraordinario. Recuerde: «no da igual cómo esté construido».

A continuación vamos a detenernos a estudiar más de cerca el problema de la legibilidad del código y cuáles son las posibles soluciones. Pero comencemos con un contraejemplo:

Just Java
Peter van der Linden
April 1, 1996.

```
u0050 u0076 u0064 u004c u0020 u0031 u0020 u0041 u0070 u0072 u0039 u0036
u002a u002f u0020 u0063 u006c u0061 u0073 u0073 u0020 u0068 u0020 u007b
u0020 u0020 u0070 u0075 u0062 u006c u0069 u0063 u0020 u0020 u0020 u0020 u0020
u0073 u0074 u0061 u0074 u0069 u0063 u0020 u0020 u0076 u0061 u0069 u0064
u006d u0061 u0069 u006e u0028 u0020 u0053 u0074 u0072 u0069 u006e u0067
u006b u006d u0061 u0029 u0020 u007b u0053 u0079 u0073 u0074 u0065 u006d
u003e u006f u0075 u0074 u002e u0070 u0072 u0069 u006e u0074 u006c u006e
u0028 u0022 u0048 u0069 u0021 u0022 u0029 u003b u007d u007d u002f u002a
```

Esto que acaba de leer no es ningún volcado de memoria. Tampoco es una broma. Se trata de un fragmento de código en Java. Sí, ha leído bien, en Java. En realidad este programa se compila correctamente y produce, al ejecutarlo, la siguiente salida por pantalla:

Hi!

Se preguntará el lector cómo un código con la apariencia de un comentario puede no sólo compilarse correctamente sino además, producir una salida como ésa. Muy sencillo, el ingenioso programador de este código utiliza, en lugar de caracteres legibles tales como la «a», la «b», etc., los códigos de dichos caracteres en formato Unicode para ocultar deliberadamente el objeto de su programa. ¿Qué dice exactamente este intrincado programa, entonces? Simplemente, en algún lugar de esta maraña de códigos Unicode se cierra el comentario inicial, se declara una clase que es la que emite el saludo y vuelve a abrirse un comentario que se cierra con el cierre de comentario «visible» al final del código. Aclarado el misterio se preguntará el lector ¿era realmente necesario todo esto para mostrar por pantalla un simple saludo?

El código anterior es uno de los muchos ejemplos existentes de «*código oscuro*» - a menudo denominado también *código ofuscado*-, código fuente que ha sido entresacado deliberadamente para ocultar su funcionalidad y hacerlo ininteligible. Hay quien tiene como afición la escritura de código de este tipo, e incluso, existen concursos internacionales sobre el tema con importantes premios en juego¹. Lamentablemente los programas ininteligibles no son exclusivos de los concursos de codificación oscura. Pueden ponerse algunos casos, no tan extremos como el anterior, pero sí suficientemente significativos de este problema. Para situar nuestro código en el extremo opuesto, es decir, para hacerlo definitivamente más legible, se recomienda poner cuidado en los siguientes puntos:

Nombres de identificadores. Deben ser significativos pero cortos. Eso sí, no tanto que no proporcionen información acerca de la utilidad (y a ser posible el tipo) del elemento al que califican. Ha de evitarse utilizar abreviaturas a menos que su significado sea evidente, ya que lo que resulta obvio para el que escribe el código puede no serlo tanto para otras personas. Cuando el nombre del identificador deba contener más de una palabra, existen dos opciones:

1. En algunos lenguajes como Java se propone escribir juntas todas las palabras, sin separaciones, pero poniendo en mayúsculas las iniciales intermedias: `calcularÁrea`, `tasaDeInterés`, `numeroDeAlumnos`, etc.
2. En otros lenguajes, como Pascal, se opta por separar las palabras por un guion bajo para hacerlas más legibles: `MAXIMO_INICIAL`, `calcular_area`, etc.

¹Posiblemente el más conocido e importante de estos concursos sea el denominado «*International Obfuscated C Code Contest*», que se organiza anualmente desde 1984.

Otras reglas complementarias sobre la elección de nombres correctos serían:

- Utilizar mayúsculas para los nombres de constantes (MAXIMO_INICIAL, PI, etc.)
- Utilizar verbos en infinitivo para los nombres de método (procedimientos y funciones), y hacer que comiencen por minúscula: (calcularSalario, ordenarArray, borrar_Formulario, etc.)
- Usar el formato anterior, pero empleando sustantivos para los nombres de atributos y variables locales: (salarioBase, numeroDeFax, etc.). Es común observar cómo a veces se hace preceder al nombre de una variable una abreviatura que indica el tipo de la misma: strPrimerApellido, intNumeroDeHijos, etc. (al respecto, véase más adelante la Tabla 6.1).
- Usar sustantivos para los nombres de clases y estructuras (tales como registros, uniones, etc.), pero en este caso, se recomienda que empiecen por mayúscula: FiguraGeometrica, Expediente_Academico, etc.
- Esforzarse en encontrar nombres significativos, evitando nombres como Procesar(), Calcular(), aux, cont, etc.

El uso homogéneo de nombres de identificadores contribuye de una manera decisiva a la legibilidad de los programas, desterrando en gran medida la confusión y haciendo que quienes realizan tareas de mantenimiento encuentren familiar un código que no han escrito.

Comentarios. El código debe acompañarse de comentarios que expliquen aquellos aspectos que no se explican por sí solos. Sin embargo, introducir comentarios como los que se muestran a continuación sería insultar la inteligencia del lector:

```
/** Calcula el importe */
public int calcularImporte(){
    // inicializar el valor de retorno a cero
    int valorDeRetorno = 0;
    ...
    //incrementar en uno el valor de retorno
    valorDeRetorno++;
}
```

Otro ejemplo de lo mismo sería el siguiente:

```
// Impleza el bucle
for (i=0; i<10; i++){
    A[i] = A[i] + 2; // Suma 2 al elemento i-ésimo del array
}
```

Se recomienda, por tanto, introducir comentarios sólo si es útil, con el objetivo de facilitar la comprensión y teniendo cuidado de no complicar más el código. Y por supuesto, recuerde que los comentarios deben mantenerse actualizados: si el código cambia, el comentario debe cambiar también.

En la Sección 6.6.1 se estudia con algo más de detalle la utilización de comentarios como parte de la documentación técnica de un programa.

Estructuración del código. La estructura del código es otro aspecto importante a tener en cuenta para aumentar su calidad. Así, las instrucciones deben disponerse en párrafos, cada uno de los cuales puede ir precedido por un comentario que lo explique:

```
// Apertura del fichero de datos
FileInputStream fEntradaDatos = new FileInputStream("datos.txt");
BufferedInputStream bufferEntrada = new BufferedInputStream(fEntradaDatos);
DataInputStream entradaDeDatos = new DataInputStream(bufferEntrada);
```

Debe utilizarse el sangrado para proporcionar legibilidad al código. Las instrucciones con y sin sangrado producen el mismo resultado cuando se ejecutan; sin embargo, el efecto que produce en el lector este código:

```
public static long factorial(int x)
    throws IllegalArgumentException {
    if (x >= tabla.length)
        throw new IllegalArgumentException("Overflow: x es demasiado grande.");
    while(ultimo < x) {
        tabla[ultimo + 1] = tabla[ultimo] * (ultimo + 1);
        ultimo++;
    }
    return tabla[x];
}
```

es muy distinto del que produce este otro:

```
public static long factorial(int x)
    throws IllegalArgumentException {
    if (x >= tabla.length)
        throw new IllegalArgumentException("Overflow: x es demasiado grande.");
    while(ultimo < x) {
        tabla[ultimo + 1] = tabla[ultimo] * (ultimo + 1);
        ultimo++;
    }
    return tabla[x];
}
```

Las reglas básicas del estructuración del código difieren de unos lenguajes a otros. No obstante, es posible citar algunas recomendaciones comunes:

- Se sugiere limitar la extensión del texto de un método a una página impresa, salvo excepciones.
- Se propone limitar la longitud de la línea para evitar efectos indeseados en la edición y/o impresión. El valor límite depende del lenguaje pero suele ser un valor entre 70 y 80 caracteres.
- Se recomienda utilizar un número de espacios fijo para el sangrado. El número de espacios recomendado también varía ligeramente dependiendo del lenguaje: por ejemplo en Java se recomienda utilizar 2 espacios mientras que en Phyton se recomienda usar 4. La recomendación habitual se sitúa en 2-3 espacios.
- No es conveniente mezclar espacios y tabulaciones en el sangrado. Se recomienda utilizar preferentemente espacios, pero si se prefiere el uso de tabulaciones, éstas deben emplearse solas. En los casos en que haya mezcla de ambos, se recomienda reconvertir todo a espacios.
- Se recomienda separar aquellas secciones lógicamente relacionadas con una línea en blanco, que debería utilizarse también tras la definición de una clase, método o elemento de alto nivel. Pero recuerde que no conviene abusar innecesariamente de las líneas en blanco.

Expresiones. Las expresiones deben escribirse utilizando paréntesis siempre que su estructura sin ellos pueda ser confusa. Cuando una expresión sea demasiado larga como para caber completa en una línea, es aconsejable cortarla en un punto donde las subexpresiones que se separen sean igualmente legibles, por ejemplo antes de una subexpresión, antes de un operador o antes de un paréntesis. Así, en lugar de escribir:

```
int suma = ((x1 * x2 + y) / (x3 * x4 * z)) + (x1 / (x2 * y * z)) - ((EPSILON *
x2) + x4);
```

preferiremos escribir:

```
int suma = ((x1 * x2 + y) / (x3 * x4 * z))
+ (x1 / (x2 * y * z))
- ((EPSILON * x2) + x4);
```

Como regla práctica, recomendamos utilizar paréntesis «de más» para aumentar la legibilidad del código cuando el orden de precedencia de los operadores sin ellos pueda resultar confuso. Recuerde que aunque la disposición física del código no afecta a la función que realiza, determina en gran medida el tiempo que tardarán otros en comprender el software que construyamos.

Disposición de los elementos e instrucciones de control. Los símbolos que marcan el inicio y final de un método, o de un bloque de código, deben disponerse en líneas dispuestas exclusivamente a tal efecto. Un ejemplo en Pascal sería el siguiente:

```
if (x > 0) then
begin
  y := y + INTERVALO;
  recalcularDistancia(x,y,z);
end;
```

Deben evitarse los bloques que únicamente contienen una línea, pues aunque son correctos desde el punto de vista sintáctico, introducen símbolos prescindibles que producen confusión. Así por ejemplo, el siguiente código:

```
if (x > 0){
  y += INTERVALO;
}
else{
  y *= x;
}
```

debería escribirse del siguiente modo:

```
if (x > 0)
  y += INTERVALO;
else
  y *= x;
```

En cuanto a la disposición de las instrucciones que conforman el cuerpo de los bucles y las sentencias selectivas, deben sangrarse uniformemente, encajando organizadamente las estructuras en bloques imaginarios, unas dentro de otras según lo indicado en la Figura 6.2.

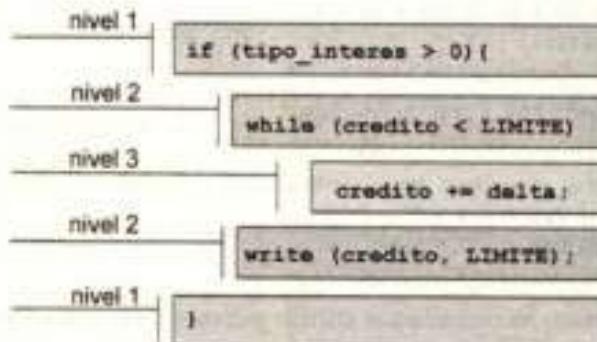


Figura 6.2: Organización de bloques de código encajados

Gestión de las condiciones de error: manejo de excepciones

Los modernos lenguajes de programación orientada a objetos incorporan el denominado *manejo de excepciones*, un potente mecanismo para gestionar las condiciones de error que pueden presentarse en un programa.

Se denomina **excepción** a cualquier anomalía o condición de error no esperada que se produce durante la ejecución de un programa

Si no fueran adecuadamente tratadas, las excepciones forzarían, generalmente, la finalización abrupta del programa, emitiendo el sistema un mensaje de error más o menos controlado. Algunas de las posibles fuentes de error son:

- División por cero.
- Desbordamientos positivos (*overflow*) o negativos (*underflow*).
- Argumentos de método o función no esperados.
- Resultados fuera de rango.
- Índices de *arrays* fuera de los límites correctos.
- Errores en acceso a ficheros, etc.

Normalmente se puede detectar cuándo ocurren ciertos tipos de excepciones, pues éstas se producen al ejecutar operaciones «potencialmente peligrosas». Se trata de errores denominados excepciones *síncronas*, pues suceden en un momento predecible. Se denominan excepciones *asíncronas* a las que se producen como consecuencia de sucesos que escapan al control del programa, tales como la pulsación por parte del usuario de una cierta secuencia de teclas que aborta la ejecución de un programa en curso (por ejemplo [CTRL]+C en programas de consola en MS-DOS, o [ALT]+F4 en Windows).

Los mecanismos de tratamiento de excepciones están orientados al tratamiento de las excepciones síncronas, siendo su objetivo la independencia entre la detección de condiciones de error (realizado por el código «de negocio» específico de la aplicación) y el tratamiento de los errores detectados. Las principales ventajas del uso de excepciones son las siguientes:

- Permiten separar el código de tratamiento de errores del resto del código.
- Posibilitan la propagación de errores hacia arriba en la pila de llamadas entre métodos.
- Permiten agrupar y clasificar los diferentes tipos de errores.

Frecuentemente, el modelo de excepciones de los lenguajes actuales se basa en el concepto de salto no local, implementado con algunas palabras reservadas tales como *throw*, *catch* y *try*, haciendo uso de los siguientes tres conceptos:

1. Las instrucciones que pueden dar lugar a posibles excepciones son aisladas en bloques especiales (*try*) para controlar la propagación de las mismas.
2. Cuando se detecta una excepción, el programa eleva (*throw*) una excepción, que será tratada en otro punto del mismo.
3. La excepción se trata mediante código específico de tratamiento de excepciones que captura (*catch*) la excepción.

Haciendo uso de los conceptos anteriores, la existencia de un mecanismo de tratamiento de excepciones permite transferir el control y transmitir una información, desde un punto en la ejecución del programa hasta un controlador de la excepción detectada. A menudo se facilita, además, una manera de agrupar excepciones similares, de modo que se puedan escribir controladores para capturar y tratar tanto excepciones individuales como grupos. El siguiente código muestra un ejemplo de utilización de lo dicho hasta el momento:

```
void inicializarFichero(File f) throws IOException{
    if (abrirFichero() == ERROR_APERTURA)
        throw new IOException("No se pudo abrir el fichero");
    else
        // resto del código...
}

void inicializarSistemaCompleto(){
    // Inicializar controladores...
    try {
        inicializarFichero(f);
    } catch (IOException e) {
        System.out.println("Inicialización incompleta: " + e.getMessage());
    }
    // resto del código
}
```

En este ejemplo, una operación de inicialización de un sistema, que incluye la inicialización de un determinado fichero como parte de sus operaciones, aísla el código de inicialización del fichero como potencial fuente de errores. Así, la invocación al método `inicializarFichero()` se realiza dentro de un bloque `try`, puesto que dicho método podría elevar una excepción del tipo `IOException`. Con un bloque `catch` en el método `inicializarSistemaCompleto()` –inmediatamente a continuación del bloque `try`– se controla la posible excepción.

Además de lo dicho anteriormente, la introducción de código para el tratamiento de excepciones no impone, con carácter general, un coste adicional en cuanto a tiempo de ejecución a un código que no eleve ninguna excepción. Como además es fácil de comprender e implementar, es una técnica cada día más ampliamente utilizada en la construcción de software de calidad.

Documentación del código

Documentar el código es añadir información al código «original» para explicar lo que hace con el objetivo de que cualquier persona que lo lea entienda lo que se está haciendo y por qué, por lo que la información debe ser suficiente. Se trata de una tarea esencial de Ingeniería del Software a la que a menudo, sin embargo, no se da la importancia que merece.

Se denomina **documentación** a cualquier información gráfica o escrita que describe, define, especifica, reporta o certifica actividades, requisitos, procedimientos o resultados, así como al proceso de generar o revisar un documento (IEEE, 1990)

Documentar el código no es un lujo, sino una necesidad que sólo se apreciará en su justa medida cuando haya que reparar errores en el código o se tenga que dotar al programa de nuevas capacidades. Como ya se ha insistido en otras partes de este mismo capítulo, por una razón o por otra, todo programa será modificado en el futuro, bien por el programador original, bien por otro programador que le sustituya.

Comúnmente se distinguen hasta cuatro tipos de documentación:

1. Documentación sobre el diseño y la arquitectura, donde se detallan los principios que guiaron la construcción y se proporciona una visión general del sistema software, que incluye su relación con el entorno. Este tipo de documentación no describe qué algoritmo debe utilizarse para implementar una cierta funcionalidad, ni por qué es necesario implementar un método concreto, sino que trata las necesidades generales que obligan a implementar un conjunto de métodos.
2. Documentación técnica, que incluye la documentación del código sobre los algoritmos, interfaces, estructuras de datos, etc. Se trata con más detalle en breve.
3. Documentación para los usuarios finales, tales como el manual de usuario, tutoriales, o la documentación específicamente orientada a los administradores del sistema y otro personal de soporte del software.
4. Documentación comercial, como los artículos blancos y otras formas de publicación comercial. Desde la perspectiva de la Ingeniería del Software, la única forma digna de mención son los denominados «artículos blancos» (*white papers*). Se trata de escritos pseudo-científicos donde se apuntan los beneficios de una determinada tecnología o producto software. Su objetivo es mostrar la importancia de la solución proporcionada por la compañía, explicando adecuadamente cuál es el enfoque tomado y en qué consiste a grandes rasgos. No obstante, y como resulta lógico, siendo documentación comercial, se da importancia sólo a las soluciones propias frente a las de la competencia y rara vez se hace alusión a los posibles puntos débiles.

La programación literaria

Hay autores que proponen cambiar radicalmente nuestra actitud acerca de la documentación pues «ha llegado el momento de mejorar significativamente la documentación de los programas, y la mejor manera de hacerlo es considerar que son obras literarias» (Knuth, 1984). Knuth ha propuesto una forma de programación, denominada «programación literaria» que combina el uso de un lenguaje de programación con un lenguaje de documentación, con el objetivo de construir programas más robustos y fáciles de mantener.

A efectos prácticos, la programación literaria combina los comentarios (dirigidos a las personas que deben comprender y modificar el código) y el código de computadora en sí (dirigido al compilador). El enfoque seguido es, en cierto modo, similar al de los lenguajes de marcado (XML, HTML, etc.), donde se combina texto junto con información que se utiliza para indicar cosas tales como la forma en que aquél ha de estructurarse, o presentarse.

La información se estructura jerárquicamente y de acuerdo a una agrupación lógica de los contenidos, pero también sigue unas ciertas reglas de formato para que las herramientas de generación automática de documentación puedan extraer la información dirigida al traductor. La herramienta CWEB, por ejemplo, permite combinar comentarios en texto en formato TeX –Un sistema de composición tipográfica desarrollado por Donald Knuth– con código en lenguaje C, facilitando el escribir el código sin tener en cuenta en qué orden será procesado por el compilador. En realidad, escribir programas en CWEB es como escribir documentos en TeX, sólo que con un modo adicional de trabajo –llamado *modo C*– que se añade a los modos tradicionales de TeX: *escritura horizontal, vertical y modo matemático*. El siguiente ejemplo en CWEB sigue las directrices de la programación literaria:

```
#include <stdio.h>
<estructuras de datos>
<cabeceras de funciones>
<programa principal>
```

Y el programa principal sería algo como:

```
<programa principal>* main() { /*
  Código del programa principal */
}
```

Cuando las descripciones de alto nivel que aparecen entre símbolos «menor que» y «mayor que» se expanden de acuerdo a lo establecido en el texto fuente, se obtiene un programa sintácticamente correcto en C.

Volvamos por un momento a la documentación técnica. Ésta consiste en comentarios empotrados dentro del código, cuyo objetivo es, como ya se señaló, explicar aquellos aspectos del código que no se explican por sí solos. No hay que traducir al español lo que se hace, sino explicar por qué se decidió hacerlo así. De este modo, los comentarios deben orientarse a detallar de qué se encarga una clase o un método, cuál es el uso esperado de una variable, interfaz o método, qué algoritmo se está empleando y (si existen) qué otras alternativas habrá disponibles para el algoritmo utilizado.

Con carácter general, e independientemente del lenguaje de programación utilizado, puede optarse por situar los comentarios de dos maneras distintas:

1. Justo antes del código que se comenta. Es lo más recomendable, pues el resultado es más legible que cuando se utilizan otros tipos de comentario. Estos comentarios suelen utilizarse para comentar conjuntamente un bloque de sentencias, y tienen la ventaja adicional de que no se encuentran limitados por la longitud de la línea:

```
/* Calcular el máximo del array para posteriormente utilizarlo
como límite en la función de reducción */
max = A[i];
for (i=0; i < MAX_ARRAY; i++)
    if (A[i] > max) max = A[i];
reducirDistancia(max,A,B);
```

2. En la misma linea (a la derecha del código que comentan). Se utilizan para describir la acción o la razón de una línea en particular, pero deben utilizarse con mesura. Además del efecto perturbador que produce mezclar en una misma línea código y comentarios, si son demasiado largos pueden situarse fuera del área de edición, con la consiguiente molestia para quien los ha de leer. Un ejemplo de uso correcto sería:

```
int longArray = array.length(); // para evitar recalcular la longitud
```

Hoy en día existen herramientas como Javadoc, ClassDoc, ROBODoc, Doxygen o TwinText que permiten autogenerar documentación a partir de los comentarios del código (ver Figura 6.3). Estas herramientas extraen los comentarios del código fuente y les proporcionan un formato más legible, habitualmente HTML, ASCII, LaTeX o RTF. Como la documentación generada suele, además, venir estructurada en forma de guía de referencia, es más fácil hacer búsquedas rápidas por un nombre de clase o método en particular y acceder directamente a la documentación sobre el mismo.

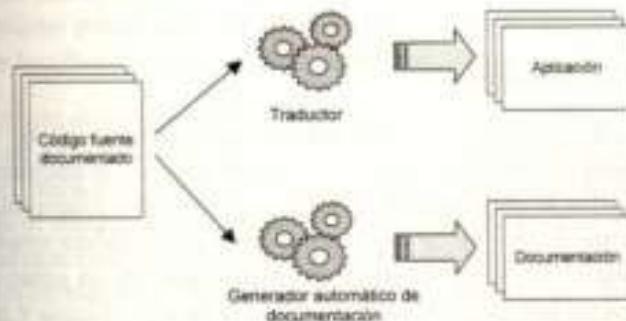


Figura 6.3: Funcionamiento de un generador automatizado de documentación

Para la generación automática de código con estas herramientas, es necesario realizar los comentarios con una sintaxis especial. En Javadoc, por ejemplo, dichos comentarios deben aparecer justo antes de la declaración de una clase, un atributo o un método en el mismo código fuente. Javadoc analiza dichos comentarios y a partir de la información incluida en los mismos genera una documentación bien organizada de las clases, métodos y atributos que aparecen en el código fuente. La Figura 6.4 muestra la aplicación de Javadoc a dos clases, Cola y Nodo, que dan soporte al concepto de estructura de colas dinámica FIFO (primero en entrar, primero en salir).

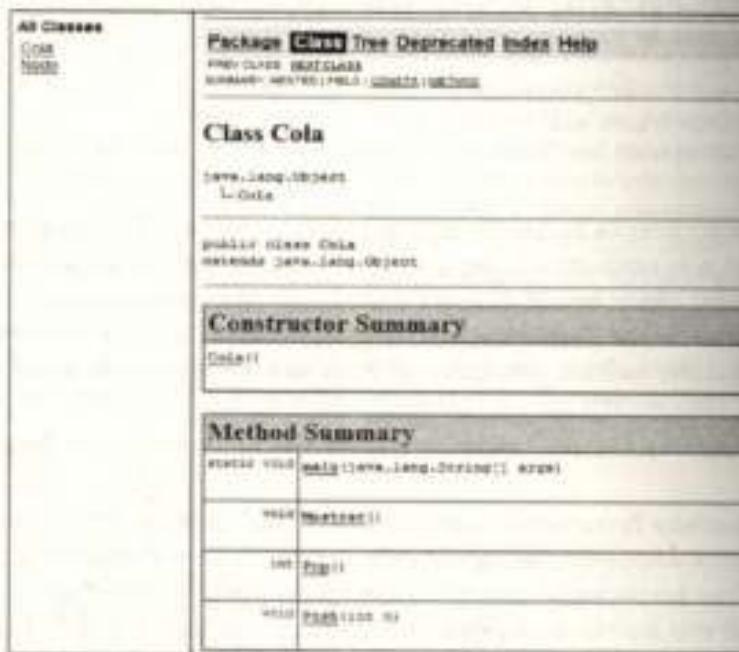


Figura 6.4: Documentación generada por Javadoc

Técnicas de afinación del código

Afinar² el código consiste en mejorarlo de acuerdo con ciertas técnicas. La mejora se enfoca a un determinado número de aspectos, fundamentalmente la eficiencia. Sin embargo, la utilización de algunas de estas técnicas no está al alcance de cualquiera, pues se requiere cierta experiencia como programador para comprender cuándo deben emplearse y cómo utilizarlas adecuadamente. No obstante, tampoco es necesario ser un genio para poder afinar un código; simplemente hay que estudiar metódicamente el problema y pensar cuidadosamente qué está ocurriendo y cuál puede ser la técnica más adecuada para resolverlo.

²La expresión *code tuning* puede también traducirse como perfeccionamiento o puesta a punto del código.

La posibilidad de transformar un código que se ejecuta en 200 milisegundos en otro que, siendo funcionalmente equivalente se ejecuta en la mitad de tiempo, es sencillamente tentadora. Sin embargo, debe quedar claro que las técnicas de afinación del código no son, si con mucho, la mejor forma de mejorar la eficiencia de nuestros programas. Es mucho más eficaz utilizar algoritmos con menor complejidad computacional, emplear máquinas más potentes o hacer uso de mejores compiladores.

Técnicas de afinación y eficiencia

Es posible comparar la diferencia de eficiencia que puede lograrse con sólo elegir el mejor algoritmo dentro de los disponibles para la ordenación de un conjunto de números. Utilizando un modesto PC con un procesador Intel Pentium M a 1.70GHz y 1GB de RAM, el algoritmo de la burbuja tardó 49,94 segundos en ordenar 100.000 números, mientras que el QuickSort tardó sólo 21 milisegundos. Esto supone un ahorro en coste computacional del 99,96%. Por mucho que nos esforzásemos en afinar la implementación del método de la burbuja, jamás podríamos alcanzar la eficiencia del método QuickSort. En cuanto a la máquina, la misma operación utilizando un PC con procesador Intel Core 2 Duo T5600 a 1.83 GHz y 2GB de RAM tardó 40 segundos en ordenar los mismos datos con el algoritmo de la burbuja (un 19,9% de ahorro con respecto a los mismos cálculos en la máquina anterior) y 19 milisegundos (un 9,5% de ahorro) con el algoritmo QuickSort.

Esto evidencia la diferencia de eficacia entre utilizar algoritmos con menor complejidad computacional y emplear máquinas más potentes. En general, siempre será deseable elegir un algoritmo de menor complejidad antes que realizar afinaciones a un algoritmo ineficiente o emplear máquinas más potentes.

Esto nos indica que algo a tener en cuenta antes de estudiar las técnicas de afinación es saber cuándo deben usarse. No se trata de técnicas que haya que poner en práctica por sistema, movidos por la obsesión de obtener el código más eficiente posible. La afinación debe utilizarse para optimizar un código que se ha demostrado ineficiente. Así, no debe usarse sistemáticamente como método de construcción: si no es necesario optimizar, es preferible no emplear técnicas de afinación.

Las técnicas de afinación del código pueden clasificarse por grupos, pues algunas de ellas abordan el mismo problema desde diferentes perspectivas. A continuación se muestra la clasificación de Jon Bentley para las técnicas de afinación del código:

- Reglas de *cesión de espacio* para ganar tiempo: estas reglas abogan por sacrificar algo de espacio de almacenamiento con tal de obtener beneficios en términos de tiempo. Por ejemplo, reducir el tiempo de acceso y manipulación de datos en una estructura aumentando el tamaño de la misma. Así, es posible hacer que se ejecute más rápido un programa que controla los valores de cierto número de bits si se accede a los datos byte a byte o en estructuras aún mayores (*word* o doble *word*). Un ejemplo de reglas de cesión de espacio para ganar tiempo son las tablas arcoíris, conjuntos de valores precalculados que se emplean en criptografía para la obtención de una contraseña a partir de su codificación encriptada.

- Reglas de *cesión de tiempo* para ganar espacio: son, en cierto modo, la antítesis de las anteriores, pues abogan por ceder en términos de tiempo para obtener una cierta ganancia en términos de espacio. Por ejemplo, una estructura de datos puede reducir costes de almacenamiento si se le permite solapar datos que no se utilizan simultáneamente, haciendo uso de alguna forma de memoria virtual, lo cual inevitablemente aumenta el tiempo de acceso a los datos que contiene.
- Reglas de *bucles*: su objetivo es mejorar la eficiencia haciendo modificaciones en los bucles del código original, por ejemplo moviendo código fuera del bucle o fusionando varios bucles que iteran sobre el mismo conjunto de valores. Así, el siguiente bucle:

```
for (i=0; i<15; i++) {
    x = n * m;
    y += x * i;
}
```

podría modificarse para sacar la asignación $x = n * m$ fuera del bucle puesto que no depende de la variable de iteración i :

```
x = n * m;
for (i=0; i<15; i++)
    y += x * i;
```

Esta modificación mejora la eficiencia ya que ahorra 14 asignaciones innecesarias.

- Reglas *lógicas*: proponen la mejora de las expresiones lógicas mediante su sustitución por expresiones algebraicas equivalentes (pero menos costosas de evaluar) y el reordenamiento de expresiones lógicas compuestas para que se evalúen antes las menos costosas y más frecuentes, y más tarde las más costosas y raras. La aplicación de estas técnicas a la siguiente sentencia en Java:

```
if ((x > 0) || (y > 0)) z += y;
```

comporta un estudio de la propia naturaleza del programa. Si a partir del conocimiento que nos aporta dicho estudio sabemos que en este punto en la ejecución la posibilidad de que x sea positivo es mucho menor que la posibilidad de que lo sea y , reordenaremos la expresión para escribir:

```
if ((y > 0) || (x > 0)) z += y;
```

puesto que la expresión lógica se evaluará a verdadero siempre que alguna de las dos subexpresiones que la integran ($y > 0$, $x > 0$) se evalúe a verdadero. Siendo mayor la probabilidad de que la subexpresión $y > 0$ se evalúe a verdadero, la reordenación ahorrará muchas evaluaciones de la expresión completa. Esto es así porque la mayoría de los lenguajes incluyen la denominada *evaluación en cortocircuito*, que consiste en

evaluar primero la subexpresión que conforma el operando izquierdo y, si el valor de éste es suficiente para determinar el resultado, no se evalúa el operando derecho.

- **Reglas de procedimientos:** también mediante la modificación de procedimientos o funciones es posible mejorar su eficiencia. Se recomienda, por ejemplo, reescribir los métodos recursivos en forma iterativa, modificar las llamadas entre métodos relacionados para evitar que el segundo en ejecutarse tenga que esperar la finalización del primero y el tercero la del segundo (trabajo en tubería), o explotar las posibilidades de trabajo en paralelo. Veamos un ejemplo práctico. La función que calcula el factorial habitualmente es el paradigma de las funciones recursivas, puesto que la propia definición de factorial incluye la noción de recursividad. Así, la escritura más habitual de dicha función en Java es la siguiente:

```
public static long factorialRecursoivo(int n){
    if (n < 2)
        return 1;
    else
        return n * factorialRecursoivo (n-1);
}
```

La solución iterativa tal vez carece del *glamour* de su equivalente recursiva, pero es mucho más eficiente:

```
public static long factorialIterativo (int n){
    int acumulador = 1;
    for (int i = 1; i <= n; i++)
        acumulador *= i;
    return acumulador;
}
```

Porque la versión recursiva de la función Factorial requiere, en cada invocación recursiva, tiempo para hacer la llamada a la función, crear las variables locales, copiar los parámetros por valor, ejecutar las instrucciones de la función, destruir las variables locales y los parámetros por valor y finalmente, salir de la función. Sin embargo, la versión iterativa únicamente realiza esta secuencia una vez, y si bien la parte ejecutiva es más compleja, no es comparable el número de operaciones que lleva a cabo ni la eficiencia de la misma. En un PC con procesador Intel Core 2 Duo T5600 a 1.83 GHz y 2GB de RAM tardó 9,219 microsegundos en ejecutar el algoritmo recursivo por los 6,984 microsegundos del algoritmo iterativo (cerca de un 25% más eficiente).

- **Reglas de expresiones:** pretenden obtener expresiones más eficientes sin modificar, como es lógico, su resultado. De este modo, proponen sustituir expresiones por otras algebraicamente equivalentes pero más eficientes, eliminar subexpresiones comunes, o explotar el paralelismo en la evaluación de expresiones. Algunos ejemplos de estas técnicas serían el reemplazo de operaciones trigonométricas costosas por sumas y

multiplicaciones, y el evitar multiplicar y dividir por múltiplos de 2 para en su lugar utilizar desplazamientos de bits, sustituyendo:

$$y = x / 2$$

por la operación equivalente con desplazamiento de bits:

$$y = x >> 1$$

Una última anotación. Antes de abalanzarse sobre el código a aplicar estas técnicas, todo desarrollador debería tener en cuenta dos cosas importantes. La primera de ellas es que los modernos compiladores realizan optimizaciones del código que llevan a cabo algunas de las mejoras enumeradas, como la eliminación de subexpresiones comunes, la mejora de eficiencia en las instrucciones iterativas moviendo código para que se ejecute fuera del bucle, o el reemplazo de multiplicaciones por sumas en instrucciones repetitivas. Simplemente, muchas veces no merece la pena el esfuerzo: es cuestión de documentarse antes de realizar la afinación. La segunda cosa a tener en cuenta es el impacto de muchas de estas técnicas sobre la legibilidad del código, lo cual debe sopesarse para no aumentar innecesariamente los costes de mantenimiento asociados a un código menos legible.

6.6.2 Anticipar los cambios

Como ya se ha dicho, los sistemas de software son complejos, y por lo tanto, difíciles de gestionar. Una de las características que más dificulta su gestión es el hecho de que con toda seguridad, el sistema no permanecerá siempre tal y como fue creado originalmente. Muy probablemente, a lo largo de la vida del software, éste se verá sometido a actividades de mantenimiento que introducirán cambios, bien para ajustarlo a nuevos requisitos o bien para eliminar errores.

Es posible que, tras cierto tiempo, el software ya no se corresponda exactamente con el diseño considerado durante su desarrollo, ni el código con la documentación disponible, por lo que cada vez resultará más complicado entender el sistema. De hecho, el cambio continuo en el software es parte intrínseca de su evolución y ha sido enunciado como una de las leyes de la evolución del software (las cuales se estudian en la Sección 8.5.2).

Los cambios introducidos en una parte del software a menudo afectan a otras, y ocurre a menudo un problema con los cambios, pues la documentación, el rediseño, y en general todos los artefactos generados como consecuencia de la nueva funcionalidad a menudo están disponibles, pero los elementos que se ven afectados por el cambio y que no sufren cambios directos no tienen tanta suerte. Ello lleva a estos elementos a convertirse en potencial fuente de problemas. El objetivo de un correcto proceso de construcción, en lo referente a los posibles cambios que sufrirá el software, es aislar aquellas áreas más inestables para que el posible fallo afecte sólo a unas pocas rutinas, cuantas menos mejor.

McConnell propone tener en cuenta los siguientes puntos para poder así anticipar los cambios (McConnell, 2004):

1. *Identificar elementos susceptibles de cambiar.* Si se ha hecho un buen proceso de obtención y formalización de requisitos, se deberían haber documentado las posibles mejoras futuras y las áreas en que periódicamente habrá que realizar ajustes. Algunas de las áreas más proclives a modificaciones son las que incluyen dependencias de hardware, formatos de entrada-salida, estructuras de datos complejas y reglas de negocio. Sin embargo, todo proyecto tiene elementos que *a priori* no parecen proclives a los cambios, pero que posteriormente pueden verse afectados. Para elementos no incluidos en la documentación de requisitos, deben seguirse los pasos 2 y 3.
2. *Separar aquellos elementos que es probable que cambien.* Deben clasificarse de tal modo que cada elemento de los identificados en el paso anterior tenga su propia clase (si estamos utilizando programación orientada a objetos). Otro enfoque sería almacenar juntos en una clase todos aquellos componentes que cambiarán a la vez.
3. *Aislar los elementos que se prevea puedan cambiar,* diseñando las interfaces entre dichos elementos de modo que los cambios dentro de un elemento queden circunscritos al mismo y no se propaguen al resto de elementos con los que éste interacciona. Lo ideal sería que una clase que utiliza otra que ha cambiado no notase el cambio si la interfaz no ha cambiado. Esto es similar a lo que ocurre cuando, acostumbrados a un ratón mecánico, utilizamos un ratón óptico. La tecnología que los hace funcionar es muy distinta, si bien el manejo desde el punto de vista del humano no ha cambiado, pues las operaciones básicas siguen siendo pulsar el botón izquierdo, pulsar el botón derecho y desplazar.

Puestos a la tarea de buscar aquellos elementos más proclives a sufrir cambios, debe tenerse en cuenta que algunas áreas son más proclives que otras a cambiar. McConnell señala las siguientes como especialmente proclives a ser modificadas:

- *Reglas de negocio:* son muy inestables porque dependen en muchos casos de factores externos. Un ejemplo concreto para un sistema de gestión administrativa de una universidad sería el siguiente. Si se modifica el cálculo de asignaturas que permiten pasar de curso a un estudiante en la universidad (número de asignaturas mínimas superadas, existencia de asignaturas llave, máximo número de convocatorias de examen disponibles, etc.), todos los cómputos asociados deben cambiar.
- *Dependencias del hardware:* como adaptación de la interfaz al tamaño de la pantalla, resolución de la imagen (por ejemplo, el modo de calcular los gráficos en tiempo real) dependiendo de la tarjeta gráfica, etc.
- *Entradas y salidas de la aplicación:* deben tenerse en cuenta los formatos de entrada y salida de los datos de la aplicación (tipo de los ficheros, formatos de fichero no

estandarizados y cambiantes, etc.), pero también cuestiones de entrada-salida relacionadas directamente con los usuarios, como el diseño de los formularios en pantalla, el número de campos de un formulario que se mostrarán simultáneamente, la posición de los mismos, etc.

- *Dependencias de las extensiones no estándar* de los lenguajes de programación. A menudo, los entornos de desarrollo proporcionan extensiones útiles del lenguaje al que dan soporte pero que no están en el estándar de dicho lenguaje. El uso de estas extensiones es a menudo fuente de errores posteriores, por lo que deben utilizarse con cuidado.
- *Áreas de especial dificultad de diseño o construcción*, donde la implementación pudo no haber sido todo lo buena que hubiera sido deseable y, por ello, puede requerirse una implementación mejor en el futuro. Una consulta compleja en una base de datos, por ejemplo, puede no implementarse del modo más óptimo posible en una primera aproximación (porque en ese punto muchas veces es suficiente con hacer algo que funcione). Con seguridad, más adelante habrá que rehacer la consulta para mejorar el tiempo de respuesta u otros factores.
- *Variables de estado*: es habitual que una variable que en principio fue pensada para contener un valor de estado booleano, necesite de pronto ser modificada para aceptar un tercer e incluso un cuarto estado. Para prever estas circunstancias, es preferible implementar estas variables como de tipo enumerado, un recurso que facilita añadir nuevos valores. Una válvula podría, inicialmente, estar cerrada o abierta, para lo cual la implementación obvia es una variable lógica `estaCerrada`, que tomará los dos posibles valores. Sin embargo, si con el tiempo los nuevos modelos de válvulas permiten estados intermedios o no determinados, habría sido mejor una implementación que facilitase acomodar nuevos valores. Así, una variable de tipo enumerado con valores `CERRADA`, `PARCIALMENTE_ABIERTA`, `ABIERTA`, `ESTADO_INDEFINIDO`, que permitiría seguir introduciendo nuevos valores si surgen, sería lo más deseable.
- *Limitaciones en el tamaño de los datos*: a menudo es necesario ampliar el tamaño máximo de almacenamiento inicialmente previsto para una estructura de datos, por ejemplo, un `array`. Es una práctica recomendada utilizar constantes con nombre (por ejemplo, `MAXIMO_ALCANZABLE`) en lugar de emplear directamente el dato (1.000, por ejemplo), ya que de este modo una modificación subsiguiente obligará a modificar sólo aquel punto del programa donde se establezca el valor de la constante, y no todas y cada una de las sentencias en que se emplee el dato (declaración del `array`, límites de bucles, etc.).

Una técnica muy atractiva y diametralmente distinta al resto de las enumeradas más arriba, es el uso de métodos dirigidos por tablas. Se trata de un esquema que permite buscar la información sobre las sentencias lógicas (`if` y `case`) en una tabla en lugar de utilizar

dichas sentencias directamente. En realidad, cualquier selección que se implementa con sentencias *if* o *case* puede implementarse con una tabla, si bien el uso de éstas sólo resulta interesante cuando la cadena de condiciones y subcondiciones se torna lo suficientemente compleja. En estos casos, la búsqueda en una tabla puede ser mucho más sencilla que un complejo entramado de sentencias selectivas anidadas. En casos más elementales, la simplicidad y claridad de las sentencias selectivas es siempre preferible.

6.6.3 Construir para verificar

Construir para verificar significa enfrentarse a las tareas de construcción de software buscando y arreglando todos los errores que pudieran generar fallos posteriores durante la ejecución. Algunas de las técnicas que facilitan la construcción de software bajo esta filosofía son el empleo sistemático de pruebas de unidad, la organización del código para permitir pruebas automatizadas, la utilización de métodos estandarizados que faciliten las revisiones del código y el uso limitado de estructuras complejas de los lenguajes de programación, a menudo difíciles de entender.

Uso sistemático de pruebas de unidad

Desde el punto de vista de la construcción, las pruebas unitarias pueden verse como pequeños módulos auxiliares que se encargan de verificar el funcionamiento de otras unidades lógicas del sistema. Su objetivo es verificar que un componente funciona correctamente por sí mismo, sin tener en cuenta las relaciones que pueda tener con otras partes del sistema. Su uso sistemático facilita la creación de software de calidad y aporta un buen número de ventajas a los desarrolladores.

En la programación orientada a objetos, las pruebas unitarias se crean para utilizar otro código fuente llamando directamente a los métodos de una clase. En dichas llamadas, se pasan ciertos parámetros y posteriormente se comparan los valores que se generan con los valores esperados. Los métodos de pruebas unitarias residen en clases separadas, pero asociadas a las clases que prueban.

Es importante apuntar que este tipo de pruebas está intimamente ligado a las actividades de construcción, siendo una actividad difícilmente clasificable como estrictamente de construcción o estrictamente de pruebas. En algunos modelos de desarrollo, como en la programación extrema, y en general en los métodos ágiles, se recomienda desarrollar las pruebas unitarias antes incluso que la propia unidad a probar.

Organización del código para permitir pruebas automatizadas

Para crear pruebas unitarias es posible bien utilizar herramientas que faciliten la generación del código fuente inicial de la prueba, o bien escribir la prueba completamente a mano. Como se verá con mayor detalle en la Sección 7.6, existen varios *marcos de pruebas*

(frameworks) denominados genéricamente xUnit, que resultan de gran utilidad para ayudar a realizar pruebas unitarias en diferentes lenguajes. Estos marcos de pruebas unitarias están formados por diversas clases que proporcionan al desarrollador gran flexibilidad para escribir pruebas unitarias a partir de un código previamente organizado a tal efecto.

Los dos elementos básicos que se manejan durante las pruebas unitarias son los casos de prueba y las colecciones de prueba. Un *caso de prueba* está formado por clases que tienen una serie de métodos que ejecutan los métodos de otra clase, la cual es objeto de la prueba. Estos casos de prueba se estructuran en *colecciones de pruebas*, conjuntos de casos de prueba sobre clases funcionalmente relacionadas que pueden automatizar el proceso. No obstante, la automatización de las pruebas unitarias se estudia en más profundidad en el capítulo siguiente, a donde remitimos al lector interesado en profundizar en su estudio.

Métodos para la revisión del código

Una revisión de código, también denominada **inspección** –o *walkthrough* haciendo uso del término original en inglés–, es una reunión en la que cierto componente software se presenta a un conjunto de actores involucrados en el desarrollo, tales como usuarios, clientes, gestores y otros interesados, para que éstos aporten sus comentarios, realicen críticas y en último término comuniquen su aprobación o reprobación del código.

Durante las sesiones de revisión se lleva a cabo un análisis sistemático del código, intentando detectar defectos en el mismo que hayan podido pasar inadvertidas. La detección y reparación de estos errores en una fase tan temprana tiene un beneficioso efecto sobre la calidad final del producto software desarrollado, pero también está orientada a la formación del programador a través de la comprobación de los fallos que haya podido cometer durante la codificación.

Las revisiones de código pueden clasificarse en dos grandes categorías, las *revisiones formales* y las *revisiones ligeras*:

- Las revisiones formales a menudo se llevan a cabo en más de una sesión. Se trata de un proceso detallado de búsqueda de errores y discusión sobre el código «línea por línea». Los revisores utilizan plantillas como la mostrada en la Figura 6.5 para documentar y clasificar los errores detectados, así como para sugerir mejoras.
- Las revisiones ligeras son más informales, pero no tienen por qué ser menos efectivas. A diferencia de las anteriores, no se plantean como una actividad separada de la codificación, sino que forman parte del propio proceso de programación. Las más relevantes son las siguientes:
 - Circulación de código nuevo mediante correo electrónico. Existen sistemas software que automáticamente envían a otros desarrolladores los ficheros de código fuente tras ser implementados e introducidos en la herramienta de control de versiones. Quienes reciben el código, lo revisan y envían sus comentarios al autor original, también por correo electrónico.

- Programación por parejas. Se trata de una técnica de desarrollo que consiste en codificar en equipos de dos programadores: mientras uno escribe el código, el otro lo lee y hace comentarios. La tarea que cada uno desempeña no es fija, y de hecho, se suelen cambiar frecuentemente los papeles.
- Uso de herramientas de revisión. Existen herramientas que detectan de forma automatizada algunos de los problemas que se detectarían en una revisión hecha por personas. El método aporta ventajas: resulta menos estresante para el programador y no requiere disponibilidad de otras personas.
- Revisiones «por encima del hombro», que se refieren a las sugerencias informales de mejora y a los comentarios hechos por otros desarrolladores que leen el código a medida que se está construyendo.

Lista de comprobaciones para inspección de código	
Proyecto:	
Autor:	
Nombre de ficheros:	
Fecha:	
Número de errores	Tipo de error
Mayor	Menor
	Temaño de función y complejidad inadecuados
	Expresión de ideas poco clara en el código
	No se cumplen los estándares internos de codificación
	Encapsulación pobre
	Prototipos de función utilizados incorrectamente
	Tipos de datos que no coinciden
	VARIABLES sin inicializar al comienzo del código
	VARIABLES sin inicializar en la entrada de bucles
	Lógica pobre: no funcionará según las necesidades expresadas
	Comentarios pobres
	Condiciones de error no tratadas
	Selecciones múltiples sin opción por defecto (switch sin default)
	Sintaxis incorrecta (ej. Uso inadecuado de ==, =, &, &&, etc.)
	Código no reentrant en sitios peligrosos
	Código lento en un área donde es importante la rapidez
	Otros (indicar):
	Otros (indicar):

Error menor: si no se elimina puede producir un problema visible por el cliente.

Error menor: falta de cumplimiento de estándares de codificación, errores leves de escritura, etc. que no producen errores mayores.

Figura 6.5: Lista de comprobaciones para las revisiones formales de código

En realidad, muchas organizaciones utilizan una mezcla de ambos enfoques para hacer las revisiones de código. Si bien los métodos formales tienen sus ventajas, pues permiten dejar por escrito los comentarios realizados, levantar acta de las discusiones y planificar las modificaciones sugeridas, se ha demostrado que los métodos ligeros permiten detectar un número similar de errores, pero en menos tiempo y a menor coste, lo que los hace especialmente atractivos.

Uso limitado de estructuras complejas del lenguaje

Un hábito que tiene un impacto negativo en la construcción del software es el uso de elementos del lenguaje complejos o difíciles de entender. Sobre todo cuando existen alternativas de menor complejidad, el uso de estos elementos debería ser objeto de reflexión. El abuso de la aritmética de punteros en lenguajes como C, la utilización de complicadas estructuras recursivas de datos (cuando no son estrictamente necesarias), o la innecesaria complejidad derivada del uso extremo de técnicas de afinación del código, pueden ser citadas entre las prácticas a evitar.

La elección del lenguaje de programación a utilizar durante la construcción, por ejemplo, es una decisión importante que tiene un impacto directo en la futura verificación del software. La utilización de ciertos lenguajes de programación, que por su propia naturaleza son especialmente indicados para la programación a bajo nivel, puede llevar al desarrollo de programas cuya tasa de errores sea mayor que los desarrollados con otros lenguajes. El soporte para estructuras como los punteros, o el hecho de su orientación hacia bajo nivel que llevan al programador a utilizar un estilo parco, brusco hasta cierto punto, son reconocidas como la principal fuente de errores en muchos programas. Tal vez el paradigma de estos lenguajes sea C (y su hermano mayor, C++).

Ya hemos hablado anteriormente del código oscuro, sin embargo, no todos los lenguajes de programación se prestan igualmente a la escritura de programas poco inteligibles. Algunos lenguajes como C y C++ son los más citados como fácilmente «ofuscables», no en vano los concursos de código oscuro comenzaron con C. Muchos de los programadores de estos lenguajes, tienden a utilizar un estilo semi-criptico que se basa en un cierto número de mitos, muchos de ellos completamente falsos. Como por ejemplo que un código fuente más corto genera código máquina más eficiente que uno largo. En particular, el siguiente programa en Java:

```
for (i = 0; i < 10; i++)
    a[i] = i;
```

se ejecuta en 5.588 microsegundos en un modesto PC con procesador Intel Core 2 Duo T5600 a 1.83 GHz y 2GB de RAM. Un tosco equivalente como el siguiente:

```
a[0] = 0;
a[1] = 1;
a[2] = 2;
...
a[8] = 8;
a[9] = 9;
```

tarda un 15% menos en hacer el mismo trabajo. 4.749 microsegundos para ser exactos.

No obstante, el ejemplo anterior no debe confundirnos. Casi siempre es preferible, por una cuestión de estilo y legibilidad, utilizar la primera de las opciones. Tampoco estamos diciendo que la primera opción sea innecesariamente compleja. El único propósito de este ejemplo es hacer reflexionar a los candidatos a futuros participantes de los concursos de código oscuro sobre el hecho de que escribir complejos programas de una sola línea tal vez

no sólo sea una forma de fastidiar a los demás con código ininteligible, sino también una manera de crear código más ineficiente si lo comparamos con un equivalente más largo.

6.6.4 Utilización de estándares

Un **estándar** puede definirse como un conjunto de especificaciones técnicas documentadas que regulan la realización de un proceso o la fabricación de un producto. En la fabricación de un producto, el objetivo de la estandarización es fundamentalmente la interoperabilidad entre artículos construidos por diferentes fabricantes. Dado que la construcción de software es un caso particular de «fabricación de un producto», resulta de aplicación lo anterior.

La elaboración de un estándar es un proceso que conlleva tiempo y en el que intervienen muchas personas y organizaciones diferentes. En principio, el uso generalizado de un producto hace surgir consorcios y asociaciones de usuarios, que son las primeras organizaciones que, tras un periodo de utilización, digamos desordenada, promueven la normatización mediante la elaboración de documentos técnicos cuyo objeto es sistematizar el uso del producto entre sus miembros. Estos documentos, con frecuencia de carácter interno, suelen denominarse *especificaciones*, y si bien no pueden considerarse estándares, son frecuentemente el germen de un estándar posterior. Así, no suelen cubrir todo el espectro de usuarios, sino sólo aquello que atañe a los miembros del consorcio donde se han generado. Es importante reseñar que una especificación siempre está asociada a comités no acreditados para la publicación y difusión formal de estándares, tales como IETF (*Internet Engineering Task Force*), OMG (*Object Management Group*), o W3C (*World Wide Web Consortium*). A partir de una o más especificaciones sobre el mismo producto, organizaciones de certificación tales como IEEE, AENOR, CEN o ISO, y con el concurso de expertos en la materia, mejoran la especificación para cubrir las necesidades de todos los usuarios y fabricantes potenciales del producto.

En la construcción de software es necesario tener en cuenta la existencia de diferentes estándares sobre métodos de comunicación (estándares sobre el formato y contenido de los documentos, de legibilidad, etc.), lenguajes de programación (escritura de código estándar Java, C, etc.), plataformas, notaciones de representación y herramientas. Las personas que se enfrentan a la construcción de software deben conocer los estándares, especificaciones y procedimientos por los que deben regirse. Los estándares pueden clasificarse en dos tipos: externos e internos.

- Los estándares externos son aquellos que afectan a la construcción de software y cuya fuente no es directamente la organización que lleva a cabo el desarrollo. Dentro de éstos podemos agrupar las especificaciones de interfaces publicadas por organismos nacionales e internacionales de estandarización tales como ISO, IEEE, IETF (*Internet Engineering Task Force*), OMG (*Object Management Group*), W3C (*World Wide Web Consortium*) y otros.
- Los estándares internos afectan a la construcción de software dentro de la organización de desarrollo. Pueden ser reglas internas de obligado cumplimiento, o simples

recomendaciones de cara a la construcción. Casi todas las compañías de desarrollo (por supuesto todas las importantes) cuentan con su estándar de construcción.

Como ejemplo de estándar interno, veamos las recomendaciones de estilo en la codificación que Microsoft exige a sus programadores. Con objeto de obtener programas más claros y consistentes, Microsoft recomienda el uso de la notación húngara³, que consiste en indicar el tipo de las variables dentro del propio nombre de las mismas. Para ello, debe incluirse en el nombre un prefijo que indique su tipo según la Tabla 6.1.

Tabla 6.1: Algunos prefijos de tipo según la notación húngara

Prefijo	Descripción
a	Array
b	Boolean
c	Char
db	Real de doble precisión (<i>double</i>)
dw	Doble Word
fn	Función
fp	Real de precisión simple (<i>floating point</i>)
i	Integer
l	Long
m_	Miembro de una clase
p	Puntero
s	String

Además de esta tabla de prefijos simples, se dan recomendaciones sobre el uso de tipos combinados, como por ejemplo, «puntero a entero» (Tabla 6.2).

Tabla 6.2: Combinación de prefijos de tipo en notación húngara

Combinación de prefijos	Descripción
<i>psMiCadena</i>	Un puntero a <i>String</i>
<i>m_pszMiCadena</i>	Un puntero a una cadena que es atributo de una clase

Sin embargo, las recomendaciones no se quedan en los nombres de variable. Microsoft detalla cómo deben hacerse los comentarios para todos los distintos tipos de elementos susceptibles de ser documentados, como ficheros, clases, definiciones de métodos, etc.

³Esta notación fue inventada por Charles Simonyi, húngaro de nacimiento, de ahí su nombre.

En Microsoft, es obligatorio este comentario al principio de todo fichero de código:

```
*****
Fichero: MIFICHERO.EXT
Resumen: Breve resumen del contenido y propósito del fichero.
Clases: Clases declaradas o utilizadas (ficheros fuente).
Funciones: Funciones exportadas (ficheros fuente).
Origen: Indicaciones sobre el origen de los contenidos. No
        se trata de una historia de los cambios sino una
        referencia a la herencia de editores que hay detrás
        de los contenidos u otras indicaciones sobre el
```

Información sobre copyright y avisos legales

El ejemplo de Microsoft ha servido para describir, brevemente, el alcance y significado de los estándares internos. Sin embargo, y como ya se ha dicho, es de esperar que cualquier organización que se dedique a la construcción de software profesional cuente con sus propias recomendaciones. De hecho, la notación húngara es un ejemplo que suscita poco consenso, ya que según muchos autores, complica innecesariamente la comprensión del código. Las convenciones publicadas por Sun Microsystems para el lenguaje Java son más neutras y, en general, mejor aceptadas. La Tabla 6.3 muestra el apartado de dichas convenciones dedicado a los nombres de identificadores.

Tabla 6.3: Convenciones de nombres para Java publicadas por Sun Microsystems

Elemento	Regla	Ejemplo
Clases	Emplear nombres que deberán comenzar por mayúscula. Si tiene más de una palabra, la inicial de cada una irá en mayúscula.	class AveRapaz
Interfaces	Utilizar la misma regla de las clases	interface Persistente
Métodos	Emplear verbos y hacer que comiencen por minúscula. En caso de múltiples palabras, las iniciales intermedias irán en mayúsculas.	calcularArea()
Variables	Utilizar nombres significativos y cortos, evitando que comiencen por subrayado o dólar. Hacer que comiencen por minúsculas, y si son varias palabras, las iniciales intermedias irán en mayúsculas.	float distanciaFocal
Constantes	Los nombres deberán ir en mayúsculas. Si el nombre tiene más de una palabra, separarlas por símbolos de subrayado. Evitar constantes ANSI como MAX_INT o MAX_FLOAT	LIMITE_INFERIOR

Aunque las dos notaciones mostradas no pueden ser consideradas estándares al no haber sido sometidas a un proceso formal de estandarización, sí es posible catalogarlas como especificaciones, al haber sido promulgadas por instituciones con cierto ascendiente en su campo. En ambos casos, su uso está ampliamente difundido.

6.7 La calidad en la construcción de software

Durante el proceso de construcción, los desarrolladores tienen a su disposición diversos mecanismos que les permiten estar seguros de que el software que están escribiendo cumple los requisitos exigidos de calidad y es conforme con las especificaciones incluidas en la documentación de requisitos. Se trata de mecanismos y técnicas heterogéneas tales como la depuración, los análisis de rendimiento, las aserciones, las revisiones de código y el análisis estático, entre otras, que estudiaremos con detalle en esta sección.

6.7.1 Aserciones y diseño por contrato

Las **aserciones** se emplean a menudo como elemento de verificación que, tras ejecutar una prueba unitaria, permite comparar los valores obtenidos con los valores esperados. En la programación orientada a objetos, las aserciones constituyen la base de una técnica más amplia denominada «diseño por contrato».

Una **aserción** es una expresión lógica que especifica el estado de un programa, o un conjunto de condiciones que las variables de ese programa deben satisfacer en un cierto punto de su ejecución (IEEE, 1990)

Como veremos más adelante, las aserciones pueden tener la forma de precondiciones, de postcondiciones o de invariantes. En Java, una aserción simple se expresa así:

```
assert Expresión1;
```

donde **Expresión1** es una expresión lógica. Cuando se ejecuta el programa, se evalúa el valor de **Expresión1**; si dicho valor es igual a **false** se eleva una excepción de tipo **AssertionError**. El empleo de aserciones ralentiza la ejecución del código, pues supone la realización de comprobaciones que no se ejecutarían en un código sin aserciones. Sin embargo, las ventajas que reporta su utilización hacen de ellas una atractiva herramienta de verificación. Meyer cita las siguientes ventajas del uso de aserciones (Meyer, 1999):

1. Ayudan a producir software fiable.
2. Proporcionan una documentación sistemática.
3. Son una importante herramienta para la comprobación y depuración del software.

No obstante, una vez verificado el código, siempre es posible deshabilitar las aserciones como opción en la compilación del programa, en cuyo caso, una instrucción con aserciones sería equivalente a la misma instrucción sin ellas.

En el Capítulo 4 se analizó cómo un caso de uso puede ser especificado en detalle con un conjunto de informaciones que incluyen precondiciones (asunciones anteriores al comienzo del caso de uso) y postcondiciones (asunciones que deben cumplirse tras finalizar el caso de uso). De un modo similar, es posible expresar la tarea que desempeña un método o función mediante precondiciones y postcondiciones en forma de aserciones.

Una **precondición** es una expresión lógica que establece las propiedades que deben cumplirse antes de invocar un código. Una **postcondición** es una expresión lógica que establece las propiedades que debe garantizar un código tras finalizar su ejecución.

El siguiente ejemplo muestra una función que calcula el factorial de un número *n*. Dicha función comprueba que *n* está dentro del rango válido de valores (es mayor o igual a cero y no supera el máximo que provocaría un desbordamiento en la pila de llamadas recursivas). La implementación de este método haría uso del tratamiento de excepciones:

```
public float factorial(int n) throws IllegalArgumentException {
    if (n < 0 || n > VALOR_MAXIMO)
        throw new IllegalArgumentException("Argumento inválido: " + n);
    return calcularFactorial(n);
}
```

No obstante, es posible implementar el método interno *calcularFactorial* del ejemplo anterior mediante aserciones para verificar que el retorno de dicho método se adhiere a la condición esperada por el método *Factorial* que lo invoca:

```
private float calcularFactorial(int n)
{
    assert (n >= 0 && n <= VALOR_MAXIMO);
    // Cálculo del factorial de n...
}
```

Las postcondiciones se pueden emplear para garantizar que, por ejemplo, tras insertar un elemento en una pila ésta no está vacía, o que el resultado de computar la media de un conjunto de números es un valor entre el mínimo y el máximo de dicho conjunto.

Otra de las aplicaciones de las aserciones es la utilización de invariantes, las cuales pueden aplicarse a un diverso número de casos: invariantes de clase, invariantes de bucle o invariantes internas.

Una **invariante** es una expresión lógica que debería ser cierta siempre en un determinado segmento de código o en un punto concreto de un programa
 (IEEE, 1990)

El siguiente es un ejemplo de aserción utilizada como invariante en un fragmento de código que incluye 3 alternativas de flujo basadas en el valor de una cierta variable n:

```
if (n >= 5) {  
    ...  
} else if (n < 0){  
    ...  
} else {  
    assert (n >= 0 && n < 5);  
    ...  
}
```

El *diseño por contrato* es una técnica de programación que ve las relaciones entre clases y clientes de las clases como si fuera un convenio entre partes que incluye cláusulas formales en forma de aserciones. Así, el cliente de una clase acudirá al contrato de la misma para saber qué se compromete a cumplir la clase cuando un cliente invoca a un cierto método (*postcondiciones*). La clase por su parte, establece las condiciones previas (*precondiciones*) que un cliente debe asumir para poder realizar la invocación a uno de sus métodos. La definición formal de las afirmaciones y responsabilidades de cada uno constituye lo que Meyer denomina *contrato entre la clase y sus clientes* (Meyer, 1999). El diseño por contrato aspira a establecer un grado de confianza entre elementos de un sistema software que garantice la calidad del mismo.

6.7.2 Análisis de rendimiento

El **análisis de rendimiento**, comúnmente conocido por el término inglés original *profiling*, es una técnica de análisis dinámico de programas que consiste en la monitorización y estudio del comportamiento de un componente software a partir de información que se recaba durante su ejecución. Su utilización permite determinar qué partes de un programa tienen fallos de rendimiento (en términos de memoria o de velocidad) y por tanto deben ser optimizadas para cumplir los requisitos no funcionales de rendimiento incluidos en la especificación de requisitos software. El análisis de rendimiento permite detectar, fundamentalmente, *escapes de memoria (memory leaks)* y problemas de rendimiento en términos de velocidad de ejecución por debajo de lo esperado.

Se llama *escape* a toda pérdida gradual de memoria causada por un programa que no retorna adecuadamente la memoria que le fue asignada para uso temporal.

Una **escape de memoria** es una pérdida gradual de la memoria disponible en el sistema causada por un defecto en la aplicación

La existencia de escapes de memoria en una aplicación frecuentemente lleva a un agotamiento de la memoria, que en último término puede ocasionar el colapso del programa.

del sistema operativo). Un escape de memoria, por pequeño que sea, resulta un problema serio en aplicaciones que se ejecutan continuamente.

Existen diversas herramientas software que facilitan el análisis de una aplicación en ejecución, proporcionando importante información sobre su comportamiento. Se trata de herramientas que permiten monitorizar el estado de los diferentes hilos de ejecución, el uso de memoria y el rendimiento de la CPU, con un coste bajo en términos de carga adicional al sistema.

La Figura 6.6 proporciona una vista dinámica de los objetos en memoria dentro del montículo de la máquina virtual de Java. Obsérvese cómo la aplicación proporciona datos por cada clase tales como el número de bytes ocupados por instancias de las mismas, la edad media de los objetos de la clase, el número de objetos activos y otros. Cuando se sospecha que existe un escape de memoria, el programador deberá examinar las columnas *live objects* (objetos activos) y *generations* (número de operaciones de recolección de basura en la memoria a que ha sobrevivido) para detectar qué clase es la causante del escape.

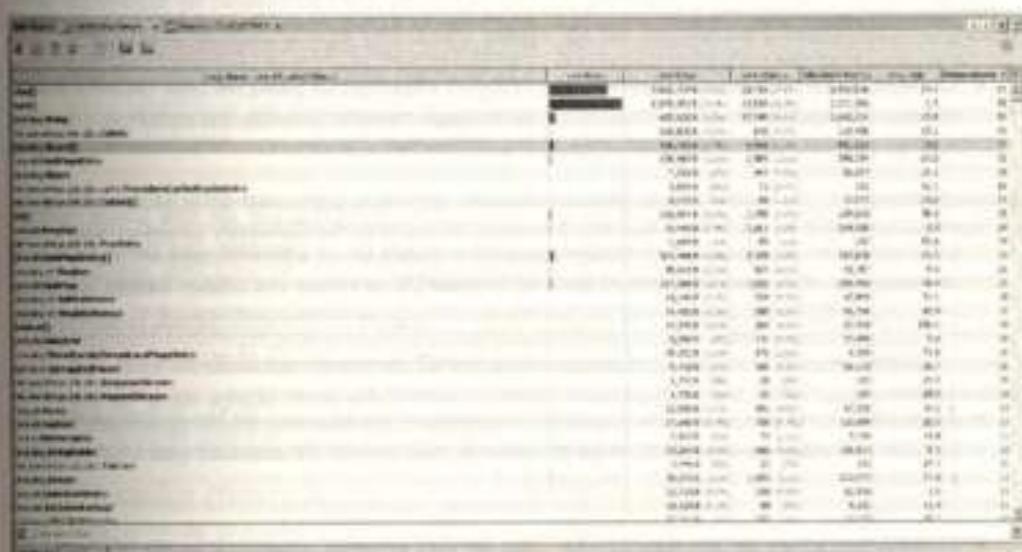


Figura 6.6: Monitorización de escapes de memoria en NetBeans profiler

Las herramientas de análisis de rendimiento resultan hoy en día imprescindibles para comprender el comportamiento de un programa. Los datos que proporcionan resultan muy valiosos para los programadores a la hora de analizar un componente software y encontrar aquellos defectos que ocasionan fallos de rendimiento.

6.7.3 Depuración

La construcción implica necesariamente la verificación de que el software construido se comporta tal y como era de esperar. El proceso de comprobación, y eventualmente de reparación de las discrepancias entre el comportamiento esperado y el comportamiento real se denomina *depuración*.

Depuración es el proceso de detectar, localizar y corregir fallos en un programa software (IEEE, 1990)

Algunas de las técnicas más utilizadas para llevar a cabo la depuración de un software son las siguientes:

- Puntos de ruptura (*breakpoints*). Puntos del programa en los que la ejecución se suspende para permitir la observación de los resultados del programa, valores de las variables, rendimiento, etc. En función de la causa que provoque la suspensión de la ejecución, pueden clasificarse en puntos de ruptura de datos, de código, estáticos, programables, etc.
- Centinelas (*watches*). Ventanas que permiten observar el valor de una variable o dato durante la ejecución de un programa, reflejando dinámicamente los cambios a medida que se van ejecutando instrucciones.
- Ejecución paso a paso. Técnica que consiste en la ejecución de una única instrucción (o parte de ella) como respuesta a una orden de la persona que lleva a cabo la depuración. Combinada con el uso de centinelas permite ver cómo varían los datos almacenados en respuesta a la ejecución de cada instrucción individual, lo que facilita la identificación precisa de la instrucción donde se encuentra el error.
- *Seguimiento de la ejecución*. Consiste en elaborar un registro de las instrucciones utilizadas durante la ejecución de un programa, con objeto de conocer qué instrucciones se están invocando y en qué secuencia.
- *Comprobaciones de escritorio*. Se trata de ejecuciones «manuales» de la lógica del algoritmo a depurar, donde el desarrollador emplea un conjunto de datos de entrada y realiza una comprobación sobre ellos. No se trata de un método exhaustivo para comprobar todos los posibles caminos, sino una aproximación. La prueba completa tal vez requeriría más de un conjunto de datos de prueba.

El proceso de depuración comienza cuando se informa de que hay discrepancias entre lo que se espera que haga el sistema y lo que hace realmente. A la hora de enfrentarse a dicho proceso no existen pautas predefinidas, si bien algunos autores han recopilado consejos prácticos para ayudar a los desarrolladores.

David Agans, por ejemplo, ha publicado un conjunto de nueve reglas «indispensables» para la depuración (Agans, 2002):

1. Comprender el sistema. La persona encargada de la depuración ha de conocer el código fuente, así como el entorno de ejecución y las herramientas que se utilizaron para construir el software. Una buena recomendación en este punto es leer la documentación que acompaña al software.
2. Hacerlo fallar. Para poder proseguir con el proceso de detección y reparación del error, es imprescindible reproducir el fallo al que nos enfrentamos. Resulta por tanto indispensable dedicar cierto tiempo a llegar a conocer qué condiciones ocasionan el fallo que se desea eliminar.
3. No pensar, sino observar. Utilizar puntos de ruptura, centinelas, ejecución paso a paso, etc. para observar el comportamiento del software y así conseguir tener una idea aproximada de cuándo y dónde se produce el fallo. Sobre todo, evitar sacar conclusiones a partir únicamente de unos pocos datos.
4. Estrechar la búsqueda. Un principio que se corresponde con la filosofía «divide y vencerás», la cual aboga por aislar partes de un problema y resolverlos separadamente como estrategia para resolver el problema completo. En la depuración, implica identificar las distintas partes que pueden estar ocasionando el fallo y probarlas separadamente.
5. Modificar sólo una cosa a la vez. Muchos cambios simultáneos, sin comprobación que permita determinar cuál es el impacto de cada uno en el funcionamiento del sistema, únicamente introducirán confusión en el proceso.
6. Realizar una auditoría. Por mínimo que pueda parecer, cualquier detalle es importante a la hora de detectar un error. La creación de un registro de eventos detallado puede ser de gran ayuda para encontrar el error o reproducir el fallo, por ejemplo si se encuentran patrones en los registros de eventos.
7. Comprobar primero lo obvio. No debe asumirse que nada es cierto o que funciona simplemente porque es obvio: debe cuestionarse todo.
8. Pedir opinión a otras personas. Es conveniente solicitar a un compañero que eche un vistazo al problema, pero sin decirle dónde creemos nosotros que se encuentra el error para no condicionar su análisis. Deberá considerarse también la opinión de expertos en foros específicos, que puede ser muy valiosa.
9. Pensar que si no se ha arreglado, el error seguirá ahí. Aunque parezca obvio decirlo, no se debe pensar que los errores se arreglan solos. Si tras introducir cambios en el código no específicamente orientados a corregir el error, no es posible reproducirlo, no debe asumirse que se ha arreglado. Por el contrario, deberían deshacerse

los cambios y analizar el impacto de los mismos en el código, intentando reproducir de nuevo el error y re aplicando los cambios para ver una vez más cómo afectan a la desaparición del error.

Estas nueve reglas se basan en la experiencia del autor y constituyen un *prontuario indispensable* para los neófitos de la depuración. Debe tenerse en cuenta que la depuración es una tarea difícilmente sistematizable, por lo que no debe esperarse un algoritmo que permita detectar y eliminar todos los errores de un software siguiendo un número fijo de pasos en secuencia. Muy al contrario, la depuración puede convertirse en una ardua tarea, especialmente la detección del error, por lo que resulta recomendable guiar nuestras acciones mediante pautas de actuación prestablecidas como por ejemplo, las nueve reglas enunciadas.

6.8 Gestión de la construcción

La gestión de la construcción depende en gran medida del tipo de ciclo de vida utilizado. En modelos de ciclo de vida tradicionales, la construcción es sinónimo de codificación. Allí, la gestión de la misma se encuentra claramente separada de la gestión de otras actividades de desarrollo tales como el diseño y las pruebas. Sin embargo, en modelos más modernos como los métodos ágiles o los basados en la evolución de prototipos, la identificación entre construcción y codificación no existe. Son enfoques que propugnan un modelo de construcción donde, por supuesto, tiene un papel central la codificación, pero donde también caben actividades de prueba y diseño.

Las tareas más importantes dentro de la gestión de la construcción de software son la planificación de la construcción y la toma de medidas durante la misma. Ambas tareas se estudian con más detalle en las secciones siguientes.

6.8.1 Planificación de la construcción

La elección de un método de construcción concreto es un aspecto clave de la actividad de planificación de la construcción. Posiblemente el más importante junto con la elección de un lenguaje de construcción. Téngase presente que esta elección afecta directamente a la forma en que se materializan los pre requisitos de la construcción, el orden en que se llevan a cabo y el grado de compleción que se espera alcanzar antes de comenzar la construcción.

La forma en que se aborda la construcción determina la forma en que se podrá reducir la complejidad en el proyecto, y en general, la manera en que los cuatro principios fundamentales de la construcción se ponen en práctica. Los objetivos que la importancia de cada uno de estos principios introduzcan en el proyecto deben ser cuidadosamente planificados para asegurar que se llevan correctamente a buen término en los plazos y circunstancias especificados.

La planificación de la construcción también define el orden en que deben crearse e integrarse los diferentes componentes que resultan de la construcción, las acciones relacionadas

con la calidad del proceso, y la asignación de tareas a las diferentes personas que participan de la construcción según el método de desarrollo elegido.

Una de las herramientas más importantes a la hora de planificar la construcción son las estimaciones del tamaño del software. Contar con medidas que permitan hacer estimaciones fiables facilitará la elaboración de mejores planificaciones, permitiendo ajustar mejor el tamaño del trabajo y dividir el trabajo en elementos separables. Estas estimaciones ayudan además a realizar un mejor seguimiento del progreso de la construcción una vez comenzada.

6.8.2 Métricas de construcción

Durante el proceso de construcción de un software es posible llevar a cabo un gran número de mediciones que posteriormente pueden resultar útiles, bien en el desarrollo actual o bien en otros futuros. Algunos de los artefactos y elementos de los que más comúnmente se toman medidas son los siguientes:

- Código desarrollado. Se puede englobar dentro de las medidas de productividad. La más común es la cuenta del número de las líneas de código fuente entregadas. También se utiliza el número de instrucciones de código objeto entregado o el número de páginas de la documentación del sistema.
- Código reutilizado. La reutilización es un principio fundamental del desarrollo de software moderno. Conocer cuánto código ha sido creado de nuevo y cuánto procede de desarrollos anteriores, de bibliotecas externas o de cualesquiera otras fuentes de código listo para su reutilización, permite mejorar las estimaciones sobre el esfuerzo necesario en la construcción de cara a desarrollos posteriores.
- Código destruido. El código eliminado o comentado puede obtenerse si se está utilizando una herramienta de control de versiones. La medida proporciona información sobre la actividad realizada dentro de un fichero o en el proyecto completo.
- Complejidad del código. La teoría de la ciencia del software propuesta por Halstead —que se estudia con detalle en el Capítulo 3— es probablemente la medida de complejidad mejor conocida. La ciencia del software utiliza un conjunto de medidas primitivas para computar la longitud global del programa, el volumen mínimo potencial para un algoritmo y su volumen real (número de bits requeridos para especificar un programa), o la complejidad del software.
- Estadísticas de inspección del código. Los datos obtenidos de las revisiones formales del código son de gran utilidad, pues incluyen (entre otros) el número de errores detectados en las mismas. Pueden utilizarse como base para calcular, por ejemplo, el porcentaje de errores detectados en dichas revisiones con respecto al total de errores encontrados a lo largo del desarrollo.
- Porcentaje de errores encontrados y reparados.

- Esfuerzo de construcción, que suele estimarse a partir del tamaño del código desarrollado (habitualmente en líneas de código) y el tiempo a emplear en la construcción.

Todas estas medidas resultan muy útiles en la gestión del desarrollo (y más concretamente del proceso de construcción), asegurando que la misma cumple con los parámetros de calidad exigidos y en todo caso, mejorando el proceso de construcción en su conjunto.

6.9 Resumen

La siguiente nube de palabras resume los principales conceptos que hemos tratado en el capítulo y muestra su importancia relativa.



Figura 6.7: Principales conceptos tratados en el capítulo

En este capítulo hemos tratado la construcción de un software a partir de su diseño y hemos visto que la construcción se entiende hoy en día, no como mera implementación del diseño en un cierto lenguaje de programación, sino como un conjunto de actividades que por supuesto incluyen la codificación, pero también la verificación del código, su depuración y ciertos tipos de pruebas.

Hemos dedicado cierto tiempo a estudiar los diferentes lenguajes de construcción, y nos hemos detenido posteriormente a entender qué es la reutilización, cuál es su papel en los desarrollos modernos y cómo la amplia disponibilidad de bibliotecas de componentes reutilizables ha cambiado sustancialmente el modo de crear aplicaciones. Finalmente hemos estudiado todo lo relacionado con la calidad del software desde el punto de vista de la construcción, para terminar el reparando en la importancia de la gestión en el proceso de construcción de un software. Nos hemos detenido particularmente en la planificación del proceso de construcción y de la toma de medidas específicas de esta parte de un proyecto como punto fundamental para su mejor comprensión y mejora.

6.10 Notas bibliográficas

La segunda edición del clásico de Steve McConnell «*Code Complete*» (McConnell, 2004), ganador del premio *Jolt Excellence* (considerados los Óscar de la industria del desarrollo de software) como mejor libro de programación, discute las mejores prácticas para la codificación de software en cualquier lenguaje de programación, por lo que resulta una interesante lectura para todos aquellos interesados en la construcción de software.

Si quiere saber más sobre las técnicas de afinación del código, resultará sin duda interesante un vistazo al apéndice 4 (*Rules for code tuning*) del libro (Bentley, 2000). Originalmente parte del libro del mismo autor «*Writing efficient programs*», publicado en 1982 y hoy descatalogado, incluye la clasificación de técnicas por reglas descrita en el capítulo junto con numerosos ejemplos de aplicación.

Si no lo ha hecho aún, es un buen momento para leer dos libros clásicos sobre cómo hacer mejor programación, «*The Pragmatic Programmer*» de Andrew Hunt y David Thomas (Hunt y Thomas, 1999), y «*Refactoring*» de Martin Fowler (Fowler, Beck, Brant, Opdyke y Roberts, 1999). El primero contiene gran cantidad de ideas, observaciones y recomendaciones para desarrolladores, las cuales resultan sencillamente impagables. El segundo acuña un nuevo término, *refactoring*, que consiste en tomar el código existente, estudiarlo y modificarlo para mejorar su diseño con el objetivo de facilitar las futuras modificaciones y mejoras del mismo. Aproveche y lea también el clásico artículo «*Producción en masa de componentes software*» de Roy McIlroy (McIlroy, 1968), una lectura muy recomendable para llegar a comprender la forma en que se produce el software hoy en día que parte de conocer qué problemas llevaron al cambio de mentalidad que dio lugar a la filosofía actual de reutilización del software.

Oracle tiene publicado mucho material sobre cómo documentar programas utilizando Javadoc. Para profundizar en el tema recomendamos la guía oficial de estilo de Javadoc «*How to Write Doc Comments for the Javadoc Tool*», disponible en la web de Oracle.

En «*La catedral y el bazar*», Eric S. Raymond (2001) analiza varios proyectos exitosos de software libre y estudia las implicaciones que el enfoque de construcción colaborativa del mundo del software libre ha tenido y tiene en el futuro del desarrollo de software, confrontando el modelo catedral –donde el código no se libera hasta que no se pone en circulación cada nueva versión–, al modelo dinámico y colaborativo del bazar –donde el código se está haciendo a la vista de todo el mundo–. Interesante y amena lectura para reflexionar sobre el futuro de la construcción de software.

6.11 Cuestiones de autoevaluación

- 6.1 Indique razonadamente si la siguiente afirmación es verdadera o falsa: «La mejor manera de mejorar la eficiencia de un código es utilizar técnicas de afinación».

R. Falso. Como hemos visto en el capítulo, las técnicas de afinación del código no son la mejor forma de mejorar la eficiencia en todos los casos. En numerosas ocasiones es más eficaz

utilizar algoritmos con menor complejidad computacional, emplear máquinas más potentes o hacer uso de mejores traductores.

- 6.2 La utilización de estándares en la construcción de software aporta un buen número de beneficios tal y como se ha visto en el capítulo pero, ¿qué tipos de estándar existen?

R. *Hay básicamente dos tipos de estándares en la construcción de software: estándares externos (aquellos cuya fuente no es directamente la organización que lleva a cabo el desarrollo) y estándares internos (reglas internas de una organización de desarrollo encaminadas a uniformizar la construcción).*

- 6.3 ¿Qué es un generador automatizado de documentación?

R. *Es una herramienta que genera la documentación de un software en un formato específico (generalmente en HTML) a partir de los comentarios embebidos en el código. Para ello analiza el código fuente, extrae los comentarios, les da el formato deseado y genera una salida que pone a disposición del usuario.*

- 6.4 ¿Qué es un White paper?

R. *Un white paper es una forma de documentación comercial, con forma de escrito pseudo-científico, donde se apuntan los beneficios de una determinada tecnología o producto software. En ellos se da importancia a las soluciones propias frente a las de la competencia, y generalmente se evitan los puntos débiles del producto que se describe de forma deliberada.*

- 6.5 ¿En qué consiste el denominado problema de la complejidad del software? Cite algunas técnicas de construcción de software que permitan reducir la complejidad.

R. *Se trata de un problema inherente a las limitaciones de la mente humana, que se resume perfectamente en una frase de Dijkstra: «no existe una mente lo suficientemente capaz como para memorizar completo un programa de computadora moderno». Las soluciones radican en escribir código sin complejidades innecesarias, con menos errores y que sea más fácil de probar y mantener. Algunas de estas técnicas abordan la legibilidad, la documentación del código, el empleo de estructuras de control, la afinación del código, etc.*

- 6.6 Utilizando notación húngara, ¿qué nombre pondría a una variable real que va a contener el importe del salario bruto de un empleado, siendo Empleado una clase y su salario bruto un atributo?

R. *El nombre habría de ser m_fpSalarioBruto*

- 6.7 Dentro de la clasificación de técnicas de afinación de código propuesta por Jon Bentley están las reglas de expresiones. ¿En qué consisten?

R. *Se trata de técnicas cuyo objetivo es mejorar la eficiencia con que se evalúan las expresiones de un programa, para lo cual proponen sustituir las expresiones originales por otras algebraicamente equivalentes pero más eficientes, eliminar subexpresiones comunes, o explotar la posibilidad de evaluación paralela de subexpresiones.*

- 6.8 Explique brevemente en qué consiste la denominada evaluación en cortocircuito.

R. *Consiste en evaluar las expresiones lógicas compuestas de modo que siempre se evita primero la subexpresión que conforma el operando izquierdo y, si el valor de la evaluación es suficiente para determinar el resultado de la expresión completa, no evaluar la subexpresión que conforma el operando derecho.*

- 6.9** ¿Cuáles son los principios fundamentales de la construcción de software?
- R *Minimizar la complejidad, anticipar los cambios, construir para la verificación y utilización de estándares de construcción.*
- 6.10** Enumere algunos de los artefactos y elementos de los que más comúnmente se toman medidas durante el proceso de construcción.
- R *Es posible (y deseable) tomar medidas relacionadas con el código desarrollado, reutilizado y destruido, con la complejidad del código, obtener estadísticas de inspección del código; calcular porcentajes de errores encontrados y arreglados, medir el esfuerzo en la construcción, o la situación a lo largo del tiempo de las tareas de construcción, entre otras.*

6.12 Ejercicios y actividades propuestas

6.12.1 Ejercicios resueltos

- 6.1** Modifique el fragmento de código oscuro puesto como ejemplo en la Sección 6.6.1 para que en lugar del saludo «Hi!» emita en su lugar el mensaje «Ingeniería del Software».

Solución propuesta: Lo primero que haremos de hacer es lógicamente traducir a Java «legible» el código que tenemos. Para ello, haremos uso de una tabla de códigos unicode pues como vimos el código se había obfuscado mediante la sustitución de los caracteres legibles del código fuente (tales como la «a», la «b», etc.) por sus equivalentes códigos en formato Unicode. El resultado de la traducción es:

```

1: /* Just Java
2:  Peter van der Linden
3:  April 1, 1996.
4:
5: PvdL 1 Apr96
6: */ class h {
7: public
8: static void
9: main(String
10: []a) {System.
11: out.println
12: ("Hi!"); } } */
13:
14: */

```

Una vez traducido, la modificación a introducir es trivial:

```
12: ("Ingeniería del Software"); } } */
```

Si compilamos el nuevo código se muestra correctamente el nuevo mensaje. Sólo nos restaría obfuscarn el código que acabamos de escribir traduciendo cada carácter a su código unicode correspondiente, tarea que por trivial y repetitiva no realizaremos aquí, pero que proponemos al lector interesado.

- 6.2 Utilizando reglas de afinación del código, mejore la eficiencia del siguiente algoritmo:

```
public static void algoritmoMejorable(int n){
    int i;
    double numerador, denominador;

    for (i=1; i<=n; i++){
        numerador = 5 * i;
        denominador = Math.pow(2,i);
        System.out.print(numerador);
        System.out.print("/");
        System.out.println(denominador);
    }
}
```

Solución propuesta: El algoritmo muestra por pantalla los términos de la serie numérica:

$$\frac{5}{2} + \frac{10}{2^2} + \frac{15}{2^3} + \frac{20}{2^4} + \dots + \frac{5 \cdot n}{2^n}$$

Para ello, calcula en cada paso de la iteración el término i y lo muestra por pantalla. Es posible eliminar una operación muy costosa (la potencia de 2) en el cuerpo del bucle y sustituirla por una de menor coste como es la multiplicación. También es posible sustituir la multiplicación $5 \cdot i$ por una suma, menos costosa en términos de eficiencia. Llevando a cabo estas sustituciones, el código quedaría así:

```
public static void algoritmoMejorado(int n){
    int i;
    double numerador = 0, denominador = 1;

    for (i=1; i<=n; i++){
        numerador += 5;
        denominador *= 2;
        System.out.print(numerador);
        System.out.print("/");
        System.out.println(denominador);
    }
}
```

La mejora en tiempos es evidente: más de un 30% en media, si bien esto dependerá del número de términos utilizado y el entorno de ejecución. No obstante, téngase en cuenta que un buen compilador casi con toda seguridad hubiera hecho esto mismo por nosotros, por lo cual el esfuerzo realizado parte más bien de una necesidad académica, que es la resolución del ejercicio, más que de una necesidad real de afinar el código.

- 6.3 Escriba una clase `pila` en java con las operaciones básicas de este tipo de estructuras y genere documentación en HTML para dicha clase utilizando Javadoc.

Solución propuesta: El primer paso consiste en escribir el código fuente en java para la implementación de las funcionalidades de la pila.

```

public class Nodo{
    public int valor;
    public Nodo sig;

    public Nodo(int v){
        valor = v;
        sig = null;
    }
}

public class Pila {
    private Nodo cima;

    // Constructor
    public Pila(){
        cima = null;
    }

    // Método para añadir un elemento a la pila
    public void push(int n){
        Nodo nuevoNodo = new Nodo(n);

        // Si la pila está vacía poner cima a apuntar al nuevo nodo
        if(cima == null){
            cima = nuevoNodo;
        }
        else{
            nuevoNodo.sig = cima;
            cima = nuevoNodo;
        }
    }

    // Método para sacar un elemento de la pila
    public int pop(){
        int n = 0;
        if (cima != null){ // Caso general: la pila no está vacía
            n = cima.valor;
            cima = cima.sig;
        }
        else{
            // Si la pila está vacía emitir un mensaje de error y retornar 0
            System.out.println("La pila está vacía");
        }
        return n;
    }
}

```

```

public void mostrar(){
    if(cima == null){
        System.out.println("La pila esta vacía");
    }else{
        while(cima != null){
            System.out.print("[ "+cima.valor+" ]-> ");
            cima = cima.sig;
        }
        System.out.print("\n");
    }
}

public static void main(String [] args){
    Pila c = new Pila();
    c.push(20); c.push(30);c.push(27);c.push(22);c.push(23);
    c.mostrar();
}
}

```

Una vez compilado correctamente el código fuente y creadas las clases Pila.class y Nodo.class, es preciso ir al directorio o carpeta donde tengamos dichas clases. Una vez allí se debe ejecutar lo siguiente:

```
javadoc Pila.java Nodo.java
```

Lo cual genera todo un conjunto de ficheros HTML que constituyen la documentación auto-generada a partir de los ficheros de clase proporcionados:

```

allclasses-frame.html
allclasses-frame.html
allclasses-noframe.html
constant-values.html
deprecated-list.html
help-doc.html
index-all.html
index.html
Nodo.html
overview-tree.html
package-frame.html
package-summary.html
package-tree.html
Pila.html

```

El fichero que enlaza los demás es index.html. Al abrirlo en un navegador podremos acceder a la documentación que ha generado Javadoc desde un punto de vista centralizado.

- 6.4** Suponga que durante una tarea de mantenimiento de un software que tiene cientos de miles de líneas de código, se encuentra con el siguiente código:

```

public static long SumaPares(int n){
    if (0 == n) //condición de salida
        return 0;
    else //caso general
        if (n\2 == 0)
            return n + SumaPares(n-2);
        else
            return SumaPares(n-1);
}

```

Proporcione una implementación iterativa de este mismo algoritmo y compare las diferencias en tiempo de ejecución entre ambos. A tenor de los datos obtenidos ¿estaría indicado sustituir la versión recursiva por su equivalente iterativa? Razone su respuesta.

Solución propuesta: En primer lugar hemos de estudiar el comportamiento del algoritmo. Se trata de un método que lleva a cabo la suma de los números pares entre 0 y un número positivo n que se pasa como argumento, incluyendo los extremos. Así, la ejecución de la función `SumaPares(10)` proporciona el resultado 30, es decir, la suma de los pares entre 0 y 10 ambos inclusive: $0 + 2 + 4 + 6 + 8 + 10 = 30$. Un equivalente iterativo podría ser el siguiente código:

```

public static long sumaParesIterativo(int n){
    int subTotal = 0;
    for (int i = 2; i <= n; i += 2)
        subTotal += i;
    return subTotal;
}

```

Implementando el siguiente código sencillo para comprobar la eficiencia de cada ejecución:

```

long inicio = System.nanoTime();
sumaPares(100);
System.out.println(System.nanoTime() - inicio);
inicio = System.nanoTime();
sumaParesIterativo(100);
System.out.println(System.nanoTime() - inicio);

```

obtenemos los siguientes datos: método recursivo, 29.892 microsegundos; método iterativo, 12.013 microsegundos. A la vista de esta diferencia tan significativa, un 60% más eficiente en la implementación iterativa, sería deseable la sustitución dado que se trata de un algoritmo sencillo (el tiempo estimado de codificación del nuevo algoritmo es muy pequeño) aunque no hubiera *a priori* una necesidad surgida de informes de bajo rendimiento del algoritmo original.

6.12.2 Actividades propuestas

- 6.1 Implemente un paquete en Java que incluya varias clases relacionadas con los algoritmos de ordenación de elementos en arrays. Una vez finalice y pruebe su implementación, utilice Javadoc para generar documentación para su paquete. Investigue el número de ficheros que se han generado como salida de la invocación por defecto y refine la ejecución de javadoc utilizando opciones que le permitan obtener sólo aquella documentación que le resulte de utilidad.
- 6.2 Instale en su computadora un generador automático de documentación de código de los que se distribuyen bajo licencia GPL que no sea Javadoc (por ejemplo ClassDoc, Doxygen o Natural Docs) y compare las diferencias entre la documentación obtenida al aplicarlo al paquete Java obtenido como resultado del ejercicio anterior y la documentación Javadoc obtenida en dicho ejercicio.
- 6.3 La serie de Fibonacci es una sucesión de números enteros descrita por primera vez en Europa en el siglo XII por Leonardo de Pisa, también conocido como Fibonacci (de ahí su nombre). Investigue su cálculo e implemente dos versiones de una función que calcule términos de la serie de Fibonacci, una iterativa y otra recursiva, comparando finalmente la diferencia de eficiencia entre ambas implementaciones.
- 6.4 Mejore el siguiente algoritmo haciendo uso de técnicas de legibilidad y, si son de aplicación, de reglas de afinación del código para mejorar su eficiencia:

```
public static void algoritmoMejorable(int n, int x){
    int i, a = x;
    double b = 100, c;

    for (i=1; i<=n; i++){
        b = 5 * i;
        while (a < 1000){
            b = 10 * i;
            a *= b;
        }
        System.out.print(a);
        System.out.print("-");
    };
    c = b + 1;
    System.out.print(b);
    System.out.print(c);
}
```

- 6.5 Para ver cómo la construcción de código poco legible afecta a las labores de mantenimiento, nada mejor que intentar comprender un código programado por otros. Acceda a SourceForge (<http://sourceforge.net/>), actualmente la mayor base de datos de desarrollo de código fuente abierto disponible en internet, y seleccione un proyecto al azar. Escoja un fichero cualquiera e intente averiguar, a partir de su código, lo que hace dicho programa. Luego acceda a la URL donde se publican los programas premiados en el concurso de escritura de software obscuro en C (<http://www.ioccc.org/>) y haga lo mismo. Compare los tiempos de comprensión y saque sus propias conclusiones.

- 6.6** Discuta en grupo cómo la utilización de un entorno de desarrollo frente a otro puede afectar a la construcción de software. Si en el grupo de discusión se conocen diferentes entornos de desarrollo para un mismo lenguaje, pongan por escrito los pros y los contras de utilizar cada uno como documento inicial de la discusión.
- 6.7** En 1980, C.A.R. Hoare recibió el premio Turing que otorga la ACM que, como dijimos en la introducción de este capítulo, es considerado por muchos como el premio Nobel de la informática. En «*The emperor's old clothes*», su discurso de agradecimiento en la ceremonia de entrega del premio –publicado posteriormente y que hoy se ha convertido ya en un clásico– Hoare relató sus experiencias en la construcción de programas, propuso ciertas recomendaciones y avanzó algunos de los problemas con que los desarrolladores se encontrarían en el futuro. De hecho, la cita que encabeza este capítulo pertenece a las conclusiones de este artículo. Léalo y discuta en grupo cuáles de las recomendaciones de Hoare son aún hoy día de aplicación en la construcción de software y cuáles se encuentran obsoletas. El discurso íntegro puede descargarse gratuitamente de <http://www.braithwaite-lee.com/opinions/p75-hoare.pdf>.
- 6.8** El código siguiente es uno de esos casos en que el uso de la sentencia GOTO puede considerarse útil. Originalmente incluido en el libro «*Code Complete*» (McConnell, 2004), se trata de una rutina escrita en Pascal que purga un conjunto de ficheros. El código mostrado busca cada uno de los ficheros a purgar, y una vez encontrado, lo abre, lo sobrescribe y finalmente lo borra, buscando en cada paso los posibles errores que pudieran haberse producido:

```

PROCEDURE PurgeFiles( var ErrorState: ERROR_CODE );
{ This routine purges a group of files }
var
  FileIndex:           Integer;
  FileHandle:          FILEHANDLE_T;
  FileList:            FILELIST_T;
  NumFilesToPurge:     Integer;

label
  END_PROC;

begin

  MakePurgeFileList( FileList, NumFilesToPurge );
  ErrorState := Success;
  FileIndex := 0;
  while ( FileIndex < NumFilesToPurge ) do
    begin
      FileIndex := FileIndex + 1;
      if not FindFile( FileList[ FileIndex ], FileHandle ) then
        begin
          ErrorState := FileFindError;
          goto END_PROC
        end;
      if not OpenFile( FileHandle ) then
        begin
          ErrorState := FileOpenError;
          goto END_PROC
        end;
      if not WriteFile( FileHandle, 'X', 1 ) then
        begin
          ErrorState := FileWriteError;
          goto END_PROC
        end;
      if not CloseFile( FileHandle ) then
        begin
          ErrorState := FileCloseError;
          goto END_PROC
        end;
    end;
  end;
end;

```

```

    end;
    if not OverwriteFile( FileHandle ) then
    begin
        ErrorState := FileOverwriteError;
        goto END_PROC
    end;
    if Erase( FileHandle ) then
    begin
        ErrorState := FileEraseError;
        goto END_PROC
    end
end; { while }

END_PROC:
    DeletePurgeFileList( FileList, NumFilesToPurge )

end; { PurgeFiles }

```

Busque una alternativa sin GOTO al código mostrado, aunque le adelantamos que una alternativa elegante (que no duplique el código que purga los ficheros) será tan complicada que seguramente estará de acuerdo en que en este caso en particular merece la pena el «coste» de utilizar GOTO en beneficio del futuro mantenimiento del software.

- 6.9 ¿Qué opinión le merece el siguiente comentario? ¿Qué argumentos utilizaría para rebatirlo?
«El tiempo que se pierde en pensar mejores nombres para los identificadores puede emplearse en cosas mejores. Si sólo es un trozo de código en el que queda claro por el contexto cuál es el significado de cada nombre ¿por qué perder el tiempo buscando los mejores nombres? Sinceramente creo que no es hacer buen uso del tiempo disponible el dedicarse a esto».
- 6.10 Según Jef Raskin (2005), los comentarios del código son más importantes que el propio código. Así, en un artículo titulado precisamente *«Comments are more important than code»*, afirma lo siguiente: *«no se debe creer a ningún programador, gestor o comercial que afirme que el código puede ser documentado automáticamente o autodocumentado. No es cierto. La buena documentación incluye una información previa de base que no puede derivarse a partir del código»*. Contraste estas afirmaciones con lo estudiado en el capítulo y exprese su propia opinión al respecto a la luz de las diversas consideraciones.

Pruebas

*La prueba de programas puede utilizarse para mostrar la presencia de errores,
pero nunca para demostrar su ausencia.*

— Edgar Dijkstra

7.1 El porqué de las pruebas

El Therac-25, una máquina de radioterapia fabricada por la compañía *Atomic Energy of Canada Limited* (AECL), fue responsable de uno de los más graves siniestros relacionados con el software de los que se tiene noticia. Entre 1985 y 1987, el Therac-25 causó al menos seis accidentes, cinco de ellos mortales, al suministrar a varios pacientes cantidades de radiación muy superiores a las pautadas por los facultativos. Para el mantenimiento de la seguridad, en el diseño del Therac-25 se había confiado completamente en el software, desciendiendo de los circuitos hardware de seguridad que sí se incluían en modelos anteriores de la misma compañía, como el Therac-20. Así, cuando la interfaz desde la que los facultativos establecían la dosis de radiación a administrar al paciente era operada muy rápido para pasar desde la posición de trabajo a una posición de descanso, el software no detectaba correctamente la nueva posición y continuaba suministrando dosis de radiación potencialmente letales a la persona en tratamiento.

Las investigaciones revelaron, entre otros muchos errores, problemas de sincronización entre el programa de control y la interfaz de usuario, errores de desbordamiento de un *flag* cuyo valor se establecía mediante una operación de incremento, y también problemas de reutilización inapropiada de código que había sido implementado específicamente para modelos más antiguos como el Therac-6.

Este es sólo un ejemplo de los catastróficos efectos que un software erróneo puede provocar. Para evitar este tipo de sucesos, la realización de pruebas sistemáticas y en profundidad es la única receta que permite minimizar los riesgos de un software que controla directamente procesos peligrosos. Sin embargo, la existencia de defectos no detectados durante el desarrollo en sistemas software no tan críticos, también implica riesgos de imagen, de responsabilidad legal y algunos otros que estudiaremos con detalle a lo largo del capítulo.

Podría argumentarse, no sin cierta razón, que casos tan extremos como el relatado no son frecuentes. Ni siquiera significativos. Sin embargo, sólo pensar en la remota posibilidad de que algo así ocurra y que sea consecuencia directa de un defecto en el software pone los pelos de punta. Como se verá en este capítulo, un riguroso proceso de prueba permite minimizar el riesgo, y si bien resulta costoso en términos económicos y de esfuerzo, se hace imprescindible para la obtención de software que cumpla unos mínimos parámetros de calidad. De hecho su coste resulta rentable a la vista del incremento en la calidad del producto final.

7.2 Objetivos

Este capítulo trata los aspectos esenciales de las pruebas de software y su terminología, describe las diferentes técnicas de prueba existentes y referencia las medidas relacionadas con las pruebas. Una vez asimilados los contenidos de este capítulo, el lector será capaz de:

- Reconocer y diferenciar los conceptos fundamentales relacionados con las pruebas de software.
- Comprender la necesidad de las pruebas como parte esencial del desarrollo de un sistema software.
- Conocer las distintas actividades que componen los procesos de prueba, desde la planificación de las pruebas hasta la evaluación de los resultados de las mismas.
- Saber diferenciar los distintos niveles de prueba en función del objeto de la misma.
- Conocer las diferentes técnicas de prueba de software.

7.3 Introducción

Tradicionalmente, tanto los ingenieros de prueba como los responsables de desarrollo han comprendido y valorado la importancia de las pruebas como parte integral de un proceso de producción que aspira a alcanzar determinados parámetros de calidad. No obstante, a menudo ambos han fracasado en el intento de transmitir este mensaje a sus superiores. Bien es cierto que cada vez son más los que creen que no sólo el éxito de un proyecto de desarrollo de software, sino también la continuidad de una línea de producción, se basa

m la obtención de un nivel adecuado de calidad, y que las actividades de prueba son sus máximos garantes. Como se verá más adelante, el riesgo de prescindir de las pruebas, o de realizarlas de manera inadecuada, resulta tan elevado que no es posible hablar de un proceso de Ingeniería del Software sin ellas.

Las pruebas de software son en realidad un elemento diferente dentro del proceso de desarrollo. Al contrario que el resto de actividades, su éxito radica en la detección de errores tanto en el propio proceso como en el software obtenido como resultado del mismo. Podría parecer extraño a primera vista el hecho de que encontrar errores en el software construido deba considerarse un éxito, pero la perspectiva del ingeniero de pruebas es distinta a la del resto de profesionales implicados en el desarrollo. Sin embargo, es en cierto modo similar a lo que ocurre en otras áreas, como por ejemplo las pruebas diagnósticas en el campo de la Medicina. Una prueba diagnóstica se considera positiva si detecta algo mal, si bien, es mucho peor para un paciente tener algo mal y no saberlo o tener conocimiento de que algo está mal sin conocer con exactitud qué.

Una prueba de software es todo proceso orientado a comprobar la calidad del software mediante la identificación de fallos en el mismo. La prueba implica necesariamente la ejecución del software

Dada su especial naturaleza, debe tenerse muy en cuenta que el éxito de las pruebas depende, en buena medida, de la actitud de colaboración que todo el equipo de desarrollo muestre respecto a las actividades de pruebas como garantía de calidad del software que se está produciendo. A este respecto, algunos autores abogan por fomentar un ambiente favorable respecto al descubrimiento de fallos, evitando en lo posible que nadie se sienta culpable por los errores que se vayan descubriendo.

Durante las pruebas se tiene presente el hecho de que los usuarios finales potencialmente ejecutarán todas las funcionalidades del sistema, lo que implica que el software será puesto a prueba inevitablemente, bien por los desarrolladores o bien por los usuarios finales. Por tanto, los errores que no sean detectados en las pruebas realizadas durante el desarrollo aparecerán cuando los usuarios finales utilicen el software, con el consiguiente impacto en la imagen y reputación del equipo de desarrollo, esfuerzo y coste asociado a la reparación, posibles implicaciones legales a consecuencia del mal funcionamiento del software, etc. Debe intentarse, y éste es uno de los objetivos fundamentales de las pruebas, que el porcentaje de fallos detectados por el usuario sea lo menor posible.

Probar un producto no es sino comparar su estado actual con el estado esperado de acuerdo a una especificación del mismo. Por extensión, y puesto que el software no es otra cosa que el producto resultante de un proceso de desarrollo, todo proceso que permite comprobar el comportamiento de un determinado software con el comportamiento que se espera del mismo según lo descrito en sus especificaciones podría ser catalogado como prueba de software, siendo los aspectos fundamentales a verificar en un software su corrección, com-

pleción y seguridad. En el resto del capítulo asumiremos que las pruebas implican necesariamente ejecución del software, dejando fuera de nuestro concepto de prueba otros tipos de verificaciones de calidad, tales como las revisiones —que vimos en la Sección 6.6.3 como parte de la construcción del software—, las cuales no implican ejecución y que a menudo se incluyen dentro de un concepto de prueba en sentido amplio.

Otro objetivo fundamental del capítulo es desmentir la errónea creencia —sin muy extendida— de que las pruebas son la última de las actividades del desarrollo, cuyo único objetivo es detectar fallos en el software. Es decir, que las pruebas se hacen al final, y que por tanto, durante el desarrollo no se planifican y preparan. Sin embargo, no debemos confundirnos. Los procesos de software actuales recomiendan comenzar a diseñar pruebas de aceptación al mismo tiempo que se realiza la especificación de requisitos. Este enfoque produce especificaciones mucho más detalladas por la obligación de determinar de antemano la manera exacta en que dichas especificaciones serán probadas. De este modo, y siguiendo la filosofía «es mejor evitar errores que arreglarlos», el proceso de prueba de un software es considerado en la actualidad una tarea compleja y constante a lo largo del proceso de desarrollo, cuyo objetivo último es la producción de software de calidad. Al estar presente durante todo el proceso de desarrollo y mantenimiento, el propio proceso de prueba, cuya planificación se lleva a cabo en las etapas iniciales de requisitos, se refina según avanza el desarrollo. Es importante remarcar que tanto la planificación de las pruebas como las propias actividades de diseño proporcionan a los diseñadores de software una información muy útil de cara a identificar debilidades potenciales en el software, tales como errores de diseño, contradicciones que pudieran haber pasado desapercibidas, ambigüedades en la documentación, etc.

Resulta por tanto esencial el diseño de un plan de pruebas y la integración de las mismas en el proceso de desarrollo, utilizando para ello un enfoque sistemático y dinámico. Sistemático porque la complejidad del software actual fuerza la planificación minuciosa de las pruebas, y dinámico porque obligatoriamente se ejecuta el software con entradas de datos preparadas al efecto, comúnmente denominadas *casos de prueba*.

Un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados, que han sido desarrollados para un objetivo particular como, por ejemplo, ejercitarse un camino concreto de un programa o verificar el cumplimiento de un determinado requisito

Como ejemplo introductorio, veamos un caso de prueba simple. Supongamos que, tras la programación de una calculadora sencilla, se desea probar la operación *sumar*. Si suponemos que la calculadora tiene 3 botones rotulados como '1', '+' y '=', y una caja de texto donde se muestran los resultados, las instrucciones para ejecutar la prueba serían:

1. Presione el botón etiquetado como '1'.
2. Presione el botón etiquetado como '+'.

3. Presione el botón etiquetado como '1'.
4. Presione el botón etiquetado como '='.
5. Observe y anote el resultado.

El resultado de la prueba será considerado correcto si el resultado obtenido es '2', e incorrecto en otro caso.

En el contexto de una determinada prueba, se considerará que un caso de prueba es bueno si tiene una alta probabilidad de descubrir un fallo no encontrado hasta entonces. Aunque el término *caso de prueba* a veces también se utiliza para referir la documentación en la que se describen las entradas, condiciones y salidas de un caso de prueba.

Durante el proceso de prueba se ha de preparar un número de casos de prueba lo suficientemente significativo como para cubrir todas las posibles causas de fallo. A esto último (cubrir absolutamente todas las posibles causas de fallo) se denomina *prueba completa* o *prueba exhaustiva*, y como se verá más adelante, resulta imposible de llevar a cabo en la práctica. Para realizar una prueba exhaustiva de la calculadora del ejemplo anterior, habría que someterla a todas las posibles variaciones de los datos de entrada, tanto si son válidos como si no lo son, lo cual es tanto como decir que hay que probar todas y cada una de las operaciones matemáticas que pudieran escribirse; sencillamente imposible.

Se denomina **prueba exhaustiva** o **prueba completa** a una prueba ideal que proporcionaría la seguridad de que se han comprobado todas y cada una de las posibles causas de fallo

En un producto tan complejo de fabricar como es el software, las posibilidades de que se hayan introducido errores durante el proceso de construcción son muy elevadas. Resulta obvio por tanto, que el subconjunto de casos de prueba escogidos nunca podrá proporcionar la fiabilidad que proporcionaría una prueba completa: estos casos son sólo una muestra de todos los casos particulares que podrían presentarse en el entorno real para el que ha sido construido el software. Descontando pues la realización de una prueba exhaustiva, es importante elegir correctamente los casos, teniendo en cuenta que los criterios de selección de las pruebas más apropiadas para cada conjunto de condiciones particulares no es un problema trivial.

El esfuerzo que requiere la realización de pruebas en el software debe tenerse muy en cuenta durante la planificación y gestión del desarrollo, y por supuesto en la estimación de los costes asociados al mismo. Un estudio del Departamento de Comercio de los Estados Unidos publicado en 2002, estimaba el coste de los errores en el software para la economía americana en 60.000 millones de dólares: el equivalente al 0,6% de su PIB. En términos generales, se calcula que entre el 30% y el 40% del esfuerzo total de un proyecto se dedica a actividades de prueba. Este porcentaje es mucho mayor cuando el software controla procesos críticos para la seguridad de las personas como puede ser el software médico, el de control de centrales nucleares, el control de vuelo en aeronaves, etc.

A continuación se definen los conceptos esenciales dentro de las actividades de prueba, se analizan los problemas asociados a la realización de las mismas, y se estudia el papel que desempeñan las pruebas en la tarea de evitar los riesgos que conlleva el software erróneo.

7.3.1 Conceptos fundamentales

En la bibliografía sobre pruebas de software se emplean términos específicos que es preciso definir adecuadamente antes de proseguir. En primer lugar es imprescindible distinguir los conceptos de fallo, defecto y error, cuyo uso equivocado puede dar lugar a malentendidos. Para nombrar la causa de un funcionamiento incorrecto del software, se emplean indistintamente los términos *defecto* o *error*, mientras que para referirse a los efectos no deseados observados en los servicios proporcionados por un software (muchas veces ocasionados por defectos en el mismo) se utiliza el término *fallo*.

Un **fallo** es un efecto indeseado observado en las funciones o prestaciones desempeñadas por un software

Diremos pues que las pruebas permiten descubrir *fallos*, que han sido causados por *defectos* en el software, los cuales deben subsanarse.

Se denomina **error** (o **defecto**) a una imperfección en el software que provoca un funcionamiento incorrecto del mismo

Debe tenerse en cuenta que no siempre puede determinarse de manera inequívoca qué error provoca un determinado fallo. Algunos autores dicen que es incluso más correcto no utilizar el término error y en su lugar referirse a entradas que causan fallos o a grupos de entradas de datos que hacen que el fallo aparezca.

Una de las confusiones más habituales consiste en no distinguir entre lo qué es exactamente *probar* un software y en qué medida no es lo mismo que *depurar* un software.

Se denomina **probar** un software al proceso de mostrar la presencia de un error en el mismo. **Depurar** un software consiste en descubrir en qué lugar exacto se encuentra un error y modificar el software para eliminar dicho error.

Tal y como se muestra en la Figura 7.1, las actividades de prueba se integran en un proceso iterativo donde se llevan a cabo, alternativamente, las dos acciones anteriores: prueba y depuración.



Figura 7.1: Carácter iterativo del proceso de prueba

El término *probable*, en su acepción de «cosa que puede verificarse o probarse», se entiende en el entorno de las pruebas de software como la capacidad de un módulo de software para ser sometido a pruebas que permitan determinar si contiene errores o no. Tiene mucha relación con la probabilidad, posiblemente cuantificable estadísticamente, de que los errores queden expuestos durante las pruebas, si es que el software es erróneo.

Otro concepto básico es el concepto de *oráculo*, definido en la guía SWEBOK de la siguiente manera:

Un **oráculo** es cualquier agente (humano o no) capaz de decidir si un programa se comporta correctamente durante una prueba y, por tanto, capaz de dictaminar si la prueba se ha superado o no

Otro concepto muy relacionado con las pruebas es la *trazabilidad* o seguimiento desde los requisitos. Toda funcionalidad del sistema, todo elemento medible, tiene su origen en algún apartado específico del documento de especificación de requisitos. Es deseable, por tanto, poder conectar los requisitos con las funcionalidades que les dan soporte, trazando una línea de unión a lo largo del proceso de desarrollo que permita planificar las pruebas de manera consistente.

Trazabilidad es la capacidad para interrelacionar dos o más entidades, inequívocamente identificables, dentro de una cadena de sucesos cronológicamente ordenada

Establecer esta línea de unión facilita las actividades de verificación y aprobación de las pruebas, y garantiza que los procedimientos de pruebas sean compatibles con las necesidades de los usuarios y/o clientes.

Las pruebas del software están relacionadas con la gestión de la calidad del software, la verificación y depuración, y por supuesto, con la programación. Sin embargo, no deben confundirse con tales actividades: las pruebas tienen entidad en sí mismas.

7.3.2 Limitaciones en la realización de pruebas

La realización de pruebas es un proceso importante que resulta crítico para el éxito de un proyecto de desarrollo. Debido en buena medida a la responsabilidad que las pruebas tienen sobre la corrección del producto final, quienes se enfrentan a un proceso de prueba encuentran numerosos problemas y limitaciones en la realización de sus tareas. Entre las más importantes podemos citar los siguientes:

- **Imposibilidad de hacer pruebas exhaustivas.** No es posible asegurar al 100% que un software esté libre de errores. Sólo una prueba exhaustiva permitiría corroborar que no hay errores. Como ya hemos visto, este tipo de pruebas son inviables en términos tanto económicos como humanos: sirva como ejemplo el de la calculadora simple de la Sección 7.3.
- **Limitaciones derivadas de la naturaleza de las pruebas.** Existen serias limitaciones a la hora de hacer pruebas derivadas de la propia naturaleza de las pruebas. Por ejemplo, el siguiente programa retorna un resultado erróneo al comparar algunas parejas de valores enteros:

```
int compararEnteros(int i, int j){
    /* Retorna 0 si i es igual a j, -1 si i es menor
     * que j, y +1 si j es mayor que i */
    if (i%35 == 0) return -1;
    if (i==j) return 0;
    if (i<j) return -1;
    else return 1;
}
```

Aunque este código funciona adecuadamente para muchas de las parejas de posibles valores, cuando *i* vale 0 el resultado es siempre «-1» independientemente del valor de *j*, lo cual es incorrecto.

Si bien errores como éste pueden resultar muy evidentes a la vista del código del programa (lo que más adelante denominaremos *pruebas de caja blanca*), son extremadamente difíciles de detectar en pruebas basadas en la introducción de valores de prueba y tras la observación de los resultados que no examinan el código fuente (pruebas comúnmente denominadas *de caja negra*). En este ejemplo, las pruebas de caja negra sólo detectarían el error cuando se introduce una pareja de números (*i,j*) donde *i* es un múltiplo de 35 y *j* un número menor que *i*, pero nunca en otros casos, pues el resto de parejas (*i,j*) no hacen funcionar incorrectamente al programa. Esto obliga a realizar pruebas de varios tipos sobre un mismo código para así detectar todos los posibles tipos de errores ocultos en el mismo.

- **Selección de los casos de prueba.** Un problema importante es decidir cuáles son los casos de prueba más adecuados en cada situación. Existen ciertos criterios de

selección de pruebas que pueden utilizarse para seleccionar casos de pruebas o para decidir si pueden darse por buenas las pruebas realizadas y por tanto, se puede terminar de hacer pruebas. No obstante, los criterios de selección dependen también de decisiones personales basadas en la intuición y la experiencia, lo que aumenta la incertidumbre acerca de su validez. Nos enfrentamos por tanto a un doble problema: seleccionar las entradas y ver cuál es la salida deseada, donde puede resultar de gran ayuda la utilización de oráculos.

- **Selección de los equipos de prueba.** Las pruebas deberían ser llevadas a cabo, en el caso ideal, por personas independientes, completamente imparciales con respecto al software a probar y, a ser posible, que no hubieran participado en su realización directamente. Sólo quienes no conocen *a priori* las interioridades de un módulo software pueden crear pruebas no condicionadas por su conocimiento sobre el mismo. Algunas compañías de software emplean grupos de ingenieros de prueba y de programadores totalmente distintos. Sin embargo, esto resulta ciertamente difícil en equipos de desarrollo pequeños. Comprobar si el grupo de casos de prueba es apropiado para aquello que se desea probar es una tarea complicada que requiere mucha experiencia.
- **Finalización de las pruebas.** La capacidad para decidir si se puede terminar de hacer pruebas o no, y cuándo, es algo difícil y para lo que se requiere experiencia, pues en buena medida condiciona la planificación de las pruebas. Se puede ver afectada, por ejemplo, por rotaciones en el equipo de pruebas.

Además de los problemas enumerados, existen muchas otras dificultades a las que los ingenieros del software se enfrentan cuando hacen pruebas. El denominado «problema de los caminos de control no alcanzables» (aquellos que no se ejecutan para ninguna entrada de datos), por ejemplo, es una de las cuestiones importantes con que se topan las técnicas de prueba basadas en código. Dado el gran número de técnicas de prueba que es posible realizar durante el desarrollo de un software, la enumeración exhaustiva de todos los problemas que podrían surgir queda fuera del alcance de este libro.

7.3.3 Las pruebas y el riesgo

La importancia de las pruebas es directamente proporcional a los catastróficos efectos que un software erróneo podría provocar. Sin embargo, si bien resultan obvios los riesgos de un software que controla directamente procesos peligrosos, existen otros riesgos no tan obvios como los derivados del empleo de datos generados por software para tomar decisiones, o del incorrecto almacenamiento y/o tratamiento de datos sensibles. Si bien nunca debe olvidarse la imposibilidad de producir software completamente libre de errores, un riguroso y exhaustivo proceso de prueba permite minimizar el riesgo.

Según la guía SWEBOK, las pruebas deben dirigirse en función de los riesgos y por tanto pueden verse, en cierto modo, como una estrategia de gestión del riesgo para una

organización de desarrollo. A continuación se enumeran los principales riesgos a que se enfrenta una organización al distribuir software insuficientemente probado:

- **Deterioro de su imagen.** Es un problema que a medio y largo plazo puede producir importantes pérdidas económicas, y si bien su impacto es difícilmente medible, puede dañar a una compañía más de lo que pueden hacerlo otras causas. Así por ejemplo, empresas como eBay que basan su negocio en la confianza mutua entre usuarios (en este caso las personas que venden y compran bienes por subasta a través de su Web), son muy sensibles a problemas de imagen como el provocado por una banda de delincuentes que, aprovechando un error en el software de verificación de acceso a las cuentas de usuarios, estafó en octubre de 2006 a varios internautas europeos. Los en apariencia leves fallos del software interno de un terminal de teléfono móvil es otro ejemplo de deterioro de imagen. El que ciertas características de un teléfono funcionen incorrectamente (por ejemplo, una errónea sincronización de la agenda que obliga a introducir manualmente los nuevos contactos, problemas en la tarjeta de memoria que ocasionan pérdida de las fotos, tonos y aplicaciones preinstaladas, etc.), pueden ser la causa de que cientos de clientes insatisfechos elijan teléfonos de la competencia en sus próximas adquisiciones. O lo que es peor, deteriore la imagen de la compañía con comentarios desfavorables en foros de opinión de consumidores.
- **Pérdidas económicas.** En septiembre de 1962, la sonda Mariner 1, un ingenio espacial valorado en 18 millones de dólares, tuvo que ser destruida durante uno de los primeros vuelos de prueba tras perder el rumbo por culpa de un fallo aparentemente pequeño en el software del sistema de guiado. El problema residía en que el programador había olvidado un símbolo en una fórmula al transcribir las órdenes desde el documento original manuscrito. La fórmula original (con el símbolo) contemplaba ciertas pequeñas variaciones de velocidad como posibles, e indicaba que debían tratarse como normales. Sin dicho símbolo, la fórmula cambiaba de modo que dichas variaciones eran tratadas como anómalas y corregidas con virajes, lo que terminó por sacar al cohete del rumbo normal y provocar su destrucción. Otro ejemplo muy conocido es el del Ariane 5. En junio de 1996, el primer vuelo de prueba del cohete Ariane 5 duró exactamente 39 segundos, ya que justo después del lanzamiento el cohete se salió de rumbo, se partió y explotó por culpa de un mal funcionamiento del software de control. Una operación mal programada de conversión de datos desde coma flotante en 64 bits a entero con signo de 16 bits indujo en el Ariane movimientos que suponían altas cargas aerodinámicas, provocando la separación de los propulsores de la etapa principal y disparando en última instancia el mecanismo de autodestrucción de la lanzadera. Errores como éstos ocasionan cuantiosas pérdidas, poniendo de relieve la falta de proporcionalidad entre el tamaño de los errores y los problemas que éstos pueden provocar.
- **Consecuencias legales.** En junio de 1990, millones de teléfonos dejaron de funcionar durante 9 horas en los Estados Unidos por culpa de una sentencia break situada en

el lugar incorrecto del programa de encaminamiento de llamadas de la compañía AT&T. Sesenta mil personas reclamaron haber sido afectadas, mientras se estima que aproximadamente 70 millones de llamadas no se completaron. Además de las cuantiosas perdidas económicas y de imagen, AT&T tuvo que enfrentarse a las consecuencias legales derivadas de poner en riesgo la seguridad pública y nacional. Este caso es un buen ejemplo de cómo los fallos de un software insuficientemente probado pueden tener importantes consecuencias legales para los responsables del desarrollo, lo que no sólo se traduce en costes económicos para hacer frente a indemnizaciones o reparación de costes materiales, sino también en pérdidas de licencias de explotación, incumplimientos de contratos, etc., cuyos costes totales son difícilmente cuantificables (por lo intangibles) pero muy graves. En aquellos casos en que las consecuencias de un mal funcionamiento del software pueda suponer riesgos graves para la vida de personas, para la seguridad nacional, o cuantiosas pérdidas materiales, las consecuencias legales pueden conllevar incluso responsabilidades penales.

La lista de casos descritos más arriba, que no pretende ser exhaustiva, tiene como objeto llamar la atención sobre el riesgo de realizar pruebas inadecuadas o insuficientes. En la siguiente sección se estudian las técnicas de prueba más usuales, cuyo objetivo es conseguir pruebas de software más sistemáticas y, en último término, obtener software menos proclive a incluir errores.

7.4 Técnicas de prueba

El objetivo de las pruebas es descubrir el máximo número de fallos en el software. Las pruebas añaden así valor al producto, pues parten de un programa con errores y señalan cuáles son, lo cual permite subsanarlos. Las técnicas de prueba son el medio que ayuda a conseguir los objetivos de las pruebas a través de su sistematización.

El objeto primordial de las técnicas que se analizan a continuación es sistematizar el proceso de prueba. Sin embargo, hay autores que consideran injustificada la búsqueda de un enfoque sistemático de las pruebas, y promueven un modelo de pruebas aleatorias, alegando que la probabilidad de descubrir errores cuando las pruebas se eligen completamente al azar es similar a cuando dichas pruebas se hacen siguiendo estrictos criterios de cobertura.

Tradicionalmente, las técnicas de prueba se han clasificado únicamente en dos categorías: pruebas de caja blanca y pruebas de caja negra. Estos términos se utilizan para describir la perspectiva desde la cual las personas responsables de la prueba se enfrentan a la misma. Así, se denominan *pruebas de caja blanca* a aquellas que están basadas en información acerca de cómo se ha diseñado o programado el software, mientras que aquellas donde los casos de prueba se basan solamente en el comportamiento de la entrada y salida de datos se denominan *pruebas de caja negra*. Las siguientes secciones explican con más detenimiento tanto esta clasificación como la exhaustiva clasificación de técnicas de prueba que propone la guía SWEBOK.

7.4.1 Pruebas de caja blanca y de caja negra

El concepto de **caja negra**, de uso frecuente en muchas disciplinas, se refiere al hecho de pensar en un mecanismo o sistema como si las entradas de datos para su desempeño se conocieran, pero su funcionamiento interno se ignorase. Debe tenerse en cuenta que la mayoría de nosotros utiliza a diario mecanismos según este enfoque. Así por ejemplo, conducimos un automóvil sin conocer detalles de su mecánica interna, simplemente proporcionándole ciertas entradas (presión sobre el acelerador, giro del volante, etc.) y observando el comportamiento que estas entradas provocan en el vehículo a lo largo del tiempo. Idéntico caso es el del uso de un televisor: no es necesario para ver nuestro programa favorito saber una sola palabra sobre los rudimentos del funcionamiento interno de un receptor de televisión. Una lavadora, un reproductor de DVD o una cámara de fotos son algunos otros ejemplos de los muchos que podríamos traer al caso.

Cuando se trata un mecanismo como si fuera una caja negra, lo importante es el comportamiento que las entradas provocan en el mismo, con atención especial a los resultados producidos o salidas. La caja es negra, es decir, sus contenidos no son visibles desde el exterior. En el campo del software, y más concretamente cuando se habla de pruebas de software, el concepto de caja negra se aplica a un módulo de programa en el que se introducen valores como entrada y se observan tanto las salidas como su comportamiento, pero no se presta atención al diseño del código del módulo ya que o bien se desconoce o bien se ignora deliberadamente. En este tipo de pruebas se parte de la especificación de requisitos del módulo con el objetivo de comprobar si lleva correctamente a cabo aquellas funciones que se esperan de él.

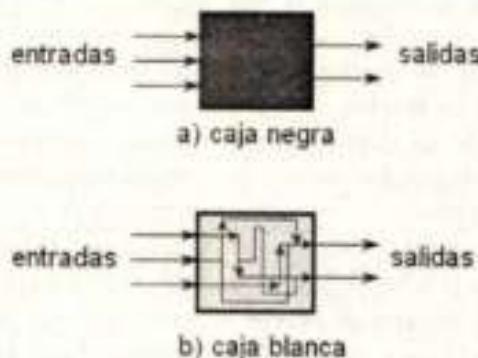


Figura 7.2: Pruebas de caja blanca y caja negra

Una de las mayores ventajas de estas técnicas es que en general resulta sencillo ejecutarlas, ya que el encargado de la prueba se limita a introducir un conjunto de datos de

entrada y a estudiar la salida. Así, su automatización más sencilla que la de otros tipos de prueba. En cuanto a las desventajas, deben reseñarse dos fundamentalmente: el alto coste de mantenimiento de las pruebas automatizadas (los cambios en la aplicación suelen influir en la validez de las pruebas de caja negra), y el riesgo de dejar parte del código sin probar si no se diseñan correctamente.

Clases de equivalencia

Un problema importante en las pruebas de caja negra es la incertidumbre sobre la cobertura de las mismas. Generalmente, la amplia casuística de los valores de entrada obliga a utilizar técnicas que permitan racionalizar el número de pruebas a realizar. La técnica de las clases de equivalencia permite reducir esta casuística de manera considerable. Las clases de equivalencia consisten en identificar un conjunto finito de datos que será suficiente para realizar las pruebas cubriendo aceptablemente los distintos casos posibles. Supongamos, por ejemplo, que se desean realizar pruebas de caja negra sobre el módulo de programa `CompararEnteros` definido en la Sección 7.3.2. El número de posibles combinaciones de enteros que podrían pasarse a esta función es elevadísimo (suponiendo enteros de 2 bytes estaríamos hablando de todas las posibles parejas de valores entre -32,768 y +32,767). Sin embargo, si fuéramos capaces de clasificar los enteros y encontrar un número reducido de clases cuyos elementos fueran equivalentes sería suficiente probar un solo elemento de cada clase, pues el resto de elementos produciría salidas similares a las de su representante de clase. Podríamos de este modo identificar tres clases de enteros: negativos, positivos y cero, para posteriormente probar sólo 9 parejas (positivo-positivo, positivo-negativo, positivo-cero, cero-positivo, etc.). Si la clasificación resulta suficiente para el módulo actual, el número de pruebas se reduce notablemente: de más de 10^9 a sólo 9.

Se considera **caja blanca** a todo componente para cuyo uso es necesario conocer su funcionamiento interno. En el contexto de la prueba de software, se habla de pruebas de caja blanca cuando la propia naturaleza de la prueba impone la observación del código del módulo a probar. Frecuentemente, el objetivo es probar todos los «caminos» posibles de ejecución en el código, lo cual se mide mediante la noción de cobertura de código.

La cobertura es una medida porcentual que indica la cantidad de código que ha sido probado.

La medida de cobertura de un software siempre será mejor cuanto más cerca del 100%, si bien una cobertura por encima del 95% es suficientemente completa como para plantearse la necesidad de seguir realizando pruebas. Las técnicas de cobertura de código (tales como la cobertura de rutas, de sentencias o la cobertura de condiciones) son técnicas de caja blanca que permiten sistematizar las pruebas de software.

Método del camino básico

Propuesto por Tom McCabe en 1976, se trata de un tipo de prueba de caja blanca donde se supone como cierto que, para todo código expresado en forma de grafo –donde los nodos representan decisiones y los arcos el flujo de control–, es posible definir un camino básico susceptible de ser sometido a pruebas rigurosas. Si se verifica que dicho camino no contiene errores, se puede asegurar que los caminos expresados en función de caminos básicos sin errores también estarán libres de ellos. El método tiene cuatro pasos:

1. Elaborar el grafo de flujo asociado al programa a partir del diseño (o del código fuente) del mismo.
2. Calcular la complejidad ciclomática $v(g)$, estableciendo el número de caminos independientes. Se denomina *independiente* a cualquier camino que introduce al menos una sentencia de procesamiento (o condición) no considerada anteriormente.
3. Seleccionar un conjunto de caminos básicos.
4. Generar casos de prueba para cada uno de los caminos del paso anterior.

Aunque existen otras formas de calcular la complejidad ciclomática (ver Capítulo 3), ésta suele calcularse como el número de regiones distintas del grafo de flujo. Este valor se corresponde con el número de caminos linealmente independientes de la estructura de control del programa, lo cual permite preparar un conjunto de casos de prueba que fuerce la ejecución de cada camino del conjunto básico. Cuando se han completado todos los casos de prueba se puede afirmar que todas las instrucciones han sido ejecutadas al menos una vez. *Ejemplo:* En la Figura 7.3 se muestra el grafo de un programa cuya complejidad ciclomática es 3 (según el método de cálculo que propone contar el número de regiones). Para probar todas las sentencias al menos una vez, habrá que ejercitarse cada camino independiente del programa. Para ello, habrá que establecer los siguientes caminos a probar:

- C1 : 1, 4, 8
- C2 : 1, 4, 7, 3, 6, 9
- C3 : 2, 5, 9

Finalmente, deben diseñarse casos de prueba que aseguren que todos estos caminos sean ejecutados, seleccionando las entradas adecuadas para ello.

Las principales ventajas de las técnicas de caja blanca son las siguientes:

- Eficiencia al tratar pruebas automatizadas, ya que es posible definir pruebas unitarias que aislen partes del código y probarlas independientemente, acelerando así el procesamiento de las colecciones de pruebas.
- Mayor facilidad a la hora de depurar los programas en busca de errores.
- Fundamentales en el denominado desarrollo guiado por pruebas (una de las técnicas características de la programación extrema), el cual depende en gran medida del conocimiento que se tiene de la implementación.

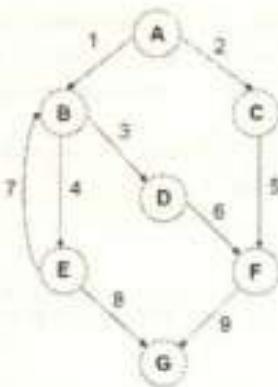


Figura 7.3: Cálculo del camino básico

En cuanto a las desventajas, enumeraremos las más importantes:

- Resulta difícil utilizar pruebas de caja blanca para validar requisitos, pues estas técnicas se centran en cómo está hecho el software, pero no en si hace lo que debería.
- No resultan adecuadas para comprobar la robustez del software ante comportamientos imprevistos, como por ejemplo, entradas no esperadas por parte de los usuarios.
- Los encargados de las pruebas deben tener un nivel de formación y experiencia más elevado que sus homólogos de las pruebas de caja negra.

La utilización de un tipo u otro de pruebas –caja blanca o caja negra– depende de cómo se haya definido el proceso de prueba (algo que se trata detalladamente en la Sección 7.8). En la Tabla 7.1 se resumen las diferencias entre ambos tipos de pruebas.

Tabla 7.1: Diferencias entre pruebas de caja blanca y pruebas de caja negra

Caja blanca	Caja negra
Es necesario conocer el código	No importa cómo esté escrito el código
No permiten validar requisitos	Adecuadas para validar requisitos
No sirven para determinar la robustez	Permiten comprobar la robustez ante imprevistos
Diseño y ejecución complejo	Más fáciles de llevar a cabo
Sin redundancia	Riesgo de dejar código sin probar
Efficientes con pruebas automatizadas	Complejo mantenimiento de las pruebas automatizadas
La prueba exhaustiva consiste en probar todos los posibles caminos de ejecución	La prueba exhaustiva consiste en probar todas las posibles combinaciones de entradas

En el aspecto práctico, y reconociendo de antemano que podría proponerse otro orden según las circunstancias, proponemos la siguiente secuencia de realización de las pruebas como guía flexible que puede ser adaptada según las necesidades:

1. Identificación de clases de equivalencia. Cuando los rangos de valores en las clases identificadas sean muy amplios, se recomienda incluir pruebas de caja negra adicionales que contemplen valores «normales» (es decir, valores alejados de los denominados valores límite).
2. Realización de pruebas de caja negra para analizar los valores límite, tanto de entrada como de salida.
3. Haciendo uso de la experiencia de los ingenieros de pruebas en situaciones similares, añadir pruebas de conjectura de errores para descubrir errores en aquellos puntos en que se presume que pueden haberse cometido fallos o en los que sea habitual descubrir errores.
4. Medición de la cobertura lograda con las fases anteriores. Si no se ha logrado la cobertura deseada, deberán añadirse más pruebas de caja blanca.

Un interesante concepto, a mitad de camino de los anteriores, es el de **caja gris**. Se trata de un tipo especial de pruebas de caja negra en las que resulta imprescindible conocer parte del código para poder realizar las pruebas. La idea en que se basan es que si la persona que realiza la prueba tiene cierto conocimiento sobre el funcionamiento interno del módulo que se prueba, podrá probarlo mejor desde el exterior. Eso sí, a diferencia de las pruebas de caja blanca, en las de caja gris no se intenta alcanzar un alto nivel de cobertura para probar todos los caminos posibles. Por el contrario, se trata simplemente de realizar pruebas de caja negra «informada» por el conocimiento sobre cómo operan e interactúan los componentes internos.

Así por ejemplo, al probar un método de una clase Java puede ser necesario saber si hay llamadas a métodos de otras clases, pues si ése fuera el caso la prueba en curso no podría considerarse estrictamente unitaria al ser imprescindible para su correcto funcionamiento que los métodos de las dos clases se comporten según lo previsto.

```
public class Contenedor{
    public boolean almacenarElementoNoRepetido(int x, Pila p){
        int cima;
        cima = p.top();
        if (cima != x)
            p.push(x);
    }
}
```

En este tipo de pruebas, se diseña un número limitado de casos de prueba para analizar el funcionamiento interno de parte del software que se está probando, y en el resto de la prueba

se toma un enfoque de caja negra, probando distintas entradas y comprobando las salidas. En el ejemplo anterior, los casos de prueba estarían dirigidos a comprobar las llamadas a métodos de otras clases (las llamadas `p.top()` y `p.push()` particularmente). Este tipo de pruebas es muy común cuando se utilizan bases de datos, pues permiten al ingeniero de pruebas configurar el estado de los datos antes de probar, por ejemplo, una consulta sobre la base de datos desde un enfoque de caja negra.

7.4.2 Clasificación exhaustiva de las técnicas de prueba

El criterio según el cual se clasifican las técnicas de prueba en pruebas de caja blanca y pruebas de caja negra es poco específico, ya que agrupa muchas técnicas diferentes dentro de una misma categoría. Por ello, estudiaremos además la clasificación propuesta por la guía SWEBOK y que se muestra en la Tabla 7.2, más completa y rigurosa. La diferencia esencial entre las distintas técnicas que se presentan en dicha tabla reside en el modo en que se escoge el conjunto de pruebas. Es éste un aspecto esencial, pues criterios de selección distintos pueden conducir a grados de efectividad muy diferentes.

Tabla 7.2: Clasificación de las técnicas de prueba propuesta por la guía SWEBOK

Base de clasificación	Técnicas específicas
Técnicas basadas en la intuición y experiencia	Pruebas <i>ad hoc</i>
	Pruebas por exploración
Técnicas basadas en la especificación	Particiones de equivalencia
	Ánalisis de valores límite
	Pruebas de robustez
	Tablas de decisión
	Pruebas basadas en máquinas de estado finito
	Pruebas basadas en especificaciones formales
	Pruebas aleatorias
Técnicas basadas en el código	Pruebas de cobertura basadas en flujo de control
	Pruebas de cobertura basadas en flujo de datos
Técnicas basadas en errores	Pruebas de conjetaura de errores
	Pruebas de mutación
Técnicas estadísticas	Pruebas de sala limpia
Técnicas basadas en el uso	Pruebas de perfil operativo
	Pruebas de fiabilidad del software

En la práctica, los responsables de las pruebas utilizan su experiencia en el diseño de pruebas para identificar el criterio de selección más apropiado en cada caso, lo cual no es en absoluto una tarea sencilla. Así, es importante señalar que no todo el mundo usa estas técnicas con la misma frecuencia. Por ello, la siguiente lista incluye las técnicas que todo ingeniero del software debería conocer, lo cual no significa que deban utilizarse todas ellas en todas las situaciones.

Técnicas basadas en la intuición y experiencia: se basan en la habilidad e intuición de los ingenieros del software, y en su conocimiento de situaciones similares. Entre ellas es posible citar las pruebas *ad hoc* y las pruebas por exploración.

- Las pruebas *ad hoc* son pruebas a la medida de la aplicación o del módulo a probar, y pueden resultar útiles cuando se desean identificar casos de prueba que no se pueden extraer fácilmente mediante técnicas sistemáticas.
- Por su parte, las *pruebas por exploración* no forman parte de un plan de pruebas preestablecido. Por el contrario, se diseñan, ejecutan y modifican dinámicamente a partir de la experiencia del ingeniero del software. Por ejemplo, por el comportamiento del producto durante otras pruebas, por su familiaridad con la aplicación, o por el riesgo asociado con un tipo de aplicaciones en particular.

Técnicas basadas en la especificación: son variadas y agrupan un gran número de técnicas muy utilizadas:

- La técnica de *particiones de equivalencia* consiste en dividir el dominio de los datos de entrada en clases de equivalencia de acuerdo con una determinada relación entre sus elementos. Esta técnica logra reducir un rango de valores amplio a un pequeño conjunto de clases de equivalencia.
- En el *análisis de valores límite*, los casos de prueba se seleccionan cerca de los límites del dominio de las variables de la entrada de datos, ya que habitualmente la mayoría de los errores se concentran cerca de los valores extremos.
- Las pruebas de robustez, pueden considerarse una variante de esta técnica en que se seleccionan casos de prueba fuera del dominio de los datos de entrada, con el objetivo de comprobar la robustez del programa ante entradas de datos erróneas o inesperadas.
- Las *tablas de decisión* representan relaciones lógicas entre condiciones (generalmente entradas) y acciones (mayoritariamente salidas), derivando casos de prueba a partir de combinaciones de condiciones y posibles acciones.
- Otras técnicas que merece la pena mencionar son las *basadas en máquinas de estado finito*, las *basadas en las especificaciones formales* y las *pruebas aleatorias*.

Técnicas basadas en el código: analizan el código de un programa o módulo de programa y realizan pruebas basadas en un estudio de la cobertura del mismo. Es decir, ante la imposibilidad de probar absolutamente todas las zonas del código (todas las iteraciones posibles de un bucle, todas las ramas de una sentencia selectiva, etc.) definen una medida de la cantidad de código que realmente se ha probado y la probabilidad de que con ese porcentaje del código probado se detecten todos los posibles fallos del módulo. A menudo utilizan una representación de la estructura de control para la prueba, bien un gráfico de flujo, bien un gráfico de llamadas entre módulos, u otras formas de representación. Las dos técnicas más relevantes se diferencian en cómo establecen el criterio de cobertura. Así, el criterio de cobertura puede estar basado en el flujo de control o en el flujo de datos.

Técnicas basadas en errores: generan casos de prueba especialmente orientados a descubrir categorías de errores probables o predefinidos.

- La técnica de *conjetura de errores* consiste en que los casos de prueba son diseñados por ingenieros del software conjeturando cuáles serán los errores más probables en un programa determinado. Lógicamente, esto no se realiza de modo casual ni aleatorio, sino basándose en su experiencia en programas similares.
- Las *pruebas por mutación* es otra técnica que se basa en los errores. Se denomina *mutante* a una versión ligeramente modificada del programa en pruebas, que sólo se diferencia de aquél en un ligero cambio sintáctico. Cada caso de prueba se aplica tanto al original como a los mutantes generados: si una prueba consigue identificar la diferencia entre el programa y el mutante, se dice que se ha «acabado» con este último. El efecto de acoplamiento, base teórica de las pruebas de mutación, dictamina que buscando errores sintácticos simples se deberían encontrar otros más complejos.

Técnicas de sala limpia: se orientan a un control estadístico del número de fallos. Estas técnicas basan la prueba de software en principios según los cuales la prueba se ve como un experimento estadístico. Se toma un subconjunto significativo de todos los posibles usos del software, y se emplea para crear casos de prueba. Los resultados obtenidos para dichos casos se consideran una muestra estadística de la que es posible inferir conclusiones acerca de la población completa, esto es, sobre el software en su conjunto.

Técnicas basadas en el uso: pueden dividirse en *pruebas de perfil operativo* y *pruebas de fiabilidad del software*.

- Las pruebas de fiabilidad del software, se llevan a cabo a lo largo de todo el proceso de desarrollo, y están diseñadas y guiadas por los objetivos de fiabilidad, el uso esperado y lo críticas que sean las distintas funcionalidades.
- Las pruebas de perfil operativo son pruebas de fiabilidad en las que el entorno de pruebas debe reproducir el entorno operativo del software lo más fielmente posible.

7.5 Niveles de prueba

Las pruebas se llevan a cabo en diferentes niveles; comenzaremos por tanto estableciendo qué es un **nivel**. Aunque es posible encontrar clasificaciones diferentes en la literatura, en este libro clasificaremos los niveles de prueba atendiendo a dos criterios fundamentales: el objeto de las pruebas y el objetivo que se persigue con ellas. Según el objeto de la prueba se distinguirán tres niveles: pruebas de unidad, pruebas de integración y pruebas del sistema. La Figura 7.4 ilustra cómo las pruebas que se realizan a diferentes niveles van actuando sobre el software en desarrollo y facilitando su refinamiento y puesta a punto.



Figura 7.4: Organización de las pruebas

Según el objetivo que persiguen, podremos encontrarnos con pruebas de aceptación, pruebas de instalación, pruebas alfa y beta, pruebas de conformidad y pruebas de regresión, entre otras. A continuación se explica más en detalle en qué consiste cada uno de los tipos de prueba enumerados, para luego detenernos y estudiar más en detalle las pruebas unitarias.

7.5.1 Pruebas según su objeto

Como ya se ha dicho, las pruebas de software se realizan a diferentes niveles, que a menudo se definen en función del objeto de la prueba. Se entiende por *objeto de la prueba* aquello que se está probando, y que puede ser un módulo de programa, un conjunto de módulos relacionados o un sistema software en su conjunto.

Pruebas de unidad

Una **prueba unitaria** es la prueba de un módulo concreto, dentro de un software que incluye muchos otros módulos, con el objeto de verificar que el funcionamiento aislado de dicho módulo cumple la función para la que ha sido construido. El objetivo final del conjunto de las pruebas unitarias que se llevan a cabo es comprobar que las partes individuales que conforman el software son correctas.

En este contexto, se entiende *módulo* como aquella unidad de código que sirve como bloque de construcción para la estructura física de un sistema. Este concepto de módulo se debe interpretar en términos del lenguaje de programación que se utiliza, y así por ejemplo, podemos considerar que una clase Java o C++ es un módulo. En lenguajes más antiguos, se pueden considerar módulos los subprogramas o los conjuntos de subprogramas relacionados, si bien generalmente se identifica como módulo toda aquella unidad de programa que se puede compilar de manera separada.

Una prueba de unidad es un componente software específicamente creado para verificar el funcionamiento de un componente del sistema en construcción

En los métodos tradicionales de desarrollo de software, las pruebas unitarias se llevan a cabo cuando los desarrolladores consideran su código preparado para someterse a las pruebas del equipo responsable de las mismas. Sin embargo, métodos más modernos donde el desarrollo se guía por las pruebas (como la programación extrema), propugnan un modelo en el que el programador realiza personalmente las pruebas de unidad de su propio código, implementando esas pruebas antes incluso de escribir el código a probar. Debido pues a la importancia que las pruebas unitarias tienen actualmente como parte integral del proceso de construcción del software, la Sección 7.6 se dedica íntegramente al estudio detallado de la realización y diseño de las mismas.

Pruebas de integración

Una vez que el resultado de las pruebas unitarias proporciona a los desarrolladores la seguridad de que los componentes individuales del software funcionan correctamente por separado, es necesario comprobar si también funcionan correctamente en conjunción con otros. Una **prueba de integración** es el proceso que permite verificar si un módulo funciona adecuadamente cuando trabaja conjuntamente con otros, por lo que es necesario comprobar si existen defectos en las interfaces entre dicho módulo y el resto. Se trata de un proceso incremental en el que se van paulatinamente involucrando, durante todo el tiempo que dura la construcción del sistema, un número creciente de módulos, para terminar probando el sistema como conjunto completo.

Existen diferentes estrategias de integración. La Tabla 7.3 muestra un resumen de las ventajas y desventajas asociadas a los dos enfoques puros de pruebas de integración: ascendente y descendente. Estas estrategias, que podríamos denominar *clásicas* –integración descendente o ascendente– se usan generalmente con software claramente estructurado de modo jerárquico:

- Las pruebas de **integración ascendente** integran primero los componentes de infraestructura y añaden componentes funcionales progresivamente. Es decir, se prueban primero los componentes de menor nivel, luego los que invocan o utilizan de

algún modo los anteriores, y así sucesivamente. Este proceso se repite hasta alcanzar el nivel más alto: aquel en el cual se ven involucrados todos los componentes (ver Figura 7.5). Para dirigir el proceso, se crean programas que invocan a un componente en particular y le pasan un caso de prueba. A estos componentes se les denomina *conductores* de la prueba (*drivers*) para un componente.

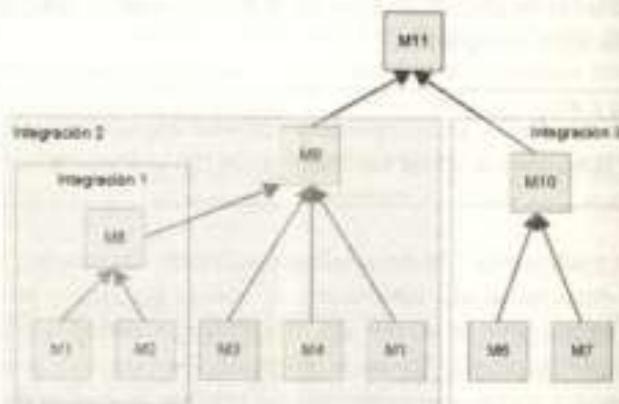


Figura 7.5: Pruebas de integración ascendente

- En las pruebas de **integración descendente** se desarrolla un esqueleto del sistema al que se van progresivamente añadiendo componentes. El proceso consiste en probar primero el componente de más alto nivel, después aquellos que son utilizados por dicho componente, y así sucesivamente con el resto de los componentes hasta llegar a los de más bajo nivel. En este tipo de pruebas existe una particularidad, y es que cuando un componente que se está probando invoca a otro que aún no ha sido probado, debe crearse lo que se denomina un procedimiento *tonto* (*stub*), una implementación temporal de una parte del programa con el único propósito de poder realizar la prueba. En la Figura 7.6 se muestra el proceso de integración descendente.

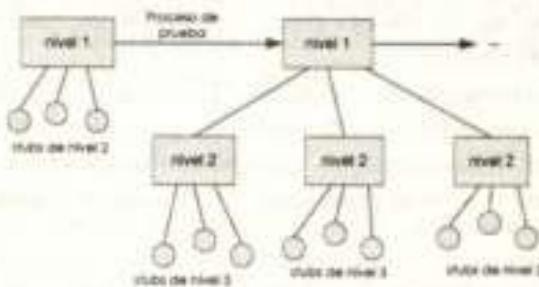


Figura 7.6: Pruebas de integración descendente

- Una estrategia combinada, denominada **integración en sandwich**, mezcla ideas del enfoque descendente con otras del ascendente. En este tipo de pruebas, el sistema se representa como un sandwich con 3 capas: la capa central es aquella que se desea probar, y las otras dos son la capa inmediatamente superior y la inmediatamente inferior. Se utiliza un enfoque ascendente para probar la capa central asumiendo que la inferior ya ha sido implementada, y uno descendente para probar la capa central desde la superior. Una de sus mayores ventajas es que una vez probado un módulo ascendentemente no será necesario programar un *stub* para probar el componente inmediatamente superior, pues el componente real ya ha sido probado y está disponible.

Tabla 7.3: Ventajas (V) y desventajas (D) de los dos enfoques puros de integración

Integración ascendente	Integración descendente
(D) Los componentes más importantes son los últimos en ser probados, lo cual a menudo postpone la detección de fallos importantes.	(V) Los casos de prueba se pueden definir en función de las funcionalidades del sistema.
(D) El flujo de control es dependiente de los elementos de más alto nivel, que se prueban los últimos, lo cual dificulta las pruebas en sistemas muy dependientes de una estricta secuencia temporal.	(V) Las cuestiones importantes de funcionalidad se prueban al principio, lo cual permite la detección temprana de defectos.
(V) Son especialmente adecuadas en la programación orientada a objetos, pues en este modelo de programación, los objetos se prueban primero independientemente y luego en conjunción con otros.	(D) Puede ser necesario programar muchos stubs, lo que constituye un importante esfuerzo de desarrollo de código que luego no será utilizado.

Las estrategias modernas de integración están dirigidas por la arquitectura, lo que supone integrar los componentes de software o subsistemas basándose en los enlaces entre módulos que dan soporte a una determinada funcionalidad. Así por ejemplo, en un sistema de gestión administrativa de una universidad todas las clases que intervienen en la impresión de un certificado de calificaciones para un alumno se probarían conjuntamente, pero no todos los métodos de todas las clases, sino únicamente aquellos métodos directamente involucrados en esta funcionalidad (es decir, aquellos que se invocan unos a otros para finalmente obtener el certificado impreso).

Las pruebas de integración son una actividad continua que se repite para cada una de las funcionalidades identificadas, cubriendo así todos los requisitos del usuario. Excepto si se trata de un desarrollo pequeño, siempre es preferible emplear estrategias de integración sistemáticas e incrementales que no probar todos los componentes juntos al final, en un caótico proceso denominado frecuentemente *pruebas en big-bang*.

En la Figura 7.7 se representa esquemáticamente un programa que puede ser sometido a diferentes estrategias de integración. La aplicación de cada tipo de prueba sería el siguiente:

- Integración descendente: probar primero el componente A. Escribir procedimientos *tontos (stubs)* para B, C y D. Una vez eliminados todos los errores de A, pasar a probar B, escribiendo *stubs* para E, F, G, H, etc.
- Integración ascendente: probar E independientemente. Escribir conductores para proporcionar a E la entrada que requiere de B. Probar F independientemente y escribir conductores para proporcionar a F la entrada que requiere de B, etc. Una vez probados E, F, G y H, se puede probar el subsistema B utilizando un conductor para A.
- *Big-bang*: probar todos los componentes (A, B, C, D, E, F, G, H, I, J, K y M) juntos.
- Integración en sandwich: combinación de las estrategias ascendente y descendente.

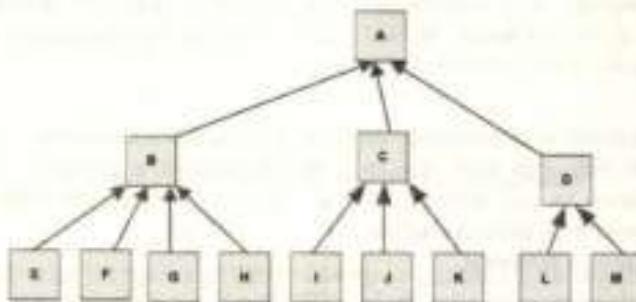


Figura 7.7: Ejemplo de pruebas de integración

Pruebas del sistema

Si se han llevado a cabo correctamente las pruebas de unidad y de integración, la mayoría de los fallos funcionales deberían haber sido identificados ya. El objetivo de dichas pruebas era asegurar que el código producido implementaba fielmente y sin defectos el diseño del sistema. Sin embargo, es necesario comprobar el funcionamiento del sistema construido antes de considerar que está listo para ser puesto en producción.

Las pruebas del sistema podrían englobarse en la categoría de pruebas de caja negra, puesto que su objetivo es comprobar la funcionalidad del sistema completo, integrado, con el objetivo primordial de comprobar si hace lo que el cliente desea. En general, puede considerarse prueba del sistema cualquier demostración o análisis que permita comprobar si el sistema en su conjunto se adecúa a las especificaciones de requisitos. Desde la perspectiva de quienes participan en ellas, son muy distintas a las pruebas de unidad o de integración, pues debe trabajarse con el equipo de desarrollo prácticamente al completo en lugar de sólo con las personas involucradas en el desarrollo del módulo o módulos implicados.

Los principales puntos de comprobación durante las pruebas del sistema son la seguridad, la velocidad, la exactitud y la fiabilidad. También las interconexiones externas con otras aplicaciones, utilidades, dispositivos hardware o con el sistema operativo, deberían ser evaluadas en este nivel. No se trata de comprobaciones a realizar únicamente sobre el software construido, sino de pruebas que deben enfocarse desde varias perspectivas. El proceso a menudo sigue un esquema como el que se representan en la Figura 7.8:

- 1. Pruebas de funcionalidad y operativa.** Se trata de pruebas de caja negra sobre las diferentes funcionalidades del sistema integrado, con el objetivo de determinar si las funciones descritas en la especificación de requisitos realmente se cumplen y si funcionan adecuadamente.
- 2. Pruebas de rendimiento.** Se comprueba la compleción de los requisitos no funcionales, midiendo el rendimiento y comparándolo con el valor esperado (y expresado en los requisitos). Al ser una categoría muy amplia, podemos encontrarnos con *pruebas de desgaste*, que evalúan cómo se comporta el sistema cuando se alcanzan o superan sus límites de operación (por ejemplo, en número de usuarios o dispositivos conectados), *pruebas de volumen*, que comprueban cómo se comporta el sistema ante elevados volúmenes de datos, *pruebas de configuración*, que comprueban si existen diferencias en el comportamiento del sistema cuando se ejecuta en diferentes plataformas, pruebas de compatibilidad, de regresión, de seguridad, de calidad, de recuperación, etc. Si las pruebas de rendimiento no resultan satisfactorias, el uso de técnicas de afinación (Sección 6.6.1) ayudará a alcanzar los objetivos.

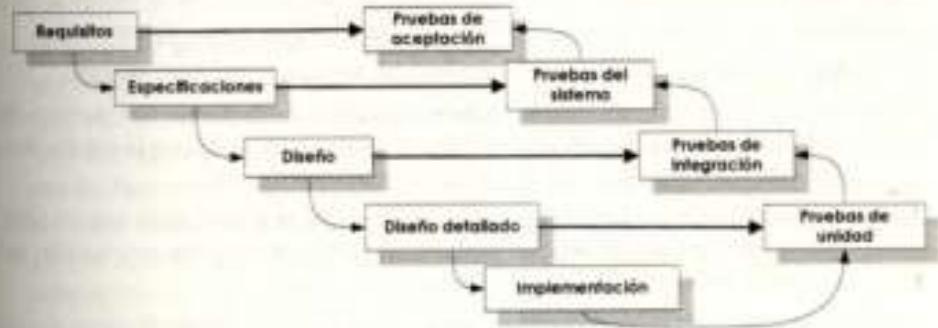


Figura 7.8: Diferentes niveles a los que se realizan las pruebas

- 3. Pruebas de aceptación.** Se realizan sobre el software completado conjuntamente con el usuario, que acepta formalmente el producto si se prueba el correcto funcionamiento del mismo. En este punto, los desarrolladores tienen el convencimiento de que el sistema construido cumple los requisitos especificados al inicio del proceso de desarrollo, por lo que son los usuarios los que diseñan los casos de prueba, ya que son ellos quienes en último término dirigen este tipo de pruebas.

Algunas de las técnicas de prueba de aceptación más comunes son las *pruebas de punto de referencia* (*benchmark tests*), donde el usuario establece un conjunto de puntos de referencia para el rendimiento de ciertas funcionalidades del sistema que sirven para evaluar su rendimiento en comparación con dichos niveles; las *pruebas piloto*, que consisten en hacer una instalación experimental del software y someterlo al trabajo diario al que se sometería una vez en producción, las *pruebas alfa* y *beta* y las *pruebas en paralelo*, que se llevan a cabo siempre que se construye un sistema que va a remplazar a uno ya en funcionamiento.

Automatización de las pruebas de rendimiento

Debido a los cambios de configuración en el entorno de ejecución o a pequeños ajustes hechos al código, pueden detectarse nuevos problemas de rendimiento en componentes o sistemas software que ya habían superado un proceso de prueba de rendimiento. Para asegurarse de que un componente software se ejecuta siempre de acuerdo a las exigencias de rendimiento especificadas resulta muy útil automatizar las pruebas de rendimiento, haciendo uso de herramientas de automatización como jUnitPerf. Estas herramientas permiten asegurar que las pruebas de rendimiento se realizan continuamente: tras cualquier modificación en el código se comprueba su rendimiento para la nueva situación.

jUnitPerf es un *framework* formado por un conjunto de extensiones de JUnit para pruebas de rendimiento, que permite medir el volumen y el tiempo de ejecución y contrastarlo con los valores esperados según la especificación. Las pruebas con jUnitPerf son transparentes, lo que significa que pueden añadirse a las pruebas funcionales codificadas con JUnit y dichas pruebas continuarán ejecutándose correctamente. Especificando *a priori* un tiempo de respuesta como medida de éxito, las pruebas de jUnitPerf revelan si el código se ejecuta lo suficientemente rápido. Si no es así, la prueba falla. El proceso es el siguiente:

- (a) Llevar a cabo una estimación del rendimiento (tiempo y volumen) deseado.
- (b) Escribir una prueba jUnitPerf con las estimaciones como límites de tolerancia de la prueba. Si se ejecuta dicha prueba sobre el software sin modificar, la prueba debería fallar, lo que indica que ha sido correctamente escrita.
- (c) Corregir los fallos identificados y volver a ejecutar la prueba hasta que no falle, lo que indicará que el software cumple los requisitos de rendimiento establecidos en los límites de tolerancia.

La automatización mediante este tipo de herramientas facilita el trabajo de mantenimiento, ya que evitan el diseño de nuevas pruebas de rendimiento. Así, si en el futuro se modifica el código por el motivo que sea, sólo habrá que volver a ejecutar la prueba jUnitPerf para asegurar que el código sigue cumpliendo los requisitos de rendimiento establecidos. También resulta muy útil el hecho de que los límites de tolerancia puedan modificarse, por ejemplo, si se establecen requisitos más exigentes, lo cual es una tarea tan sencilla como modificar el valor de una constante en el código de la prueba.

4. **Pruebas de instalación.** Se realizan para asegurar tanto el funcionamiento correcto de las opciones y funcionalidades de la instalación como para asegurar que todos los componentes necesarios se instalaron correctamente. Es importante probar las

diferentes opciones de instalación, así como el buen funcionamiento de las opciones de desinstalación, entre otros.

Tenga en cuenta que, en paralelo a este proceso y dependiendo de las características específicas del sistema construido, es posible tener que realizar otros tipos de pruebas, tales como pruebas de recuperación, de seguridad o de interfaces.

15.2 Pruebas según el objetivo que persiguen

En la sección anterior hemos visto como en función del objeto de la prueba (aquel que se está probando) las pruebas de software se realizan a diferentes niveles: un módulo, un conjunto de módulos o el sistema completo. Ahora bien, es necesario tener en cuenta que algunos tipos de pruebas son apropiados para unos niveles de prueba pero no lo son para otros, lo que permite afirmar que los objetivos de las pruebas variarán según el objeto de las mismas. En realidad, según la intención con que se lleve a cabo una prueba, el nivel de la misma será diferente.

Todas las pruebas se llevan a cabo para conseguir un determinado objetivo. Así, pueden realizarse para comprobar que las especificaciones funcionales se han implementado correctamente (*pruebas de conformidad*) o para evaluar la facilidad con la que los nuevos usuarios utilizan el software (*pruebas de usabilidad*), pero también para evaluar muchas otras características del software tales como el rendimiento, la seguridad, la fiabilidad o la facilidad de mantenimiento. A continuación se enumeran y describen los tipos de prueba más frecuentemente citados por la literatura sobre el tema atendiendo a sus objetivos:

- **Pruebas de aceptación.** Se dice que un software es aceptable cuando hace aquello que debe hacer y solamente esto. Las pruebas de aceptación comparan el comportamiento de un módulo o sistema software completo con las necesidades del cliente. Lo habitual es que sea el cliente quien compruebe que el sistema que recibe posee todas las funcionalidades exigidas en el cuestionario de especificaciones técnicas, tanto en lo que respecta a las características generales como en lo relativo a cada uno de los grupos de funciones y utilidades, si bien esta actividad de pruebas puede ser realizada conjuntamente con los desarrolladores.
- **Pruebas de instalación.** Una vez concluidas las pruebas de aceptación, se debe verificar que todos los productos han quedado instalados satisfactoriamente en el entorno final en el que serán utilizados. Desde esta perspectiva, las pruebas de instalación son en realidad pruebas del sistema que se llevan a cabo bajo los requisitos que impone una determinada configuración de hardware.
- **Pruebas alfa y beta.** Es bastante frecuente, dependiendo del producto software que se esté desarrollando, proporcionar una primera versión del software (una versión que tal vez no contenga todas las funcionalidades que contendrá la versión final) a un cierto número de usuarios para su verificación. Típicamente, este proceso tiene

dos etapas: *pruebas alfa*, realizadas por personas de la organización que desarrolla el software, y *pruebas beta*, en la que participa un número limitado de usuarios externos. Los usuarios que participan en estos procesos notifican a la organización los problemas detectados en el producto para su corrección. A veces, las versiones beta de ciertos productos se ponen a disposición de una amplia comunidad de usuarios –beta masivas– con el objeto de incrementar la información que dichos usuarios retornan a la organización. No obstante, es preferible hacer este tipo de pruebas con un número limitado de usuarios con responsabilidades, evitando en lo posible las a menudo poco útiles betas masivas a que acostumbran algunos fabricantes.

- **Pruebas de conformidad.** Consisten en verificar si el comportamiento del software es conforme con ciertas especificaciones a las que debería ajustarse según lo establecido en los requisitos. En realidad, es muy frecuente que existan ciertas normas o estándares por los que el software deba regirse. Así por ejemplo, aquellos programas que tratan información sobre datos de carácter personal registrados en soporte físico, deberían cumplir la normativa vigente sobre protección de datos. Frecuentemente estas pruebas las realiza una organización externa especialmente acreditada para esa función. Dicha organización emite un veredicto final sobre si el software es o no conforme con la norma en cuestión.
- **Pruebas de regresión.** Se trata de un tipo de pruebas aplicable a diferentes niveles, por lo que en general se habla de componentes como objeto de las pruebas de regresión. De este modo, las pruebas de regresión pueden definirse como aquellas que se llevan a cabo sobre un componente para verificar que los cambios realizados no han producido efectos no deseados. El fundamento de este tipo de pruebas es comprobar que un componente que ya había sido probado no se ha visto afectado por modificaciones en otros componentes. Cuando se aplican a un sistema en su conjunto, por ejemplo, cuando el sistema en desarrollo reemplaza a un sistema previo en funcionamiento, consisten en verificar que el rendimiento del nuevo sistema es, al menos, tan bueno como el del sistema antiguo. En los métodos ágiles son pruebas de particular importancia, pues se hace uso de ellas en todas las fases del desarrollo.
- **Pruebas de rendimiento.** Estas pruebas verifican que el software alcanza los objetivos de rendimiento especificados por el cliente y expresados en el documento de requisitos. Estas pruebas son realizadas internamente por el equipo de desarrollo, si bien los resultados de las mismas frecuentemente se proporcionan al cliente para su comprobación.
- **Pruebas de desgaste.** Cuando los requisitos de un componente software marcan que éste debe poder gestionar, por ejemplo, un determinado número de conexiones simultáneas, es peligroso asumir que el sistema trabajará siempre bajo esas condiciones. Las pruebas de desgaste llevan al sistema más allá de sus límites normales de operación con el objeto de comprobar que no sólo es capaz de soportar los límites

de carga marcados sino que también puede funcionar correctamente por encima de los mismos (si bien, es de esperar que la necesidad de funcionar por encima de los límites sólo ocurra de manera puntual).

- **Pruebas de recuperación.** Cuando el software termina de manera incorrecta o incontrolada, cuando las operaciones en curso no pueden completarse debido a una terminación abrupta por causas indeterminadas, cuando se produce un fallo de hardware o, en general, cuando el sistema sufre algún «desastre», es necesario poner en marcha mecanismos de recuperación. Las pruebas de recuperación verifican que dichos mecanismos funcionan correctamente.
- **Pruebas de configuración.** Evalúan el software en los diferentes entornos que pueden configurarse según lo especificado en los requisitos. Se llevan a cabo en aquellos casos en los que el software debe funcionar en diferentes plataformas o se orienta a diferentes destinatarios, casos en los que resulta imprescindible probar que el funcionamiento es igualmente correcto en todas las posibles configuraciones.
- **Pruebas de usabilidad.** Según la definición más comúnmente utilizada, la usabilidad se refiere a la capacidad de un software de ser comprendido, aprendido, usado y ser atractivo para el usuario, en condiciones específicas de uso. El objetivo de estas pruebas es limitar y eliminar dificultades para los usuarios. Para ello, se selecciona un conjunto representativo de usuarios finales a los que se solicita que hagan uso del software. Del resultado de su interacción se toman datos directos (preguntando a la persona que hizo la prueba), indirectos (comportamiento observado en los usuarios durante la prueba) y, siempre que esto sea posible, datos tomados automáticamente (por ejemplo, el tiempo que les lleva a los usuarios procesar la información de los diálogos, de los mensajes de ayuda, etc.). Es importante destacar que las pruebas de usabilidad no sólo evalúan la facilidad con la que nuevos usuarios utilizan el software, sino también la claridad de la documentación del usuario, y muchos otros aspectos relacionados con la usabilidad.
- **Otras pruebas.** Existen muchas otras pruebas que podrían enumerarse, como las de *continuidad* (grupos de pruebas que se ejecutan en varias versiones diferentes de un mismo componente software con el objetivo de comparar los resultados y encontrar posibles discontinuidades), las de *compatibilidad* (necesarias cuando el sistema en desarrollo debe comunicarse con otros sistemas) y otras. Si está interesado en ampliar información, acuda a las referencias bibliográficas incluidas al final del capítulo.

En esta sección se han clasificado las pruebas atendiendo al objetivo que persiguen. En lo relativo a su utilización práctica, el lector debe tener en cuenta que algunos de los tipos de pruebas descritos son más apropiados para paquetes de software hechos a medida, como por ejemplo, las pruebas de instalación, mientras que otros, como las pruebas beta, son más apropiados para productos dirigidos a un mercado más amplio y no tan específico.

Prototipos «Mago de Oz»

En algunos procesos de pruebas de usabilidad se utiliza un interesante sistema que combina la utilización de prototipos con técnicas de usabilidad más convencionales. A esta técnica se le denomina *prototipado estilo Mago de Oz*. Un prototipo Mago de Oz es un tipo de prototípico que sólo funciona si hay alguien entre bambalinas que dirige la prueba y proporciona los resultados a conveniencia. El usuario no sabe que los resultados que genera la aplicación como respuesta a su interacción con la interfaz están siendo generados por un humano en lugar de por el sistema informático, lo cual permite probar algunos elementos de interfaz que tienen especial complejidad.

Así por ejemplo, para examinar el funcionamiento de un sistema de búsqueda podría llevarse a cabo una prueba donde el usuario introdujera una consulta y un experto la reescribiera para obtener mejores resultados, o simplemente seleccionarse a mano los resultados. Esto permite comprobar diferentes teorías de formulación de consultas y filtrado de resultados. Otro escenario interesante es la prueba de interfaces de usuario de lenguaje natural, pues permiten diseñar la sintaxis recomendada a partir de un estudio de la sintaxis más comúnmente utilizada por los usuarios.

7.6 Pruebas unitarias con JUnit

En los últimos años han aparecido varios *frameworks* para ayudar a realizar pruebas unitarias en diferentes lenguajes, a los que se denomina genéricamente xUnit. En esta sección se estudia cómo utilizar JUnit (el xUnit para Java), si bien los conceptos fundamentales son similares a los de los xUnit para otros lenguajes (por ejemplo, NUnit para .Net). Hemos elegido JUnit para nuestros ejemplos por la gran aceptación y penetración del lenguaje Java entre los desarrolladores y en las aplicaciones empresariales en general. Se trata de un *framework* de código fuente abierto para escribir casos de prueba cuya ejecución y comprobación puede realizarse de manera automatizada (la Figura 7.9 muestra los beneficios de la automatización de las pruebas en términos de recursos).

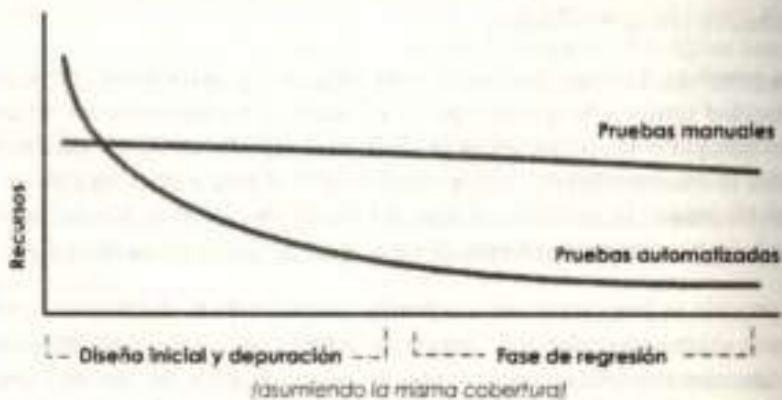


Figura 7.9: Beneficio de la automatización de las pruebas en términos de recursos

En realidad, JUnit no es más que un conjunto de clases Java, que permite y facilita la prueba de clases en este mismo lenguaje según el modelo de caja negra. Los dos elementos básicos de JUnit son los casos de prueba y las colecciones de pruebas (también conocidas como *suites de pruebas*). Los **casos de prueba** son métodos que ejecutan los métodos de otra clase, la cual es objeto de la prueba. De esta manera, cada clase tiene asociada otra de prueba, excepto clases triviales para las que no merezca la pena codificar casos de prueba.

En este enfoque las pruebas son también programas, y no simplemente especificaciones de datos de entrada y datos de salida esperados, gracias a lo cual pueden ejecutarse automáticamente. Los casos de prueba y la comprobación de si se ha pasado la prueba o no no se descartan, sino que son un producto más del desarrollo, como lo es el código fuente. Este modelo de desarrollo paralelo de pruebas de software conjuntamente con el software a probar tiene, al menos, dos ventajas importantes:

- Los casos de prueba se pueden estructurar en colecciones, lo que permite ejecutar las pruebas relativas sólo a una parte del sistema o bien a todo el sistema si así se desea.
- Segun se va desarrollando la aplicación, se va generando una batería de pruebas cada vez más completa que servirá posteriormente para realizar pruebas de regresión.

Esta última característica es especialmente importante desde el punto de vista de la calidad del software resultante. Cuando introducimos un cambio en una clase que ya ha sido probada existe la posibilidad de que ese cambio genere un fallo en otra parte del código. Para garantizar que el cambio no ha generado este tipo de efectos colaterales, deben volver a ejecutarse todas las pruebas que se habían hecho anteriormente sobre la clase. Como hemos visto en la sección anterior, a este proceso se denomina *prueba de regresión*.

La estructuración de los casos de prueba se realiza mediante **colecciones de pruebas**, conjuntos de casos de prueba sobre clases funcionalmente relacionadas. Para mostrar la utilidad de las colecciones de pruebas, imaginemos una aplicación integral para la gestión de una universidad. Para dicha aplicación, es posible agrupar en una colección todos los casos de prueba de clases relacionadas con la matriculación de alumnos, y en otra, todas las relacionadas con la contratación del profesorado. Ambas colecciones pueden, a su vez, estructurarse más detalladamente para de esta manera organizar todas las pruebas de nuestro sistema en un árbol cuyas hojas serían los casos de prueba. Esta estructuración en forma de árbol permite ejecutar cualquier subárbol a partir de un determinado nodo intermedio.

En el resto de esta sección se introducirán algunas de las técnicas esenciales de prueba unitaria con JUnit. Únicamente se tratará un subconjunto representativo del mismo, pues no es objeto del capítulo centrar todo el estudio en las pruebas unitarias. Para ejemplos más completos, remitimos al lector interesado a la documentación sobre JUnit que se referencia al final del capítulo.

7.6.1 Ejemplo sencillo de uso de JUnit

A continuación, mostramos el uso de JUnit en su versión 3; al final de la sección se mostrarán las principales diferencias y mejoras introducidas con la versión 4. Como primer

ejemplo vamos a considerar una clase muy sencilla. Probablemente, esta clase no requeriría casos de prueba por su extremada simplicidad, pero los utilizaremos por motivos pedagógicos. Supongamos que se desea crear una clase para manipular números complejos. Una primera versión de la clase podría ser la siguiente:

```
public class Complejo{  
    private float parteReal, parteImaginaria;  
  
    public Complejo(float parteReal, float parteImaginaria){  
        parteReal = parteReal;  
        parteImaginaria = parteImaginaria;  
    }  
  
    public float getParteReal(){  
        return parteReal;  
    }  
  
    public float getParteImaginaria(){  
        return parteImaginaria;  
    }  
  
    public Complejo sumar(Complejo c){  
        float real = this.getParteReal() + c.getParteReal();  
        float imag = this.getParteImaginaria() + c.getParteImaginaria();  
        return new Complejo(real, imag);  
    }  
}
```

Sólo se ha implementado una funcionalidad relevante, la suma de dos números complejos. Partiendo de aquí, utilizaremos JUnit para construir un caso de prueba. Debe tenerse en cuenta que en JUnit los casos de prueba son clases que derivan de la clase `TestCase`, e implementan métodos sin parámetros de nombre `testXXX`, donde `XXX` es una descripción de lo que está probando ese método. Como convención, el nombre de la clase que implementa el caso de prueba suele ser el mismo que el de la clase probada seguido por el sufijo `Test`. En el ejemplo actual, la clase de prueba se llamará `ComplejoTest`:

```
import junit.framework.TestCase;  
public class ComplejoTest extends TestCase{  
  
    public ComplejoTest(String nombre){ super(nombre); }  
  
    public void testSumaComplejos(){  
        Complejo c1 = new Complejo(3, 5);  
        Complejo c2 = new Complejo(1, -1);  
        Complejo resultado = c1.sumar(c2);  
        assertEquals(resultado.getParteReal(), 4);  
        assertEquals(resultado.getParteImaginaria(), 4);  
    }  
}
```

Analicemos el código anterior como arquetipo de este tipo de métodos. Se observa que el método de prueba `testSumaComplejos()` realiza la siguiente secuencia de operaciones:

1. Crea los objetos implicados en la prueba, `c1` y `c2`.
2. Ejecuta operaciones sobre los objetos creados en el paso 1 sobre el método a probar (`sumar`) en la clase principal (`Complejo`).
3. Verifica que los resultados obtenidos son los esperados, utilizando para ello aserciones que se encuentran en la clase `Assert`, una superclase de `TestCase`. Un ejemplo de este tipo de aserciones es `assertEquals`, que recibe dos valores y eleva una excepción en caso de que éstos no sean iguales, mostrando así un fallo.

Tras la ejecución aparecerá una ventana en la cual o bien tendremos que escribir el nombre del caso de prueba que queremos ejecutar, o bien habrá que seleccionarlo de una lista. Tras seleccionar la prueba deseada se lleva a cabo su ejecución. Finalmente, se mostrarán los resultados de dicha ejecución tal y como aparece en la Figura 7.10, lo cual proporciona información sobre los fallos del método de prueba seleccionado –si hubo alguno–, así como un resumen de los métodos que se han ejecutado correctamente y los que han fallado.



Figura 7.10: Resultado de la ejecución de una prueba en JUnit

Cuando la prueba falla por culpa de un error de programación, el entorno muestra el lugar del caso de prueba en el que falla la aserción, tal y como se puede ver en la Figura 7.11.

En ocasiones, la invocación al ejecutor de pruebas se incluye en el propio caso de prueba, lo cual se lleva a cabo escribiendo un método `main()`. Para seguir esta práctica

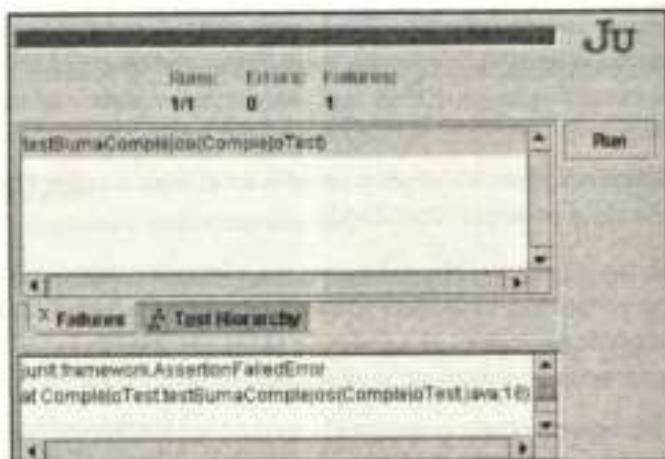


Figura 7.11: Detección de un fallo durante una prueba

sería necesario añadir el siguiente método a la clase `ComplejoTest` con objeto de que se pueda invocar al ejecutor mediante la clase que implementa el caso de prueba:

```
public static void main(String args[]) {
    String[] testCaseName {
        ComplejoTest.class.getName()
    };
    junit.swingui.TestRunner.main(testCaseName);
}
```

La instrucción para su ejecución sería la siguiente (nótese que se pasa como argumento el nombre del caso de prueba obtenido mediante la clase `ComplejoTest`):

```
java -classpath .;${backslash}junit.jar ComplejoTest
```

7.6.2 Complicación del ejemplo inicial

Para mostrar la potencia de JUnit como marco de pruebas unitarias, vamos a complicar el ejemplo con la introducción de métodos y funciones de inicialización y terminación. Además, se introducirá al lector en la escritura de pruebas que elevan excepciones. Comencemos añadiendo una operación de división a la clase `Complejo` anterior:

```
public Complejo dividir(Complejo cc){
    // Mecánica de la división: (a + bi)/(c + di) = ((ac+bd) + (bc-ad)i )/(c*c + d*d)
    float ddiv = cc.getParteReal() * cc.getParteReal() +
        cc.getParteImaginaria() * cc.getParteImaginaria();
    // Java no eleva excepciones de división por cero en números flotantes:
    if (ddiv == 0) throw new RuntimeException("División por cero");
    return new Complejo((ac+bd)/ddiv, (bc-ad)/ddiv);
}
```

```

if (ddor==0) throw new ArithmeticException("Division por cero");

float nuevaReal = this.getParteReal() * cc.getParteReal() +
    this.getParteImaginaria() * cc.getParteImaginaria();

float nuevaImag = this.getParteImaginaria() * cc.getParteReal() -
    this.getParteReal() * cc.getParteImaginaria();

return new Complejo(nuevaReal/ddor, nuevaImag/ddor);
}
}

```

Una vez completado lo anterior es necesario añadir el código que permita probar el nuevo método. El método de prueba podría ser el siguiente:

```

public void testDivisionComplejos() {
    Complejo c1 = new Complejo(3, 5);
    Complejo c2 = new Complejo(1, -1);
    Complejo resultado = c1.dividir(c2);
    assertEquals(resultado.getParteReal(), -1);
    assertEquals(resultado.getParteImaginaria(), 4);
}
}

```

Como se ve, la creación de los objetos se repite. En estas situaciones es posible sobrescribir el método `setUp()` de la clase `TestCase` para poner allí el código común. Este método será invocado justo antes de ejecutarse cada uno de los métodos de prueba de la clase. Si se lleva a cabo esta reestructuración, el código quedaría como sigue:

```

public class ComplejoTest2 extends TestCase{
//...
private Complejo c1;
private Complejo c2;

public void setUp(){
    c1 = new Complejo(3, 5);
    c2 = new Complejo(1, -1);
}

public void testSumaComplejos(){
    Complejo resultado = c1.sumar(c2);
    assertEquals(resultado.getParteReal(), 4);
    assertEquals(resultado.getParteImaginaria(), 4);
}

public void testDivisionComplejos(){
    Complejo resultado = c1.dividir(c2);
    assertEquals(resultado.getParteReal(), -1);
    assertEquals(resultado.getParteImaginaria(), 4);
}
}
//...

```

Existe también un método `tearDown()` en la clase `TestCase` que podría reescribirse para liberar los objetos iniciales creados en el método `setUp()`. En nuestro ejemplo no es necesario, ya que el recolector de basura se encargará de liberar los objetos `c1` y `c2`. Sin embargo, podría ser necesario cuando, por ejemplo, se insertan registros en una base de datos, en cuyo caso el método `tearDown()` debería incluir código para cerrar las conexiones con la base de datos, lo que es frecuente causa de problemas.

Finalmente, un ejemplo avanzado sería la ejecución de un caso particular de prueba cuya ejecución correcta debe terminar con la elevación de una excepción en un método. En nuestro ejemplo, el caso de una división por cero nos proporciona el marco idóneo. Codificamos el método de prueba correspondiente de la siguiente manera:

```
public void testDivisionPorCeroComplejos(){
    Complejo c3 = new Complejo(0, 0);
    try{
        Complejo resultado = c1.dividir(c3);
        fail("El método debería elevar una excepción");
    }
    catch (ArithmaticException e){}
}
```

La idea fundamental que subyace en esta forma de operar es que el caso de prueba tiene éxito si se eleva la excepción, por lo que si la excepción no se eleva, habrá que provocar el fallo del caso de prueba. Esto se realiza invocando el método `fail` heredado de `TestCase` justo después de la sentencia que debe producir la excepción.

7.6.3 Colecciones de pruebas

Como ya se ha dicho, la utilización de *frameworks* xUnit tiene la característica de permitir agrupar un conjunto de métodos de prueba en una colección. Para hacerlo en JUnit, se debe incorporar a la clase un método `suite()` de la clase `TestCase` que devuelva una instancia de la clase `TestSuite`.

Las colecciones pueden estructurarse en árbol para dividir las pruebas en diferentes categorías. Siguiendo el ejemplo anterior, podríamos modificar la clase `ComplejoTest` para crear una colección con dos sub-colecciones y los correspondientes casos de prueba dentro de cada una de estas últimas:

```
import junit.framework.TestCase;
import junit.framework.TestSuite;
import junit.framework.Test;

public class ComplejoTest extends TestCase{
    //...
    public static Test suite(){
        TestSuite suite = new TestSuite("suiteRaiz");
        TestSuite suite1 = new TestSuite("suiteSuma");
        suite1.addTest(new ComplejoTest("testSumaComplejos"));
    }
}
```

```

TestSuite suite2 = new TestSuite("suiteDivision");
suite2.addTest(new ComplejoTest("testDivisionComplejos"));
suite2.addTest(new ComplejoTest2("testDivisionPorCeroComplejos"));
suite.addTest(suite1);
suite.addTest(suite2);
return suite;
}

public static void main(String args[]){
junit.swingui.TestRunner.run(ComplejoTest.class);
}
}

```

El método estático `suite()` construye la colección. Este método es el que invoca implícitamente la clase `TestRunner` cuando invocamos a `run()`. En la interfaz del ejecutor aparecerá una vista de árbol con la estructura de la colección tal y como se muestra en la Figura 7.12:

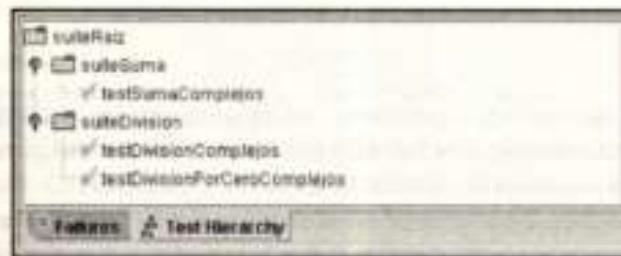


Figura 7.12: Estructura de una colección de pruebas

Obsérvese cómo la ficha «Test Hierarchy» del ejecutor de interfaz gráfica permite seleccionar una colección concreta, lo que facilita la ejecución selectiva de las pruebas.

7.6.4 JUnit 4

En su última versión hasta el momento, JUnit explota las ventajas introducidas con Java 5:

- Los `import` estáticos. Permite a una clase hacer uso de métodos y atributos estáticos de otra clase sin el incluir el cualificador (nombre de la clase a la que pertenece). Por ejemplo, todos los métodos de la clase `java.lang.Math` son estáticos y en lugar de:

```

import java.lang.Math.*;
...
double x = Math.PI * Math.cos(y);

```

es posible utilizar los métodos y variables directamente:

```

import static java.lang.Math.*;
...
double z = PI * cos(y);

```

- Las anotaciones. Son metadatos que permiten añadir información para el compilador u otras herramientas. En Java se utilizan antes de los métodos y van precedidos del símbolo @, por ejemplo @test.

Para escribir las pruebas unitarias en JUnit 4, se sigue la misma filosofía que hemos visto para JUnit versión 3, si bien teniendo en cuenta las siguientes reglas:

- El nombre de la clase de pruebas es el nombre de la clase más el sufijo Test, pero sin heredar de la clase TestCase. Las aserciones están disponibles mediante importes estáticos, por ejemplo:

```
import static org.junit.Assert.assertEquals;
```

- Los métodos setUp() y tearDown() se sustituyen por las anotaciones @Before y @After respectivamente. Los métodos pueden tener ahora cualquier nombre, aunque se recomienda mantener la convención setUp() y tearDown(). Con JUnit 4, se han definido dos anotaciones nuevas, @BeforeClass y @AfterClass que se ejecutan una única vez al principio y final de todas las pruebas respectivamente, en lugar de ejecutarse antes y después de cada caso de prueba como setUp() y tearDown().
- Los métodos de prueba se definen con la anotación @Test, aunque se recomienda llamar a los métodos de prueba igual que los métodos de la clase a probar o con la convención de JUnit 3, es decir, testXXX().

El siguiente código muestra cómo serían las pruebas de la clase Complejo en JUnit 4:

```

import org.junit.*;
import static org.junit.Assert.*;

public class ComplejosTest {

    @Before
    public void setUp() {
        c1 = new Complejo(1, 1);
        c2 = new Complejo(1, 0);
        c3 = new Complejo(0, 1);
    }

    @Test
    public void testSumar() {

```

```
Complejo resultado = c3.sumar(c2);
assertEquals(c1, result);
}

@Test
public void testMultiplicar() {
    Complejo resultado = c3.multiplicar(c2);
    assertEquals(c3, result);
}
...
}
```

También se ha simplificado el manejo de colecciones de pruebas con el uso de anotaciones. Simplemente ahora necesitamos especificar con directivas las clases que queremos incluir en la colección, como se muestra en el siguiente ejemplo:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({ClasePrueba1.class, ClasePrueba2.class})
public class EjecutarColeccionSuite {
}
```

Otras mejoras sustanciales incluyen el poder indicar que no se ejecuten métodos y clases de pruebas que aún no están implementadas, la posibilidad de definir pruebas parametrizadas y otras cuya explicación detallada queda fuera del alcance de este libro.

7.7 Métricas relacionadas con las pruebas

Como vimos en el Capítulo 3, las métricas son un elemento cada vez más importante y aceptado a medida que las organizaciones maduran y se preocupan por la calidad del software producido. Las personas responsables de las actividades de prueba deben medir los resultados de los procesos de prueba, ya que de dichas medidas se obtienen datos muy valiosos que permitirán evaluar y gestionar la eficacia de estos procesos. Estas medidas son, por otra parte, esenciales en los análisis de calidad, y servirán como base para introducir mejoras en la planificación y realización de las pruebas.

La utilización de métricas aporta al proceso de prueba un conjunto de beneficios importante. Así, las medidas que se obtienen del proceso de prueba permiten realizar estimaciones en procesos futuros, proporcionan un medio para controlar el propio proceso de prueba, y ayudan a identificar áreas de riesgo donde es necesario hacer más pruebas, entre otros. Además, las métricas sobre el tamaño del componente a probar, o su complejidad, permiten orientar las pruebas y adaptarlas mejor al componente a probar, utilizando las mejores técnicas de prueba entre las disponibles.

Las métricas de pruebas pueden clasificarse en dos grandes grupos: las que llevan a cabo mediciones durante el proceso de prueba y las métricas de evaluación de las pruebas realizadas. A continuación se estudian las medidas de pruebas según estos criterios.

7.7.1 Medidas durante las pruebas

Se trata, como su propio nombre indica, de medidas tomadas o utilizadas a la vez que se están llevando a cabo las pruebas. Dentro de esta categoría se encuentran medidas que ayudan en la planificación y diseño de pruebas de programas, medidas para clasificar y analizar errores, medidas de densidad de fallos, medidas que permiten determinar si es necesario seguir probando o si por el contrario, las pruebas se pueden dar por finalizadas, y medidas que permiten construir modelos de crecimiento de la fiabilidad. Veamos ahora algunos aspectos de cada una de ellas y en especial, para qué pueden ser utilizadas:

- **Dirección de las pruebas.** Las medidas para ayudar en la planificación y diseño de pruebas de programas se utilizan para dirigir las pruebas. Entran dentro de esta categoría las métricas de tamaño (por ejemplo, el número de líneas de código) o las que miden la estructura de un programa (por ejemplo, su complejidad).
- **Tendencias que muestran los casos de prueba.** Una de las métricas más importantes durante las actividades de prueba es la tendencia que muestran los casos de prueba ejecutados, en especial el número de defectos que quedan aún sin resolver.
- **Aumento de la efectividad.** Las medidas sobre los tipos de errores detectados permiten hacer pruebas más efectivas mediante la clasificación de las mismas en función de sus causas, su severidad, la etapa del desarrollo en que son detectados, etc. Conocer qué tipos de errores se pueden encontrar y la frecuencia con que suelen aparecer permite elaborar estadísticas que pueden ser muy útiles no sólo para realizar estimaciones sobre la calidad del producto, sino también para mejorar el proceso de prueba.
- **Densidad de fallos.** La densidad de fallos encontrados en un módulo de software permite valorarlo de acuerdo a un baremo, o en comparación con otros módulos. Para ello, será necesario contar y clasificar los errores descubiertos por su tipo y calcular el valor de densidad, que se obtiene como el cociente entre el número de errores encontrados y el tamaño del programa.
- **Conclusión de las pruebas.** Uno de los elementos críticos durante el proceso de prueba es determinar si las pruebas se pueden dar por concluidas o no. Dicha decisión debe tomarse en función de la fiabilidad del software que se está probando. La medición de los datos que permiten estimar la fiabilidad del software resulta extremadamente importante en este punto.
- **Modelos de crecimiento de la fiabilidad.** Los modelos de crecimiento de la fiabilidad permiten estimar la tasa de fallos proporcionando predicciones de la fiabilidad

del software que se basan en los fallos observados. El número de fallos que tendrá un sistema en la fase de producción puede predecirse, lo que proporciona a los gestores del proyecto una útil herramienta de decisión. Estos modelos asumen, en general, que los errores que causan los fallos observados se han solucionado (aunque existen algunos modelos que también aceptan soluciones imperfectas), y por tanto, el producto experimenta una fiabilidad incremental a lo largo de su ciclo de vida (Figura 7.13).

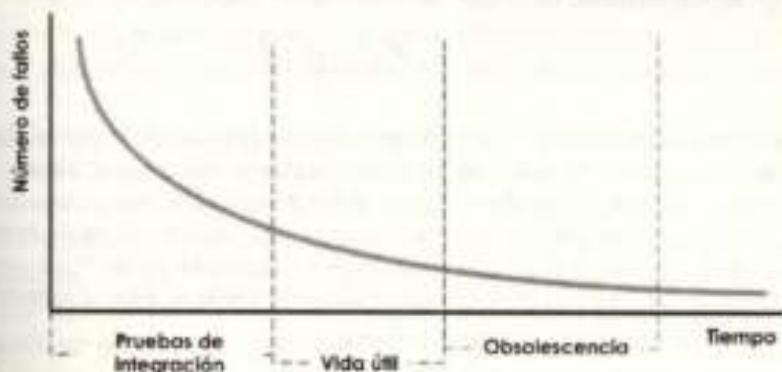


Figura 7.13: Evolución de la tasa de fallos en sistemas software

7.7.2 Evaluación de las pruebas realizadas

Una vez realizadas las pruebas, es posible recabar datos que serán de utilidad tanto para el propio proceso de desarrollo en curso como para procesos futuros. Las medidas de cobertura y compleción, la medición de errores artificiales introducidos a propósito, el cómputo de datos sobre mutantes en pruebas de mutación y los estudios para comparar la efectividad relativa de las diferentes técnicas de prueba permiten evaluar las pruebas realizadas:

- **Evaluación de la cobertura.** Las medidas de cobertura y compleción permiten controlar qué parte del software ha sido ya probada entre las definidas en la especificación. A la hora de evaluar la compleción de las pruebas realizadas, se suelen comparar los elementos cubiertos con su número total.
- **Estimación de errores.** La introducción de errores ficticios en un programa permite comprobar cuántos de estos errores son posteriormente detectados durante el proceso de prueba. En teoría, la medición del número de errores de este tipo que se detectan permite estimar el número de errores genuinos aún no detectados.
- **Medición de la efectividad.** Los datos de las pruebas de mutación, en particular la relación entre el número de mutantes con los que se ha acabado y el total de mutantes generados, sirve como medida de la efectividad del conjunto de pruebas realizadas.

Técnicas de crecimiento de la fiabilidad del software

Los modelos de crecimiento de la fiabilidad son modelos paramétricos que permiten predecir la fiabilidad del software a partir de los fallos observados. Estos modelos se clasifican en dos categorías: los que analizan el tiempo entre fallos y los que analizan el número de fallos observados por unidad de tiempo. Para realizar las predicciones, miden determinados parámetros; por ejemplo, el número esperado de fallos acumulado $D(t)$ se determina en el modelo de Schneidewind en función del tiempo según la expresión:

$$D(t) = \frac{\alpha(1 - \exp(-\beta t))}{\beta}$$

donde α representa el número de fallos al comienzo de las pruebas y β la tasa de fallos por unidad de tiempo. La aplicación de estos modelos permite reducir considerablemente los costes de garantía del producto, lo que incide positivamente tanto en la imagen de la compañía como en la satisfacción de sus clientes. El funcionamiento de estos modelos se basa en la realización de una curva de garantía que permite predecir qué ocurrirá en el futuro respecto de los fallos en el software. Para determinarla, se sigue el siguiente proceso:

1. Mantener un registro de fallos anteriores.
2. Representar los fallos en un gráfico temporal
3. Determinar la curva que mejor se ajuste a las observaciones.
4. Medir la exactitud de la curva tomada como modelo.
5. Predecir la tasa de fallos en cualquier punto futuro utilizando el modelo.

Resaltemos que en absoluto se trata de técnicas teóricas con poca aplicación práctica. La NASA, por ejemplo, utiliza estas técnicas para asegurar la fiabilidad del software instalado en sistemas particularmente críticos para el éxito de las misiones que desarrolla. Una prueba más de su uso habitual es la existencia de herramientas específicas como SMERPS o CASRE que automatizan y facilitan el proceso de aplicación de modelos, tomando datos, analizándolos, probando varios modelos, aconsejando el que mejor se ajusta según las circunstancias y prediciendo el comportamiento futuro a partir del modelo elegido.

- **Estudio de adecuación de las pruebas.** La efectividad relativa de las diferentes técnicas de prueba permite estimar qué técnica resulta más adecuada en una situación concreta, complementando en buena medida la experiencia de los ingenieros de pruebas. Sin embargo, la elección del criterio que se utiliza para la comparación de la efectividad es opinable y está, por tanto, sujeta a controversia.

7.8 El proceso de prueba

Todos los conceptos vistos –las técnicas de prueba, los niveles de prueba y las métricas– deben integrarse en un proceso definido, controlado y cuya gestión corre a cargo de una o más personas responsables del mismo. Este proceso debe estar oportunamente documentado mediante un plan de pruebas, guía para todos los involucrados en la realización de las pruebas y que constituye un registro del proceso de planificación de las mismas.

En el plan de pruebas se describen, entre otras cuestiones, los entornos de prueba, el grado de independencia del equipo encargado de las mismas y la evaluación de los resultados, se detalla y justifica el uso de las técnicas a emplear para el diseño de casos de prueba y las métricas a utilizar, y en general se hace explícito el objeto, ámbito, y directrices principales del esfuerzo de prueba.

El proceso de prueba abarca diferentes aspectos que conviene analizar por separado y que deben tratarse oportunamente como parte del mismo: la finalidad que guía el propio proceso de prueba, la gestión del mismo, la documentación producida, etc. A continuación analizaremos algunos de los más importantes contenidos de todo proceso de prueba:

- **Finalidad que guía el proceso de prueba.** Es importante definir, como parte del proceso de prueba, qué finalidad guía la realización de las pruebas, ya que muchas decisiones pueden variar en función de ello. Así por ejemplo, cuando las pruebas están dirigidas en función del riesgo, los encargados de las mismas diseñan las estrategias de prueba y asignan las prioridades. Por el contrario, cuando se trata de pruebas de escenario, los casos de prueba se definen y basan en ciertas situaciones del software previamente definidas.
- **Gestión del proceso de prueba.** Durante el proceso de prueba se realizan pruebas a diferentes niveles. La interrelación de muchas de ellas y la dependencia mutua obliga a organizar cuidadosamente aspectos tales como cuándo van a realizarse, quiénes van a ser las personas implicadas, qué herramientas deberán utilizarse y qué normas y medidas van a emplearse como referencia. Todo ello debe enmarcarse dentro de un proceso claramente definido que será una parte integral del ciclo de vida del software.
- **Documentación y productos de las pruebas.** El proceso de prueba debe ser un proceso formal y documentado. El estándar IEEE para la documentación de las pruebas del software (IEEE, 1998a) define los documentos a generar durante las pruebas, entre los que se incluyen el *plan de pruebas*, la *especificación del diseño de las pruebas*, la *especificación del procedimiento de pruebas*, la *especificación de los casos de prueba* y un *informe sobre las incidencias encontradas*, entre otros.
- **Estructura del equipo de pruebas.** En un proceso de prueba formalizado es común también formalizar la organización del equipo de pruebas. Las personas que integran este equipo pueden ser miembros internos (personas que participan en el proyecto, y que pueden estar o no involucrados en la construcción del software), o miembros externos. El empleo de miembros internos suele ser más sencillo y aporta la ventaja de que se trata de personas que ya tienen un cierto conocimiento del sistema en desarrollo, lo cual puede resultar fundamental a la hora de enfrentarse a las actividades de prueba. Por otra parte, los miembros externos pueden aportar una visión independiente y sin prejuicios de aquello que se está probando, si bien su participación es generalmente más cara y complicada. Las decisiones acerca de quién integrará finalmente el equipo de pruebas se toman en función del coste del proyecto, de su

planificación, del nivel de madurez de las organizaciones involucradas y de lo crítico que sea el software a probar.

- **Medidas del proceso.** Con el objeto de mejorar el proceso de prueba, pero también para recabar datos objetivos sobre el mismo que puedan ser utilizados en proyectos futuros, a lo largo de todo el proceso de prueba se utilizan diferentes medidas. Como se trató en la Sección 7.7, dichas medidas permiten conocer, por ejemplo, si las actividades de pruebas han sido o no eficaces, cuáles de ellas han sido más eficientes, qué técnicas proporcionaron mejores resultados, etc. La especificación de qué métricas se utilizarán durante el proceso es un importante aspecto que debe ser motivo de estudio previo al inicio de las actividades de prueba y que puede variarse a lo largo de su desarrollo.
- **Fin del proceso de prueba.** Antes de comenzar el proceso de prueba debe decidirse el número de pruebas que serán consideradas suficientes de acuerdo con la calidad que se desea obtener, y cuándo puede finalizar la fase de pruebas. Para tomar esta decisión, los responsables de las pruebas se apoyan en medidas como la consecución de un determinado nivel de cobertura, si bien otras consideraciones tales como el coste y los riesgos en que se incuraría por culpa de los errores potenciales que aún queden, y la estimación del coste que conllevaría continuar realizando pruebas, son también tomadas muy en cuenta. Otro aspecto a considerar es el hecho de que, en términos generales, la evolución del número de errores a lo largo del desarrollo sigue una curva similar a la mostrada en la Figura 7.13. A medida que se avanza en el proceso de pruebas, el equipo adquiere mayor experiencia y es capaz, por tanto, de detectar un mayor número de defectos, por lo que es de esperar que cerca del final de las pruebas sólo algunos defectos especialmente difíciles de detectar permanecerán en el software.

Aparte de todos los anteriores, hay muchos otros aspectos a tener en cuenta como parte del proceso de prueba. Así, las herramientas que van a ser utilizadas, las decisiones relativas a la automatización de las pruebas (como por ejemplo qué pruebas se automatizarán y con qué criterios), o la reutilización de pruebas existentes, son temas relevantes y que deberían hacerse explícitos para clarificar y facilitar a todos los involucrados su tarea.

7.9 Resumen

En este capítulo hemos estudiado y analizado las pruebas de software. El estudio de las pruebas comienza con un conjunto de conceptos básicos relacionados con las actividades de prueba del software, con el objetivo fundamental de aclarar el –a menudo– poco uniforme discurso. Así, hemos distinguido entre fallo (efecto no deseado en las prestaciones del software) y error (imperfección en el software que provoca su incorrecto funcionamiento) o

entre probar un software (mostrar la presencia de errores en el mismo) y depurarlo (descubrir en qué lugar exacto se encuentra el error y eliminarlo), entre otros.

Una vez presentados los conceptos fundamentales, hemos estudiado las limitaciones del proceso de prueba así como los principales riesgos a que se enfrenta una organización cuando distribuye software insuficientemente probado.

Dado que existen multitud de técnicas de prueba distintas que es preciso clasificar, hemos presentado una de las clasificaciones clásicas de las técnicas de prueba –aquella que las divide en técnicas de prueba de caja blanca y técnicas de caja negra– para posteriormente detallar la clasificación que propone la guía SWEBOK, más exhaustiva y sistemática. Y dado que las pruebas se llevan a cabo en diferentes niveles, hemos establecido una clasificación de los niveles de prueba.

El estudio de las pruebas unitarias, uno de los tipos de pruebas de mayor importancia durante la construcción del software –y dado que nuestros lectores en buena medida serán también programadores y querrán crear pruebas unitarias para sus programas– nos ha llevado a xUnit. Se trata de un conjunto de *frameworks* para ayudar a realizar pruebas unitarias en diferentes lenguajes, de los cuales JUnit es posiblemente el más utilizado y del que hemos hecho uso a la hora de profundizar y poner ejemplos. Finalmente, las métricas durante las pruebas y el estudio del propio proceso de prueba en sí, su finalidad, su gestión y la documentación producida durante el mismo, han sido tratados y discutidos en el capítulo.

La Figura 7.14 representa en una nube de palabras los conceptos, por importancia relativa, que han sido tratados en el capítulo.

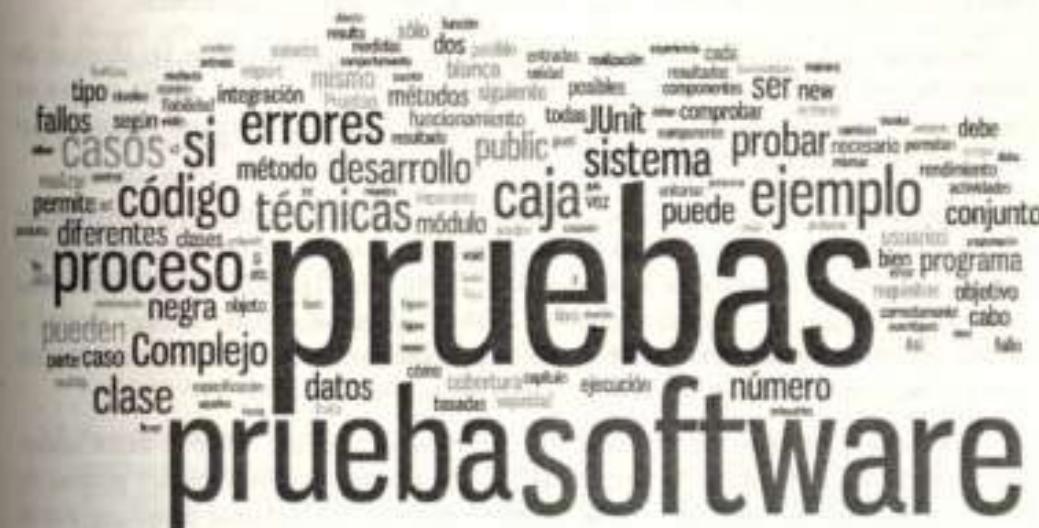


Figura 7.14: Principales conceptos tratados en el capítulo

7.10 Notas bibliográficas

El libro «*Lessons learned in software testing, a context-driven approach*» (Kaner, Bach y Pettichord, 2001) es considerado por muchos un texto imprescindible para conocer con detalle cómo se lleva a cabo la realización y gestión de las pruebas. A partir de su experiencia, los autores relatan un conjunto de lecciones aprendidas a través de los años que incluye prácticas útiles, advertencias sobre cómo seleccionar las mejores técnicas de acuerdo con las circunstancias del proyecto, y muchas otras enseñanzas de interés.

Como libro general sobre el proceso de pruebas, resulta altamente recomendable la segunda edición de «*Software Testing*» (Patton, 2005), donde se analiza desde un punto de vista práctico el mundo de la prueba de software y su relación con los procesos de aseguramiento de la calidad.

Es muy interesante la lectura del manual «*Joint software system safety handbook*» del Departamento de Defensa de los Estados Unidos, elaborado conjuntamente por militares, académicos y empresas relacionadas con el mundo de la defensa, que proporciona directrices para la construcción de software capaz de ejecutarse de acuerdo a niveles aceptables de seguridad. Está accesible en: <http://www2.urscorp.com/dahlgren/files/ssshandbook.pdf>

El libro de Nancy G. Leveson «*Safeware: system safety and computers*» (Leveson, 1995) es una excelente introducción al tema de la seguridad de los sistemas ante fallos. Escrito de manera amena y rigurosa, incluye interesantes anexos donde se analizan varios casos famosos de errores de software que llevaron a consecuencias catastróficas.

Para aquellos lectores que quieran profundizar en el conocimiento de las técnicas de prueba unitaria, el libro «*Unit analysis and testing*» (Morell y Deimel, 1992), realiza un tratamiento exhaustivo de las mismas. Este libro puede además descargarse gratuitamente en <http://www.sei.cmu.edu/reports/89cm009.pdf>

Para sacarle todo el partido a JUnit, «*JUnit recipes: practical methods for programmer testing*» (Rainsberger, 2004) es considerado el libro más completo y con más aplicaciones y trucos sobre JUnit actualmente en el mercado. Otros libros igualmente interesantes son «*JUnit pocket guide*» de Kent Beck (Beck, 2004) y «*JUnit in action*» de Ted Husted y Vincent Massol (Husted y Massol, 2003), libros que, aun siendo breves, cubren el tema con la profundidad necesaria e incluyen un número de ejemplos suficiente.

Por último, en el libro «*Software testing techniques*» (Beizer, 1990) el lector interesado en profundizar en aquellas técnicas de prueba que por las limitaciones propias de un libro como éste, no dedicado exclusivamente a pruebas, no pudieron estudiarse con la profundidad y el detalle debidos, encontrará en él un útil resumen de las estrategias de prueba más utilizadas. Hay otros textos en la misma línea, el capítulo de introducción de (Nguyen, 2000) por ejemplo, que contiene una detallada explicación de los fundamentos de las pruebas de caja gris.

7.11 Cuestiones de autoevaluación

- 7.1 En las técnicas de cobertura se considera que un 95% es un hito que permite tomar la decisión de no seguir realizando pruebas. ¿Podemos suponer por tanto que conseguir una cobertura del 95% es en sí mismo un objetivo de este tipo de pruebas? Razona tu respuesta.
- R *No debería ser un objetivo de las pruebas en sí mismo. Alcanzar un 95% de cobertura debería ser un medio para mejorar las posibilidades de encontrar fallos realizando pruebas sistemáticas en cada módulo. Nada más.*
- 7.2 Indique si es verdadera o falsa la siguiente afirmación y razona su respuesta: «Las pruebas de usabilidad evalúan únicamente la facilidad con la que los usuarios utilizan el software».
- R *Es cierto que las pruebas de usabilidad evalúan la facilidad con la que los usuarios utilizan el software, pero no sólo esto, ya que este tipo de pruebas también evalúa la claridad de la documentación del usuario, la efectividad con que el software soporta las tareas de usuario, y muchos otros aspectos relacionados con la usabilidad.*
- 7.3 La introducción de errores ficticios en un programa permite comprobar el progreso de las pruebas. ¿Cómo se lleva a cabo la estimación del progreso según esta técnica?
- R *La medición del número de errores ficticios que se detectan en un momento determinado del proceso de pruebas permite estimar el número de errores genuinos aún no detectados, ya que se conoce el número de errores introducidos al inicio del proceso. El progreso porcentual se estima como el cociente entre el número de errores ficticios detectados dividido por el número total de errores ficticios introducido.*
- 7.4 ¿Por qué es imposible asegurar que un software está completamente libre de errores?
- R *Porque sólo una prueba exhaustiva permite afirmarlo. Puesto que este tipo de pruebas son casi siempre inviables puede concluirse que siempre existirá un porcentaje de errores ocultos no detectados o al menos la incertidumbre sobre su existencia.*
- 7.5 Los frameworks genéricamente denominados xUnit ayudan a realizar un determinado tipo de pruebas en diferentes lenguajes. ¿Qué tipo de pruebas?
- R *Las pruebas unitarias.*
- 7.6 ¿Qué es una colección de pruebas en JUnit?
- R *Un elemento de organización que permite estructurar y clasificar los casos de prueba. Así, la ejecución de una colección comporta la ejecución de un conjunto de casos de prueba que se lleva a cabo sobre un conjunto de clases funcionalmente relacionadas.*
- 7.7 Razona si es verdadera o falsa la siguiente afirmación: «Un inconveniente importante de las pruebas de caja negra es lo difícil que resulta comprobar mediante las mismas la robustez del software ante comportamientos imprevistos, como por ejemplo, entradas no esperadas por parte de los usuarios».
- R *Falso, las pruebas de caja negra precisamente sirven para detectar este tipo de anomalías. Son las pruebas de caja blanca las que no pueden utilizarse con este fin.*
- 7.8 ¿Qué son las pruebas de caja gris?

- R. Son técnicas de prueba basadas en el conocimiento parcial del código. Al ser su objetivo fundamental el probar el código tal y como se ve desde fuera, del mismo modo que se trataría en una prueba de caja negra, se dice a veces que son un tipo especial de caja negra informadas por el conocimiento que se tiene de los componentes internos sobre los que opera el código que se está probando.
- 7.9 Razone si es verdadera o falsa la siguiente afirmación: «En desarrollos pequeños, siempre es preferible emplear la estrategia de pruebas de integración en big-bang».
- R. Falso. Siempre es preferible utilizar estrategias de integración sistemáticas e incrementales que no probar todos los componentes juntos al final. Sin embargo, en desarrollos muy pequeños puede no merecer la pena el esfuerzo y podría considerarse aceptable la estrategia del big-bang.
- 7.10 ¿En qué consisten las pruebas beta?

- R. En las pruebas beta el software desarrollado se distribuye a un número limitado de usuarios externos para que éstos lo prueben y notifiquen los problemas detectados.

7.12 Ejercicios y actividades propuestas

7.12.1 Ejercicios resueltos

- 7.1 Identifique clases de equivalencia para el parámetro *x* en el siguiente código escrito en Pascal:

```
function transformar (x:integer): real;
begin
  if (x mod 2 = 0) then
    transformar := 2.0 * x;
  else if (x mod 3 = 0) then
    transformar := 3.0 * x;
  else if (x mod 5 = 0) then
    transformar := 5.0 * x;
  else
    transformar := x;
end;
```

Solución propuesta: Existen claramente tres clases de equivalencia para el parámetro *x*: el conjunto de números pares, el conjunto de los múltiplos impares de tres y el resto de valores enteros. Los valores pertenecientes a la primera clase entrarían por el primero de los caminos de la sentencia *if*, los múltiplos de 3 por el segundo camino (*else if*) y el resto, por el camino que determina el *else*.

- 7.2 ¿Cuál es la diferencia esencial entre las técnicas de prueba de caja blanca y las de caja negra? Ponga ejemplos de situaciones en las que sea oportuno realizar pruebas de caja blanca y compárelas con otras situaciones en las que sea preferible utilizar pruebas de caja negra.

Solución propuesta: En las pruebas de caja blanca se ejecuta un módulo de programa y, mediante la observación de su código, se pretende probar todos los caminos posibles de ejecución. Las pruebas de caja negra, por el contrario, se basan en la introducción de valores de prueba para un módulo y en observar si los resultados son los deseados, pero no es parte de la prueba inspeccionar el código.

El problema del año 2000 es un buen ejemplo de cuándo es necesario hacer pruebas de caja blanca. Los antiguos programas que hacían uso de la fecha para sus cálculos internos (por ejemplo para establecer la edad de una persona en función de su fecha de nacimiento) tenían codificado internamente el año con dos dígitos por simple imprevisión. Pasaron cientos de pruebas de caja negra y nunca fallaron. De hecho, jamás hubieran fallado de haberse acabado el mundo la noche del 31 de diciembre de 1999. Sin embargo, y dado que el fin del mundo aún parece lejano, lo más oportuno parecía arreglar los errores en aquellos programas. Errores sólo subsanables mediante la realización de pruebas de caja blanca.

Un ejemplo de situación en la que es necesario realizar pruebas de caja negra sería la prueba de un procedimiento de ordenación de los elementos en un *array*. Si la especificación dice que deben ordenarse ascendente, sólo podremos comprobar si funciona adecuadamente si ejecutarmos una prueba de caja negra con un *array* de datos de prueba. Si dicho código ordena correctamente los datos pero en orden descendente, sería imposible detectar el error con pruebas de caja blanca: el código es correcto, pero no hace lo que se espera.

- 7.3 El programa en C que se muestra a continuación fue escrito para comprobar que un conjunto de caracteres de entrada está formado por cualquier sucesión de caracteres encerrados entre dos símbolos de \$. Decida cuál es el tipo de prueba más efectiva para comprobar la corrección de este programa (caja negra o caja blanca) y justifique su respuesta.

```
include <stdio.h>
int main(){
    char c, aux;
    while ( c != '$' ){
        printf("Introduzca la cadena a analizar:\n");
        c = getchar();
    }
    while ( aux != '$' ) aux = getchar();
    printf("Lectura de fórmula completada");
    return 0;
}
```

Solución propuesta: Para ser exhaustivos habría que utilizar los dos tipos de prueba. En primer lugar, una prueba de caja negra determinaría cuál es el comportamiento de este módulo ante entradas correctas (cadenas de caracteres encerradas entre símbolos de \$) y cadenas incorrectas. En este último caso, hay múltiples posibilidades: cadenas que no comienzan por \$ pero que contienen dicho símbolo una vez (por ejemplo 'abc\$defg'), cadenas que contienen dicho símbolo en distintas posiciones dos o más veces ('abc\$de\$fg', 'abc\$defg\$', 'abc\$de\$fg\$', etc.), cadenas que no contienen dicho símbolo ('abcdefg'), etc. De todos modos, tras observar el comportamiento del programa es más que recomendable inspeccionar el código para observar los diferentes caminos de ejecución posibles y corregirlo si procede, es decir, realizar pruebas de caja blanca sobre el mismo.

- 7.4 Diseñe una clase Cola que incluya las operaciones básicas de este tipo de estructuras de datos (al menos los métodos que permitan la creación de la cola, introducir un elemento y sacar un elemento), e implemente una clase ColaPrueba para probarla utilizando JUnit.

```

class Nodo{
    public int valor;
    public Nodo sig;

    public Nodo(int v){
        valor = v;
        sig = null;
    }
}

class Cola {

    // Referencias internas para la gestión de la cola
    private Nodo frente, fin;

    // Constructor
    public Cola(){
        frente = fin = null;
    }

    // Método para añadir un elemento a la cola
    public void push(int n){
        Nodo nuevoNodo = new Nodo(n);

        // Si la cola está vacía ponemos frente y fin a apuntar al nuevo nodo
        if (frente == null){
            frente = fin = nuevoNodo;
        }
        else{
            // Se añade siempre por el final
            fin.sig = nuevoNodo;
            fin = nuevoNodo;
        }
    }

    // Método para sacar un elemento de la cola
    public int pop(){
        int n = 0;
        // Si la cola no está vacía, sacar por el frente
        if (frente != null){
            n = frente.valor;
            frente = frente.sig;
            // Si sólo había un elemento, poner fin a null
            if (frente == null) fin = null;
        }
        else{
            // Si la cola está vacía emitir un mensaje y retornar 0
        }
    }
}

```

```

        System.out.println("La cola esta vacia");
    }
    return n;
}

public void mostrar(){
    if(frente == null){
        System.out.println("La cola esta vacia");
    }else{
        Nodo tmp = frente;
        while(tmp != null){
            System.out.print("[ "+tmp.valor+" ]-> ");
            tmp = tmp.sig;
        }
        System.out.print("\n");
    }
}

```

Solución propuesta: El código de pruebas para JUnit de la clase Cola no tiene excesivo interés si lo que se hace es programar pruebas para cada método por separado. Lo más interesante es comprobar que al sacar un elemento de una pila vacía el resultado es el esperado según la especificación del método: cero. Por lo demás, resulta difícil probar los métodos aislados y no en conjunción unos con otros, pero no obstante, el código de prueba sería el siguiente:

```

import junit.framework.*;

public class ColaPrueba extends TestCase {

    public ColaPrueba(String testName) {
        super(testName);
    }

    public void testPush() {
        System.out.println("push");
        int n = 0;
        Cola c = new Cola();
        c.push(n);
    }

    public void testPop() {
        System.out.println("pop");
        Cola c = new Cola();
        int expResult = 0;
        int result = c.pop();
        assertEquals(expResult, result);
    }
}

```

7.12.2 Actividades propuestas

- 7.1 Utilizando como base la clase `Complejo` de la Sección 7.6, implemente varios métodos susceptibles de ser probados con un framework xUnit, ampliando la clase `ComplejoTest` para incluir la prueba de las nuevas funcionalidades.
- 7.2 En la dirección <http://www.junit.org> se puede obtener JUnit además de una gran cantidad de artículos y software relacionado con este *framework*. Instale JUnit en su computadora y rehaga el ejemplo de la Sección 7.6 con objeto de afianzar los conceptos sobre prueba unitaria vistos en el capítulo.
- 7.3 Considere la siguiente especificación: «Un programa lee por teclado dos palabras, P1 y P2, y muestra por pantalla aquellas letras de P1 que no están en P2». Escriba un conjunto de casos de prueba que permitan probar adecuadamente este programa. Pista: los casos de prueba tendrán la forma (x, y) , donde x e y representan palabras.
- 7.4 Ejecute las pruebas de ejemplo que se proporcionan junto con JUnit. Cuando esté suficientemente familiarizado con estos conjuntos de pruebas modifique `junit.samples.VectorTest` añadiéndole más pruebas o modificando alguna de las existentes.
- 7.5 Lea el relato detallado de los graves incidentes de seguridad del Therac-25 que realiza Nancy Leveson en el artículo *«Medical devices: the Therac-25»*. Este artículo puede descargarse en línea en: <http://sunnyday.mit.edu/papers/therac.pdf>
- 7.6 La siguiente noticia saltaba en abril de 2007 a las cabeceras de los diarios:

Un error informático «mató» a la cápsula Mars Global Surveyor

WASHINGTON.- La cápsula Mars Global Surveyor, tras estar en funcionamiento cuatro veces más tiempo de lo planificado en su misión de estudio del planeta rojo, dejó de funcionar el mes de noviembre pasado aparentemente por un fallo informático, según ha revelado la NASA.

Una comisión especial, creada por la NASA para estudiar la pérdida repentina de la cápsula hace más de cinco meses, indicó que probablemente la culpa la tuvo un fallo de programas de computadora y los comandos desde la Tierra.

El 2 de noviembre pasado, después de que los técnicos ordenaran a la MGS ajustes rutinarios de sus paneles solares, la cápsula dio una serie de alarmas y luego indicó que se había estabilizado. Aquella fue su última comunicación.

Posteriormente la nave se reorientó en un ángulo que exponía directamente a la luz del Sol una de sus dos baterías. Esto causó un recalentamiento de la misma y, en última instancia, el agotamiento de ambas baterías en unas once horas.

La posición incorrecta de la antena impidió que la cápsula informara de su situación a los técnicos de control, y la respuesta de seguridad programada no incluía el asegurarse de que su orientación fuese segura desde el punto de vista térmico.

«La causa de la pérdida fue resultado de una serie de acontecimientos vinculados con un error del software de computadora cometido cinco meses antes del probable fallo de baterías», dijo Dolly Perkins, subdirectora técnica del Centro Goddard de Vuelo Espacial en Maryland, y una de las integrantes de la junta revisora.

A lo largo del capítulo se hace referencia a otros errores de software cuyas consecuencias, como en el caso de la Mars Global Surveyor, fueron desastrosas. Entre los muchos ejemplos que encontrará en la historia de la seronáutica y la ingeniería aeroespacial, investigue algunos casos en que los errores que tuvieran relación con un software que se sometió a un proceso de pruebas inadecuado.

- 7.7 Como estudiamos en otros capítulos, la programación extrema es un modelo de desarrollo de software ligero que surge para superar los tradicionales riesgos de un proyecto de desarrollo. Para paliar las altas tasas de defectos del software, la programación extrema da una especial relevancia a la prueba como elemento clave del desarrollo, proponiendo que se escriban los casos de prueba incluso antes de desarrollar el código que implementa la funcionalidad a probar. Consulte la referencia principal al respecto (Beck, 2002) y analice el papel de los distintos tipos de prueba estudiados en el capítulo dentro de este novedoso enfoque de desarrollo.
- 7.8 Instale en su computadora una herramienta de medición de la cobertura de código (por ejemplo, *Cobertura*) y compare los resultados de la medición automatizada con los estimados «a mano» para una prueba de un módulo software determinado. **Nota:** *Cobertura* puede descargarse gratuitamente de internet en la dirección <http://cobertura.sourceforge.net/>.
- 7.9 El artículo de Kent Beck «*Simple Smalltalk Testing: With Patterns*» puso la primera piedra de lo que hoy conocemos como xUnit. En grupo, léalo y discuta las diferencias de aplicación de la técnica entre diferentes implementaciones, por ejemplo, entre la original del artículo en Smalltalk y la detallada en el capítulo en Java. El artículo puede descargarse gratuitamente de internet en la dirección <http://www.xptoolkit.com/testfram.htm>.
- 7.10 El estándar IEEE 829.98 (IEEE, 1998a) trata la documentación de las pruebas de software. Lea el apartado dedicado al plan de pruebas, donde se especifica el formato del mismo, y discuta en grupo cómo podría adaptarse y/o completarse dicho formato para ajustarse a las especificidades del desarrollo guiado por pruebas de la programación extrema.

Mantenimiento

Los programas, como las personas, envejecen. No podemos prevenir el envejecimiento pero si entender sus causas, tomar medidas para limitar sus efectos, revertir temporalmente sus daños y prepararnos para cuando llegue.

— David Parnas

8.1 La mente de los otros

Los pensamientos contenidos en la mente de los otros son privados y no pueden observarse directamente. Así, la única manera de conocer los procesos intelectuales de otras personas es observando bien sus acciones o bien los resultados de las mismas. Esto último es lo que muchas veces sucede durante el mantenimiento del software. Resulta cada vez más frecuente que las personas que tienen que corregir, mejorar o extender un producto software no sean sus creadores originales, con el inconveniente añadido de que estos últimos no están disponibles para pedirles explicaciones. Por lo tanto, sólo se dispone de los resultados del desarrollo para comprender la estructura, funcionamiento y diseño del software. Como en todas las obras humanas que resultan en artefactos complejos, si el creador no tuvo el cuidado de hacer el diseño comprensible, o la deferencia de proporcionar documentación para que los que vengan detrás lo entiendan, la tarea de comprender el software es una labor más propia de Sherlock Holmes que de un ingeniero. Y la investigación detectivesca, aunque concluya con éxito, es una tarea que consume mucho tiempo y energía.

El diseño y el código fuente pueden hacerse tremadamente oscuros e impenetrables, o pueden hacerse razonablemente entendibles a través de las explicaciones de los autores. También pueden haber sido creados de un modo tal que permita y facilite modificaciones posteriores. Es sólo una cuestión de intención. Pero es importante resaltar que en la Ingeniería del Software no podemos contar con que los desarrolladores sean bienintencionados,

pues esto no sucede en todos los casos. La ingeniería como actividad disciplinada debe garantizar que artefactos tales como el código fuente estén razonablemente documentados para facilitar su comprensión futura. En los casos en que no se llevó a cabo este proceso de documentación, es importante que una vez hecho el esfuerzo de comprender el código se cree la documentación que se debió hacer en su momento. A esta actividad se denomina *ingeniería inversa*, y su objetivo es no lastrar futuros mantenimientos con un esfuerzo de comprensión que ya se ha realizado y que sería poco inteligente desaprovechar. Si es la propia estructura del código la que es manifiestamente mejorable, se puede añadir una reingeniería o reestructuración del mismo, para que las futuras actividades de mantenimiento no se encuentren en adelante con las mismas dificultades de comprensión.

Como veremos en este capítulo, el mantenimiento del software tiene como principal problema la comprensión del diseño y el código existente, prerequisito indispensable para proceder a cambiarlo o mejorarlo. Este hecho no sería importante si el mantenimiento como actividad posterior a la entrega del software fuese ocasional y poco frecuente. Pero la realidad es otra. Todos los productos software evolucionan, y si no lo hacen quedan condenados a no ser utilizados por sus usuarios, cuyas necesidades y expectativas evolucionan constantemente. En consecuencia, la gestión del mantenimiento y las técnicas orientadas a facilitarlo tienen hoy día una importancia capital en la Ingeniería del Software. Y ello a pesar de que en muchas ocasiones la presión de unas fechas de entrega muy próximas no permite prestar la suficiente atención a preparar el software para que mañana otros comprendan lo que nosotros decidimos y pensamos al crear el software. Como decíamos al principio, no se puede conocer el proceso intelectual de otras mentes salvo por la observación de sus acciones o la inspección de sus resultados. Si hacemos los resultados oscuros, estamos condenando a otros desarrolladores a un esfuerzo que podría haberse evitado.

8.2 Objetivos

El objetivo general de este capítulo es comprender la importancia del mantenimiento y las actividades que, como una fase concreta dentro del ciclo de vida del software, éste cubre.

Se introducirán los conceptos fundamentales, las técnicas, métodos, estándares internacionales y herramientas relacionadas con el proceso de mantenimiento del software. Concretamente, las competencias específicas que se ejercitarán son las siguientes:

- Saber definir el mantenimiento del software, y diferenciarlo del concepto de evolución del mismo.
- Saber distinguir los distintos tipos de mantenimiento del software.
- Saber entender las labores de reingeniería e ingeniería inversa.
- Conocer y saber aplicar métricas relacionadas con el mantenimiento del software.
- Conocer y saber utilizar herramientas propias de ingeniería relacionadas con el mantenimiento del software.

8.3 Introducción

Comenzaremos la discusión de este capítulo recordando la definición posiblemente más utilizada de la Ingeniería del Software. Esta definición, que se citó en el capítulo de introducción, decía:

La Ingeniería del Software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, la operación y el **mantenimiento** del software; esto es, la aplicación de la ingeniería al software

En la propia definición aparece el mantenimiento como una de las fases de la Ingeniería del Software. Ahora bien ¿en qué se diferencia el mantenimiento de otras actividades de la Ingeniería del Software? –más concretamente, ¿en qué se diferencia de lo que se suele denominar desarrollo? Para clarificar esta delimitación, hay que buscar un criterio que separe unas actividades de las otras, como se intentará a lo largo de este capítulo.

En una primera aproximación podemos decir que las actividades de mantenimiento son actividades de Ingeniería del Software orientadas a la *modificación o cambio* del mismo. Pero para introducir cambios, primero se necesita una comprensión del objeto que se ha de cambiar, y sólo después se podrá hacer efectiva la modificación requerida. Esto hace que la comprensión del software –como actividad humana– sea un elemento esencial en la Ingeniería del Software y obliga a que este último tenga una cierta estructura e incluya una documentación asociada que facilite su comprensión.

La importancia del mantenimiento radica en sus implicaciones económicas, al igual que sucede en otras actividades de ingeniería. En un sistema fácil de mantener se puede implementar un cambio con un menor esfuerzo, lo que permite pensar que el trabajo de hacer el software más fácil de mantener puede resultar rentable. Claro está, esto sólo será cierto si se cumple la premisa de que los cambios son frecuentes en el software. Dado que la experiencia nos demuestra que todo software evoluciona para adaptarse a las necesidades de sus usuarios, la premisa anterior se cumple para la mayor parte de los casos. En consecuencia, prever el mantenimiento del software y trabajar –aun antes de la entrega– con el objetivo de hacerlo más fácil de mantener, aunque ello en un principio aumente los costes de producción, merece la pena pues estos sobrecostes se recuperarán con creces si el mantenimiento resulta más fácil.

En la siguiente sección se definen y comentan algunos conceptos fundamentales que se manejarán en el resto del capítulo, importantes para comprender el estado actual de la cuestión y conseguir una comprensión de las actividades relacionadas con el mantenimiento.

8.4 Conceptos fundamentales

Antes de profundizar en los temas que se tratan en este capítulo, es preciso definir formalmente conceptos fundamentales como *mantenimiento* o *facilidad de mantenimiento*, detallar aquellos aspectos que influyen en la facilidad de mantenimiento, analizar sus propiedades y comprender el papel que otorgan al mantenimiento de software algunos modelos de calidad ampliamente aceptados.

8.4.1 ¿Qué es el mantenimiento del software?

La consideración de qué tipo de actividades de Ingeniería del Software se consideran mantenimiento ha evolucionado con el tiempo. Es importante conocer las diferentes concepciones del término, pues las actividades de mantenimiento suelen regirse por contratos de mantenimiento donde se especifican claramente las responsabilidades de cada parte (los desarrolladores y el cliente) en las actividades post-entrega. Es decir, en los contratos o planes de mantenimiento se especifica qué actividades se consideran dentro del contrato y cuáles no, y se delimita la forma y alcance de las solicitudes de actuación. Todo lo cual necesita un consenso previo sobre lo que se considera mantenimiento y lo que no.

Cualquier esfuerzo de Ingeniería del Software –si termina con éxito– acaba por producir un determinado producto software, orientado a satisfacer ciertos requisitos previamente establecidos. El mantenimiento en este contexto se entiende de manera general como las actividades que producirán cambios en el producto inicialmente acordado. Ahora bien, el cambio se puede entender de diferentes maneras. El estándar 1219 de IEEE para el mantenimiento del software (IEEE, 1998c) define mantenimiento como:

El **mantenimiento** del software es la modificación de un producto software después de la entrega para corregir fallos, para mejorar su rendimiento u otros atributos, o para adaptar el producto a un entorno modificado

Esta definición considera el mantenimiento una actividad post-entrega, situando las actividades de mantenimiento de un producto sólo después de que dicho producto haya sido entregado y puesto en operación.

Otras fuentes consideran que algunas actividades de mantenimiento pueden comenzar antes de la entrega del producto, como ocurre, por ejemplo, cuando se revisa el diseño antes de la implementación para mejorar la facilidad de mantenimiento futura. Algunas de estas actividades son la planificación de las actividades posteriores a la entrega o las acciones orientadas a facilitar el mantenimiento posterior, como por ejemplo la revisión de la documentación. Si bien se trata, en cierto modo, de actividades de preparación para el mantenimiento más que de mantenimiento en sí, su inclusión aporta una visión más amplia del mantenimiento del software. En esta línea, Pigoski resalta la necesidad de comenzar a considerar el mantenimiento desde el mismo momento en que comienza el desarrollo:

Mantenimiento del software son la totalidad de las actividades necesarias para proporcionar un soporte rentable al sistema software. Estas actividades se desarrollan tanto antes como después de la entrega. Las actividades previas a la entrega incluyen la planificación de las operaciones posteriores a la entrega, la planificación del soporte y la determinación de la logística. Las actividades posteriores a la entrega incluyen la modificación del software, la formación de usuarios y la puesta en marcha de un servicio de soporte técnico y atención al cliente (*help desk*) (Pigoski, 1997)

La guía SWEBOK considera que el mantenimiento ocurre durante todo el ciclo de vida, y coincide en su definición con la definición anterior de Pigoski.

Teniendo en cuenta las diferencias entre las distintas definiciones de actividades *pre* y *post* entrega del software, en esta obra diferenciaremos entre actividades de preparación para el mantenimiento y actividades de mantenimiento propiamente dichas (posteriores a la entrega). El criterio para diferenciar unas actividades de otras es la entrega del producto software. En ocasiones esa entrega es un acto formal dentro de un contrato, mientras que en otras es una simple decisión de disponibilidad pública de un grupo de desarrollo. Por ejemplo, en los proyectos de código fuente abierto, que se desarrollan de manera voluntaria, el qué es una entrega lo determinan los propios desarrolladores cuando piensan que la funcionalidad implementada ha llegado a un determinado nivel de perfección.

8.4.2 La facilidad de mantenimiento

El glosario IEEE de términos de Ingeniería del Software define facilidad de mantenimiento (*maintainability*) del siguiente modo:

La facilidad de mantenimiento es la disposición de un sistema o componente software para ser modificado con objeto de corregir fallos, mejorar su funcionamiento u otros atributos, o adaptarse a cambios en el entorno

Esta definición está directamente conectada con la definición del estándar IEEE para mantenimiento del software (IEEE 1219) que dimos anteriormente. En consecuencia, la facilidad de mantenimiento es una característica de calidad del software relacionada con el esfuerzo –medido en términos de coste y recursos– para llevar a cabo un mantenimiento de manera satisfactoria: a mayor facilidad de mantenimiento, menores costes (y viceversa). La facilidad de mantenimiento debe establecerse como objetivo tanto en las fases iniciales del ciclo de vida, para reducir las posteriores necesidades de mantenimiento, como durante la fase de mantenimiento propiamente dicho, para reducir los efectos laterales y otros inconvenientes ocultos (y seguir así reduciendo las futuras necesidades de mantenimiento).

Hay varios factores que afectan directamente a la facilidad de mantenimiento, de forma que si alguno de ellos no se satisface adecuadamente, ésta se resiente:

- El proceso de desarrollo: la facilidad de mantenimiento debe formar parte integral del proceso de desarrollo del software. Las técnicas utilizadas deben ser lo menos intrusivas posible con el software existente. Desde este punto de vista, las organizaciones se enfrentan esencialmente a dos problemas: mejorar la facilidad de mantenimiento, y convencer a los responsables de que la mayor ganancia se obtendrá cuando la facilidad de mantenimiento sea intrínseca a los productos software.
- La documentación: en ocasiones, ni la documentación ni las especificaciones de diseño están disponibles. En estas circunstancias, los costes del mantenimiento se incrementan debido al tiempo requerido para que un desarrollador entienda el diseño del software antes de poder comenzar a modificarlo. Las decisiones sobre la documentación que debe desarrollarse son muy importantes cuando la responsabilidad del mantenimiento de un sistema se va a transferir a una organización diferente.
- La comprensión de los programas: la causa básica de la mayor parte de los altos costes del mantenimiento del software reside en la presencia de obstáculos a la comprensión humana de los programas y sistemas existentes. Estos obstáculos generalmente tienen su origen en la información disponible (que o bien es incomprendible, o incorrecta o simplemente insuficiente), en la complejidad del software o de la propia naturaleza de la aplicación, y también en malas interpretaciones u olvidos sobre el por qué de algunas decisiones de diseño en el software.

Pero hay otros factores que influyen en la facilidad de mantenimiento. Así por ejemplo, la forma en que se haya construido el software es un factor decisivo en la facilidad de mantenimiento del producto resultante. Los generadores de código, por lo general, no producen un código claro ni fácil de comprender, por lo que el mantenimiento del software así generado es más complejo. Por otro lado, las técnicas de programación estructurada, la aplicación de metodologías de Ingeniería del Software y el seguimiento de estándares, permiten la obtención de software con menos necesidades de mantenimiento, y en el caso de que se produzcan, son mucho más fáciles de llevar a cabo. En general, la aplicación de los principios fundamentales de la construcción de software que se estudiaron en el Capítulo 6 (minimizar la complejidad, anticipar los cambios, construir para verificar y utilizar estándares), es la mejor forma de aumentar la facilidad de mantenimiento. Empero, se han identificado varios factores específicos que influyen en la facilidad de mantenimiento:

- Falta de cuidado en las fases de diseño, codificación o prueba.
- Pobre configuración del producto software.
- Adecuada cualificación del equipo de desarrolladores del software.

- Estructura del software fácil de comprender.
- Facilidad de uso del sistema.
- Empleo de lenguajes de programación y sistemas operativos de uso generalizado o, si es posible, estandarizados.
- Estructura estandarizada de la documentación.
- Documentación disponible para los casos de prueba.
- Incorporación en el sistema de facilidades de depuración.
- Disponibilidad del hardware adecuado para realizar el mantenimiento.
- Disponibilidad de la persona o grupo que desarrolló originalmente el software.
- Planificación del mantenimiento.

8.4.3 Mantenimiento y calidad

El estándar ISO/IEC 9126 para la evaluación de la calidad del software define un modelo de calidad en el que ésta se define como la totalidad de características relacionadas con su habilidad para satisfacer necesidades establecidas o especificadas. Este modelo, que se estudia en profundidad en el Capítulo 9, clasifica los atributos de calidad según seis características (funcionalidad, fiabilidad, usabilidad, eficiencia, facilidad de mantenimiento y portabilidad), las cuales, a su vez, se subdividen en subcaracterísticas.

Concretamente la *facilidad de mantenimiento* se subdivide a su vez, según este modelo, en cinco subcaracterísticas que idealmente deben estar presentes para que un software sea fácil de mantener y, según las cuales, un producto software debe ser:

1. Fácil de analizar: lo cual implica que deben poderse diagnosticar sus deficiencias o causas de fallos, o de identificar las partes que deben ser modificadas.
2. Fácil de cambiar: el software debe permitir implementar una modificación especificada previamente. La implementación incluye los cambios en el diseño, el código y la documentación. Si el software es modificado por el usuario final, entonces la facilidad de cambio puede afectar a la operabilidad.
3. Estable: los efectos inesperados de las modificaciones deben tener un impacto mínimo.
4. Fácil de probar: debe permitir evaluar las modificaciones.
5. Conforme: debe satisfacer los estándares o normas sobre facilidad de mantenimiento.

Cada una de estas propiedades pueden ser medidas con métricas específicas, algunas de las cuales se estudian con mayor detalle en la Sección 8.8.

8.4.4 Aspectos de la facilidad de mantenimiento

La facilidad de mantenimiento se puede considerar como la combinación de dos propiedades diferentes: la susceptibilidad de ser reparado (*repairability*) y la flexibilidad. Vamos a continuación a definir estas propiedades y a discutir sobre ellas.

Un sistema software es **reparable** si permite la corrección de sus defectos con una cantidad de trabajo limitada y razonable

Esta propiedad es crítica, por lo que el diseño viene muchas veces condicionado para que el resultado final sea reparable. Se trata de una propiedad a la que afecta especialmente la cantidad y tamaño de los componentes o piezas que componen el software. Así, un producto software formado por módulos bien diseñados será más fácil de analizar y reparar que uno monolítico, si bien el incremento del número de módulos no implica un producto más reparable, ya que también aumenta la complejidad de las interconexiones entre módulos. En consecuencia, se debe buscar un punto de equilibrio con la estructura de módulos más adecuada para garantizar que el software sea reparable, facilitando la localización y eliminación de los errores en unos pocos módulos.

En cuanto a la flexibilidad, la definiremos del siguiente modo:

Un sistema software es **flexible** si permite la introducción de cambios para satisfacer nuevos requisitos, es decir, si puede evolucionar

Por su naturaleza inmaterial, el software es mucho más fácil de cambiar o incrementar –por lo que respecta a sus funciones– que otros productos de naturaleza física como, por ejemplo, equipos hardware. Sin embargo, esta flexibilidad se ve disminuida con cada nueva versión, ya que cada versión del software complica su estructura y hace más difíciles las futuras modificaciones.

Es importante destacar que la aplicación de técnicas y metodologías apropiadas puede minimizar el impacto en la flexibilidad de cada nueva modificación en el software. Por ello, la flexibilidad es una característica tanto del producto software como de los procesos relacionados con su construcción. En términos de estos últimos, los procesos deben podese acomodar a nuevas técnicas de gestión y organización, o a cambios en la forma de entender la ingeniería, entre otros.

8.5 La práctica del mantenimiento del software

Según diferentes estudios, el esfuerzo consumido en mantenimiento del software es, en proporción al tiempo de desarrollo, muy elevado, con cifras que oscilan entre el 50% y el

80% dependiendo de los estudios. O lo que es lo mismo, cada 100 euros gastados en producir un software suponen posteriormente un gasto de entre 150 y 400 euros en mantenerlo. Además, el porcentaje que este coste representa con respecto al total se ha incrementado notablemente con los años, ya que en la década de 1970, por ejemplo, el mantenimiento suponía tan solo un 35-40% del total, en los 80 entre el 40% y el 60%, etc. En definitiva, el mantenimiento constituye con creces la mayor parte del esfuerzo total del desarrollo, siendo una actividad que consume muchos recursos (ver Figura 8.1). Las actividades de mantenimiento incluyen entre otros, los siguientes elementos:

1. Comprender el software y entender los cambios que se deben realizar.
2. Modificar el software y actualizar la documentación.
3. Volver a realizar las pruebas del software (pruebas de regresión), además de probar específicamente las partes añadidas.

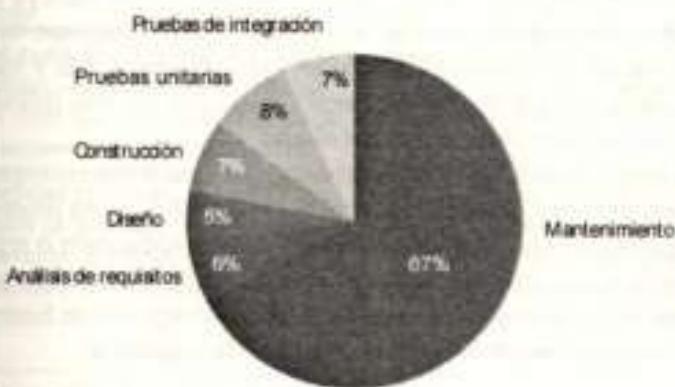


Figura 8.1: El esfuerzo de mantenimiento dentro del ciclo de vida del software

Además de esos aspectos que implican costes directos, hay otros que se traducen en costes indirectos u ocultos, también de gran importancia, como los siguientes:

- Costes de oportunidad, es decir, desarrollos que se han de posponer o que nunca se realizarán, debido a que los recursos disponibles están dedicados a las tareas de mantenimiento.
- Insatisfacción del cliente cuando no se puede atender en un tiempo aceptable una petición de reparación o modificación que parece razonable.
- Los errores ocultos introducidos al cambiar el software durante el mantenimiento reducen la calidad global del producto. Esto se traducirá en costes futuros, cuando los nuevos errores se detecten.

- Perjuicio causado a otros proyectos de desarrollo cuando el personal tiene que dejarlos, total o parcialmente, para atender peticiones de mantenimiento.

En 1970 ya se había popularizado el término *crisis del software* para hacer referencia a la situación que acabamos de describir. Los síntomas de esta crisis han estado afectando desde entonces a la industria del desarrollo de software y todavía se sienten sus efectos. Ahora sabemos que una de las principales causas de esta situación ha sido la poca importancia que se le ha dado históricamente al proceso de mantenimiento del software.

8.5.1 El mantenimiento del software como un caso especial de mantenimientos

A diferencia de los materiales mecánicos que son producto de otras actividades de ingeniería, el software no se deteriora con el uso ni con el paso del tiempo. Mejor dicho, no sufre un deterioro físico. En la ingeniería mecánica, el mantenimiento consiste en las acciones de reparación necesarias para que la máquina siga funcionando. En la Ingeniería del Software, el mantenimiento tiene un significado más amplio que incluye su adaptación a nuevas necesidades. No obstante, si se suele considerar que el software tiene un deterioro en su estructura cuando a lo largo del tiempo se van incluyendo más y más cambios que hacen que su organización y composición interna sean cada vez más difíciles de entender. A este fenómeno se conoce como *erosión del diseño*.

Esta idea del deterioro de la estructura está muy relacionada con el concepto de facilidad de mantenimiento. Ésta es, como ya vimos en la Sección 8.4.2, una propiedad del diseño de un software que mide la probabilidad de realizar una modificación o reparación en éste en un cierto tiempo. La preocupación por la facilidad de mantenimiento puede traducirse en la introducción de cambios en el software con el único objetivo de hacerlo más fácil de mantener, algo con apenas paralelos en otras ramas de la ingeniería.

8.5.2 La evolución del software y sus leyes

El término *evolución del software* se utiliza desde los años 1970 para denominar la dinámica de crecimiento del software. Una definición atribuida a Lehman y Ramil lo define de la siguiente manera:

La evolución del software es el conjunto de todas las actividades de programación que se orientan a generar una nueva versión de un software a partir de una versión anterior operativa

Otros autores inciden en la generación de una nueva versión operativa de un software con cambios tangibles respecto a una versión anterior, pero mencionando a su vez los procesos y actividades de garantía de calidad y la gestión de dichos procesos que se llevan a cabo durante la aplicación de un cierto proceso de mantenimiento.

Lehman comenzó el compendio de sus leyes de la evolución del software en 1968, con la observación del proceso de programación de IBM, punto de partida para un estudio más amplio sobre la evolución del sistema operativo OS/360. La elaboración completa de las 8 leyes se prolongó durante más de 20 años, viendo finalmente la luz en 1996.

Según el Diccionario de la Real Academia, una ley es una «*regla y norma constante e invariable de las cosas, nacida de la causa primera o de las cualidades y condiciones de las mismas*». En el ámbito de las ciencias de la ingeniería, una ley debe entenderse como una característica común a muchos fenómenos o que se presenta con regularidad. La Tabla 8.1 resume la formulación de las leyes del mantenimiento tal y como se describen en el documento final publicado por sus autores (Lehman, 1996). Todas ellas hacen referencias a programas «de tipo E», es decir, a aquellos programas destinados a solucionar un problema del mundo real determinado.

Tabla 8.1: Leyes de mantenimiento del software

Ley	Enunciado	Notas
I	Cambio continuo. Un software que se utiliza debe adaptarse continuamente, en caso contrario, el software se hace progresivamente menos satisfactorio.	Esas adaptaciones son el resultado del cambio en la operación del entorno en el cual el software cumple una cierta función.
II	Complejidad creciente. A medida que evoluciona un software, su complejidad se incrementa, a menos que se trabaje para mantenerla o reducirla.	Esta ley implica un tipo de «degradación» o «entropía» en la estructura del software. Esto a su vez implica un aumento progresivo del esfuerzo de mantenimiento, a menos que se realice algún tipo de mantenimiento preventivo a este respecto.
III	Autorregulación. El proceso de evolución del software se autoregula con una distribución de medidas de atributos de producto y procesos cercana a la normal.	La evolución del software la realiza un equipo que opera en una organización más grande. Las decisiones de gestión respecto a los cambios en el software constituyen una dinámica que determina las características de crecimiento del producto.
IV	Conservación de la estabilidad organizativa (velocidad de trabajo invariable). La velocidad de actividad global efectiva media en un sistema en evolución no varía a lo largo del ciclo de vida del producto.	Usualmente se considera que el esfuerzo gastado en la evolución del sistema lo determinan las decisiones de dirección. Esto es, por supuesto así, en un cierto grado, pero su influencia está limitada por factores externos como por ejemplo, la disponibilidad de personal competente y por los propios atributos del sistema, por ejemplo, la complejidad. En la práctica, la actividad se estabiliza en un nivel que puede considerarse constante.

Las leyes que hemos enunciado son el resultado del estudio científico y de la experiencia acumulada en la Ingeniería del Software durante muchos años. Como tales, pueden servir como base para la planificación de las actividades de mantenimiento y son especialmente útiles en la toma de decisiones sobre los trabajos relacionados con el mismo.

8.6 El proceso de mantenimiento

Los procesos de mantenimiento detallan las actividades que se deben realizar durante el mantenimiento y las interacciones entre las mismas en términos de entradas y salidas. La guía SWEBOK considera un esquema de actividades como el mostrado en la Figura 8.2 para los cambios del software, que a su vez está adaptado de estándares del IEEE.

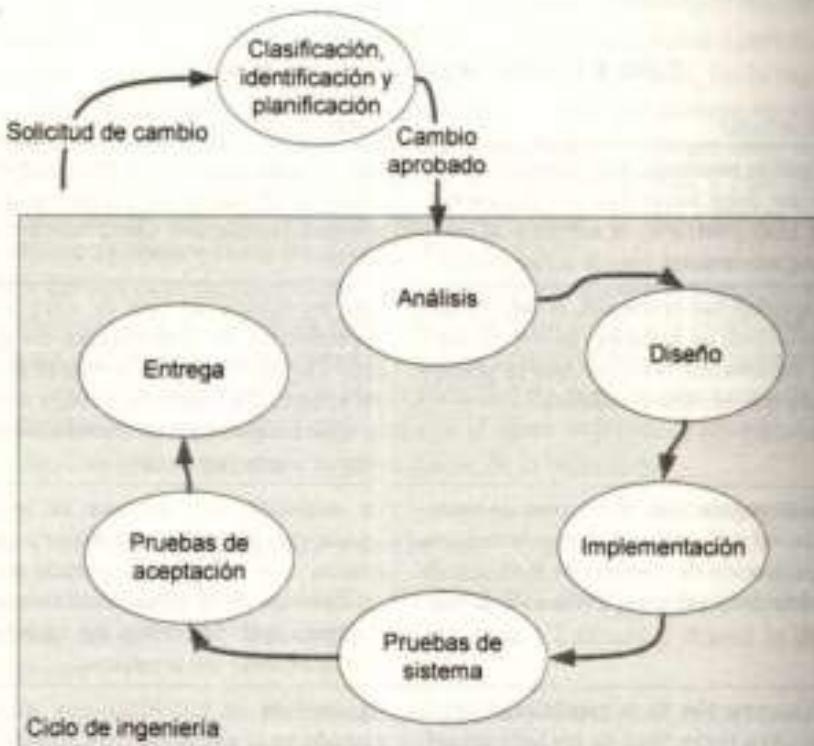


Figura 8.2: Esquema de las actividades de mantenimiento

Lo fundamental del esquema es que un cambio en el software sigue un microciclo completo de ingeniería, al que debe preceder una actividad específica de clasificación, identificación y toma de decisiones respecto a los cambios que deben hacerse en el software. En concreto, el paso de clasificación de los cambios es de capital importancia para las actividades de gestión de configuración del software, que se tratarán en el Capítulo 11.

8.6.1 Las actividades de mantenimiento

Acabamos de ver que las actividades que se llevan a cabo durante el mantenimiento del software son similares a las del desarrollo en general, pues como parte del mantenimiento se analiza, diseña, se implementa y se hacen pruebas. Se trata pues de actividades que se desencadenan por la aparición de ciertas causas, cuyo origen pertenece a tres grupos:

1. Eliminación de defectos del producto software. Por ejemplo, el cliente encuentra un fallo en el software que le han entregado, tal como un cálculo incorrecto en una aplicación de contabilidad. La organización que lo desarrolló es responsable de encontrar el defecto, y **corregirlo**.
2. Adaptación del producto software a nuevos requisitos. En este caso no hay ningún fallo, pero el cliente tiene nuevas necesidades. Por ejemplo, puede que haya nueva legislación fiscal que haga que un software quede obsoleto y haya que **adaptarlo**, o puede que haya que hacer cambios para acomodar el software a una nueva versión de un sistema operativo. También es posible que los usuarios tengan nuevas necesidades que den lugar a nuevos requisitos funcionales. Esos usuarios necesitan las nuevas funcionalidades, que no fueron previstas dado que simplemente, aún no existían. No obstante, si no se añaden dichas funcionalidades, los usuarios no querrán o no podrán seguir usando la aplicación (y la cambiarán por otra eventualmente). Un ejemplo simple sería el cambio en la legislación contable, que conlleve nuevos cálculos o formas de informar de los resultados contables. En otras palabras, para que los usuarios sigan satisfechos con un software, éste deberá dar soporte a las nuevas necesidades que vayan surgiendo. A este tipo de cambios se les suele denominar **perfectivos**.
3. Inclusión de mejoras en el diseño. En ocasiones, los desarrolladores que hacen mantenimiento de un software consideran que su estructura interna es difícil de comprender, lo que dificulta el cambio y extensión del mismo. En casos así puede ser aconsejable mejorar el diseño del software como **prevención** de cambios futuros.

Todas estas causas son el resultado de tener que modificar el software para que cumpla con requisitos del usuario ya establecidos (caso 1), para que siga cumpliéndolos cuando cambia su entorno (caso 2), o cuando se quiere mejorar la manera en que los cumple (caso 3). El mantenimiento debido a los defectos, por ejemplo, se realiza siempre *a posteriori*, pues se desencadena cuando el defecto provoca un fallo y éste se detecta. En otras ocasiones, se realizan actividades de mantenimiento preventivo, que intentan detectar y corregir fallos latentes que se supone pueden existir, pero que aún no se han manifestado.

Se puede, por tanto, establecer una correlación directa entre las causas enumeradas y las categorías fundamentales del mantenimiento definidas por Lientz y Swanson, que son:

- Mantenimiento correctivo: modificaciones reactivas a un producto software hechas después de la entrega para corregir defectos descubiertos.

- Mantenimiento adaptativo: modificación de un producto software realizada después de la entrega para permitir que dicho producto siga pudiéndose utilizar en un entorno diferente.
- Mantenimiento perfectivo: modificación de un producto software después de la entrega para mejorar el rendimiento o la facilidad de mantenimiento.

Según esta clasificación, no se considera mantenimiento a los cambios introducidos para incluir nuevos requisitos funcionales. No obstante, no hay un consenso unánime en este sentido, y de hecho, el concepto de evolución del software que vimos en la Sección 8.5.2, amplía el espectro del mantenimiento según una concepción del término cambio mucho más amplia. En realidad, muchos autores consideran que el mantenimiento perfectivo sí incluye cambios en la funcionalidad, y que las categorías adaptativa y perfectiva abarcan todo tipo de mejoras, en contraposición al mantenimiento correctivo. El estándar ISO/IEC 14764 introduce un nuevo tipo de mantenimiento orientado a prevenir errores que puedan surgir en el futuro: el mantenimiento preventivo. El equipo de desarrollo puede iniciar actividades de mantenimiento preventivo para corregir errores, o iniciar acciones para mejorar la facilidad de mantenimiento del software, que el estándar clasifica en las categorías que se muestran en la Tabla 8.2, que nos puede ayudar a ver sus diferencias. Por otro lado, dicho equipo puede reaccionar con modificaciones del software para corregir errores detectados o para adaptarlo a nuevos entornos.

Tabla 8.2: Clasificación ISO/IEC 14764 de las categorías de mantenimiento

	Corrección	Mejora
Proactiva	Mantenimiento preventivo	Mantenimiento perfectivo
Reactiva	Mantenimiento correctivo	Mantenimiento adaptativo

El estándar 1219 de IEEE para el mantenimiento del software, por último, define una categoría adicional: mantenimiento de emergencia. Dicha categoría engloba aquellos cambios que deben llevarse a cabo sin planificación previa, pues resultan esenciales para mantener un sistema en operación.

No obstante todo lo anterior, la clasificación más exhaustiva que se ha publicado es la realizada por Ned Chapin y sus colaboradores, la cual se muestra en la Figura 8.3 en forma de árbol de decisión. Según esta clasificación existe una primera categoría –etiquetada con una A en la figura– que agrupa las actividades de mantenimiento que no resultan en un cambio en el software. Éstas incluyen el uso del software para formación de usuarios (A1), cualquier actividad de consultoría sobre el software (A2) y en general, estudios sobre el mismo (A3), que pueden incluir incluso la evaluación sobre si una solicitud de cambio debe atenderse o no. Por ejemplo, las actividades de soporte telefónico o de consultoría sobre

como usar el software para propósitos de usuarios específicos entran en esta categoría, que en el caso del software empaquetado (productos software preparados para la venta al público en general¹), tiene una importancia muy grande.

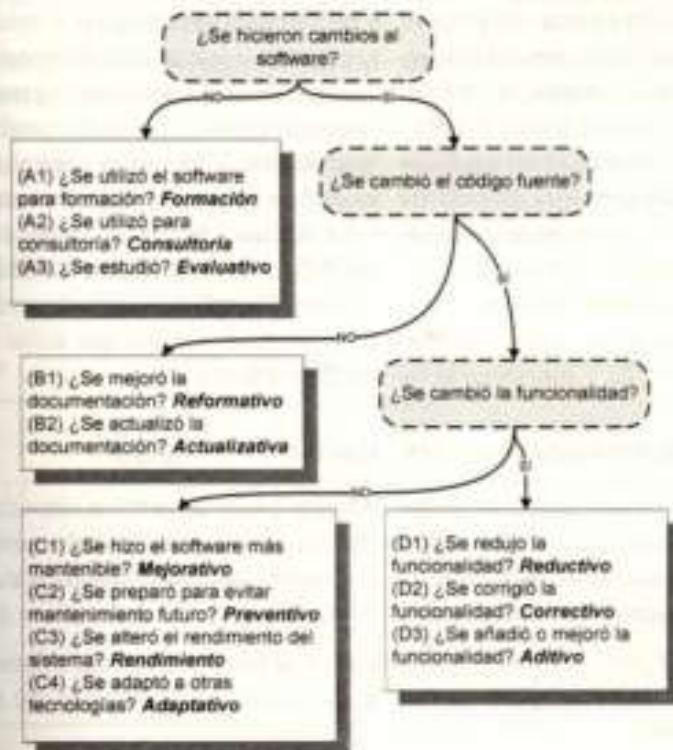


Figura 8.3: Tipos de actividades de mantenimiento según Chapin et al.

Considerando las actividades de mantenimiento en las que se cambia la documentación (de cualquier tipo) pero no el código fuente en sí, estamos ante una segunda categoría de actividades -etiquetadas como B en la figura-. Dentro de ella, si lo que se hace es mejorar la documentación para las necesidades de los usuarios (o incluso de los desarrolladores) diremos que se trata de mantenimiento reformativo (B1). Esto puede incluir mejorar los manuales de usuario, crear material de formación o reestructurar la documentación del diseño utilizando una herramienta CASE, por ejemplo. En otros casos, los cambios en la documentación se hacen para hacerla más ajustada y correcta con respecto al código que describe (B2). Por ejemplo, añadir documentación que faltaba en el código, corregir posibles errores en los comentarios, o completar diagramas que no estaban completamente actualizados con respecto a la última versión de la implementación.

¹En inglés se denominan COTS, *commercial off-the-shelf software*.

Un tercer grupo –C en la figura– es el que agrupa las actividades mediante las que se cambia el código, pero no la funcionalidad. En esta categoría tenemos el mantenimiento adaptativo (C4), cuyo objetivo es adaptar el software a nuevos o diferentes entornos tecnológicos. Por ejemplo, cuando se ajusta a un nuevo sistema operativo o a un nuevo protocolo de red. También en esta categoría están los cambios para mejorar el rendimiento en sus diferentes aspectos (C3), como el tiempo de respuesta o la cantidad de espacio que ocupa en disco. Además de lo anterior, la categoría incluye cambios de carácter preventivo (C2), es decir, orientados a evitar gastos futuros de mantenimiento, y también cambios de carácter mejorativo (C1), como cambiar algoritmos por otros más elegantes, mejorar la facilidad de comprensión de los mensajes de error o ajustar el código a una cierta convención de codificación. Por último, si se realizaron cambios que afectan a la funcionalidad, éstos pueden ser de varios tipos, según se elimine (D1), corrija (D2) o añada y mejore (D3) la funcionalidad.

La anterior clasificación permite comprender el amplio abanico de actividades que se pueden categorizar como mantenimiento. Es importante resaltar que todas ellas deben ser objeto de planificación y control en la Ingeniería del Software.

8.6.2 El mantenimiento como preparación

Las actividades de mantenimiento suelen definirse como las propias actividades de cambio del software (sea con motivo de corrección, mejora o extensión). No obstante, con el tiempo se han generalizado como actividades de mantenimiento otras acciones cuyo objetivo es preparar el mantenimiento futuro. Estas actividades pueden clasificarse en dos tipos:

- Controles *a priori*. Esta categoría incluye la planificación del mantenimiento y los controles e inspecciones de calidad para asegurar que el software que se crea sea fácil de mantener.
- Trabajo *a posteriori*. Esto incluye la mejora de la facilidad de mantenimiento por sí misma después de que el software esté en uso.

La última categoría incluye un conjunto de técnicas, como la ingeniería inversa o la reestructuración del código (*refactoring*), que han adquirido gran importancia en los últimos años y que estudiaremos con mayor profundidad en la siguiente sección.

8.7 Técnicas para el mantenimiento del software

Para paliar los problemas asociados al mantenimiento, dentro de la Ingeniería del Software se proporcionan soluciones técnicas que permiten abordarlo de manera que su impacto –en términos de coste– dentro del ciclo de vida sea menor. Las soluciones técnicas pueden ser de tres tipos:

- Ingeniería inversa: análisis de un sistema para identificar sus componentes y las relaciones entre ellos, pero también para elaborar nuevas representaciones del sistema, generalmente de más alto nivel.

- Reingeniería: modificación de componentes software en la que se emplean técnicas de ingeniería inversa para su análisis y herramientas de ingeniería directa para su reconstrucción. Este cambio está dirigido generalmente a facilitar el mantenimiento, aumentar su reutilización, mejorar su comprensión o facilitar su evolución.
- Reestructuración (*refactoring*): cambio de representación de un producto software, pero dentro del mismo nivel de abstracción.

El objetivo de estas técnicas es proporcionar métodos para reconstruir el software, bien reprogramándolo, volviéndolo a documentar, rediseñándolo, o rehaciendo algunas características del producto. La diferencia entre las soluciones descritas radica en cuál es el origen y cuál es el destino de las mismas (producto inicial y/o producto final). Gráficamente, estas tres soluciones técnicas se enmarcan en el ciclo de vida como muestra la Figura 8.4:

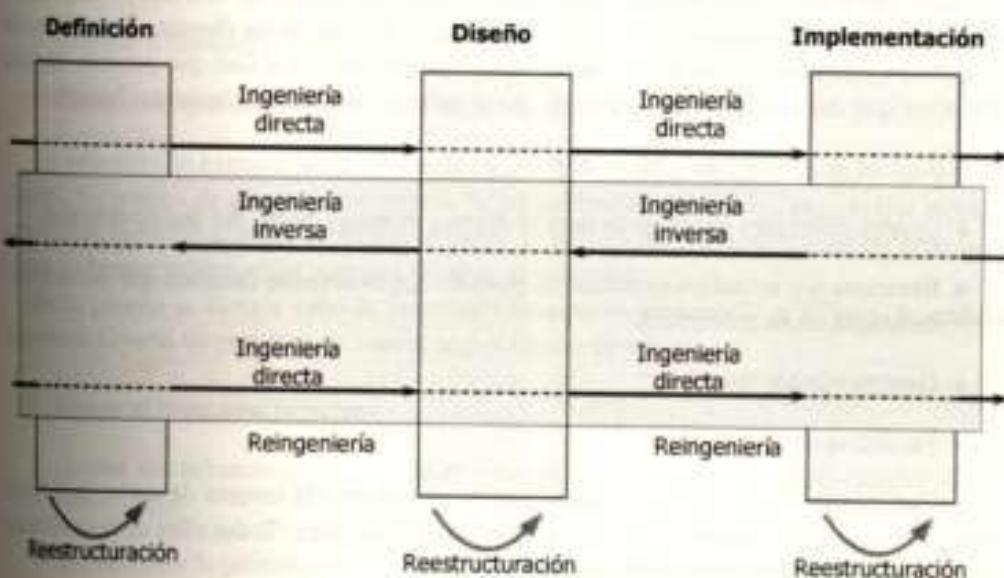


Figura 8.4: Aplicación de técnicas para el mantenimiento en el ciclo de vida del software

Como se puede ver, la reingeniería hace referencia a un ciclo, esto es, se aplican técnicas de ingeniería inversa para conseguir representaciones de mayor abstracción del producto sobre las que se aplican técnicas de ingeniería directa para rediseñar o reimplementar el producto. Cualquiera de estas técnicas se puede aplicar a lo largo de todas las fases del ciclo de vida o bien entre algunas de sus fases. En lo que sigue se estudiará el concepto de ingeniería inversa de procesos, de datos y de interfaces, el de reingeniería, donde se estudiarán los costes y beneficios de la reingeniería y por último, se tratará el concepto de reestructuración y *refactoring* como un conjunto de técnicas para llevar a cabo reestructuración de código orientado a objetos.

8.7.1 Ingeniería inversa

La ingeniería inversa es el proceso de construir especificaciones de un mayor nivel de abstracción partiendo generalmente del código fuente de un sistema software, pero también de cualquier otro producto, pues se puede utilizar –por ejemplo– un elemento de diseño. Más formalmente, puede definirse así:

La ingeniería inversa consiste en analizar un sistema para identificar sus componentes y las relaciones entre ellos, así como para crear representaciones del mismo en alguna forma que no exista, generalmente en un nivel de abstracción más elevado

Estas especificaciones pueden volver a utilizarse para construir una nueva implementación del sistema empleando, por ejemplo, técnicas de ingeniería directa. La aplicación de ingeniería inversa nunca cambia la funcionalidad del software sino que permite obtener productos que indican cómo se ha construido el mismo. Aporta los siguientes beneficios:

- Reduce la complejidad del sistema.
- Genera diferentes alternativas para el diseño, distintas de las del diseño original.
- Recupera y/o actualiza información perdida (generalmente cambios que no se documentaron en su momento).
- Detecta efectos laterales.
- Facilita la reutilización.

La ingeniería inversa puede ser de varios tipos: ingeniería inversa de datos, ingeniería inversa de lógica o de proceso e ingeniería inversa de interfaces. Todas ellas se desarrollarán con más detalle en lo que resta de sección.

Ingeniería inversa de datos

La **ingeniería inversa de datos** se aplica sobre algún código de bases datos (aplicación, código SQL, etc.) para obtener los modelos relacionales, o también sobre el modelo relacional para obtener el diagrama conceptual (por ejemplo, el modelo entidad-relación). Este tipo de ingeniería se utiliza para:

- Modificar una base de datos.
- Migrar a un nuevo sistema de gestión de base de datos.
- Crear el modelo de datos del sistema software.

Se pueden extraer, dependiendo del punto de partida, entidades, relaciones, atributos, claves primarias o ajena, etc., y a partir de estos elementos crear modelos de datos, como por ejemplo, diagramas entidad-relación.

Ingeniería inversa de procesos

Cuando la ingeniería inversa se aplica sobre el código de un programa para extraer su lógica, o sobre un documento de diseño para obtener documentos de análisis o de requisitos, se habla de **ingeniería inversa de procesos**. Habitualmente, este tipo de ingeniería inversa se usa para:

- Entender mejor la aplicación y regenerar el código.
- Migrar la aplicación a un nuevo sistema operativo.
- Generar o completar la documentación.
- Comprobar que el código cumple las especificaciones de diseño.

La información extraída son las especificaciones de diseño: se crean modelos de flujo de control, diagramas de diseño, documentos de especificación de diseño, etc., lo que permite tomar estas especificaciones como nuevo punto de partida para aplicar ingeniería inversa de nuevo y obtener información a mayor nivel de abstracción.

Pero, ¿cómo se lleva a cabo la ingeniería inversa de procesos? A la hora de realizar ingeniería inversa de procesos se suelen seguir los siguientes pasos:

1. Buscar el programa principal.
2. Ignorar inicializaciones de variables y sentencias similares de relativamente menor importancia.
3. Inspeccionar la primera rutina llamada y determinar si es importante.
4. Inspeccionar las rutinas invocadas por la primera rutina del programa principal, para proceder a examinar aquéllas que nos parecen importantes.
5. Repetir los pasos 3 y 4 en el resto del software.
6. Recopilar esas rutinas «importantes», a las que se denominará *componentes funcionales*.
7. Asignar significado a cada componente funcional según el siguiente proceso:
 - (a) explicar qué hace cada componente funcional en el conjunto del sistema y
 - (b) explicar qué hace el sistema a partir de los diferentes componentes funcionales.

A la hora de encontrar los componentes funcionales hay que tener en cuenta que los módulos generalmente incluyen este tipo de componentes. Además, suele haber componentes funcionales cerca de grandes zonas de comentarios y sus identificadores a menudo son largos y formados por palabras fáciles de entender. Una vez encontrados los posibles componentes funcionales, conviene repasar la lista y filtrarla para obtener el listado definitivo, teniendo en cuenta que un componente es funcional cuando su ausencia complica el funcionamiento de la aplicación, afecta negativamente a la legibilidad del código, impide la comprensión de todo el componente o de otro componente funcional, o hace caer a niveles muy bajos la calidad, la fiabilidad, o la facilidad de mantenimiento.

Ingeniería inversa de interfaces de usuario

Muchos programas gozan de gran fiabilidad, capacidad de procesamiento, etc., pero sus diseñadores han olvidado la comodidad y facilidad de uso del usuario final (usabilidad). En dichos casos es posible aplicar **ingeniería inversa sobre la interfaz de usuario** con objeto de, manteniendo la lógica interna del programa, obtener los modelos y especificaciones que sirvieron de base para su construcción, y tomarlas como punto de partida en procesos de ingeniería directa que permitan modificar dicha interfaz. En general, como resultado de este tipo de procesos se obtiene la relación entre los distintos componentes de la interfaz de usuario, siendo interesante poder obtener aspectos específicos de modelo de interfaces por separado: modelo de tareas, modelo de presentación, etc. No obstante, la consecución de este objetivo depende en gran medida de la especificación que en su momento se utilizara para generar la interfaz.

8.7.2 Reingeniería

Cuando una aplicación lleva años siendo utilizada, es fácil que se vuelva inestable como fruto de las múltiples correcciones, adaptaciones o mejoras que ha podido sufrir a lo largo del tiempo. Como consecuencia de ello, cada vez que se pretende realizar un cambio se producen efectos colaterales inesperados y hasta de gravedad, por lo que se hace necesario, si se prevé que la aplicación seguirá siendo de utilidad, aplicar reingeniería a la misma. En la Sección 8.7 ya se definió la reingeniería como «la modificación de un producto software, o de ciertos componentes, usando para el análisis del sistema existente técnicas de ingeniería inversa y, para la etapa de reconstrucción, herramientas de ingeniería directa».

Entre los beneficios de aplicar reingeniería a un producto existente se pueden incluir los siguientes:

- Facilita el mantenimiento del software,
- Permite la adaptación a nuevos estándares, tecnologías, etc.
- Es posiblemente la alternativa más económica frente a un nuevo desarrollo del sistema cuando se analizan las dos opciones en términos de coste/beneficio.

Procesos implicados en la reingeniería

La reingeniería debe ser entendida como un proceso mediante el cual se mejora un software existente haciendo uso de técnicas de ingeniería inversa y reestructuración de código. La reingeniería consiste en una suma de tareas que requiere tiempo y esfuerzo, y que se puede dividir en una serie de procesos separados que se llevan a cabo secuencialmente. Los procesos fundamentales en la reingeniería son:

1. Análisis de inventario: estudio de la antigüedad, importancia de la aplicación en el negocio y la facilidad de mantenimiento actual, entre otros criterios, para estudiar la posible conveniencia de la reingeniería.
2. Reestructuración de documentos, proceso en que se puede elegir una de las siguientes tres opciones:
 - (a) Obviar la documentación de los módulos estáticos, que no van a sufrir cambios.
 - (b) Documentar sólo lo que se va a modificar.
 - (c) Documentar toda la información del sistema, si es que éste es fundamental para el negocio.
3. Ingeniería inversa: en este proceso se extraen modelos de alto nivel de abstracción que ayuden a la comprensión de la aplicación, para poder modificarla. Estos modelos sirven como punto de partida del siguiente proceso (4), y se deben almacenar en un repositorio para que las personas o herramientas CASE que lleven a cabo los siguientes pasos puedan accederlos. Del mismo modo, se conforma también la documentación de análisis y diseño que facilitará el posterior mantenimiento.
4. Reestructuración del código y de los datos y/o aplicación de técnicas de ingeniería directa: a la luz de los resultados de la ingeniería inversa, se reestructuran el código y los datos, o se aplican técnicas de ingeniería directa para rehacer la aplicación.

Valoraciones sobre la reingeniería

Ante la perspectiva de aplicar procesos de reingeniería a un software, cabe preguntarse si existen alternativas a este proceso, tales como comprar un software que lo sustituya, o desarrollar el software de nuevo.

Debe tenerse en cuenta que cualquiera que sea la opción elegida (incluyendo la reingeniería) se incurre en costes de mantenimiento y de operaciones. No obstante, la reingeniería se suele considerar una buena opción frente al desarrollo de una nueva aplicación cuando:

- La aplicación tiene fallos frecuentes que son difíciles de localizar.
- La aplicación es poco eficiente, pero realiza la funcionalidad esperada.

- Existen dificultades para integrar la aplicación con otros sistemas.
- El software de la aplicación es de poca calidad.
- Cuando no se dispone de personal suficiente para realizar todas las modificaciones necesarias que podrían surgir.
- Cuando no sea fácil hacer las pruebas oportunas a todos los cambios que se debería realizar.
- Cuando el mantenimiento de la aplicación consume muchos recursos.
- Cuando es necesario incluir nuevos requisitos en la aplicación, si bien los fundamentales se mantienen.

De manera cuantitativa, se puede calcular el beneficio de las dos opciones disponibles. Si se mantiene el software como está, el beneficio se puede calcular de la siguiente manera:

$$B_M = [V_A - (C_{MA} + C_{OpA})] \cdot T_{Vida}$$

Siendo B_M el beneficio de mantenimiento, V_A el valor de negocio actual (anual), C_{MA} el coste de mantenimiento actual, y C_{OpA} el coste actual de operación de la aplicación, es decir, los costes derivados de mantener la aplicación en uso (servicios de atención al cliente, administración, etc.).

Si por el contrario se elige hacer reingeniería, el beneficio obtenido será:

$$B_R = [(G_F \cdot T_{Vida}) - (C_R \cdot F_R)] - B_M$$

$$G_F = V_F - (C_{MF} \cdot C_{OpF})] - B_M$$

$$T_{Vida} = T_{Vida(Estimado)} - T_{Reingenieria}$$

Donde B_R es el beneficio de reingeniería, G_F la ganancia final, C_R el coste de reingeniería, F_R el factor de riesgo de la reingeniería, B_M el beneficio de mantenimiento, V_F el valor de negocio tras la reingeniería (anual), C_{MF} el coste de mantenimiento final y C_{OpF} el coste de operación final.

8.7.3 Reestructuración

La **reestructuración** es una modificación del producto software dentro del mismo nivel de abstracción. En la reestructuración se produce una modificación de una parte del sistema (código fuente, documentación, etc.), sin que se altere el resultado de otras etapas. Por su naturaleza, no requiere un conocimiento total del sistema, sino únicamente de la parte a modificar. Un ejemplo podría ser la reestructuración de un programa para mejorar su

legibilidad, proceso tras el cual los documentos de la etapa de diseño no habrán sufrido modificación y el sistema seguirá realizando la misma funcionalidad.

Uno de los casos de reestructuración más habituales es la reestructuración del código. Algunos sistemas heredados² tienen una arquitectura de programa relativamente sólida, pero los módulos individuales han sido codificados de una forma que los hace difíciles de comprender, comprobar y mantener. En estos casos, se puede reestructurar el código ubicado dentro estos módulos «problemáticos». Para llevar a cabo esta actividad, se analiza el código fuente indicando las violaciones de las normas de la programación estructurada, para posteriormente reestructurar el código. El código resultante se revisa y se comprueba para asegurar que no se hayan introducido anomalías, así como para verificar que funciona correctamente. Por último, se actualiza la documentación interna del código.

Refactoring

El *refactoring*³ es un tipo de reestructuración de código que se aplica en desarrollos orientados a objetos (Fowler et al., 1999):

Refactoring es el proceso de cambiar el software de un sistema para mejorar su estructuración interna pero sin alterar su comportamiento externo

Como cualquier reestructuración de código, el *refactoring* tiene como objetivo limpiar el código para que sea más fácil de entender y modificar. Esta acción consigue, como efecto lateral, mejorar el diseño del software y ayudar a encontrar errores ocultos que puede que no hayan salido a la luz todavía. Es muy importante tener en cuenta la importancia de preservar el comportamiento original del código, por lo que quien vaya a ponerlo en práctica debe asegurarse de tener una clara idea de lo que hace dicho código.

Existe una gran variedad de técnicas para hacer *refactoring*, que pueden agruparse en las siguientes categorías:

- (Re-)composición de métodos. En esta categoría se incluyen las técnicas que se basan en acortar métodos demasiado largos (de acuerdo a la funcionalidad), sustituir llamadas a los métodos por su código, etc.
- Movimiento de características entre clases. Bajo esta categoría se incluyen técnicas para reasignar las responsabilidades de una clase a otra, por ejemplo, separar una clase en varias, eliminar una clase, introducir uno o más métodos nuevos en una clase, etc.

²Se denominan habitualmente como «heredados» (*legacy*) a los sistemas que funcionan sobre hardware o software antiguo, y que llevan en funcionamiento muchos años.

³Preferimos conservar el término original en inglés, ya que la traducción directa por «refactorización» no es tan habitual.

- Reorganización de datos. Esta categoría incluye las técnicas de *refactoring* que permiten facilitar el trabajo con datos, como la creación de métodos *get* y *set* para consultar los propios atributos dentro de una clase, reemplazar ciertas estructuras de datos por objetos, reemplazar literales por constantes, etc.
- Simplificación de expresiones condicionales. Esta familia de técnicas pretende facilitar la comprensión, depuración y mantenimiento del software mediante la simplificación de las estructuras condicionales, por ejemplo, dividiendo una condición compleja en varias, o eliminando expresiones condicionales redundantes.
- Simplificación de las llamadas a los métodos. Las técnicas de esta categoría pretenden simplificar la interfaz de una clase para que sea más fácil de usar. Pertenece a esta categoría algunas técnicas de *refactoring* como el cambio de nombre de los métodos, la eliminación o agregación de parámetros a los métodos, y otros.
- Reorganización de la jerarquía de generalización. Bajo esta categoría se engloban técnicas para mover métodos a lo largo de la jerarquía de herencia, como añadir un método de subclases a una superclase, añadir constructores a las superclases, dotar a las superclases de mayor funcionalidad, o crear nuevas subclases o superclases.

Uno de los autores más relevantes en el mundo del *refactoring*, Martin Fowler, propone un catálogo de transformaciones de código sobre el código original. Así, identifica el código a limpiar (código duplicado o casi duplicado, métodos muy largos, etc.), e indica las transformaciones recomendadas sobre el mismo. Este catálogo de transformaciones se describe de manera similar a los patrones de diseño, utilizando una plantilla con la siguiente información:

- Nombre, para construir un vocabulario que pueda ser conocido por la comunidad.
- Un resumen con contextos de aplicación de la técnica de *refactoring* y una explicación de lo que ésta hace.
- Los mecanismos para llevarla a cabo y una descripción paso a paso de cómo hacerlo.

Un ejemplo sencillo para ilustrar cómo aplicar estas técnicas es el siguiente. Si dentro de una misma clase se detecta el código duplicado o casi duplicado de varios métodos, podríamos aplicar la técnica de *refactoring* denominada *extraer método*, para sacar la parte de código duplicado del lugar en que está actualmente y crear con ella un nuevo método privado. Esta misma técnica podría aplicarse cuando nos encontramos con métodos muy largos, pues en la orientación a objetos suele considerarse óptimo el que los métodos o clases no tengan excesivas responsabilidades, siendo aconsejable descomponerlos en métodos o clases más pequeños.

En resumen, el *refactoring* es un conjunto de técnicas que persiguen mejorar la estructura del código. Y puesto que cada vez hay disponibles más herramientas para ayudar a aplicar técnicas de *refactoring*, su aplicación se ha extendido considerablemente.

8.8 Métricas de mantenimiento

En el Capítulo 3 aprendimos que obtener medidas es una labor fundamental en cualquier actividad relacionada con la ingeniería. En el caso del mantenimiento del software, la medición se puede considerar como un atributo de calidad del software. De hecho, el aseguramiento de la calidad, una actividad de protección que se aplica a lo largo de todo el proceso de Ingeniería del Software, incluye el mantenimiento del mismo.

Para determinar mejor los costes del mantenimiento, los encargados del mismo pueden medir la facilidad de mantenimiento y utilizar dicha medida como factor en la evaluación de costes. Si las mediciones se realizan durante el desarrollo, ayudarán a determinar en qué grado se está incorporando la facilidad de mantenimiento al proceso de producción del software. Si se realizan una vez que el desarrollo se ha completado, pueden servir de guía durante el proceso de mantenimiento, bien para evaluar el impacto de un cambio, o bien para realizar un análisis comparativo entre varias propuestas.

Las métricas de mantenimiento no pueden evaluar el coste de realizar un cierto cambio en el sistema software, pero sí aspectos de su complejidad y calidad. Esto se debe a la gran correlación existente entre la facilidad de mantenimiento y dichas propiedades del software (calidad y complejidad). Dicha correlación es inversa con respecto a la complejidad, pues a mayor complejidad menor facilidad de mantenimiento, y directa con respecto a la calidad, pues a mayor calidad mayor facilidad de mantenimiento y viceversa.

Desde el punto de vista del mantenimiento, existen dos grupos de métricas que resultan especialmente interesantes: las métricas del producto y las del proceso. Las métricas del producto describen aquellas características del software que de alguna forma determinan la facilidad de mantenimiento, como por ejemplo, su tamaño, complejidad o ciertas características de su diseño. Por su parte, las métricas del proceso se pueden utilizar para mejorar el desarrollo y facilidad de mantenimiento del software. Algunos ejemplos incluyen la eficacia de eliminar defectos durante el desarrollo, patrones de aparición de defectos durante las pruebas o el tiempo fijo de respuesta del proceso.

A continuación analizaremos algunas métricas de utilidad específica durante el proceso de mantenimiento divididas por categorías. Debe tenerse en cuenta que algunas métricas pueden pertenecer a múltiples categorías, por lo que la clasificación que se muestra a continuación no ha de tomarse como exclusiva.

8.8.1 Métricas del producto

Dentro de las métricas del producto relacionadas con el mantenimiento, nos encontramos con dos grupos claramente diferenciados. Por una parte, aquellas que miden la complejidad –como las métricas de McCabe y la ciencia del software de Halstead–, que por su carácter son de aplicación no sólo al mantenimiento, sino también en otros momentos del ciclo de vida del software. Como ya se han descrito con detalle en el Capítulo 3, no volveremos sobre ellas aquí. Por otra parte, existen métricas como el índice de madurez del software, la densidad de comentarios, las métricas de calidad del diseño o la técnica del índice de facilidad.

dad de mantenimiento, cuya aplicación más evidente son las actividades de mantenimiento, aunque algunas de ellas no puedan considerarse exclusivamente para esta actividad. Será en este grupo de métricas en el que nos detendremos durante las siguientes líneas.

Índice de madurez del software

El diccionario estándar de IEEE sobre medidas para la producción de software fiable -IEEE 982.1- sugiere el empleo de un **índice de madurez del software** como métrica específica de mantenimiento. Esta métrica proporciona una indicación de la estabilidad de un producto software.

Para calcular el índice de madurez de un software es necesario obtener varias medidas previas que son las siguientes:

- M_T = número de módulos en la versión actual.
- F_m = número de módulos en la versión actual que han sido modificados.
- F_a = número de módulos en la versión actual que han sido añadidos.
- F_e = número de módulos de la versión anterior eliminados de la versión actual.

A partir de estas medidas, el índice de madurez (IMS) se calcula de la siguiente forma:

$$IMS = \frac{M_T - (F_a + F_m + F_e)}{M_T}$$

Si el IMS se aproxima al valor 1, se dice que el producto comienza a estabilizarse, por lo que el esfuerzo de mantenimiento requerido será menor. Nótese que esta métrica requiere el registro histórico detallado de los cambios en el software, ya que se basa en contar los cambios entre versiones. Ésta es una función básica de la gestión de la configuración, que se verá más adelante en este libro (en el Capítulo 11 concretamente).

Métricas de calidad de la documentación

Aunque no existen muchas métricas conocidas a este respecto, es significativo para el mantenimiento de un sistema o componente software lo bien documentado que se encuentre. Una de las formas de documentación más habituales de los programas son los comentarios que se incluyen en el propio código. Para observar la densidad de comentarios que hay en el código hay que realizar una inspección del código fuente. Si el código fuente está realizado en Java, por ejemplo, una medida fácilmente obtenible es la que estudia la proporción de comentarios en formato javadoc por número de líneas de código significativas, entendiendo por significativas aquellas que no sean de comienzo o fin (llaves, en el caso de Java) ni comentarios:

$$\text{DensidadComentarios} = n^{\circ} \text{JavaDocs} / \text{LoC}$$

donde *LoC* es el número total de líneas de código (*Lines Of Code*). Asumiendo que los comentarios del código son relevantes, es decir, que tal y como se dijo en la Sección 6.6.1 explican ciertos aspectos del código que no se explican por sí solos, cuanto mayor sea la densidad de comentarios, más fácil de mantener será el software examinado.

Otras métricas que pueden resultar útiles son las métricas de la documentación, algunas de las cuales fueron estudiadas en el Capítulo 3. De entre ellas, son de particular interés las que permiten medir la legibilidad de los documentos. Se trata de métricas que provienen del campo de la lingüística y que no son, por tanto, específicas de la Ingeniería del Software. Remitimos al lector interesado en su estudio a la Sección 3.4.

Métricas de calidad del diseño

Robert Martin, en 1995, estableció un conjunto de métricas para evaluar la calidad de los diseños orientados a objetos en términos de la interdependencia entre subsistemas (por ejemplo, paquetes en Java). Se parte de la base de que, aunque las interrelaciones entre los subsistemas del diseño son necesarias, los diseños con subsistemas fuertemente interrelacionados son muy rígidos, poco reutilizables y difíciles de mantener. Martin propone una medida para cada subsistema, que llama inestabilidad (*I*):

$$I = \frac{C_r}{C_a + C_r}$$

donde C_r es el número de clases dentro del subsistema que dependen de clases de otros subsistemas y C_a es el número de clases de otros subsistemas que tienen una dependencia con clases del subsistema en estudio. La medida de inestabilidad está entre 0 y 1, siendo 0 la medida de la máxima estabilidad (y máxima facilidad de mantenimiento desde el punto de vista expuesto) y 1 la máxima inestabilidad. Por supuesto, no todos los subsistemas deben ser extremadamente estables, pues estaríamos ante un sistema que no admite cambios.

A partir de esta medida, Martin afirma que un subsistema será tanto más estable cuantas más clases abstractas incluya, ya que las clases abstractas pueden ampliarse y restringirse mediante la definición de diferentes clases concretas derivadas. Así, define una medida de la abstracción (*A*) como:

$$A = \frac{\text{Num_Clases_Abstractas}}{\text{Clases_Totales}}$$

esta métrica produce valores entre cero y uno, donde 0 indica completamente concreto y 1 completamente abstracto.

A partir de las métricas anteriores, Martin define un punto de equilibrio donde se sitúan aquellos diseños que no son demasiado abstractos ni demasiado inestables, pues tienen el número ideal de clases en relación al número de interdependencias.

Índice de facilidad de mantenimiento

Este índice mide la facilidad de mantenimiento del producto software considerado teniendo en cuenta que toda acción de mantenimiento puede dividirse en 3 tareas:

- Comprensión de los cambios que deben hacerse.
- Realización de las modificaciones necesarias.
- Pruebas de los cambios realizados.

Welker, en 1995, observó las tareas básicas de una acción de mantenimiento, y planteó este índice para intentar cuantificar la facilidad de mantenimiento de un sistema a partir de dichas tareas. Este concepto ha sido perfeccionado por múltiples autores a lo largo de los años. Es sin duda un buen ejemplo de planteamiento de un modelo teórico de métrica sobre un atributo difícil de medir.

El índice de facilidad de mantenimiento se modela mediante un polinomio donde se ponderan los diferentes elementos o tareas que influyen en la facilidad de mantenimiento, de la siguiente forma:

$$\begin{aligned} IFM = & 171 - 5.2 * \ln(\text{aveV}) \\ & - 0.23 * \text{aveV}(g') \\ & - 16.2 * \ln(\text{aveLOC}) \\ & + 50 * \text{seno}(\sqrt{2.4 * \text{perCM}}) \end{aligned}$$

Donde aveV es la media del volumen V por módulo según Halstead, $\text{aveV}(g')$ es la media de la complejidad ciclomática por módulo, aveLOC es la media del número de líneas de código por módulo y perCM es la media porcentual de líneas de código comentadas. Cuanto mayor sea este índice, más fácil de mantener será el sistema, si bien es importante remarcar que se trata de un modelo con parámetros ajustados empíricamente, y por tanto los coeficientes que aparecen en la ecuación anterior no son los únicos posibles.

8.8.2 Métricas relacionadas con el proceso

Estas métricas evalúan el proceso de fabricación de un producto software. Algunos ejemplos clásicos son el tiempo de desarrollo del producto, el esfuerzo que conlleva dicho desarrollo, el número y tipo de recursos empleados, el número de peticiones de acciones correctivas o el coste del proceso. El tiempo requerido para completar un proceso en particular (tiempo total del proyecto, por ingeniero, por actividad, etc.), es un indicador de la facilidad de mantenimiento del sistema muy a tener en cuenta. Cuanto mayor es el tiempo –total y por ingeniero– necesario para desarrollar un sistema, mayor será su complejidad y por lo tanto más difícil será de mantener, en general. De la misma manera, cuanto mayores sean los costes requeridos para su proceso de fabricación (esfuerzo en personas/día, costes de viajes, recursos de hardware), menor será su facilidad de mantenimiento.

Además de las anteriores, otras métricas que resultan interesantes son las métricas del número de defectos descubiertos durante la fase de pruebas y las métricas relacionadas con las reparaciones:

- El número de defectos encontrados durante la fase de pruebas una vez que el código está integrado está correlacionado positivamente con el número de defectos del software que se encontrarán durante su explotación. Un gran número de defectos durante las pruebas indica que durante la fase de desarrollo se han cometido muchos errores y, salvo que medie un gran esfuerzo en la fase de pruebas, parte de estos errores se arrastrarán hasta que el sistema esté en explotación.
- Métricas relacionadas con las reparaciones. Puede medirse la facilidad de mantenimiento de un software durante su proceso de construcción midiendo la frecuencia de fallos debidos a efectos laterales producidos después de una modificación (X):

$$X = \frac{1 - A}{B}$$

siendo A el número de fallos debidos a efectos laterales detectados y corregidos y B el número total de fallos corregidos. Cuanto mayor sea X , más difícil será de mantener el sistema en el futuro.

Existen, finalmente, un conjunto de métricas muy útiles que pueden obtenerse gracias al uso de herramientas que recolectan y procesan datos del desarrollo. El tiempo medio de reparación (MTTR, *Mean time to repair*) es tal vez la más común de ellas, el cual se define como el tiempo medio necesario para reparar un fallo y hacer que el sistema vuelva a funcionar en condiciones normales tal y como operaba antes del mismo. Existen dentro de este grupo métricas para obtener el tiempo medio de mantenimiento correctivo, preventivo o el tiempo máximo de mantenimiento correctivo, aunque en realidad no son sino medidas más detalladas y específicas con el mismo o muy similar ámbito de aplicación.

Otras, no tan ampliamente utilizadas, son el porcentaje de aislamiento de un único elemento (que se define como el porcentaje de tiempo que los fallos detectados pueden aislarse dentro de un único elemento reemplazable), la media de hombres/hora necesarios para una tarea de mantenimiento, los hombres/hora medios necesarios por acción de mantenimiento, el tiempo medio de mantenimiento activo, o el número medio de horas de mantenimiento por hora de operación, por citar sólo unas pocas.

8.9 Resumen

Habitualmente se entiende que las actividades de mantenimiento se producen después de la entrega del producto. No obstante, el cómo se diseña y construye un software afecta mucho a lo fácil o difícil que será mantenerlo después de la entrega. O dicho de otra forma, los productos tendrán diferentes grados de facilidad de mantenimiento.

Obtener una facilidad de mantenimiento elevada requiere atención durante el desarrollo del producto software, por ejemplo, poniendo un especial cuidado en la documentación del diseño, o haciendo ese diseño claro y flexible para que sea más fácilmente modificable en el futuro. Puesto que la facilidad de mantenimiento se debe tener en cuenta antes de la entrega del producto, hoy se considera que las actividades de mantenimiento también son de algún modo previas a la entrega. Como el diseño afecta a la facilidad de mantenimiento, las métricas de diseño sirven como medidas indirectas de la facilidad de mantenimiento, y se usan con frecuencia para ello, dado que hay pocas métricas propuestas cuyo enfoque sea directamente la facilidad de mantenimiento.

Hoy en día se asume que todo software tiene que cambiar con el tiempo. Esos cambios son de diferentes tipos, pero incluyen los cambios en la propia funcionalidad. Dicho de otra forma, el software evoluciona. Aunque diferentes tipos de software evolucionen de formas diferentes, hay algunas *leyes* que son comunes a todos ellos, y deben tenerse en cuenta al planificar la vida a medio y largo plazo de los productos software.

En ocasiones, es necesario investigar sobre la estructura interna de un software, debido a que no se tiene documentación de su diseño. A este proceso se le denomina *ingeniería inversa*. Frecuentemente la ingeniería inversa se combina con *reingeniería*, técnica que consiste en rehacer el diseño o, en escala más pequeña, reestructurar el diseño original. Esta ingeniería adicional tiene en ocasiones como motivo principal el hacer el software más fácil de mantener.

La siguiente figura resume los conceptos fundamentales del capítulo. En ella se muestra la importancia relativa de cada concepto tratado con respecto al conjunto total de conceptos que hemos estudiado en el capítulo.



Figura 8.5: Principales conceptos tratados en el capítulo

8.10 Notas bibliográficas

Existe una extensa literatura sobre mantenimiento del software, tanto sobre la gestión del propio proceso, como sobre las diferentes técnicas y herramienta para llevarlo a cabo. Para situar el mantenimiento del software en general y tener un primer contacto con los problemas a que se enfrentan quienes han de llevarlo a cabo, el estándar de mantenimiento del IEEE es una referencia esencial (IEEE, 1998c). Para conocer más sobre las diferentes clases de mantenimiento y profundizar en sus clasificaciones, merece la pena leer el artículo «*Types of software evolution and software maintenance*» (Chapin, Hale, Khan, Ramil y Tam, 2001), así como la clasificación de las diferentes categorías de mantenimiento definidas por Lientz, Swanson y Tompkins (Lientz, Swanson y Tompkins, 1978).

Sobre la gestión del proceso de mantenimiento, y su relación con el concepto de calidad y mejora continua, el libro «*Software Maintenance Management: Evaluation and Continuous Improvement*» (April y Abran, 2008) trata en profundidad la gestión del mantenimiento y propone tareas de mejora en forma de hojas de ruta para las organizaciones que se dedican al mantenimiento. Otra referencia clásica es «*Practical Software Maintenance: Best Practices for Managing Your Software Investment*» (Pigoski, 1996), donde se analiza con detalle todo aquello que es importante en términos de necesidades de mantenimiento, y se relatan interesantes experiencias prácticas extraídas de desarrollos reales.

En lo relativo a la evolución del software, la referencia fundamental es el trabajo de Lehman y Ramil (2003) en el que enuncian las leyes de la evolución del software. Para profundizar en el tema desde un punto de vista más técnico, el libro de Stanislaw Jarzabek «*Effective Software Maintenance and Evolution: A Reuse-Based Approach*» resultará sin duda de interés. En él se estudian con profundidad técnicas específicas para el mantenimiento y evolución del software, se presentan ejemplos en proyectos reales y en general se trata todo lo relacionado con el cambio en el software.

Para aquellos interesados en las técnicas de mantenimiento, específicamente en el *refactoring*, es muy recomendable la lectura del libro de Martin Fowler, Ken Beck y sus colaboradores sobre reestructuración del código (Fowler et al., 1999), que contiene técnicas para el mantenimiento del software sin cambios funcionales. En lo referente a ingeniería inversa de software, la referencia clásica es el artículo «*Reverse Engineering and Design Recovery: A Taxonomy*» de Chikofsky y Cross donde los autores definen el término y proporcionan una taxonomía de ingeniería inversa del software (Chikofsky y Cross, 1990). Para ver la técnica en funcionamiento, nada mejor que el caso de estudio que Byrne relata en «*Software reverse engineering: a case study*» (Byrne, 1991), donde se muestra paso a paso la extracción del diseño de un viejo programa en Fortran y su reimplementación en otro lenguaje.

Para saber más sobre las métricas relacionadas con el mantenimiento, se recomienda la lectura del artículo «*OO design quality metrics: An analysis of dependencies*» donde Robert Martin describe un conjunto de métricas para medir la calidad de los diseños orientados a objetos (Martin, 1995). Otra referencia interesante sobre el índice de facilidad de mantenimiento es «*Software maintainability metrics models in practice*» (Welker y Oman, 1995).

8.11 Cuestiones de autoevaluación

- 8.1 Razona si la siguiente sentencia es verdadera o falsa: «El mantenimiento correctivo del software consiste en la realización de cambios en el código fuente para implementar todas las solicitudes de cambio de los usuarios».
- R. Falso. *El mantenimiento correctivo se limita a los cambios necesarios derivados de defectos encontrados en el software, y no incluye ningún otro tipo de cambio.*
- 8.2 ¿Cuál es el objetivo de la reingeniería?
- R. *La reingeniería se aplica cuando una aplicación es rentable a la organización pero debe ser actualizada, bien en su funcionalidad, o bien porque su mantenimiento resulta complicado (porque su diseño original es malo, o por el deterioro producido por las actualizaciones, por ejemplo). Podemos, por tanto, afirmar que su objetivo principal es mejorar la facilidad de mantenimiento de un sistema software.*
- 8.3 Indique si la siguiente sentencia es verdadera o falsa y razona su respuesta: «El mantenimiento adaptativo, según Lientz y Swanson, incluye modificaciones para añadir funcionalidades al software provenientes de cambios en los requisitos de operación de las empresas».
- R. Falso. *El mantenimiento adaptativo en la definición de Lientz y Swanson hace referencia a la introducción de cambios en el software para adaptarlo a nuevas condiciones del entorno, pero en ningún caso incluye nuevas demandas funcionales.*
- 8.4 ¿Por qué la medida de inestabilidad del software tiene impacto en su facilidad de mantenimiento? Describa razonadamente si para obtener una aplicación fácil de mantener (en grado máximo) todos sus subsistemas deben ser extremadamente estables.
- R. *La inestabilidad refleja las interrelaciones de un subsistema con otros. Un valor 0 indica la máxima estabilidad del subsistema, y un valor igual a 1 la máxima inestabilidad. Cuanto más estable es un sistema menos interrelaciones hay entre paquetes y por lo tanto, más fácil es de comprender y documentar a la hora de hacer modificaciones del tipo que sean, lo que implica mayor facilidad de mantenimiento.*
- 8.5 Razona si es verdadera o falsa la siguiente afirmación: «La ingeniería inversa es el proceso por el cual se consigue un software mejor diseñado, mejor documentado o con menos fallos que el original».
- R. Falso. *El objetivo principal de la ingeniería inversa es obtener una especificación de mayor nivel de abstracción partiendo del código fuente (o de cualquier otro elemento resultado de las distintas fases del ciclo de vida del software). Para conseguir el objetivo afirmado en la sentencia es necesario aplicar un proceso de reingeniería, una de cuyas partes es la ingeniería inversa.*
- 8.6 Indique bajo qué condiciones es recomendable realizar un proceso de reingeniería y cuándo conviene construir una nueva aplicación.
- R. *La reingeniería se suele considerar una buena opción frente al desarrollo de una nueva aplicación en las circunstancias que se mencionan en el texto, que incluyen la existencia de fallos frecuentes en la aplicación que son difíciles de localizar, la existencia de dificultades para integrar la aplicación con otros sistemas, la no disponibilidad de personal suficiente para realizar todas las modificaciones necesarias que podrían surgir o cuando el mantenimiento de la aplicación consume muchos recursos (entre otros).*

- 8.7** ¿Es cierto que las técnicas de *refactoring* están pensadas para ajustar el código a una mejor documentación, obtenida en la fase de diseño, dentro de un proceso de reingeniería?
- R.** No, el refactoring fue concebido para «limpiar» el código sin necesidad de realizar previamente ninguna labor de ingeniería inversa o proceso de reingeniería. De hecho, como pasa con otras técnicas de reestructuración, se puede aplicar sobre productos de una fase del ciclo de vida del software sin necesidad de aplicar técnicas adicionales.
- 8.8** Como parte del capítulo hemos estudiado las denominadas leyes de la evolución del software, ¿recuerda a qué hace referencia la ley de complejidad creciente?
- R.** Esta ley establece que la complejidad de un software se incrementa a medida que éste evoluciona, a menos que se realice un trabajo específico de reducción o control de la complejidad. Según esta ley, se produce un aumento progresivo del esfuerzo de mantenimiento durante la vida del software, a menos que se realice mantenimiento perfectivo para paliarlo.
- 8.9** Razone por qué las medidas M_T (número de módulos en la versión actual), F_m (número de módulos en la versión actual que han sido modificados), F_a (número de módulos en la versión actual que han sido añadidos) y F_e (número de módulos de la versión anterior que se han eliminado en la versión actual) influyen en el índice de madurez del software y, por lo tanto, en la facilidad de mantenimiento del sistema.
- R.** Estas medidas hacen referencia al nivel de inestabilidad debida a las modificaciones. Es por eso que su interrelación según la fórmula $IMS = \frac{M_T - (F_m + F_a + F_e)}{M_T}$ da como resultado la medida de la estabilidad del software. Cuanto más estable, más maduro es, puesto que esto indica que el software no ha tenido que ser modificado en su versión actual. Cuanto más maduro es el software, más fácil de mantener resulta.
- 8.10** Si un software se diseña con una buena facilidad de mantenimiento, ¿cree que tendrá menor propensión a cambiar?
- R.** En absoluto. Muchos cambios en el software se originan simplemente por la evolución de las necesidades de los usuarios. Si el software es más fácil de mantener, estos cambios serán menos costosos y más fiables, simplemente.

8.12 Ejercicios y actividades propuestas

8.12.1 Ejercicios resueltos

- 8.1** El repositorio Sourceforge (<http://sourceforge.net/>) es una plataforma para la gestión de proyectos de código fuente abierto que ofrece posibilidades únicas para analizar código creado por otras personas. Tras acceder a un cierto proyecto, se ha detectado la necesidad de aplicar algún tipo de reestructuración que mejore su código facilitando su mantenimiento futuro.
- Se trata en concreto de una aplicación que implementa un juego sobre un tablero de 8×8 donde cada casilla tiene un valor, diferenciándose con letras los diferentes tipos de casillas (A, X, C, P, etc.). La clase RatingBasedAlgorithm genera un conjunto de posibles movimientos en función del tipo de casilla, para lo cual necesita conocer exactamente de qué tipo de casilla se trata. Observando dicha clase, se detecta código duplicado (en realidad lo hay en varias clases, pero nos detendremos en ésta). Observemos en particular los métodos `isA`, `isX`, `isC` e `isP`, que determinan el tipo de una casilla:

```

private boolean isX(Square s) {
    int r = s.getRow();
    int c = s.getColumn();
    return ((c == 1 && r == 1) ||
            (c == 1 && r == 6) ||
            (c == 6 && r == 1) ||
            (c == 6 && r == 6));
}

private boolean isA(Square s) {
    int r = s.getRow();
    int c = s.getColumn();
    return ((c == 3 && r == 3) ||
            (c == 3 && r == 4) ||
            (c == 4 && r == 3) ||
            (c == 4 && r == 4));
}

private boolean isP(Square s) {
    int r = s.getRow();
    int c = s.getColumn();
    return ((c == 2 && r == 2) ||
            (c == 2 && r == 5) ||
            (c == 5 && r == 2) ||
            (c == 5 && r == 5));
}

private boolean isC(Square s) {
    int r = s.getRow();
    int c = s.getColumn();
    return ((c == 0 && r == 0) ||
            (c == 0 && r == 7) ||
            (c == 7 && r == 0) ||
            (c == 7 && r == 7));
}

```

A la vista del código de estos métodos, utilice la técnica de *refactoring «extraer método»* para extraer la parte común y crear un nuevo método.

Solución propuesta: Utilizando la técnica sugerida, se extrae la parte común con la que se crea un nuevo método al que denominaremos `testPoints`, como se muestra:

```

private boolean testPoints(Square s, int p1, int p2) {
    int r = s.getRow();
    int c = s.getColumn();
    return ((c == p1 && r == p1) ||
            (c == p1 && r == p2) ||
            (c == p2 && r == p1) ||
            (c == p2 && r == p2));
}

```

Una vez hecho esto, los métodos `isA`, `isL`, `isC` e `isP`, etc. quedan como sigue:

```
private boolean isK(Square s) {
    return testPoints(s, 1, 6);
}
private boolean isA(Square s) {
    return testPoints(s, 3, 4);
}
private boolean isP(Square s) {
    return testPoints(s, 2, 5);
}
private boolean isC(Square s) {
    return testPoints(s, 0, 7);
}
```

La aplicación de esta técnica permite, al separar funcionalmente las partes comunes, obtener una importante ganancia en legibilidad del código, así como un incremento en su facilidad de mantenimiento y reutilización.

8.2 El contrato de mantenimiento del software para la gestión automatizada de una biblioteca incluye como resumen el siguiente párrafo: «El contrato de mantenimiento garantiza la prestación de los siguientes servicios:

- Revisiones del producto completamente probadas para las plataformas contratadas.
- Soporte técnico telefónico con acceso a desarrolladores del producto.
- Notas de cambios en las diferentes revisiones con todos los cambios realizados entre revisiones.
- Correcciones al código (parches) a medida que se van detectando y corrigiendo los defectos.
- Servicio de análisis e información sobre defectos encontrados.»

Teniendo en cuenta que el producto puede comprarse para una o varias plataformas (Linux, Windows, etc.), y de acuerdo con la clasificación de Chapin vista en el capítulo, ¿qué tipos de mantenimiento incluye el contrato?

Solución propuesta: En primer lugar, se proporciona mantenimiento de carácter *conservativo* ya que hay soporte técnico. No parece que ese soporte se extienda a evaluaciones del producto para propósitos específicos, y parece claro que no incluye formación de usuarios.

Por otro lado, el contrato proporciona acceso a los resultados del mantenimiento *correctivo*, ya que permite informar de posibles defectos y el usuario se beneficia de recibir de manera continua la corrección de esos defectos en la forma de parches. El concepto de revisión del producto no se especifica si incluye o no actualizaciones funcionales, por lo que no puede considerarse que se esté contratando la evolución funcional del producto.

Las notas de las revisiones pueden considerarse un tipo concreto de revisión para actualizar la documentación, ya que indican los cambios a la última versión actualizada del código.

Por otro lado, parece claro que los posibles beneficios de un mantenimiento adaptativo para nuevas plataformas quedan excluidos del contrato, ya que la política de licencias es que se debe pagar aparte por cada plataforma en la que se quiere utilizar el producto.

- 8.3** En una empresa de software se ha desarrollado una aplicación comercial de control de flota denominada FLT, la cual ha sido implementada en lenguaje Java. Se han recopilado datos durante los cinco primeros años de comercialización del software, que han dado lugar a 6 versiones del producto. La siguiente tabla muestra los datos de cambio entre versiones.

Versión	Clases Java	Clases añadidas	Clases eliminadas	Clases modificadas
1	3.763	0	0	0
2	4.213	1.450	1.000	360
3	4.402	516	527	243
4	4.958	686	130	145
5	5.282	489	165	140
6	5.496	432	36	138

¿Qué podemos afirmar sobre la estabilidad del software FLT a la luz de los datos recopilados?

Solución propuesta: Si calculamos el índice de madurez de este software (IMS) para las diferentes versiones (menos la primera, para la que el concepto de cambio sobre la versión anterior no es aplicable), tendremos la siguiente tabla:

Versión	IMS
1	-
2	0.333017
3	0.70786
4	0.806172
5	0.849678
6	0.889738

En primer lugar, hay que considerar que las versiones se han sucedido en cinco años, lo cual es un intervalo muy corto para seis versiones, algo que ya de por sí indica que ha habido cambios importantes. Si nos fijamos en los IMS de la tabla, encontramos en primer lugar que el paso de la versión 1 a la 2 tuvo un impacto de reestructuración muy grande en el código, lo cual se refleja en un IMS muy bajo. Esto pudo ser debido a que se encontraron defectos graves o se tuvo que hacer un cambio funcional de gran impacto. El siguiente salto de versión siguió teniendo un impacto importante, pero ya más reducido. Por el número de clases añadidas y eliminadas, podría ser debido a la sustitución de una parte del código. A partir del siguiente salto de versión, las diferencias se van reduciendo, y parecen deberse más a incrementos funcionales progresivos que a reestructuraciones, dado que el número de clases añadidas es superior al número de clases eliminadas o modificadas.

- 8.4** Una cierta compañía ha desarrollado diferentes productos software, los cuales han sido implantados con éxito en diferentes clientes a los que da soporte. La solicitud cada vez más frecuente por parte de los clientes de cambios y reparaciones en uno de sus productos hace a la compañía replantearse la aplicación de alguna de las técnicas de mantenimiento del software vistas en el capítulo. Para ello, se desea estudiar la calidad del diseño mediante las métricas de Martin. Así, el software está formado por 25 paquetes, si bien el estudio se centrará en los dos paquetes que incluyen las funcionalidades centrales. El paquete P1 está formado por 8 clases, de las cuales 3 dependen de clases de otros subsistemas. Analizando el resto del código, se han identificado 12 clases que dependen de las clases de P1. El paquete P2, por su parte, está formado por 13 clases, de las cuales 4 dependen de clases de otros subsistemas. Analizando el resto del código, se han identificado 7 clases que dependen de las clases de P2. Evalúe la estabilidad del diseño de ambos paquetes y discuta los resultados.

Solución propuesta: Según Robert Martin, es posible evaluar la calidad de los diseños orientados a objetos en términos de la interdependencia entre subsistemas. La inestabilidad (I) de Martin se calcula según la siguiente fórmula:

$$I = \frac{C_r}{C_a + C_r}$$

en nuestro caso haremos de evaluar dicha inestabilidad de manera separada para P1 y P2. En el caso de P1, C_r (número de clases de P1 que dependen de clases de otros paquetes) es 3, mientras que C_a (número de clases de otros paquetes que tienen una dependencia con clases de P1) es 12. Así:

$$I_{P1} = \frac{3}{3+12} = 0,2$$

En el caso del paquete P2:

$$I_{P2} = \frac{4}{4+7} = 0,36$$

Ambos paquetes son bastante estables, con valores de inestabilidad más cercanos a cero que a uno. Esto indica que, por lo que respecta a la inestabilidad al menos, el diseño no parece necesitar de grandes modificaciones. Obviamente, los resultados de P1 y P2 no son inmediatamente extrapolables a todos los paquetes que conforman el software en estudio. Para poder aplicar la afirmación anterior al software en su conjunto con total confianza, habría que extraer una muestra mayor de paquetes del sistema y asegurar que dicha muestra es significativa.

8.12.2 Actividades propuestas

- 8.1** Con el fin de usar herramientas y métricas relacionadas con el mantenimiento, se pide:

- Elegir un software de Sourceforge. El producto elegido puede ser de cualquier tipo, siempre que el lenguaje sea Java, y el producto se componga de más de 100 clases o interfaces. Se debe seleccionar un software estable (en Sourceforge hay indicadores para esto, como Development Status 5-Production/Stable) y que esté documentado de manera razonable.

- Tomar métricas de orientación a objetos que pueden obtenerse automáticamente con la herramienta ejgen para métricas Java de Chidamber y Kemerer (entre otras).
 - Elaborar un breve informe a partir de las métricas anteriores y la inspección hecha al código sobre: (a) los módulos «problemáticos» desde el punto de vista del mantenimiento y (b) las mejoras de la facilidad de mantenimiento que podrían hacerse al software.
- 8.2** Con el objetivo de examinar una herramienta de ingeniería inversa de código abierto como Netbeans, proponemos llevar a cabo la ingeniería inversa de un proyecto Java. Para hacerlo, en Netbeans tenemos que, a partir del proyecto original, crear otro proyecto de tipo «UML - Reverse engineer a Java project».

Al crearlo tendremos que especificar la ubicación de los ficheros fuente que se deberán procesar. Si no se quieren procesar todos, se puede reestructurar el código en varias carpetas y seleccionar sólo la parte del código que nos interese.

Después de la operación de ingeniería inversa, tendremos todos los elementos detectados bajo la opción *Model* en el proyecto UML, incluyendo también las clases de las bibliotecas estándar Java (que normalmente no interesan para la creación de diagramas).

En el proyecto UML podemos seleccionar del repositorio *Model* diferentes clases, paquetes o interfaces y podemos crear diagramas a partir de ellas con el menú contextual, usando la opción «*Create Diagram from Selected Elements*», como se muestra en la Figura 8.6.



Figura 8.6: Creación de diagramas para ingeniería inversa en NetBeans

La extensión de Netbeans es fácil de utilizar. Para practicar con ella se pide lo siguiente:

- Hacer la ingeniería inversa para una parte del código del software seleccionado en el ejercicio anterior.

- Tomar una clase que tenga bastantes dependencias y crear el diagrama de clases del paquete en el que se encuentre.
- Tomar uno de los métodos de esa clase y generar un diagrama de secuencia que muestre la interacción entre esa clase y las clases dependientes.

La integración del código y los modelos en Netbeans nos permiten en todo momento navegar desde los diagramas UML al código Java correspondiente. Este tipo de herramientas permite navegar el código de manera visual a través de los diagramas, permitiendo centrarnos en ciertas partes o relaciones de los mismos.

8.3 Suponga que empieza usted a trabajar en una empresa y, como parte de su trabajo, se le solicita estudiar el modo de crear productos con una mayor facilidad de mantenimiento. Utilizando los conocimientos adquiridos, diseñe una estrategia que incluya métricas de mantenimiento para alcanzar el objetivo que se le ha encomendado.

8.4 Indique, a la vista del siguiente fragmento de un contrato de mantenimiento de software, qué tipos de mantenimiento cubre y cuáles no:

Los servicios de mantenimiento incluyen:

- La prestación de asesoramiento sobre el uso del software por correo electrónico con carácter prioritario.
- El diagnóstico de fallos en el software, e instrucciones para su rectificación.
- La creación y envío al cliente de entregas de mantenimiento.
- Responder al cliente dentro de 2 días hábiles a partir de una solicitud de servicios. La respuesta deberá incluir un primer análisis de los informes de fallos. Posteriormente, se proporcionará una rectificación a los fallos tan pronto como sea posible.

Los servicios no incluyen el diagnóstico y rectificación de fallos derivados de:

- La modificación del software o su fusión (en todo o en parte) con cualquier otro software, salvo aquellos permitidos por la licencia.
- El uso indebido o negligente del software o el equipo en el que se ejecute.
- Cualquier ajuste o alteración de cualquier reparación o modificación del software hecha por personas ajena a la compañía que desarrolló el software.
- La pérdida o daños causados directa o indirectamente por error u omisión del usuario.

8.5 En el artículo «*Problems in application software maintenance*» (Lientz y Swanson, 1981) se relata un estudio llevado a cabo en cerca de 500 organizaciones acerca de los problemas de mantenimiento de las aplicaciones software. El estudio permitió identificar seis fuentes de problemas: el conocimiento del usuario, la eficacia de los programadores, la calidad del producto, la disponibilidad de tiempo del programador, los requisitos de la máquina, y la fiabilidad del sistema. Además, se llegó a la conclusión de que los problemas de eficacia de los programadores y de calidad del producto eran mayores en sistemas más grandes y antiguos, en los que se dedica más esfuerzo al mantenimiento correctivo. Lea el artículo y reflexione sobre la vigencia hoy día de las conclusiones a que llegaron los autores.

- 8.6** El «*Journal of Software Maintenance*», publicado por John Wiley & Sons, es la única revista científica totalmente dedicada al mantenimiento de software. En ella aparecen regularmente artículos que constituyen la punta de lanza de la investigación en el área. Acuda a una biblioteca con servicio de acceso a publicaciones científicas especializadas y consulte los últimos dos números de la revista. Elabore una tabla comparativa que incluya el porcentaje de artículos que tratan temas expuestos en el capítulo, y destaque las novedades en el campo del mantenimiento del software.
- 8.7** En un intento de conocer más en profundidad sus técnicas y modos de trabajo, el *Software Engineering Institute* de la Universidad Carnegie Mellon entrevistó a diferentes personas involucradas en 8 proyectos de mantenimiento de software dentro de una agencia del Gobierno de los Estados Unidos. Sus resultados se plasmaron en el informe CMU/SEI-93-TR-008 (Dart, Christie y Brown, 1993). Las entrevistas realizadas pusieron de relieve los problemas típicos de muchas organizaciones de mantenimiento de software, particularmente la falta de una filosofía de diseño para el mantenimiento durante la fase de desarrollo. Este informe destaca las conclusiones de estas entrevistas, y presenta recomendaciones para la mejora del proceso de desarrollo. Lea el informe y discuta en grupo algún conjunto de recomendaciones que le parezca especialmente interesante.
- 8.8** En un famoso artículo titulado «*Software aging*» (Parnas, 1994), se comentan informalmente los fundamentales aspectos del mantenimiento del software, utilizando como metáfora el envejecimiento humano. Léalo y describa la problemática de las revisiones del diseño en relación al envejecimiento del software. La situación descrita por el autor ¿sigue siendo actual? Discuta en grupo a este respecto.
- 8.9** Acceda a la página web sobre *refactoring* que mantiene Martin Fowler y consulte de primera mano el catálogo de técnicas de refactoring que este importante autor ha publicado. Aplique alguna de ellas al código de proyectos de código fuente abierto publicados en Sourceforge y anote los más comunes errores detectados.
- 8.10** Acceda al repositorio Sourceforge y elija un proyecto que haya sido desarrollado en Java o en cualquier otro lenguaje orientado a objetos. Evalúe las métricas de Martin para dicho software y extraiga conclusiones sobre la calidad de su diseño. A la vista de dichas conclusiones, proponga alguna técnica de mantenimiento que sería aplicable para mejorar, en caso de ser necesario, al diseño del software.

Parte III

Gestión y Calidad en la Ingeniería del Software

Introducción a la Tercera Parte

Esta tercera y última parte del libro retoma la visión general de la Ingeniería del Software que se introdujo en la primera parte, desde la perspectiva de la gestión. Toda actividad de gestión se basa en unos objetivos. Objetivos que, desde una perspectiva de negocio, habrían de estar alineados con los objetivos del negocio. Así, el software es un medio para un fin y como tal resulta necesario que sea al menos suficientemente bueno para cumplir su misión. Juzgar si un producto es bueno es hablar de su calidad, y a esto se dedica el primero de los capítulos de esta tercera parte. Es importante entender que los productos buenos provienen habitualmente de procesos buenos, por lo que la calidad tiene una apariencia bifrontal, la del proceso y la del producto. En el capítulo dedicado a la calidad se examinan ambos aspectos y se discuten los principales estándares y modelos relacionados con la calidad del producto o del proceso.

El segundo capítulo en esta parte (Capítulo 10) gira la atención a cómo gestionar los recursos dentro de un proyecto para conseguir la calidad requerida dentro de unos plazos económicamente admisibles, tanto en coste como en calendario. La gestión del proyecto software es independiente del tipo de proceso utilizado, así como de los métodos y herramientas seleccionados. Y la ingeniería no se hace en el contexto de un mundo ideal de recursos ilimitados: los recursos son escasos, por lo que deben gestionarse, controlarse y seguirse con cuidado. Dado que los proyectos software generan multitud de artefactos (documentos, código y otros), de forma muy rápida y casi siempre en grupos de ingenieros, es de capital importancia la gestión y control de todos esos productos intermedios y sus relaciones. De esto se encarga la gestión de la configuración, que es objeto del tercer capítulo en este bloque.

El bloque termina con un capítulo dedicado a las herramientas. El software que soporta la producción de software, aun no siendo más que un medio para facilitar las actividades, puede marcar la diferencia en cuanto a la eficacia o productividad de un equipo de desarrollo. Por ello, merece un tratamiento separado, con el objeto de entender su importancia, sus tipos, y los criterios para su selección.

Calidad

Mirando la Ingeniería del Software desde una perspectiva histórica, hasta la década de los años 1960 estaríamos en la era funcional, los años 70 fueron la era de la planificación, los 80 la era de los costes y a partir de los 90 la era de la calidad y la eficiencia.

— Stephen H. Kan

9.1 La especial naturaleza de la calidad

En España es muy común organizar una reunión social en casa con la excusa de compartir una paella. Generalmente son los anfitriones quienes personalmente se encargan de cocinar, y se da por supuesto que su deseo y satisfacción reside en que sus invitados degusten una paella de la más alta calidad. Suponga que usted ha organizado un evento así, ¿qué haría para conseguir una paella de calidad con la que obsequiar a sus amistades? Seguramente lo primero que vendrá a su cabeza es que si utiliza ingredientes mediocres, la paella no será todo lo buena que le gustaría, por lo que acudiría al mercado y compraría los mejores ingredientes: marisco fresco, arroz con denominación de origen, azafrán manchego, verduras ecológicas, aceite de oliva virgen extra, etc. Sin embargo, ¿será esto suficiente para garantizar la calidad de su paella? Mucho nos tememos que no: si usted no es un experto cocinero (darle el punto preciso a una paella no es algo trivial) o si durante el proceso de cocinado recibe una llamada de teléfono por la que desatiende la paella el tiempo suficiente como para dejar que se queme, la paella resultante no tendrá la calidad mínima que sus invitados merecen. Y ello a pesar de haber empleado los ingredientes de más alta calidad en su elaboración. Del mismo modo, una paella cocinada por un experto cocinero, pero cuyos ingredientes no cumplan con un mínimo de calidad, será igualmente decepcionante. Es la combinación de ambas cosas, utilización de productos de calidad y proceso de preparación adecuado, la única receta que garantiza el éxito.

El software, como producto elaborado que es, se construye de acuerdo a procesos preestablecidos y controlados, en cuya producción se emplean «ingredientes humanos», tecnológicos, etc. Por ello, la producción de software de calidad debe seguir una receta similar a la de cualquier otro producto: buenos ingredientes, construidos, ensamblados y verificados en un proceso de calidad. Sólo esto garantiza la calidad del producto final.

La calidad es un término del que todos tenemos una noción clara, y sin embargo nos resulta difícil definirla con palabras. Tiene además muchos aspectos, pues es un concepto aplicable a prácticamente cualquier objeto, proceso o acción. Así, nos referimos a una tela de calidad, a una enseñanza de calidad, e incluso, a una jugada de fútbol de gran calidad individual. Existen agencias dedicadas a velar por la calidad, a certificarla o a acreditar que un proceso o producto cumple unos ciertos parámetros de calidad pero... ¿qué es en realidad la calidad? La definición de Raymond Paul es una referencia importante a la hora de estudiar este concepto desde la perspectiva de la Ingeniería del Software: *«la característica que distingue el grado de excelencia o superioridad de un proceso, producto o servicio»*.

Una forma sencilla de evaluar la calidad de un producto es identificar aquellos atributos que diferencian a un objeto de calidad de uno que no la tiene. Si el producto en cuestión posee dichos atributos, entonces se considera que tiene calidad y en caso contrario, se considera que no la tiene. Así, cuando hablamos del software, la calidad se identifica a menudo con la ausencia de fallos. Sin embargo, esta definición es bastante laxa, puesto que hay fallos que estamos dispuestos a tolerar, por ejemplo, en un navegador web, pero que consideraríamos inaceptables en un software que controlase procesos críticos tales como el sistema de control de una aeronave comercial. Éste es en realidad el enfoque de algunos de los modelos de calidad del software que estudiaremos en el capítulo.

A lo largo de las diferentes secciones analizaremos cómo se entiende el concepto de calidad desde la perspectiva de la Ingeniería del Software. Como veremos, éste no se limita a la noción comentada de calificativo aplicable al producto resultante de un desarrollo, sino también al propio proceso de ingeniería y a la organización que lo lleva a cabo.

9.2 Objetivos

El objetivo general de este capítulo es presentar el concepto de calidad y analizar su importancia en el desarrollo de software. Así, tras la lectura de este capítulo el lector debería:

- Conocer los conceptos fundamentales de calidad según la perspectiva de la Ingeniería del Software.
- Conocer los diferentes modelos de calidad del software.
- Saber qué es un modelo de evaluación y mejora de procesos y los desafíos que su implantación plantea.
- Conocer los estándares vigentes sobre calidad del software, así como también otros modelos y especificaciones de actualidad relacionados con la misma.

9.3 Introducción

En la entrada de este capítulo hemos reflexionado sobre el concepto de calidad en general, como característica deseable, como calificativo de un producto, de un proceso o de una acción. Durante esa discusión se hizo referencia a los atributos de calidad de un producto software. Ahora que vamos a comenzar su estudio desde la perspectiva de la Ingeniería del Software, es necesario comenzar reconociendo los múltiples aspectos con los que está relacionado el término *calidad*. La guía SWEBOK indica expresamente que los ingenieros del software deben conocer el significado y características de la calidad, así como su valor en el desarrollo y mantenimiento del software.

En el desarrollo de un sistema de software, la calidad aparece por vez primera en los requisitos, que es donde se establecen los parámetros y criterios de calidad del software que se construirá. Las características de calidad que se definan en este momento serán la referencia de ahí en adelante, por lo que todo aquello que se establezca como requisito de calidad en este punto tendrá una enorme influencia, tanto en la forma en que posteriormente se medirá la calidad, como en los criterios utilizados para evaluar si los parámetros de calidad establecidos se cumplieron o no al final del desarrollo. La Figura 9.1 muestra los diferentes aspectos de la calidad desde el punto de vista de la Ingeniería del Sofwtare. A continuación estudiaremos cada uno de ellos con detalle.



Figura 9.1: Los múltiples aspectos de la calidad en la Ingeniería del Software

9.3.1 Cultura y ética de la calidad

En su libro «*Creación de una cultura de la Ingeniería del Software*», Karl Wiegers hace un detallado catálogo de aquellos elementos que contribuyen al éxito de un proyecto de software, poniendo un énfasis especial en la mejora de la cultura de la calidad en una organización. Esta cultura de la Ingeniería del Software se podría definir como el compromiso de los ingenieros del software de una organización con las metas de calidad de su organización y particularmente, con aquellas que tienen que ver con la obtención de software de calidad.

La asimilación de esta cultura de calidad por parte de los ingenieros del software se plasma en una serie de prácticas que en conjunto se constituyen en lo que Wiegers (1996) denomina *cultura de calidad* de la organización. Según este autor, algunos de los más importantes principios a tener en cuenta son:

- Nunca deje a su jefe o cliente hablarle de hacer mal un trabajo.
- La gente necesita sentir que su trabajo es apreciado.
- La formación continua es responsabilidad de todos los miembros del equipo.
- El factor más importante en la calidad del software es que el cliente esté implicado en el desarrollo.
- Su mayor reto es tener el mismo concepto del producto final que su cliente.
- La mejora continua del proceso de desarrollo no sólo es posible: es esencial.
- Los procedimientos escritos para el desarrollo de software pueden ayudar a construir una cultura compartida de buenas prácticas.
- La calidad es la prioridad suprema. La productividad a largo plazo sólo es una consecuencia natural de la alta calidad.
- Luche porque los errores sean detectados por sus compañeros y no por los clientes.
- Una de las claves de la calidad del software es iterar muchas veces en todos los pasos del desarrollo excepto en uno, en la codificación, que sólo debe hacerse una vez.
- Gestionar correctamente los informes de defectos y las solicitudes de cambios es esencial para controlar la calidad y el mantenimiento.
- Si mide lo que hace, aprenderá a hacerlo mejor.
- No podrá cambiar todo a la vez. Identifique aquellos cambios que aportarán mayores beneficios y comience a aplicarlos hoy mismo.
- Haga aquello que tenga sentido y huya de los dogmas.

Además de la importancia que tiene ser conscientes de lo que significa la cultura de la Ingeniería del Software, las consideraciones éticas también tienen un papel en la calidad del software. Hoy en día, prácticamente todas las profesiones tienen un código deontológico para regular la ética de la profesión, inspirar buenas conductas y expresar los ideales a los que se debe aspirar. En las ciencias de la computación y la Ingeniería del Software, las dos asociaciones más relevantes y con mayor número de miembros (la IEEE Computer Society y la ACM) han desarrollado conjuntamente un código deontológico que enumera los principios morales básicos, así como los derechos y responsabilidades de los profesionales

del área. En uno de los principios de dicho código se hace mención expresa a la calidad, con un texto que reza lo siguiente:

Promover la máxima calidad, un coste aceptable y un plazo razonable, garantizando que los compromisos significativos al respecto quedan claros, que el empresario y el cliente los aceptan y que están disponibles para consideración del usuario y del público en general.

Si bien no se trata de la única mención dentro de este texto, pues cuando se abordan las cualidades personales o las de gestión también se hacen referencias expresas a la calidad, lo cual da una idea de la importancia de la calidad como parte de las buenas prácticas y la ética personal y profesional. Para aquellos lectores interesados en el tema, recomendamos la lectura completa del código deontológico cuya referencia aparece al final de este capítulo.

9.3.2 Valor y costes de la calidad

En general la obtención de un producto de calidad es, como hemos mencionado ya, positiva. Sin embargo, hemos de ser conscientes de que si bien existen unos beneficios claramente asociados con la obtención de un software de calidad, también hay costes asociados que deben asumirse. Así, el coste de la calidad tiene diferentes componentes:

- Costes de prevención: son los derivados de controlar que el proceso de producción se atenga a los criterios de calidad establecidos y prevenir la aparición de defectos. Dichos costes se asocian a la planificación de las actividades de calidad, a la formación de nuevos integrantes del equipo de desarrolladores o a la realización de informes de calidad del producto que está siendo desarrollado, entre otros.
- Costes de evaluación de la calidad: asociados con las actividades de verificación de la calidad a llevar a cabo. Entran en esta categoría los costes para verificar que lo obtenido cumple los requisitos de calidad establecidos al comienzo del proceso, los de inspección del producto entregado, los del mantenimiento de los equipos de pruebas, etc.
- Coste de los fallos internos: producidos como consecuencia de las tareas de detección y reparación de defectos internos del software. Se denomina *defectos internos* a aquellos que existen en el software antes de que éste sea entregado a los usuarios. Si bien estos defectos no son perceptibles por el usuario, no por ello son menos importantes, ya que su no reparación podría ocasionar fallos externos. Algunos factores de coste relacionados son la corrección de errores en el producto para hacerlo utilizable, o el coste de revisar un producto reparado para ver si está efectivamente libre de errores.
- Coste de los fallos externos: son costes derivados de corregir los fallos encontrados en el software después de entregado al cliente. Los factores de coste más habituales son los derivados del análisis del software que desencadena una reclamación por parte

del cliente, los costes de restitución o reemplazo del producto y el coste asociado a los servicios y reparaciones efectuadas durante el periodo de garantía.

Algunos autores han estimado todos estos costes cuantitativamente. Un ejemplo de ello es el del modelo Six Sigma, que trataremos en la Sección 9.5.5, donde se emplea la siguiente fórmula para calcular los costes de la calidad:

$$Y_{qc} = \frac{(X_{md} + X_{fd} + X_{rc}) + (X_{dpc} + X_{ac})}{X_{cmc}}$$

donde Y_{qc} representa el coste de la calidad y X_{md} , X_{fd} , etc., representan los diferentes factores de coste enumerados más arriba (prevención, evaluación, fallos internos y fallos externos).

A pesar de los costes que conlleva, el valor que proporciona la calidad a una organización es difícilmente evaluable. Subjetivamente puede decirse que los costes merecen la pena dado el valor que la calidad aporta, aunque esta percepción subjetiva no puede ser suficiente respaldo para una organización que pretende desarrollar software de manera profesional y competitiva. Existen varios modelos que han tratado de determinar cuál es el impacto, en términos de valor añadido, que la calidad tiene en una organización. Uno de los más ampliamente difundidos es el denominado *modelo de cuatro dimensiones*, que basa la calidad en el estudio de los siguientes cuatro aspectos de negocio:

- **El cliente.** Es importante, ya que el objetivo fundamental es su plena satisfacción. Si no se consigue dicha satisfacción, el software habrá fracasado.
- **Los procesos.** La mejor forma de obtener productos de calidad es implementar procesos productivos que tengan la calidad presente, por lo que este aspecto resulta clave.
- **Valoración de los recursos disponibles.** Las personas de la organización que participan en un desarrollo no deben verse como mera fuerza laboral, sino también como una importante fuente de ideas para la búsqueda constante de la calidad.
- **Adaptación al cambio.** Los cambios que se produzcan tanto en la organización como en el proceso de producción deben ser asumidos rápidamente, intentando que tengan el menor impacto posible en el mantenimiento de los niveles de calidad.

La motivación de los desarrollos de software es, fundamentalmente, crear productos que proporcionen valor a los clientes. Si bien, como hemos visto, la incorporación de la calidad a la producción de software conlleva unos costes, hoy en día está ampliamente aceptado el hecho de que los beneficios compensan con creces dichos costes.

9.3.3 Los múltiples aspectos de la calidad

Uno de los más influyentes estudiosos de la calidad del software, David Garvin, afirmó en 1982 que la calidad es un concepto complejo y con diferentes facetas, y que en consecuencia debe ser estudiado desde diferentes perspectivas. Tras analizar campos tan diferentes como

la filosofía, la economía o el marketing. Garvin identificó las siguientes cinco perspectivas desde las cuales la calidad puede ser definida y entendida:

- La visión *trascendental* de la calidad, también denominada *calidad relativa*. Hace referencia al hecho de que la calidad es fácil de percibir y reconocer, pero difícil de definir. Según esta perspectiva, todos tenemos un concepto similar de lo que es la calidad del software, algo así como un ideal que habría que alcanzar. Se reconoce, no obstante, que es difícil que el software, una vez construido, tenga la perfección de un software ideal que sirviese al mismo fin.
- La perspectiva *del usuario*, según la cual la calidad se entiende como conformidad con aquello que el cliente espera recibir y que fue establecido en las especificaciones del software. A diferencia de la perspectiva anterior, la perspectiva del usuario permite medir la calidad en términos concretos: cuanto mayor sea el grado de cercanía entre las necesidades de los usuarios y las características proporcionadas por el software, mayor será su calidad. Nótese cómo esta visión de la calidad varía en función del contexto, ya que el mismo producto, dirigido a diferentes mercados o tipos de usuarios, tendrá diferentes percepciones de calidad global.
- La perspectiva *de la producción* identifica la calidad del producto con la calidad de los procesos de producción y post-venta. Según esta perspectiva, todo producto fabricado de acuerdo con estándares regulados de calidad podrá ser considerado un producto de calidad, posiblemente mejor que otros que no hayan sido fabricados según este tipo de criterios. Ésta es la visión de la calidad del estándar ISO 9001 y del modelo de madurez CMM, que veremos más en detalle en secciones subsiguientes.
- La perspectiva *del producto* relaciona la calidad con ciertas características de éste, tales como la facilidad de mantenimiento, la funcionalidad o su fiabilidad. A diferencia de las anteriores, que observan la calidad del software tal y como es percibida exteriormente, esta perspectiva apunta la calidad interna del software. Es la perspectiva que recoge, por ejemplo, el estándar IEEE 1061-1992, donde se enuncian un conjunto de atributos a estudiar en el software construido.
- La perspectiva *del valor* establece una relación entre la cantidad de dinero que el cliente está dispuesto a pagar y la calidad del producto. Se trata de una forma pragmática de entender la calidad, pues llegado un punto donde existe un conflicto entre lo que el cliente está dispuesto a pagar y el coste real de lo que solicita, los gestores del desarrollo tendrán que decidir qué nivel de calidad puede implementarse para satisfacer las necesidades del cliente, pero teniendo en cuenta que dicho cliente no está dispuesto a asumir los costes de la mejor de las implementaciones posibles.

Conocer que la calidad puede ser vista desde perspectivas tan diferentes resulta esencial para comprender mejor la noción de calidad del software. Además, es relativamente habitual que la visión de diferentes actores involucrados en el desarrollo de un software se

ajuste a alguna de las perspectivas estudiadas por Garvin. Los clientes, o el departamento de marketing, por ejemplo, suelen tener una perspectiva de la calidad que tiene que ver más con la denominada *perspectiva de usuario*, y buscan por tanto que el software satisfaga sus necesidades y expectativas, que sea razonable el esfuerzo necesario para aprender a utilizarlo, que sea fácil de usar y que la frecuencia e impacto de los fallos sea mínima. Los desarrolladores tienden, por el contrario, a ver la calidad desde la perspectiva del producto: cantidad y tipos de errores, bajo impacto de las modificaciones o facilidad de comprensión del código, son algunos de los puntos que tienen más sentido desde esta perspectiva. En cuanto a los compradores del software, éstos suelen tener una visión mucho más cercana a la *perspectiva de valor* comentada anteriormente: máxima relación calidad-precio, principalmente. Todo ello hace que en numerosas ocasiones surjan conflictos entre los diferentes actores por cuestiones relacionadas con la calidad, ya que cada uno tiene, como hemos visto, diferentes perspectivas de la misma.

Conscientes de todo lo anterior, en el resto de este capítulo enfocaremos la calidad desde los dos puntos más relevantes dentro de las tendencias actuales de la Ingeniería del Software: la perspectiva de producto, y la evaluación y mejora de procesos en pro de la calidad.

9.4 Calidad del producto

Como hemos dicho en la sección anterior, la perspectiva del producto apunta a los atributos internos del software como fuente de calidad, a diferencia de otras perspectivas que evalúan la calidad desde un punto de vista externo, midiendo la calidad en función de cómo ésta se percibe sin evaluar las interioridades del producto en sí. Desde este punto de vista, la calidad del software viene definida según el grado de observancia de un conjunto de criterios de calidad pre establecidos y medibles. Esta es, por ejemplo, la definición de calidad de software que adopta el estándar IEEE 1061-1998 (IEEE, 1998b):

Se denomina **calidad del software** al grado en el que un software posee una combinación de atributos deseables

Esta definición de calidad, orientada a la cuantificación y medida de la misma, coincide con la noción de calidad de los modelos más clásicos como el de McCall, el de Boehm u el que define el estándar de calidad ISO/IEC 9126. A continuación nos detendremos en el estudio de estos modelos.

9.4.1 El modelo de calidad de McCall

Reconociendo la naturaleza intangible y hasta cierto punto abstracta de la calidad, muchos autores han publicado modelos que tratan de caracterizar el software de modo que resulte

más fácil evaluar y medir los costes y beneficios de la calidad del software. El modelo de McCall (1977) fue creado para las fuerzas aéreas norteamericanas con la intención de acercar las visiones de calidad de los desarrolladores y los usuarios. Es de especial importancia por ser históricamente el primero y la base de esfuerzos posteriores, y se organiza en torno a tres tipos de características de calidad:

- *Factores de calidad*, que permiten especificar cómo ven el software sus usuarios desde el exterior.
- *Criterios de calidad*, que indican cómo debe construirse internamente el software desde la perspectiva del desarrollador.
- *Métricas de calidad*, que indican cómo controlar y medir la calidad.

Tal y como se muestra en la Figura 9.2, este modelo define tres perspectivas desde las que deben estudiarse los once factores que en total se computan en la medida de la calidad de un producto de software. Estas perspectivas son las siguientes:

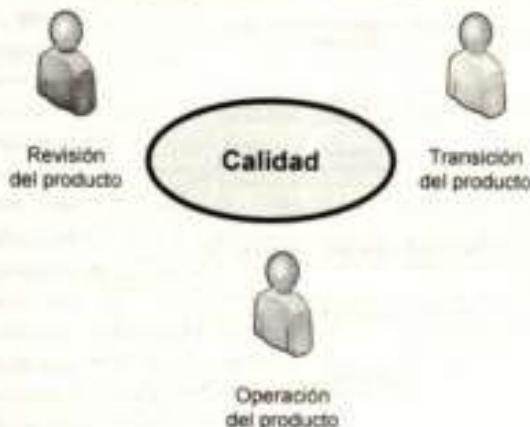


Figura 9.2: Las tres perspectivas del modelo de calidad de McCall

- Revisión del producto. Esta perspectiva estudia la capacidad del producto para adaptarse a los cambios. Se tienen en cuenta aquellos factores que influyen en la capacidad de adaptación del producto, tales como la *facilidad de mantenimiento* (disposición para ser modificado para ser corregido, adaptado o ampliado), la *flexibilidad* (capacidad para introducir cambios en función de las necesidades de negocio) y la *facilidad de evaluación* (capacidad de validar los requisitos establecidos para el software).
- Transición del producto. Esta perspectiva identifica los factores de calidad que influyen en la capacidad que tiene un cierto software para adaptarse a distintos contextos de operación. Así, tiene en cuenta factores tales como la *reusabilidad*, la *portabilidad* o la *interoperabilidad*.

- Operación del producto. Esta perspectiva identifica aquellos factores de calidad que tienen que ver con la forma en que el software lleva a cabo sus funcionalidades, y la medida en la cual cumple con sus especificaciones. Así, tiene en cuenta la *correctitud* (que las funcionalidades solicitadas en su especificación se encuentren disponibles), la *fiabilidad* (qué fallos tiene el sistema en operación), la *eficiencia* en términos de uso de recursos, la *integridad* (protección contra accesos no autorizados a la información) y la *usabilidad*.

En suma, los once factores de calidad apuntados por McCall están organizados en las tres perspectivas anteriores. Para evaluar la calidad de un software, será necesario medir dichos factores, para lo cual el modelo establece el siguiente proceso:



Figura 9.3: Factores de calidad del modelo de McCall

1. Especificar los requisitos de calidad del producto software a desarrollar, seleccionando aquellos aspectos que tengan relación con la calidad deseada.
2. Establecer los factores de calidad (de entre los once descritos) sobre los que aplicar los requisitos de calidad establecidos para el proyecto.

3. Evaluar los factores seleccionados mediante los criterios que el método proporciona para cada factor.

Así por ejemplo, si en la evaluación de la calidad de un cierto software hemos seleccionado la facilidad de mantenimiento como factor de calidad, evaluaremos dicho factor mediante los criterios específicos para el mismo. En el caso de la facilidad de mantenimiento dichos criterios son la *modularidad*, la *simplicidad*, la *concisión* y la *autodescripción*.

9.4.2 El modelo de Boëhm

El modelo de Boëhm (1978) es otro modelo de calidad basado en la identificación de un cierto número de características de calidad para el software. Posterior al modelo de McCall (Figura 9.3), su aportación fundamental es la definición de lo que Boëhm denomina *utilidades principales*, un reconocimiento explícito de que para ser considerado de calidad, un sistema de software debe ser fundamentalmente útil.

A partir de este concepto de utilidad, Boëhm plantea un modelo jerárquico en el que se definen tres utilidades de alto nivel, que serían los requisitos básicos del software. Dichas utilidades son las siguientes:

- Utilidad tal y como está, que representa hasta qué punto el software «tal y como está en este momento» es fácil de usar, fiable y eficiente.
- Facilidad de mantenimiento, que se concreta en la facilidad para identificar qué es necesario modificar, así como la facilidad de modificación o de ejecución de las pruebas sobre el elemento modificado.
- Portabilidad, esto es, la facilidad para utilizar el software en un nuevo entorno, distinto a aquel en que se está utilizando en este momento.

Estos tres usos principales representan el primer nivel de la jerarquía del modelo de Boëhm (ver Figura 9.4). En el segundo nivel, se identifican siete factores de calidad que se asocian con los tres usos del primer nivel. Estos factores son los siguientes:

- Portabilidad, que como ya se ha indicado, representa la facilidad para utilizar el software en nuevos entornos (sistemas operativos, bases de datos, etc.)
- Fiabilidad, que viene indicada por la ausencia de defectos.
- Eficiencia, es decir, mínimo uso de recursos durante el correcto funcionamiento del sistema.
- Usabilidad, entendida desde el punto de vista de la ingeniería humana y la ergonomía, aunque comúnmente se resume como la facilidad de uso del software.
- Facilidad de evaluación, en concreto, la validación de que el software cumple con los requisitos establecidos.

- Comprensibilidad, o facilidad para entender el propósito y estructura del software.
- Flexibilidad, esto es, facilidad para modificar el software ante cambios en los requisitos o aparición de otros nuevos.



Figura 9.4: Jerarquía del modelo de calidad de Boëhm

Estos factores de calidad se descomponen a su vez en elementos primitivos que pueden medirse. Así, y tal y como se muestra en la Figura 9.4, la *portabilidad* puede medirse en función de dos elementos, su independencia con respecto al dispositivo y el grado en que dicho software está autocontenido. Al igual que en el modelo de McCall, el objetivo final es medir la calidad desde los elementos de más bajo nivel del modelo, y utilizar estas medidas para mejorar los productos desarrollados.

9.4.3 El modelo de calidad ISO/IEC 9126

El estándar ISO/IEC 9126, parcialmente basado en esfuerzos anteriores como el modelo de McCall y el de Boëhm, es un estándar internacional para la evaluación de la calidad del software. Su objetivo principal es proporcionar tanto una especificación de la calidad de productos software como un modelo para su evaluación. Define, para ello, un lenguaje común que permite a los usuarios especificar sus requisitos de calidad y a los desarrolladores y evaluadores entender dichos requisitos, para posteriormente tratar de incorporarlos al software en desarrollo. ISO/IEC 9126 aspira a establecer medidas objetivas de calidad, huyendo deliberadamente de lo opinable y eliminando en lo posible toda subjetividad. También pretende conseguir que la evaluación de la calidad sea reproducible y sistemática, de modo que evaluaciones de un mismo software realizadas por personas diferentes en momentos distintos deberían dar el mismo resultado si el software no ha sufrido modificación alguna entre ambas evaluaciones (ver Figura 9.5).

El estándar ISO/IEC 9126 se divide en cuatro partes:

1. Modelo de calidad (ISO/IEC 9126-1:2001). Describe el marco del modelo de calidad y las relaciones entre los diferentes enfoques de la misma, e identifica las distintas características de calidad de los productos de software.
2. Métricas externas (ISO/IEC TR 9126-2:2003). Proporciona un conjunto de métricas que permiten medir las características de calidad externas definidas en el modelo de calidad descrito en ISO/IEC 9126-1:2001.
3. Métricas internas (ISO/IEC TR 9126-3:2003). Describe métricas para medir aquellas características internas de calidad definidas en el modelo descrito en ISO/IEC 9126-1.
4. Calidad en las métricas en uso (ISO/IEC TR 9126-4:2004). Identifica las métricas que permitirán medir la calidad desde el punto de vista del usuario.

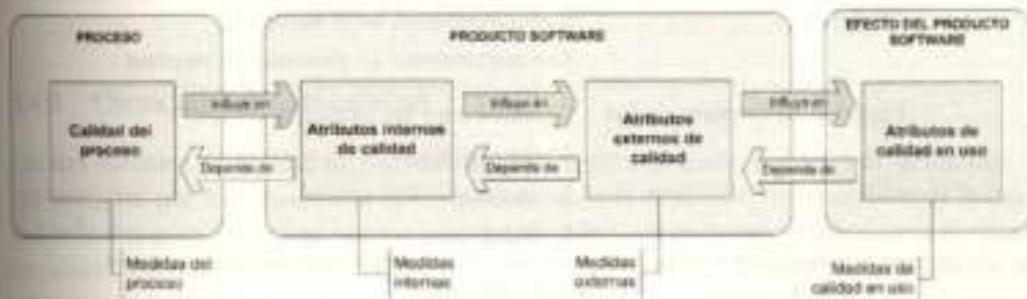


Figura 9.5: Interacción entre las diferentes partes del estándar ISO/IEC 9126

La primera parte –ISO/IEC 9126-1:2001– establece el modelo de calidad. Similar a los modelos de McCall, Boehm, FURPS y otros que veremos más adelante, se basa hasta cierto punto en las ideas aportadas por dichos modelos. Así, define seis características de calidad externas del software que son las siguientes: *funcionalidad, fiabilidad, usabilidad, eficiencia, facilidad de mantenimiento y portabilidad*. Cada una de estas características se divide, a su vez, en varias sub-características (atributos tanto externos como internos) que pueden medirse con métricas específicas según se muestra en la Tabla 9.1.

Las métricas externas de la segunda parte –ISO/IEC TR 9126-2:2003– miden el comportamiento del sistema computacional en su conjunto, lo que incluye el software pero no se limita únicamente al mismo. Las métricas internas de la tercera parte del modelo –ISO/IEC TR 9126-3:2003–, por el contrario, miden el propio software. Las métricas de calidad en uso –ISO/IEC TR 9126-4:2004–, por último, miden los efectos del software en un contexto específico de utilización. Esta parte del estándar, la calidad en uso, especifica cuatro características que son *efectividad, productividad, seguridad y satisfacción*, las cuales se toman

Tabla 9.1: Atributos de calidad del estándar ISO/IEC 9126

Área	Atributo de calidad
Funcionalidad	Adecuación Exactitud Seguridad Interoperabilidad
Fiabilidad	Madurez Tolerancia a fallos Recuperabilidad
Usabilidad	Comprensibilidad Facilidad de aprendizaje Operabilidad
Eficiencia	Comportamiento en el tiempo Comportamiento en términos de recursos
Facilidad de mantenimiento	Facilidad para ser analizado Modificabilidad Estabilidad Facilidad para ser probado
Portabilidad	Adaptabilidad Facilidad de instalación Conformidad Facilidad para ser reemplazado

como indicadores de la calidad tal y como se percibe en función del cumplimiento de las características de calidad de las otras tres partes.

Teniendo en cuenta todo lo anterior, la calidad de un software puede evaluarse según el modelo de calidad del estándar ISO/IEC 9126 (ver Figura 9.6) bien midiendo los atributos de calidad internos –con medidas estáticas de productos intermedios, no del software en ejecución–, o bien midiendo los atributos de calidad externos –a través de medidas del código cuando se ejecuta–, o bien midiendo los atributos de la calidad en uso sobre el software –cuando éste se ejecuta en el entorno final y trabaja en condiciones reales–.

En definitiva, el modelo definido por el estándar ISO/IEC 9126 presupone que una mayor calidad interna/externa del producto software incidirá de manera positiva en la percepción que el usuario tiene acerca de la calidad de la aplicación, y reconoce que el modelo propuesto puede necesitar adaptarse a las características específicas de ciertas aplicaciones.



Figura 9.6: Modelo de calidad ISO 9126

94.4 Otros modelos de calidad

Los tres modelos de calidad de software anteriormente reseñados son los más difundidos y por tanto, los que han tenido una influencia mayor en la evolución del modelado de la calidad en la disciplina de Ingeniería del Software. Existen, sin embargo, muchos otros modelos de calidad. En esta sección mencionaremos algunos otros especialmente interesantes por su relevancia histórica.

El **modelo de Dromey** (1995) aporta una visión completamente diferente a la de los modelos de McCall y Boehm. Este modelo enfatiza la perspectiva de calidad del producto, y así, niega que la calidad de un producto software pueda manifestarse a partir de un conjunto de atributos que por sí mismos indiquen calidad. Por el contrario, afirma que todo producto posee un conjunto de características que contribuyen positivamente a su calidad y otras (como por ejemplo, los defectos) que contribuyen negativamente.

Dromey plantea en consecuencia un análisis de los componentes del software para, a partir de los mismos, extraer información sobre su calidad. Las variables o las funciones de un programa de computadora serían componentes del modelo de implementación, las cuales tendrían propiedades que influyen en la calidad del producto en su conjunto. Así por ejemplo, proporcionar nombres significativos a las variables influye positivamente en la facilidad de mantenimiento del producto. Si no se respeta este principio, se generará un defecto de calidad en el modelo al que pertenece el componente (en este ejemplo en el modelo de implementación) que a su vez generaría una deficiencia en la calidad del producto software.

El **modelo FURPS / FURPS+** fue ideado y publicado por Robert Grady en Hewlett-Packard (1992) y posteriormente ampliado por Rational/IBM y denominado FURPS+. El

modelo define un conjunto de atributos considerados esenciales para el diseño, uso y mantenimiento del software divididos en dos categorías principales: funcionales (F) y no funcionales (URPS). Los funcionales representan las características principales del producto, mientras que los no funcionales representan usabilidad, fiabilidad, rendimiento y soporte¹. El signo + hace referencia a un conjunto de categorías adicionales que generalmente representan restricciones (de diseño, de implementación, de interfaz, etc.) Los factores de calidad FURPS, y los atributos descritos se emplean en este modelo para establecer métricas de calidad para todas las actividades del proceso del software. La diferencia esencial con el resto de modelos es que FURPS no tiene en cuenta la portabilidad de un producto como criterio de calidad.

El estándar IEEE 1061 define una metodología para métricas de calidad del software. Este estándar define la calidad, como ya apuntamos anteriormente, según el grado en que un software posee un conjunto de atributos deseables. El estándar define específicamente una metodología para establecer requisitos de calidad, así como para identificar, implementar, analizar y validar métricas de calidad, tanto para los productos como para los procesos de producción de software. Esta metodología aspira a aplicarse a cualquier software y trata todas las etapas del ciclo de vida.

9.5 Calidad del proceso

Los modelos estudiados en la sección anterior tratan la calidad desde la perspectiva del producto. Según estos modelos, un software de calidad es aquel que cumple con unos ciertos atributos o características de calidad. En esta sección, enfocaremos el estudio de la calidad desde otro punto de vista: el del proceso de producción.

Como el ejemplo sobre la elaboración de una paella descrito en la introducción del capítulo deja en evidencia, el proceso de producción influye decisivamente en la calidad del producto resultante. Teniendo esto muy en cuenta, en esta sección estudiaremos un conjunto de modelos de mejora de los procesos de producción de software cuyo objetivo primordial es la obtención de software de la mayor calidad. Comenzaremos analizando el concepto de *aseguramiento de la calidad* para posteriormente detenernos en el estudio de los métodos más importantes para asegurar, evaluar y mejorar los procesos de software.

9.5.1 Aseguramiento de la calidad

Los procesos de aseguramiento de la calidad garantizan que tanto los productos como los procesos de producción del software cumplen con los requisitos de calidad establecidos. Pero analicemos, antes de proseguir, algunas de las definiciones más ampliamente aceptadas de aseguramiento de la calidad (Juran, Gryna y Bingham, 1974):

¹Las iniciales de los equivalentes términos ingleses *Usability*, *Reliability*, *Performance* y *Supportability* conforman el acrónimo URPS.

Aseguramiento de la calidad es la actividad de proporcionar las evidencias necesarias para garantizar que la función de calidad se lleva a cabo adecuadamente.

Tomando como base esta definición, merece la pena comentar también la que recoge el glosario IEEE de términos de Ingeniería del Software (IEEE, 1990):

Aseguramiento de la calidad es un patrón sistemático y planificado de todas las acciones necesarias para afirmar con certeza que un producto es conforme con los requisitos técnicos establecidos.

Algunos autores han criticado que se considere el aseguramiento de la calidad como un conjunto de actividades vagas y sin implicaciones reales y definidas. Sin embargo, y como indica Fenton, muy a pesar de dichas apreciaciones el aseguramiento de la calidad impone obligaciones reales –en términos de medición– para las diferentes etapas del ciclo de vida:

- En la fase de **requisitos** es necesario determinar la viabilidad, estimando los recursos necesarios y asignando dichos recursos en función de las necesidades. También es necesario en esta etapa establecer los objetivos de calidad.
- En las fases de **especificación y diseño**, el aseguramiento de la calidad impone revisiones e inspecciones de la documentación que se genera.
- Durante la **construcción** hay que validar, verificar, y decidir cuándo el producto está listo para su entrega.
- De cara al **mantenimiento** y a la **revisión** del proyecto, el aseguramiento de la calidad impone el análisis de defectos, la auditoría del proyecto y la planificación de la mejora del proceso de producción.

Algunas de estas actividades, no obstante, se llevan a cabo a lo largo de varias fases del desarrollo y no sólo en aquellas que se ha indicado aquí. Así, la asignación de recursos es algo que difícilmente puede hacerse una única vez pues necesitará ser ajustado constantemente en función del desarrollo del proyecto.

En el ámbito de los procesos, un plan de aseguramiento de la calidad debe garantizar que éstos son adecuados, que se ejecutan de acuerdo con el plan inicialmente previsto y que durante cada proceso se toman medidas relevantes para la organización que desarrolla dicho proceso. La organización debe, para ello, cumplir un conjunto de requisitos:

- Fijar con precisión los objetivos, plazos y costes en términos de recursos.
- Ser consistente con la gestión de la configuración.

- Identificar inequívocamente las normas aplicables al proyecto y las directrices que lo guían.
- Establecer procedimientos para todo aquello que afecte al desarrollo: qué medir, cómo informar de los errores, qué metodología emplear, cómo documentar, qué acciones correctivas tomar, etc.
- Establecer procedimientos de seguimiento y monitorización que garantizan que todo lo anterior se cumple.

En las siguientes secciones estudiaremos los modelos de evaluación y mejora de procesos de software orientados finalmente a la obtención progresiva de un software de mayor calidad. Los beneficios de introducir un método de mejora en una organización son evidentes: la calidad de un producto viene en gran medida determinada por la calidad de su proceso de desarrollo.

Dado que este libro está dedicado a la Ingeniería del Software en general y no al estudio exhaustivo de la calidad del software en particular, nos detendremos únicamente en aquellos modelos de evaluación y mejora de procesos de software más ampliamente aceptados y utilizados. Incluso en éstos, no podremos realizar un estudio detallado y profundo, por lo que remitimos al lector interesado en profundizar en algún modelo en particular a las referencias bibliográficas que incluimos al final del capítulo.

9.5.2 El modelo CMMI

El modelo CMMI, acrónimo del inglés *Capability Maturity Model Integration*, es una evolución de un modelo anterior denominado CMM inicialmente desarrollado por el *Software Engineering Institute* (SEI) de la Universidad Carnegie Mellon. El SEI llevó a cabo el encargo de desarrollar un modelo de calidad que sirviera como base para establecer un sistema de capacitación de las empresas que suministraban software al Gobierno de los Estados Unidos. Dicho modelo fue definido como:

«Un enfoque para la mejora de procesos que proporciona a una organización los elementos esenciales para llevar a cabo sus procesos de manera efectiva. Puede utilizarse para guiar la mejora de procesos en un proyecto, en un departamento, o en una organización completa. CMMI ayuda a integrar funciones de la organización tradicionalmente separadas, a establecer prioridades y objetivos en la mejora de procesos, proporciona guías para los procesos de calidad y sirve como referencia para la evaluación de los procesos actuales».

CMMI no es un proceso de desarrollo de software, sino más bien una guía que describe las características que hacen efectivo a un proceso. Las ideas que aporta pueden ser, por tanto, utilizadas como un conjunto de buenas prácticas, como un marco para la organización y priorización de actividades, o como una forma de alinear los objetivos de la organización con los objetivos del proceso en estudio. Como se muestra en la Figura 9.7, CMMI se

interesa por la mejora de los procedimientos y métodos (procesos) que las personas de una organización llevan a cabo con ayuda de tecnología y otras herramientas ya que, si los procesos no están correctamente definidos, son maduros y ampliamente conocidos, ni las personas más cualificadas serán capaces de rendir a su mejor nivel aun disponiendo de las mejores herramientas.

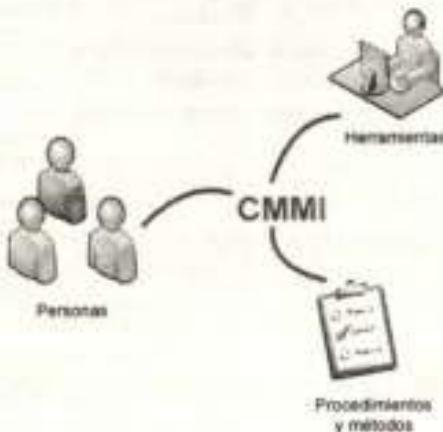


Figura 9.7: Modelo CMMI

Para comprender correctamente lo que es CMMI, es necesario primero definir el concepto de **modelo de madurez**, noción sobre la que CMMI se asienta firmemente. Un modelo de madurez no es sino un conjunto de características que describen ciertos aspectos de equilibrio, experiencia y formalidad en una organización. Estos modelos generalmente incluyen un punto de partida, los beneficios que la progresión dentro del camino de la madurez aporta a la organización y, lógicamente, una definición clara de lo que significa *madurar* en esta organización en particular.

Los modelos de madurez, a menudo, se emplean como referencia para la comparación o la mejora. El modelo de madurez de capacidades CMMI, por ejemplo, se emplea para comparar (y mejorar) el proceso de desarrollo de software de una organización describiendo una progresión continua en cinco niveles (Figura 9.8). Cada uno de dichos niveles define, por un lado, la escala de medida de la capacidad de los procesos de la organización, y por otro, los objetivos que ayudarán a la organización a ordenar sus esfuerzos de cara a mejorar sus procesos. En el nivel más bajo de la escala se encuentran aquellas organizaciones sin procesos repetibles, donde gran parte del trabajo se realiza sin procedimientos prestablecidos y a medida (*ad hoc*) para cada proyecto al que se enfrentan. En el nivel más alto, las organizaciones que usan procesos definidos y repetibles, que aplican métricas para la mejora continua de sus procesos y que, en definitiva, están continuamente en busca de hacer las cosas mejor.

CMMI sirve además para certificar el nivel en el que se encuentra una organización o área de procesos –ver Figura 9.9–, lo cual es interesante desde el punto de vista de las organizaciones, tanto para su prestigio como entidades certificadas CMMI en los niveles más altos, como para optar a contratos en cuyos pliegos de condiciones se solicite como requisito de concurso la demostración de un nivel determinado.



Figura 9.8: Niveles de madurez en CMMI

El modelo establece dos formas diferentes de medir la manera en que las organizaciones pueden mejorar sus procesos. Una forma es la denominada *representación continua*, que se basa en los denominados niveles de capacitación, y otra es la denominada *representación por etapas*, que emplea los denominados niveles de madurez (ver Tabla 9.2). Tanto los niveles de capacitación como los de madurez proporcionan una forma adecuada de medir la mejora de procesos, si bien el enfoque asociado con la mejora de procesos es diferente para cada una de las representaciones.

Representación por etapas En este modo de representación mediante niveles de madurez, CMMI define cinco niveles en los que una organización puede categorizarse de acuerdo con la disposición global de sus procesos internos. Es decir, no se enfoca a un área en particular como la representación mediante niveles de capacitación, sino que se refiere a múltiples áreas de procesos. Los cinco niveles que define CMMI son los siguientes:

- En el nivel *inicial* se encuentran aquellas organizaciones en las que no existen áreas de proceso y en las que, por lo tanto, los procesos no están definidos. Aunque este nivel, a menudo, se etiqueta como nivel de procesos *ad hoc* o caóticos, esto no quiere decir que no haya procesos en la organización que se lleven a cabo de manera controlada, sino simplemente que no están definidos de antemano y que, por tanto, no puede asumirse que siempre se realizarán de manera predecible. En estas organizaciones es frecuente, o al menos más probable que en organizaciones que se sitúan en niveles superiores, el abandono de un proceso en momentos de crisis o la incapacidad para repetir éxitos anteriores.

- En el nivel 2 (*repetible*) la organización ha establecido las actividades de gestión de proyectos, lo que le confiere la capacidad de repetir procesos con resultados consistentes. Sin embargo, los procesos pueden no repetirse para todos los proyectos de la organización ya que no existe una disciplina rigurosa sobre los mismos.
- Las organizaciones que se encuentran en el nivel 3 de madurez (*definido*) tienen documentados y estandarizados todos sus procesos de desarrollo y mantenimiento de software, y dichos procesos están sujetos a algún tipo de mejora continua. Todos los proyectos de la organización se llevan a cabo de acuerdo con los procedimientos establecidos.
- En el nivel 4 (*gestionado*) las organizaciones tienen un programa detallado y organizado de medición de procesos de desarrollo de software. Los procedimientos de gestión para ajustar y adaptar los procesos a las particularidades de cada proyecto que se aborde son públicos y están documentados.
- El nivel 5 (*optimización*) describe a aquellas organizaciones que tienen completamente implementado un proceso de mejora continua para todos sus procesos, recopilan datos de todos sus proyectos, y el estudio de dichos datos lo emplean en la mejora e innovación de los propios procesos de la organización (ver Figura 9.9).



Figura 9.9: Certificación de nivel 5 CMMI

Los diferentes niveles de madurez descritos están estrechamente relacionados con lo que CMMI denomina *áreas de proceso*. Cada una de estas áreas de proceso representa

un conjunto de buenas prácticas relacionadas que cuando se implementan conjuntamente satisfacen objetivos importantes para la consecución de mejoras significativas en dicha área.

Las áreas de proceso se agrupan por nivel de madurez, indicando qué áreas de proceso hay que implementar para alcanzar cada nivel de madurez. Una organización que quiera alcanzar el nivel 4 de madurez, por ejemplo, debería fijarse en las áreas de proceso marcadas como de nivel 4 y alcanzar los objetivos marcados para cada una de dichas áreas. Una vez certificada en el nivel 4, es posible afirmar que dicha organización no sólo utiliza las áreas de proceso de nivel 4 a los niveles adecuados de madurez, sino también todas las de nivel inferior, esto es, todas las de nivel 2 y 3 (puesto que el nivel 1 no tiene áreas clave de proceso establecidas).

Representación continua La representación mediante niveles de capacitación consiste en la definición de objetivos y prácticas generales para cada área de procesos. Estos niveles pueden considerarse, por tanto, un medio para mejorar progresivamente los procesos de una cierta área. CMMI define seis niveles de capacitación, etiquetados de 0 a 5:

0. Incompleto: un proceso que no se lleva a cabo, o que se lleva a cabo parcialmente.
1. Realizado: proceso que satisface los objetivos específicos del área a que pertenece.
2. Gestionado: el proceso se planifica y ejecuta de acuerdo con ciertas reglamentaciones, emplea personal cualificado, se monitoriza y controla, etc.
3. Definido: el proceso se ajusta a los estándares de la organización y proporciona, tanto medidas de la producción como otras informaciones valiosas desde la perspectiva de la mejora de procesos.
4. Gestionado cuantitativamente: un proceso definido que además, es controlado mediante técnicas cuantitativas o estadísticas.
5. En optimización: un proceso gestionado cuantitativamente sujeto a mejoras basadas en la comprensión de las causas de la variabilidad inherentes al propio proceso.

CMMI es hoy en día un modelo prestigioso y ampliamente difundido, por lo que la certificación en cualquiera de los niveles (pero especialmente en los más altos) es exhibida por las organizaciones como un importante marchamo de calidad. De hecho, en CMMI se han certificado organizaciones de la talla de Boeing, Nokia, Motorola, BMW, J.P. Morgan, Intel, el Departamento del Tesoro de los Estados Unidos, Reuters, IBM o la NASA, por citar sólo algunos ejemplos.

El modelo CMMI es, sin embargo, un modelo complejo cuya descripción detallada no es posible en este libro. Remitimos a aquellos interesados en alcanzar un conocimiento profundo del método, a las fuentes referenciadas al final del capítulo (*ver Tabla 9.2*).

Tabla 9.2: Las diferentes representaciones del modelo CMMI

Representación continua	Representación por etapas
Niveles de capacitación	Niveles de madurez
La organización selecciona áreas de proceso y niveles de capacitación en función de sus objetivos de mejora de procesos	La organización selecciona áreas de proceso en función de los niveles de madurez
Las mejoras se miden utilizando los 6 niveles de capacitación (0-5), que marcan la madurez de un proceso concreto en una organización	Las mejoras se miden mediante 5 niveles de madurez (1-5), que miden un conjunto de procesos de una organización
Para fijar los objetivos y corregir el rendimiento de la mejora de procesos se emplean perfiles de nivel de capacitación	Se emplean niveles de madurez para fijar los objetivos y corregir el rendimiento de la mejora de procesos
Las tablas de equivalencias permiten a una organización que usa este enfoque continuo deducir su nivel de madurez	No son necesarios mecanismos de equivalencia para calcular el nivel de madurez de la organización

9.5.3 Modelo SPICE: El estándar ISO/IEC 15504

El estándar ISO/IEC 15504 define un marco de trabajo de evaluación y mejora de procesos que puede ser utilizado por las organizaciones para planificar, gestionar, monitorizar, controlar y en definitiva, mejorar la adquisición, desarrollo, operación, evaluación y soporte del software. Este estándar partió de la iniciativa SPICE, un proyecto internacional cuyo objetivo fundamental era desarrollar un estándar para la evaluación de procesos de software. El proyecto SPICE se realizó bajo los auspicios del grupo de trabajo de evaluación de procesos de software del comité internacional de estandarización ISO y produjo finalmente el estándar ISO/IEC 15504 sobre evaluación de procesos y tecnología de la información, a menudo conocido también como modelo SPICE.

A lo largo de su desarrollo, que comenzó en 1993, el modelo ha evolucionado principalmente en el objeto de estudio: de un modelo de referencia para las buenas prácticas de software, hacia un marco de trabajo de evaluación de procesos aplicable a cualquier disciplina en el área de las tecnologías de la información. El estándar, en su versión actual (2003-2006), está formado por cinco documentos:

- ISO/IEC 15504-1 (conceptos y vocabulario): perspectiva general de introducción al estándar que incluye un glosario de definiciones de términos y una guía de la norma.
- ISO/IEC 15504-2 (cómo realizar una evaluación): descripción de requisitos para llevar a cabo evaluaciones consistentes y fiables.

- ISO/IES 15504-3 (guía para realizar una evaluación): descripción de las acciones precisas para alcanzar los requisitos mínimos para la realización de una evaluación descritos en la segunda parte de la norma.
- ISO/IES 15504-4 (guía sobre el uso del estándar para la mejora de procesos y determinación de la capacitación): indica cómo utilizar una evaluación de procesos conforme con el estándar dentro de un programa de mejora de procesos.
- ISO/IES 15504-5 (ejemplo de modelo de evaluación de procesos): proporciona ayuda sobre los modelos de evaluación de procesos mediante la exposición de ejemplos.

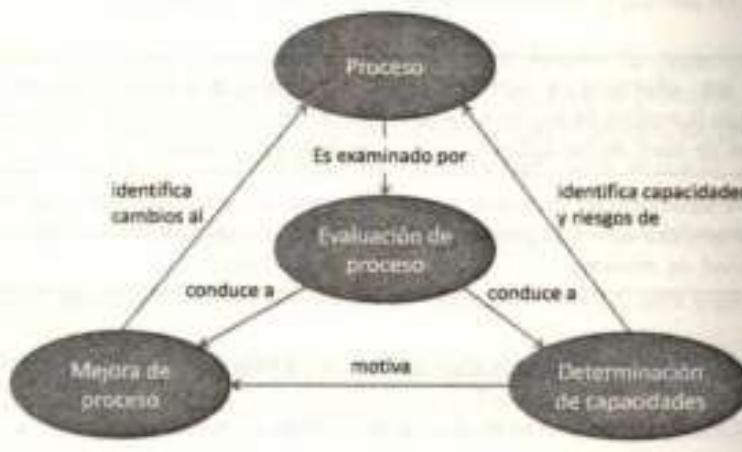


Figura 9.10: El modelo de mejora de procesos de SPICE

El modelo SPICE es bidimensional, pues trata la evaluación de procesos de software desde dos dimensiones: (i) la dimensión de los procesos, relacionada con (ii) la dimensión de la capacitación. La Figura 9.10 resume el proceso de mejora que propone SPICE.

La dimensión de los procesos viene dictada por el modelo de referencia –primer documento de la norma (ISO/IEC 15504-1)– donde se detallan tres tipos de procesos:

- Procesos primarios: todos aquellos relacionados con la adquisición, suministro, ingeniería y operación de la organización.
- Procesos de soporte: aquellos procesos que pueden ser utilizados por otros en determinadas circunstancias.
- Procesos de la organización: relacionados con la gestión, mejora del proceso, recursos e infraestructura y reutilización.

En cuanto a la dimensión de la capacitación, la norma ISO/IEC 15504 define para cada proceso un nivel en una escala igual que la del modelo continuo de CMMI y que va, por

tanto, de 0 (proceso incompleto) a 5 (proceso en optimización). La capacitación de los procesos se mide mediante atributos que se asocian a los procesos. Así, se definen nueve atributos del proceso (PA, del inglés *Process Attribute*), los cuales se muestran en la Figura 9.11.



Figura 9.11: Niveles de capacitación en SPICE

Como cada uno de estos atributos valora un cierto aspecto de la capacidad del proceso, la evaluación completa de un proceso se lleva a cabo observando el valor de sus atributos, que se miden según la siguiente escala:

- No alcanzado (0 - 15%).
- Parcialmente alcanzado (15% - 50%).
- En su mayoría alcanzado (50% - 85%).
- Completamente alcanzado (85% - 100%).

SPICE se construye tomando como base modelos anteriores, y así toma información y fundamentos teóricos de Trillium (que veremos más adelante) y de CMMI.

9.5.4 Los estándares de la familia ISO 9000

El calificativo genérico ISO 9000 designa a un conjunto de estándares para sistemas de gestión de la calidad. Aunque, desde sus inicios en los años 1980, algunas de las normas originales han desaparecido por pura evolución natural, las más relevantes y conocidas han permanecido, aunque no siempre sus actuales nombres significan lo mismo que significaban anteriormente. Hoy en día existen dos normas en esta familia: el estándar ISO 9000 (publicado en 2000) y el estándar ISO 9001 (publicado en 2005). ISO 9000 cubre los fundamentos de los sistemas de gestión de calidad y define los términos relacionados con la misma. ISO 9001, por su parte, especifica los requisitos de un sistema de gestión de la calidad dentro de una organización.

Colectivamente, las normas de la serie ISO 9000 tienen como meta ayudar a las organizaciones a definir y mantener sistemas de calidad. Es importante apuntar que, a diferencia de los modelos CMMI y SPICE, la familia de normas ISO 9000 no tiene nada que ver con programas de aseguramiento de la calidad. Estas normas han sido diseñadas, por el contrario, para definir los requisitos que debe cumplir una organización con un buen sistema de gestión de la calidad, aunque no se especifica qué deben hacer las organizaciones para alcanzar tales fines. Su objetivo primordial es por tanto la consistencia en la calidad de los productos y servicios, junto con una continua mejora en la satisfacción del cliente y en la disminución de los ratios de error. Así, uno de sus conceptos fundamentales es la conformidad de los procedimientos de la organización con sus regulaciones internas (Figura 9.12).

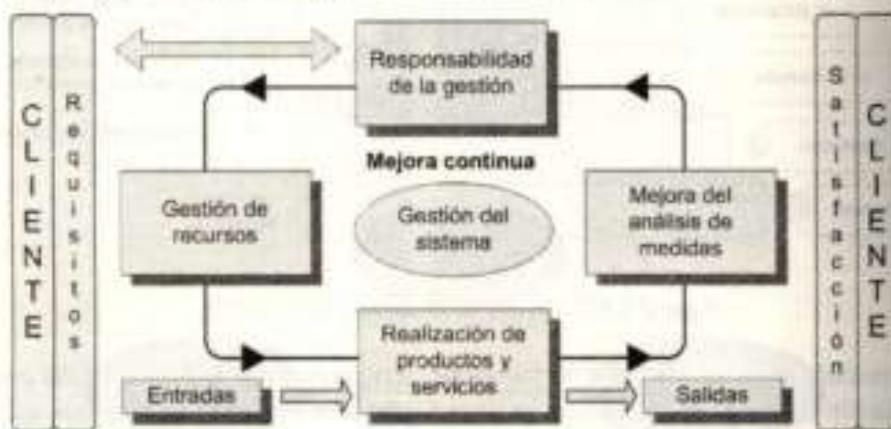


Figura 9.12: Sistema de gestión de la calidad según la norma ISO 9000

Dado que se trata de normas genéricas, aplicables, por tanto, a un amplio abanico de organizaciones, la forma y los métodos de implementar el sistema de calidad pueden variar tremadamente de unas organizaciones a otras. No obstante, todas deben compartir los siguientes principios:

- **Orientación al cliente:** las organizaciones dependen de sus clientes y, por tanto, deben luchar por colmar e incluso superar sus expectativas.
- **Liderazgo:** los estamentos superiores de gestión de la organización deben definir políticas de calidad y crear un entorno en el cual el personal se comprometa completamente con los objetivos de calidad.
- **Implicación de los empleados:** los trabajadores, el mayor capital con que cuenta una organización, sólo emplearán todas sus capacidades y aptitudes en beneficio de la organización si están completamente involucrados en el proceso de calidad.
- **Modelo de procesos:** los resultados esperados sólo se alcanzarán si las actividades y los recursos pertinentes se gestionan y controlan como procesos.

- **Modelo de gestión orientado a sistemas:** las organizaciones en las que es claramente reconocido, gestionado y controlado el enlace entre los procesos que conforman el sistema completo, están mejor situadas para alcanzar sus objetivos eficientemente.
- **Mejora continua:** la mejora continua del rendimiento global de la organización es el objetivo último.
- **Enfoque a la toma de decisiones objetiva:** las decisiones se basan en el análisis de datos e información.
- **Relaciones con los proveedores mutuamente interdependientes:** la organización depende de sus proveedores, por lo que las relaciones deberán basarse en el mutuo beneficio.

Las organizaciones certificadas de acuerdo a los estándares de calidad ISO 9000 pueden lucir el distintivo que reconoce su consistencia, fiabilidad, valor y servicio al cliente de cara al exterior. Se trata de un distintivo muy ampliamente difundido y conocido, y cuya obtención suelen ostentar las organizaciones, en general, como marchamo de calidad ante sus clientes y público en general.

Todos los requisitos y recomendaciones de esta familia de estándares son genéricos y por tanto, aplicables a cualquier tipo de organización, independientemente de su tamaño, tipo y sector productivo. Para abarcar tan diferentes organizaciones, los estándares han sido definidos de manera deliberadamente genérica, lo que ha tenido un impacto menor en la certificación de organizaciones específicamente dedicadas a la producción de software. No obstante, el nuevo enfoque a procesos y sistemas de las últimas versiones del estándar permiten adivinar un cambio en esta tendencia en un futuro a medio o largo plazo.

9.5.5 Otros modelos, estándares y especificaciones

Además de los anteriormente reseñados, existen otros métodos, estándares y especificaciones de algún u otro modo relacionados con la calidad en el software. A continuación vamos a hacer una reseña breve de algunos de ellos.

ITIL

ITIL (*Information Technology Infrastructure Library*) es el modelo de gestión de servicios de tecnologías de la información más aceptado actualmente. Está formado por un conjunto de documentos de buenas prácticas para facilitar la implementación de un marco de gestión de este tipo de servicios. Inicialmente creado por el Gobierno del Reino Unido a través de su Oficina de Comercio (OGC), en la actualidad se utiliza en todo el mundo.

ITIL comprende cinco volúmenes, cada uno de ellos dedicado a una disciplina específica dentro de la gestión de servicios de tecnologías de la información. Así, uno define la estrategia del servicio, otro el diseño del servicio, otro la transición de servicios, y los dos

restantes se dedican a la operación de los servicios y a la mejora continua de los mismos, respectivamente.

ITIL divide la gestión de servicios en dos áreas principales: los *servicios de soporte* y la *prestación de servicios*. Conjuntamente engloban diez disciplinas que abarcan todas las áreas para la provisión y gestión de servicios eficaces:

- Los *servicios de soporte* están formados por las prácticas que permiten la prestación de los servicios de tecnologías de la información, y sin las cuales sería imposible proporcionar dichos servicios. Engloba la gestión de la configuración, la gestión de incidencias, la gestión de cambios, el soporte técnico al usuario, el control de versiones y la gestión de problemas.
- La *prestación de servicios* tiene que ver con los servicios que las organizaciones requieren de sus proveedores para así poder dar a su vez servicios de negocio a sus usuarios. Engloba la gestión de los niveles de servicio, la gestión de la capacidad y de la continuidad de la prestación de servicio, la gestión de la disponibilidad y finalmente la gestión financiera.

El Gobierno británico, a través de OGC, promueve activamente la acreditación de expertos en ITIL. Estos expertos, que son reconocidos internacionalmente, pueden actuar como consultores especializados en la adopción de las buenas prácticas que ITIL propugna en cualquier organización dedicada al software. Dichas prácticas son conformes con los estándares de gestión ISO 20.000 para la adopción de un enfoque integrado de procesos a la prestación de servicios entre organizaciones.

Trillium

El modelo Trillium, desarrollado por Bell Canadá en 1992, se apoya en varios estándares de calidad de software, especialmente en la versión 1.1. del modelo de madurez CMM (precursor del actual CMMI). Concebido como un modelo para la evaluación del desarrollo de software de los proveedores de Bell, su objetivo inicial era minimizar el riesgo y asegurar tanto el correcto rendimiento como la entrega en plazos de los sistemas de software adquiridos por la organización.

En Trillium, la capacitación se define como la capacidad del desarrollador para entregar un producto software maximizando la corrección y la fiabilidad, cumpliendo las expectativas del cliente y minimizando los costes de desarrollo y los plazos de entrega. Esta capacitación se mide en una escala de cinco niveles, similar a la escala de CMMI. En su momento, Bell Canadá tenía la intención de que todos sus proveedores estuvieran certificados a un nivel 3 o superior dentro de esta escala, no por el simple hecho de establecer unas prácticas rígidas de calidad, sino para, en último término, dar mejor servicio y en definitiva satisfacer a sus clientes.

Aunque, como hemos dicho, está basado en CMM, existen diferencias importantes. El principal punto que caracteriza a Trillium con respecto a otros enfoques es el hecho de

que está orientado a los denominados *mapas de ruta*, en lugar de estar enfocado a áreas clave de proceso. Un mapa de ruta es, en Trilium, un conjunto de prácticas relacionadas que se aplican a un área o necesidad concreta de la organización, o también un elemento específico dentro del proceso de desarrollo. Cada mapa de ruta representa una capacitación significativa para una organización de desarrollo de software, y dentro de él, el nivel de las prácticas se basa en su grado de madurez. Así, las prácticas fundamentales se encuentran en los niveles más bajos, mientras que las más avanzadas están situadas en los más altos. Según este modelo, las organizaciones maduran cuando progresan en el mapa de ruta.

No obstante, el modelo Trilium no se ideó únicamente para la certificación del software específicamente creado para Bell, sino que se dirigía a cualquier software convencional. Así, los criterios del modelo están relacionados con prácticas de aseguramiento de la calidad ampliamente difundidas en la Ingeniería del Software. El reto era adaptar dichas prácticas a los posibles proveedores de todo tipo de sistemas. Para ello, Trilium incorporaba algunas características distintivas como su especial orientación al cliente, una más amplia cobertura de las incidencias y problemas que afectan a la capacitación, y la incorporación de prácticas de disciplinas tales como la ingeniería de la usabilidad, la calidad de procesos en la organización y la gestión de la confianza, entre otras.

Bootstrap

Lo que hoy se conoce como Bootstrap es el resultado de un proyecto financiado por la Unión Europea a través del extinto programa Esprit para el fomento de la investigación en tecnologías de la información. El objetivo de dicho proyecto era desarrollar una metodología para la evaluación de procesos de software, la medición cuantitativa y la mejora continua, y posteriormente validarla mediante un proceso de prueba en diversas organizaciones. Tras la finalización del proyecto se creó el Instituto Bootstrap para explotar los resultados del proyecto y continuar el desarrollo y promoción de la metodología. La última versión, Bootstrap 3.0 es conforme tanto con SPICE ISO/IEC 15504 como con el estándar ISO 12207 sobre procesos de ciclo de vida del software.

La metodología Bootstrap es aplicable a compañías de desarrollo de software de tamaño pequeño o medio, así como a departamentos de software de organizaciones más grandes. Proporciona soporte para la evaluación de la capacitación de procesos mediante su comparación con un conjunto de buenas prácticas reconocidas en la Ingeniería del Software, asegurando la fiabilidad y certeza de que evaluaciones repetidas proporcionarían el mismo resultado. La metodología da como resultado un conjunto de puntos fuertes y puntos débiles dentro de la organización evaluada. Para ello, sigue un modelo de capacitación de procesos que se basa en seis niveles de capacitación numerados entre 0 y 5, de forma similar a los modelos de CMMI y SPICE.

El elemento distintivo de Bootstrap es, no obstante, su procedimiento de evaluación, parte esencial de un proceso de mejora según esta metodología. Durante una evaluación Bootstrap, los procesos de la organización se evalúan y se definen adecuadamente. Para ello, se recogen datos mediante dos tipos de cuestionarios estandarizados: uno para tomar

datos sobre la organización del desarrollo de software y otro para tomar datos sobre los proyectos de la organización. Una vez definidos los procesos, y con los datos obtenidos mediante los cuestionarios, se evalúa cada uno de ellos mediante los niveles definidos de capacitación. Como resultado se elaboran y entregan informes y directivas de mejora para la organización, donde se aporta información, por ejemplo, sobre los procesos que mayor impacto tienen en la consecución de los objetivos de la organización, o las sugerencias sobre la priorización de procesos con baja capacitación y alto impacto.

El proceso de software personal (PSP)

PSP es un proceso de mejora de procesos de software diseñado para controlar, gestionar y mejorar la forma de trabajo individual de los ingenieros del software. Aplicación de las ideas de CMMI al trabajo individual, fue creado por Watts Humphrey a partir de sus investigaciones y su experiencia en la aplicación de los principios de CMMI. Su conclusión más importante fue que los principios del modelo son aplicables, no sólo a las áreas o a las organizaciones en su conjunto, sino también a los individuos. En consecuencia, enunció que su puesta en práctica para cada individuo involucrado en un desarrollo revertiría positivamente en la obtención de mejores resultados en la aplicación del método CMMI.

El principal objetivo del PSP es introducir disciplina en el proceso de desarrollo de software de cada individuo, para lo cual describe prácticas para el desarrollo individual de programas pequeños, desde la asignación del problema hasta las pruebas de unidad. Como tal proceso de construcción de software que es, PSP proporciona al ingeniero diversos elementos para la mejora de su trabajo: guiones, instrucciones, formularios y plantillas preparadas y estandarizadas. La lógica fundamental que guía el modelo PSP es que una persona entiende mejor lo que hace cuando define claramente el proceso que lleva a cabo, mide y controla su propio trabajo, se evalúa y aprende de la propia experiencia. El modelo se basa en los siguientes cinco principios:

- Un proceso definido y estructurado mejora la eficiencia en el trabajo.
- El proceso personal definido debe alinearse con las habilidades y preferencias del individuo.
- Cada persona se debe involucrar en la definición de su proceso.
- El proceso de cada persona debe evolucionar según evolucionan sus habilidades y capacidades.
- La mejora continua del proceso se consigue si existe una retroalimentación rápida y explícita.

Humphrey observó que la mayoría de ingenieros de software no llevan a cabo ningún seguimiento ni planificación de su trabajo, y que no creen en métodos que les obliguen a llevar a cabo dichas actividades, a menos que vean los resultados con sus propios ojos.

Como reacción, y con ánimo de revertir la situación, ideó un camino de formación donde deben realizarse diez entregas de software y cinco informes para conseguir simultáneamente formarse en las seis etapas del método. Un ingeniero de software debe escribir uno o dos programas en cada nivel y analizar los datos del trabajo realizado y entregado, para posteriormente utilizar sus datos para la mejora de su trabajo personal. Al final del proceso formativo, los ingenieros son capaces de planificar y controlar su trabajo. Su rendimiento y la calidad de los productos que entregan es significativamente mayor, tal y como han demostrado numerosos estudios llevados a cabo para comprobar la eficacia del método PSP.

Un ejemplo concreto de estos estudios es la disminución del retraso de un proyecto (medido en meses) respecto al calendario inicialmente previsto, en función del modelo de procesos empleado por la organización que lleva a cabo el desarrollo (véase la Figura 9.13).

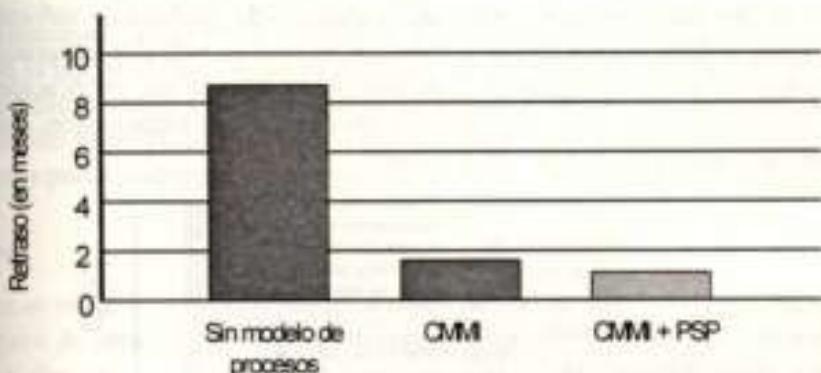


Figura 9.13: PSP en combinación con CMMI (fuente: AIS, Advanced Information Systems)

Team Software Process (TSP)

Si el objetivo del PSP es hacer que los profesionales del software tomen conciencia y control de su trabajo, los objetivos del TSP (*proceso de software para equipos*, creado también por Watts Humphrey) son proporcionar un entorno de trabajo en equipo de soporte al proceso de software personal y que ayude a construir y mantener equipos de trabajo autodirigidos.

PSP y TSP se constituyen, pues, en dos potentes metodologías orientadas fundamentalmente a alcanzar mejores resultados en la producción de software y, en definitiva, a proporcionar a los individuos y equipos el grado de compromiso y formación necesarios para realizar con éxito proyectos de software.

Según Humphrey, los objetivos de mejora continua de procesos deben enfocarse en paralelo desde tres niveles distintos:

- La organización, para lo cual lo ideal es (según Humphrey) utilizar CMMI.
- Los equipos de trabajo, para lo cual propone TSP.

- Los ingenieros del software, para lo cual propone PSP.

TSP está enfocado principalmente a la formación y gestión de los equipos de trabajo. Utilizando este modelo, una organización debe construir equipos autónomos que planifiquen y hagan seguimiento de su trabajo, estableciendo sus propios objetivos y gestionando sus procesos. Estos equipos pueden ser equipos en los que sólo haya personas formadas en Ingeniería del Software, o equipos mixtos de integración, con un número de miembros que se sitúa idealmente entre 3 y 20 ingenieros con formaciones diferentes (no todas necesariamente relacionadas con el software).

El funcionamiento integrado de PSP y TSP (ver Figura 9.14) se resume en lo siguiente: si se desea tener equipos con las habilidades necesarias para autogestionar sus procesos, sus miembros deben contar con ciertas habilidades individuales. La formación dentro de los equipos es, por tanto, esencial. Además, la creación de módulos de software es una tarea individual, si bien la elaboración y posterior entrega de módulos integrados es una labor de equipo, no sólo de codificación, sino también de integración, verificación, etc. En definitiva, el software es producto del trabajo de un equipo cuya capacitación, disciplina y compromiso son claves para el éxito del proyecto.

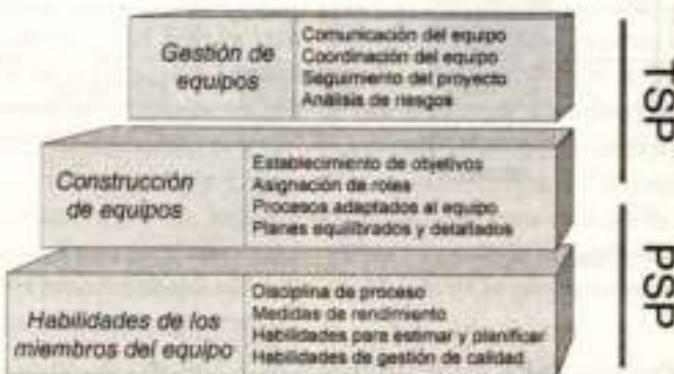


Figura 9.14: Elementos de PSP y TSP para la mejora continua de procesos

TickIT

TickIT es una iniciativa del Departamento de Comercio e Industria británico que desde 1993 es administrada y mantenida por la *TickIT Office*, un departamento del Instituto Británico de Estándares (BSI) que es quien tiene en último término la responsabilidad sobre todos los aspectos de normalización en sistemas de información y comunicaciones en el Reino Unido.

El programa TickIT define un esquema de certificación para aplicar el estándar ISO 9001 con el objeto de ayudar a las organizaciones de software a desarrollar sistemas de gestión de la calidad adecuados a sus procesos de negocio y que cumplen con los requisitos

de la norma. Así, interpreta y adapta los requisitos del estándar según las especificidades de la industria del desarrollo de software, habiendo sido especialmente diseñado para tratar con los requisitos particulares de este tipo de organizaciones.

La aplicación de TickIT supone el establecimiento de métodos de control y la gestión de dichos controles a lo largo de todo el proceso de desarrollo de software, desde el inicio hasta la entrega del software. Es, en cierto modo, un sistema de prevención de problemas que confía en que el establecimiento de controles rígidos minimizará la aparición de problemas. La certificación de conformidad con el estándar de gestión de la calidad ISO 9001 debe ser llevada a cabo por organismos de certificación acreditados por TickIT, para lo cual es necesaria una acreditación como auditor que sólo se consigue tras un examen y una trayectoria en la que se requiere experiencia directa en la industria del software y en sus procesos.

Recientemente TickIT se ha actualizado y «rebautizado» como TickITplus, una nueva versión de la iniciativa que mejora el modelado de procesos, y remozza la versión anterior. Particularmente, actualiza las prácticas de la versión anterior en lo referente al esquema de calificación y certificación, revisa la documentación, establece una nueva estructura de regulaciones para el esquema, y mejora el proceso de evaluación y planificación, entre otros.

Six Sigma

Six Sigma es una metodología orientada a procesos para la mejora del rendimiento a través de la mejora de áreas específicas de procesos de negocio. Se trata de una metodología rigurosa y disciplinada que utiliza datos y análisis estadísticos para medir y mejorar el rendimiento operativo de una compañía, identificando y eliminando defectos en todos sus procesos. Originalmente desarrollada por Motorola, hoy día es utilizada en numerosas organizaciones.

Para alcanzar los estándares de calidad que Six Sigma propone, un proceso debe producir como máximo 3,4 defectos por cada millón de oportunidades (DPMO). Tengase en cuenta que Six Sigma define *defecto* como cualquier cosa fuera de los requisitos de usuario, y *oportunidad* como cualquier área dentro del proceso, producto o servicio donde se podría producir un defecto.

Algunas de las medidas más utilizadas en Six Sigma son las siguientes:

- *Oportunidad de defecto* (OD), representa un posible defecto en cada unidad de entrada/salida importante para los requisitos o especificaciones del cliente. Así por ejemplo, en un formulario de registro con cinco entradas para que el usuario introduzca sus datos (nombre, dirección de email, región, código postal y edad) existen, *a priori*, cinco oportunidades de defecto. Si una de las entradas es de opción múltiple, con cuatro posibles respuestas, se contabilizará como 3, ya que hay tres oportunidades de error, dentro de las cuatro posibles.
- *Defectos por oportunidad* (DPO), valor que representa el cociente entre el número total de defectos y el número total de oportunidades. Si, por ejemplo, tenemos 25

defectos en 100 unidades, en cada una de las cuales hay cuatro oportunidades, DPO será igual a $25/4 \times 100$, es decir, $DPO = 0,0625$. Se trata de un valor que no suele utilizarse directamente, sino para calcular la tasa de defectos por millón.

- *Defectos por millón de oportunidades (DPMO)*, número medio de defectos por unidad observados durante un cierto número de ejecuciones del sistema, que suele normalizarse a un millón. Para calcular DPMO se ha de multiplicar DPO por un millón.

La filosofía Six Sigma aboga por obtener un amplio abanico de medidas acerca del número de defectos encontrados, para posteriormente utilizarlas en un conjunto de procesos orientados a la mejora de la calidad del software producido:

1. Reducir los defectos en el software, mediante la realización de pruebas de unidad, de integración y del sistema.
2. Encontrar y arreglar los defectos lo más cerca posible de su origen, realizando inspecciones y adelantando a fases tempranas los procesos de detección de defectos.
3. Predecir los porcentajes de defectos encontrados y reparados durante el desarrollo y tras la entrega al cliente, para lo cual se recomienda utilizar modelos de introducción y reparación de defectos, así como modelos de estimación de defectos.

La aplicación de Six Sigma permite una gestión cuantitativa de la calidad del producto, lo que en definitiva facilita las pruebas de integración, permite desarrollar productos de mayor calidad con pocos defectos latentes y en general, mejora la predictibilidad del proceso de desarrollo en su conjunto. Sin embargo, debe tenerse en cuenta que la introducción de Six Sigma hace necesario modificar el enfoque de desarrollo y gestión del proceso, y que algunos modelos, como el proceso en cascada, no soportan bien una filosofía de cambio continuo como la que propugna Six Sigma.

9.6 Resumen

La calidad es hoy día un concepto de gran importancia en todos los ámbitos, y también, por supuesto, en la Ingeniería del Software. Todos somos capaces de reconocerla cuando la vemos, tanto en un producto, como en un proceso o servicio. Sin embargo, nos resultaría difícil elaborar una definición que abarcara todos sus matices. En este capítulo hemos presentado el concepto de calidad en relación con nuestra disciplina, y analizado su importancia dentro del desarrollo de software, pero hemos hecho especial énfasis en las perspectivas del producto y del proceso de producción.

Los modelos clásicos de calidad, como el de McCall, el de Boehm, o el que define el estándar de calidad ISO/IEC 9126, se decantan por un concepto de calidad que coincide con la perspectiva del producto de Galvin. Así, conciben la calidad del software como el grado en el que un software posee una cierta combinación de atributos «de calidad». Todos

estos métodos definen de hecho un conjunto de atributos que pueden medirse para evaluar el nivel de calidad de un producto software.

Actualmente, sin embargo, la calidad del software se enfoca preferentemente desde la perspectiva del proceso de producción. El concepto de *aseguramiento de la calidad*, que significa esencialmente «proporcionar las evidencias necesarias para garantizar que la función de calidad se realiza adecuadamente» es, en esta perspectiva, especialmente importante para garantizar que, tanto los productos como los procesos de producción del software cumplen con los requisitos de calidad establecidos.

Dentro de los modelos y estándares que dan soporte a esta visión de la calidad, hemos estudiado los más modernos y ampliamente difundidos, muchos de los cuales se definen como modelos de madurez y mejora continua de procesos, y un gran número de ellos, como CMMI, la familia de estándares ISO 9000, ITIL o TickIT están asociados, además, a esquemas de certificación. Otros, como TSP/PSP, Bootstrap, SPICE, Trillium o Six Sigma, comenzaron como iniciativas más limitadas –o locales– y con el tiempo, se han instaurado firmemente en el mundo de la calidad dentro de las tecnologías de la información y la Ingeniería del Software.

Al igual que en capítulos precedentes, en la siguiente nube de palabras se muestran los principales conceptos que hemos tratado. Esta figura resume de una forma intuitiva la importancia relativa de cada concepto del capítulo en comparación con los demás.



Figura 9.15: Principales conceptos tratados en el capítulo

9.7 Notas bibliográficas

El libro donde Wiegers explica su concepto de cultura de la Ingeniería del Software (Wiegers, 1996), recibió en su día el premio a la productividad otorgado por la revista *Software Development*. En él se describen catorce principios importantes que deben guiar la forma en que se construye el software. Para aquellos interesados en ideas que sirvan para mejorar la cultura de la ingeniería en sus organizaciones, ésta es la referencia fundamental.

El código de ética y práctica profesional de la Ingeniería del Software elaborado por la ACM y la IEEE Computer Society fue publicado en español en el número 140 de la revista *Novática*. Para aquellos que prefieran consultar la referencia original en inglés, ésta puede encontrarse en (Anderson, 1992).

En un famoso artículo denominado «*Calidad del software: el objetivo escurridizo*», Kitchenham y Pfleeger (1996) hacen un interesante análisis de los conceptos y visiones de la calidad, y proporcionan un resumen de bibliografía sobre artículos científicos relacionados con el estudio de la calidad desde los principios hasta el momento de su publicación. Aunque desde entonces han surgido ideas y modelos nuevos, no deja de ser una lectura recomendada para quienes quieran saber más sobre la calidad en el software.

Animamos a aquellos lectores interesados en profundizar en los modelos de calidad que no hemos podido tratar con el detalle que habría sido necesario, a seguir estudiando dichos métodos a partir de las referencias fundamentales de cada uno:

FURPS:	(Grady, 1992)
Familia ISO 9000:	(Bamford y Deibler, 2004)
CMMI:	http://www.sei.cmu.edu/cmmi/
TickIT y TickIT+:	http://www.tickitplus.org/
SPICE:	Estándares ISO/IEC 15504-1 a ISO/IEC 15504-6
Bootstrap:	(Staff, 1993)
ITIL:	(Bon, 2002)
Trillium:	(April y Coallier, 1995a) y (April y Coallier, 1995b)

Sobre PSP y TSP, las referencias fundamentales son de Watts S. Humphrey. Sobre PSP, sobre todo su «*Introducción al Proceso Software Personal*» (Humphrey, 2001). En cuanto a la integración entre TPS y PSP recomendamos «*Introduction to the Team Software Process*» (Humphrey, 1999) y «*TSP: Leading a Development Team*» (Humphrey, 2005).

Un libro ya incluido en las notas bibliográficas del capítulo de pruebas, y que puede resultar también de interés desde el punto de vista de la calidad, es la segunda edición de «*Software Testing*» (Patton, 2005), donde se analiza desde un punto de vista práctico la relación entre las prueba de software y los procesos de aseguramiento de la calidad.

Finalmente, recomendamos encarecidamente la lectura de las referencias clásicas sobre calidad. El artículo donde Dromey esbozó su modelo de calidad (Dromey, 1995), o las monografías de Ishikawa (1985) y Juran (1988), donde se plantean los fundamentos de la calidad, tratan conceptos fundamentales y sirvieron de inspiración para trabajos posteriores.

9.8 Cuestiones de autoevaluación

- 9.1 ¿Qué es el aseguramiento de la calidad?
- R *Dentro de las muchas definiciones posibles del término, tal vez la definición de Juran sea la más aceptada: «la actividad de proporcionar las evidencias necesarias para garantizar que la función de calidad se lleva a cabo adecuadamente».*
- 9.2 ¿Cuáles son las cinco perspectivas de la calidad de Garvin?
- R *La visión trascendental de la calidad, la perspectiva del usuario, la perspectiva de la producción, la perspectiva del producto y la perspectiva del valor.*
- 9.3 ¿Cuáles son los modelos clásicos de calidad?
- R *El modelo de McCall, el de Boehm y el que define el estándar de calidad ISO/IEC 9126 son los más conocidos, aunque existen algunos otros.*
- 9.4 Cite, al menos, cinco factores de calidad definidos por el modelo de calidad de McCall.
- R *Cualquiera dentro de los once que define el modelo, y que son: facilidad de mantenimiento, flexibilidad, facilidad de evaluación, reusabilidad, portabilidad, corrección, fiabilidad, interoperabilidad, eficiencia, integridad y facilidad de uso.*
- 9.5 ¿Cómo está estructurado SPICE?
- R *Se trata de un estándar multipartito, formado por 5 documentos: ISO/IEC 15504-1 (conceptos y vocabulario), ISO/IEC 15504-2 (cómo realizar una evaluación), ISO/IEC 15504-3 (guía para realizar una evaluación), ISO/IEC 15504-4 (uso del estándar para la mejora y determinación de la capacitación de procesos) y finalmente ISO/IEC 15504-5 (ejemplo de modelo de evaluación de procesos).*
- 9.6 ¿Cuáles son los dos modelos de representación presentes en CMMI?
- R *CMMI establece dos formas diferentes de medir el modo en que las organizaciones pueden mejorar sus procesos, la representación continua, que utiliza como medida los niveles de capacitación, y la representación por etapas.*
- 9.7 ¿En qué medida pueden emplearse PSP y TSP conjuntamente con CMMI?
- R *Aunque no es imprescindible su empleo conjunto, el creador de PSP y TSP, Watts Humphrey, aboga por una utilización complementaria de los 3: CMMI para la organización, TSP para los equipos de trabajo y PSP a nivel individual.*
- 9.8 ¿Cómo clasificaría el método Bootstrap?
- R *Bootstrap es una metodología para la evaluación de procesos de software, la medición cuantitativa y la mejora continua, creada como parte de un proyecto de investigación financiado por la UE y validada mediante un proceso de prueba en diversas organizaciones.*
- 9.9 ¿Cuál es la relación entre ISO/IEC 15504 y los estándares CMMI e ISO 9001?
- R *ISO/IEC 15504 está basado en las ideas de CMMI y de ISO 9001, pero a diferencia de éstos, representa el intento de ISO/IEC para armonizar varios modelos diferentes, lo que incluye, por supuesto, a los anteriores, pero también a CMM, ISO 12207, Trillium y Bootstrap, entre otros. Siendo así, ISO/IEC 15504 es más genérico, pues se ha intentado que abarque a los demás para que los modelos conformes con ellos también lo sean con ISO/IEC 15504.*

9.10 Uno de los elementos que caracterizan a Trilium con respecto a otros enfoques son los denominados *mapas de ruta*. ¿Qué es un mapa de ruta en este método?

- R *Un mapa de ruta es, en Trilium, un conjunto de prácticas relacionadas que se aplican a un área o necesidad concreta de la organización. Cada mapa representa una capacitación significativa, siendo el progreso en dicho mapa lo que marca el ascenso en los niveles de madurez de la organización.*

9.9 Ejercicios y actividades propuestas

9.9.1 Ejercicios resueltos

- 9.1** Compare los modelos de calidad de McCall y Boehm.

Solución propuesta: Ambos enfoques, que forman parte de los denominados modelos clásicos de calidad, tienen una estructura jerárquica, y su propósito es similar. En los dos modelos existen características del producto o factores de calidad, muchos de los cuales se repiten.

El modelo de Boehm es algo más elaborado al incluir un nivel jerárquico superior adicional que no existe en el de McCall, donde se establece el uso del software, para a partir de este uso realizar la categorización de los factores de calidad.

La siguiente tabla resume sus diferencias:

McCall (1977)	Boehm (1977)
Jerárquico en 2 niveles	Jerárquico en 3 niveles
3 perspectivas del producto: revisión, transición y operación	3 utilidades del software: facilidad de mantenimiento, utilidad como está y portabilidad
11 factores de calidad	7 factores de calidad
23 elementos que pueden ser medidos	15 elementos que pueden medirse

- 9.2** Se pretende aplicar el método Six Sigma al software de una compañía de comercio electrónico por internet. Una revisión de las oportunidades de defectos de uno de sus formularios de pedido incluye los siguientes campos a rellenar por parte del usuario: número de la tarjeta con la que se realizará el pago*, fecha de caducidad de la tarjeta*, fecha de emisión de la tarjeta, importe total del pedido*, nombre del titular*, aceptación por parte del titular de las normas de entrega de la compañía y nivel de estudios del cliente. De las anteriores oportunidades, aquellas marcadas con un asterisco están asociadas a errores críticos para el proceso.

Si se toman datos de 200 pedidos y se encuentran 60 errores en 30 pedidos, ¿cuál es la tasa de defectos por cada millón de oportunidades (DPMO)?

Solución propuesta: En primer lugar hemos de calcular el número de defectos por oportunidad (DPO), valor que representa el cociente entre el número total de defectos y el número total de oportunidades. Así:

$$DPO = \frac{60 \text{ errores}}{200 \text{ pedidos -4 críticos}} = 0,075$$

pero, como es sabido, este valor se emplea para calcular la tasa de defectos por millón, multiplicándolo por 10^6 :

$$DPMO = DPO \cdot 10^6 = 0,075 \cdot 10^6 = 75,000$$

- 9.3** Como usuario de productos de software, usted se habrá encontrado a lo largo de su vida con numerosos fallos. Intente recordar algunos de los que más hayan llamado su atención y resuma en un par de líneas cómo dichos fallos afectan a la calidad del producto software desde la perspectiva de alguno de los modelos clásicos estudiados durante el capítulo.

Solución propuesta: Lógicamente las respuestas a esta cuestión dependerán en gran medida de las experiencias personales individuales de cada usuario de software. Sin embargo, la información esencial que se solicita es posible reflejarla mediante un informe cuyo formato sea similar al siguiente:

Descripción del fallo	...
Características de calidad incumplidas	...
Possible modo de repararlo (sugerencias de modificación)	...
Modelo de calidad utilizado como referencia	...

- 9.4** Imagine un sistema de venta de entradas en un cine, donde los clientes pueden comprar sus boletos bien en las taquillas –al modo tradicional– o bien en máquinas instaladas a tal efecto, donde, además, es posible consultar los datos de las sesiones y las películas que se proyectan. Todos los datos de las sesiones son introducidos en el sistema por personal especializado, que da de alta sesiones, películas, información sobre los actores, sinopsis, etc.

Analice el sistema propuesto e indique desde cuál de las siguientes perspectivas sería preferible evaluar la utilidad del sistema si dicha evaluación va a llevarse a cabo siguiendo el modelo de calidad Boehm:

- Desde la perspectiva del propietario del cine.
- Desde la perspectiva de los taquilleros y los clientes del mismo.
- Desde la perspectiva de quienes mantienen el sistema.

Solución propuesta: Como sabemos, el modelo de calidad de Boehm evalúa la utilidad de un sistema desde tres puntos de vista: su utilidad tal y como está (lo cual resultará adecuado tanto para los cajeros y clientes del sistema), su facilidad de mantenimiento (adecuado para las personas que mantienen el sistema, quienes estarán más satisfechos si éste es fácil de

entender y modificar) y finalmente desde el punto de vista de la portabilidad, que resulta importante para el propietario del sistema, ya que se asegura que si es necesario un cambio de plataforma su sistema seguirá siendo útil y funcional.

Es por ello por lo que la utilidad de este sistema debería evaluarse según los puntos de vista de los tres tipos de usuarios identificados.

9.9.2 Actividades propuestas

- 9.1 Comparar, tras un estudio exhaustivo que se apoye en las referencias proporcionadas al final del capítulo, los métodos Trillium, Bootstrap y CMMI, apuntando las diferencias que existen entre dichos modelos.
- 9.2 Acceda a una de las múltiples páginas de Internet desde las que es posible descargar shareware (software disponible gratuitamente). Seleccione alguna aplicación de baja calidad, utilizando como regla práctica para ayudarse en el proceso de selección aquellas cuya valoración por parte de los usuarios sea especialmente baja. Una vez seleccionada e instalada en su computadora, haga una lista de sus fallos, apuntando las características de calidad que viola cada fallo. Tome como referencia el modelo de calidad de Boëhm.
- 9.3 Durante el proceso de auditoría de un sistema de software, los auditores han de visitar los diferentes departamentos de una gran compañía y consignar en un formulario a tal efecto un conjunto de datos tomados de la instalación a auditar. En un formulario, de los varios que se han de llenar, los auditores deben consignar la siguiente información sobre el personal contratado: número de personas contratadas*, número de programadores*, número de técnicos de soporte, número de roles distintos definido, número de gestores por equipo*, número de puestos de trabajo físicos y porcentaje de puestos de trabajo no utilizables. De todos estos datos, los que se identifican con un asterisco son aquellos que resultan críticos y que pueden hacer que –si son consignados erróneamente– la auditoría resultante no tenga validez.
Al final del proceso los auditores visitan 50 departamentos, y de éstos, tan sólo en 15 encuentran errores, concretamente 85 errores repartidos entre estos 15 departamentos. Calcule la tasa de defectos por cada millón de oportunidades (DPMO) según el modelo Six Sigma.
- 9.4 Philip B. Crosby, reconocido experto en calidad norteamericano, afirma que la calidad es algo que los individuos y las instituciones perciben de manera subjetiva, lo que no resulta útil en la Ingeniería del Software. Según este autor, en nuestra disciplina la calidad debería definirse como la conformidad con los requisitos, siendo la ausencia de conformidad un problema de calidad. En consecuencia, los problemas de calidad se transforman en problemas de falta de conformidad, lo que hace que la calidad pueda ser claramente definida. Contraste la postura de Crosby con las teorías y modelos de calidad estudiados en el capítulo.
- 9.5 Aparte de los modelos clásicos de calidad estudiados en el capítulo, existen algunos otros que, por las limitaciones lógicas de un texto como el presente, no han podido ser incluidos. El modelo de calidad de Dromey, por ejemplo, resulta interesante como complemento de lo estudiado al respecto de los modelos clásicos, sobre todo si se lo compara con los modelos de McCall y Boëhm. Consulte alguna referencia en la bibliografía donde se explique ese modelo de calidad y compárello con los modelos de calidad de McCall y Boëhm, resumiendo sus similitudes y diferencias en una tabla.

- 9.6 Dentro de los modelos clásicos de calidad, el estándar ISO/IEC 9126 se basa en los modelos de McCall y Boéhm, pero incluye ciertas modificaciones que se han destacado en el capítulo. Compare los modelos de calidad de McCall y Boéhm con el modelo del estándar ISO 9126 y resuma sus diferencias en una tabla.
- 9.7 Existe un interesante informe, denominado «*Cómo acelerar el proceso de mejora mediante la integración de TSP y CMMI*» (Wall, McHale y Pomeroy-Huff, 2006), donde se describe cómo dos organizaciones involucradas en la organización y control de sistemas aeronavales norteamericanos (NAVAIR) integraron la metodología TSM y el modelo de madurez de CMMI para progresar desde un nivel CMMI inicial 1 hasta un nivel 4 en 30 meses (menos de la mitad del tiempo medio para conseguirlo). En grupo, analice el informe y discuta sobre los factores clave que permitieron a NAVAIR conseguir algo así, analizando especialmente la influencia que TSP pudo tener en la aceleración de los tiempos medios de adquisición de un nivel 4 de madurez.
- 9.8 El artículo «*Tres perspectivas del proceso: la organización, los equipos y las personas*» (Humphrey, 2002) muestra la visión personal de Watts Humphrey sobre el proceso de mejora de software. Este autor, que cuenta con una muy dilatada experiencia en la asesoría tecnológica a compañías de primer nivel, describe en dicho artículo algunos de los esfuerzos en los que se ha visto involucrado, como el desarrollo de los primeros métodos de evaluación de procesos, el diseño de CMM o la introducción de PSP y TSP dentro de los procesos de mejora. Lea el artículo y haga una lista de los problemas que, desde su punto de vista, resulten más relevantes entre los que menciona Humphrey, y que la creación y publicación de los métodos de mejora estudiados intentan atajar.
- 9.9 Consulte unos cuantos números –recientes– de revistas científicas donde se trate la calidad del software, como el «*Software Quality Journal*» de la editorial Springer o el «*Software Quality Professional*» que publica la asociación americana de calidad (ASQ) y evalúe las cuestiones más en boga hoy en día dentro del área de la calidad del software desde una perspectiva científica e investigadora. La actividad le resultará aún más enriquecedora si compara los resultados obtenidos con la consulta de unos cuantos artículos recientes de *Better Software magazine*, una publicación comercial orientada a otro público bien distinto: los profesionales y gestores de compañías de desarrollo de software.
- 9.10 Elabore un pequeño catálogo sobre las compañías de desarrollo de software de su país que hayan sido certificadas en el nivel 5 de CMMI, las fechas en que obtuvieron dicha certificación y si es posible, el tiempo que les llevó conseguirlo. Puede para ello consultar la información disponible en internet, y dado que generalmente las compañías están interesadas en publicitar un hecho así, no le será complicado encontrar información al respecto.

10

Gestión

Sacar del equipo a una persona que rinde a un mal nivel puede ser más productivo que añadir a otra muy buena.
— Tom DeMarco

10.1 El desarrollo de proyectos no es sólo tecnología

Hay en la Ingeniería del Software muchos ejemplos de mala gestión, pero existe un caso especialmente cómico –si no fuera por lo trágico– donde todo aquello que podía haber ido mal fue mal: el caso del servicio de ambulancias londinense a principios de los años 1990. El servicio de ambulancias londinense, quizás el más grande del mundo en aquel momento, se gestionaba de forma completamente manual. Los operadores, al recibir una llamada de emergencia, rellenaban un formulario de papel que remitían a un sitio central en el que se decía a qué división de las tres en las que se descomponía el servicio de ambulancias (noreste, noroeste y sur) se asignaba la emergencia. Después se asignaban los recursos y se enviaban al lugar donde eran requeridos.

La organización adolecía de diversos problemas, como la falta de precisión en la localización del accidente o el exceso de papel en el proceso. Por ejemplo, la identificación de llamadas duplicadas se hacía de forma manual, lo que tenía un impacto negativo en los tiempos de respuesta –desde que se realizaba la llamada hasta que la ambulancia se presentaba en el lugar del accidente–, hasta el punto de no cumplir con los estándares establecidos para tiempos de llegada al accidente. La solución era aparentemente sencilla: crear un sistema automático capaz de reconocer llamadas duplicadas y de localizar y movilizar ambulancias eficientemente y con la mínima intervención humana posible.

Antes de describir en detalle los fallos de este proyecto, situémoslo en su contexto. Por aquellas fechas no existían los sistemas GPS, y los sistemas automáticos de rutas habían sido cancelados o simplemente no intentados en otros 25 de los 62 proyectos planificados para servicios similares en el Reino Unido. Además, el proyecto se desarrolló en un momento en el que el sistema sanitario británico se encontraba en plena reorganización y había problemas de disputas laborales y huelgas. En este contexto, se inició la especificación y licitación del sistema. Para ello se creó un comité que apenas contaba con personal de ambulancias, si bien los requisitos y el diseño del nuevo sistema incluían las nuevas prácticas laborales. La especificación se completó en febrero de 1991 sin la firma formal de sus componentes, estableciéndose como fecha de entrega final del sistema el 8 de enero de 1992. En mayo de 1991 se asignó el contrato al consorcio, lo que dejaba un exiguo plazo de desarrollo de 6 meses, con el agravante de que en septiembre se incluyeron nuevos requisitos.

A partir de aquí, enumeraremos algunos de los problemas que fueron surgiendo durante el proyecto:

- *Selección del contratista.* En la fase de licitación se recibieron 17 propuestas, y teníase en cuenta que por ley se debía aceptar la oferta de menor coste. La oferta del consorcio ganador era mucho más baja que el resto con diferencia, pero no se cuestionó nada al respecto: las referencias de las empresas que lo integraban no fueron analizadas con detalle y las opiniones negativas de varios asesores externos no fueron tenidas en cuenta.
- *Planificación inicial.* En lo referente al hardware, se optó por una arquitectura relativamente nueva en aquella época: PC en modo distribuido siguiendo una arquitectura cliente/servidor con distintos lenguajes. En cuanto al software, para ciertas partes se optó por MS Visual Basic Versión 1, que aún no estaba en su versión final. Por último, todo el desarrollo debía de hacerse en una sola fase, sin tiempo asignado para hacer revisiones.
- *Gestión de recursos humanos.* No había nadie contratado a tiempo completo, y las condiciones laborables eran pobres. Además, se utilizó una metodología en la que el personal no tenía experiencia previa.
- *Implementación y pruebas.* Después de haber planificado la construcción en una sola fase, se pasó a un modelo en tres fases a través de las cuales iban integrándose los subsistemas progresivamente. En cuanto a las pruebas, aunque se llevaron a cabo pruebas funcionales y de carga, éstas nunca se realizaron con el sistema completo. Además, la integración del subsistema de movilización de ambulancias y el de comunicaciones no fueron probados suficientemente. Los sistemas de respaldo (*backup*) eran incompletos y no se habían probado antes de poner el sistema en operación.

Finalmente, el sistema se puso en operación a las 7.00 am del lunes 26 de octubre de 1992 y esa misma mañana, en plena hora punta, el sistema ya perdía llamadas. Este hecho

generaba más llamadas duplicadas y esperas de hasta 30 minutos. Tampoco se reconocían ciertas carreteras, por lo que a mitad del día el volumen de llamadas ahogaba el sistema. En realidad, casi nada funcionó correctamente: localización inadecuada del vehículo más cercano al accidente, sobrecarga en las comunicaciones, bloqueo de las estaciones de trabajo, lentitud general de la operación, identificación incorrecta de las llamadas duplicadas (lo que hacía que fueran innecesariamente asignadas varias ambulancias a un mismo accidente, y que provocaba una mayor carga en el sistema de lo necesario) o problemas de los reportes del personal (por ejemplo, al llegar a un accidente o al hospital, no se indicaba el código correspondiente o se indicaba mal). Al día siguiente se apagó parte de sistema para hacerlo semi-manual, siendo necesario emplear personal extra para ello. Diez días después de su puesta en producción (concretamente el 4 de noviembre a las 2:00 am), el sistema se bloqueó de tal modo que no pudo ser reiniciado, por lo que se apagó definitivamente y se volvió al sistema manual.

La raíz de todos los problemas relatados fue la pobre gestión de este proyecto, desde los recursos humanos a la estimación del tamaño de los sistemas, pasando por supuesto por la gestión del tiempo necesario para implementar las especificaciones y probar suficientemente los componentes del sistema hasta su puesta en operación. Como parece evidente, tal desastre tuvo sus consecuencias. El 28 de octubre de 1992, a los dos días de ponerse en marcha el sistema, el director ejecutivo dimitió. Más adelante, en febrero de 1993, el Gobierno Británico decidió abrir una investigación tras la cual el presidente de LAS tuvo también que dejar su puesto. La investigación determinó, entre otros resultados, que un futuro sistema debería construirse en un plazo no menor de 4 años y con un presupuesto muchísimo más elevado.

El caso descrito es la prueba evidente de que muchos de los problemas que afectan a la ingeniería del Software están relacionados con la gestión, desde la licitación de los proyectos hasta su ejecución. En el resto de este capítulo estudiaremos cómo resolver las dificultades asociadas a esta parte tan importante de la disciplina, prestando especial atención a la estimación de esfuerzo y plazos, a la gestión de los recursos (para evitar mantener al personal descontento y bajo presión) o a la correcta elección de metodologías de desarrollo y tecnologías adecuadas, entre otros.

10.2 Objetivos

El objetivo fundamental de este capítulo es comprender la importancia de la gestión de un proyecto de desarrollo de software, así como las diferentes tareas a realizar y los modelos de gestión existentes. Al final de este capítulo, el lector debería:

- Comprender la problemática inherente a la gestión de un proyecto de desarrollo de software.
- Conocer que existen diferentes formas de afrontar la gestión de un proyecto de software y más concretamente:

- Conocer las características generales de un modelo de gestión de proyectos.
 - Comprender los principales modelos de gestión existentes.
- Saber utilizar técnicas específicas de gestión tales como la estimación, planificación y seguimiento de proyectos.

10.3 Visión general de la gestión de proyectos

De entre las múltiples definiciones de gestión de proyectos existentes, destacamos la dada por la guía del conocimiento de la gestión de proyectos (PMBOK – *Project Management Body of Knowledge*):

La gestión de proyectos es la aplicación de conocimiento, habilidades, herramientas y técnicas a las actividades de un proyecto para cumplir los requisitos del mismo

Otra definición interesante, pues se refiere a proyectos específicamente de nuestra disciplina, es la del glosario IEEE de términos de Ingeniería del Software:

La gestión en la Ingeniería del Software se puede definir como la aplicación de las actividades de gestión –planificación, coordinación, medición, monitorización, control y realización de informes– para asegurar que el desarrollo y el mantenimiento del software se realiza de una forma sistemática, disciplinada y cuantificable

En los más importantes estándares y guías de referencia (el estándar IEEE/EIA 12207, las guías SWEBOK y PMBOK), la gestión de proyectos se lleva a cabo a través de los siguientes procesos:

1. *Iniciación y alcance del proyecto.* Incluye todas las actividades necesarias para decidir si el proyecto debe llevarse a cabo. Estas actividades incluyen la determinación y alcance de los objetivos, los estudios de viabilidad del proyecto y la revisión e inspección de los requisitos.
2. *Planificación del proyecto.* Incluyen las actividades de preparación del proyecto tales como la *planificación del proceso*, donde se decide el ciclo de vida y las herramientas a utilizar, la *determinación de entregables*, la *estimación de coste, plazos y esfuerzo*, la *asignación de recursos* a actividades, la planificación de la *gestión de riesgos* y la *gestión de la calidad*.
3. *Seguimiento del proyecto.* Una vez que el proyecto está en marcha resulta necesario un constante control del mismo que permitirá llevar a cabo acciones correctivas si

la planificación y el actual curso del proyecto divergen. Otras tareas de seguimiento son el mantener alta la moral del equipo y, dependiendo del proyecto, mantener a la dirección de la organización informada del progreso.

4. *Revisión y evaluación del proyecto.* Es importante analizar si los requisitos han sido satisfechos, así como la eficiencia y eficacia con la que se ha llevado a cabo el proyecto.
5. *Cierre del proyecto.* Análisis *post-mortem* del proyecto, donde se aprende de los errores y se analizan posibles mejoras de cara a futuros proyectos.

Para ello, los gestores de proyectos deben mantener en equilibrio los tres parámetros principales que definen un proyectos software a lo largo de su ciclo de vida:

- La funcionalidad que el sistema en construcción ha de proporcionar.
- El plazo en que se debe desarrollar.
- Los recursos de los que se dispone para ello (económicos, humanos, herramientas, métodos, etc.)

Estos parámetros, que a menudo se ven como independientes, influyen unos en otros de manera importante. Así por ejemplo, si en un cierto momento se decide aumentar la funcionalidad esto implicará probablemente un aumento del plazo y/o de los recursos. Por otro lado, una disminución en el plazo traerá como consecuencia una reducción, bien de la funcionalidad o bien de la calidad del producto a desarrollar. En resumen, hay que llegar a una solución de compromiso entre las tres dimensiones que componen lo que se denomina *triángulo mágico* (ver Figura 10.1) para lograr el equilibrio al que antes nos hemos referido, y son los gestores de proyectos quienes se han de encargar de equilibrar estas tres variables a lo largo del proyecto para mantener constantes los objetivos del mismo.

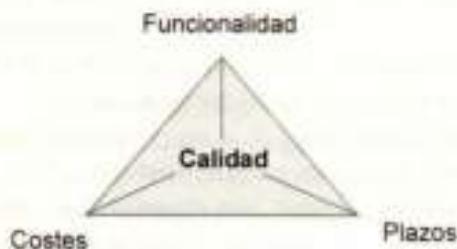


Figura 10.1: El triángulo mágico en la gestión de proyectos

Si como indica DeMarco «*No puede controlarse lo que no se puede medir*», parece evidente que para llevar a cabo la gestión de proyectos, la organización debe establecer un conjunto de procesos de medición. La guía SWEBOK resalta la importancia de la medición

(cuyos fundamentos vimos en el Capítulo 3) y de la calidad (Capítulo 9) en la gestión de proyectos, así como la necesidad de establecer y mantener los compromisos de medición, planificar el proceso de medición, llevarlo a cabo y evaluar tanto procesos como productos para identificar posibles mejoras en ambos.

10.4 La estimación de coste, plazos y esfuerzo

En general, se puede definir *estimación* como el proceso de determinar una variable no conocida a partir de otras conocidas. La estimación de coste, plazos y esfuerzo es parte esencial de la planificación de cualquier tipo de proyectos, y no lo es menos en el caso de la Ingeniería del Software.

Estimar no es una tarea fácil, por múltiples razones, desde aspectos relacionados con el proyecto (por ejemplo, proyectos nuevos sin parecido a otros realizados anteriormente, o en los que hay que utilizar nueva tecnología, proyectos con requisitos cambiantes o pobremente definidos, etc.), hasta otros otros relacionados con la estimación en sí misma (por ejemplo, falta de experiencia en técnicas de estimación, utilización de métodos de estimación que no son apropiados, falta de registros históricos de proyectos que permitan reutilizar experiencias pasadas, etc.).

La correcta estimación del coste y el esfuerzo es una tarea crítica. *Subestimar* supondrá casi con seguridad que el proyecto se llevará a cabo con pérdidas para la organización, pues al no poder dedicarle los recursos necesarios, es muy posible que el proyecto se retrase, que el producto resultante sea de una calidad inferior a la requerida y que el personal tenga que hacer horas extra (a costa de la propia moral del equipo). En un caso extremo, el proyecto tendrá que ser cancelado. Una *sobreestimación*, por el contrario, lleva a lo que se conoce humorísticamente como la «ley de Parkinson»: *el trabajo se expande hasta llenar todo el volumen disponible*. Cuando se trata de proyectos que han de desarrollarse tras licitación pública, hay más posibilidades de perder el concurso de licitación en favor de la competencia, ya que en estos casos suele utilizarse la técnica de «coste ganador» –*price to win* en inglés–. Según esta técnica, lo que se estima fundamentalmente es un presupuesto que supere al de los competidores para después ajustar el resto de parámetros que permitirán llevar a cabo el proyecto en función de dicho presupuesto.

Para llegar a una correcta estimación de coste y esfuerzo, evitando los problemas asociados a una estimación inadecuada, es necesario establecer con claridad los objetivos de proyecto, procurar que los requisitos estén bien especificados y recabar toda la información disponible en ese momento. Todo ello debería realizarse varias veces durante el desarrollo, pues según avanza el proyecto tendremos más información disponible: si las nuevas estimaciones varían mucho de las originales, deberán tomarse medidas correctivas.

En cuanto a los métodos de estimación, existen varios, y no puede decirse que unos sean mejores que otros, o que haya uno que aporte los mejores resultados. Resulta habitual, en consecuencia, aplicar más de uno en un mismo proyecto, pues unos métodos pueden complementar a otros como forma de verificar las estimaciones.

La mayoría de los métodos de estimación utilizan datos de proyectos anteriores para calcular las nuevas predicciones, con diferente grado de acierto. A continuación se describen brevemente algunas categorías en las que se pueden agrupar las técnicas de estimación:

- Estimación basada en el juicio (experiencia) de expertos. Esta técnica se basa simplemente en la experiencia y conocimiento de personas que han hecho un cierto número de estimaciones en el pasado.
- Modelos algorítmicos o paramétricos. El coste y esfuerzo se calcula mediante ecuaciones que tienen en cuenta distintas variables de un proyecto. La técnica de los puntos de función pertenece a esta categoría, aunque dada su importancia –es quizás el método más utilizado en la actualidad– merece mención aparte y dedicaremos a su descripción una sección separada.
- Otros métodos. Principalmente aquellos basados en técnicas de inteligencia artificial y minería de datos.

10.4.1 Estimación mediante juicio de expertos

La **estimación mediante juicio de expertos** está basada en el conocimiento y experiencia de proyectos similares. Aunque existen estudios empíricos que muestran que las estimaciones son habitualmente bastante acertadas en sí mismas, esta técnica suele utilizarse como confirmación de los resultados de otras técnicas de estimación, o como factor de corrección. Uno de sus mayores inconvenientes consiste en que se obtienen como *caja negra*, pues no definen detalladamente cómo se ha llegado a la estimación.

Entre las técnicas más conocidas de este tipo destacan el método *Delphi* y la *estimación por analogía*, si bien para poder aplicar esta última es necesario disponer de un archivo histórico con información de proyectos pasados.

Método Delphi

Este método fue creado en los años 1940 en el ámbito militar, donde se empleaban juntos una serie de cuestionarios y las opiniones de expertos para obtener predicciones. En la Ingeniería del Software, fue Boehm (1981) quien formalizó y adaptó esta técnica para las estimaciones de tamaño del software bajo la denominación *Wideband Delphi* (lo que podría traducirse como *Delphi de banda ancha*).

Se trata de una técnica en la que existe un coordinador que proporciona a cada experto una especificación y un formulario. El coordinador reúne a los expertos para intercambiar opiniones sobre la estimación y al final de la reunión, cada experto rellena su formulario de forma anónima. A continuación, el coordinador proporciona a los expertos un resumen de la estimación y se organizan nuevas reuniones para debatir las estimaciones de la ronda anterior hasta el punto donde ya no sean necesarias más revisiones. No obstante, existen

diversas variantes de estos pasos básicos, como por ejemplo, comentar las estimaciones por adelantado y justificar los costes entre los expertos.

De los beneficios del método, Steve McConnell se hace eco en su libro «*Estimación de software: desmitificando este arte oscuro*» (McConnell, 2006). El autor comparó los datos de un conjunto de estimaciones con los datos finales tomados al término de los proyectos, obteniendo como resultado que en un 33% de los casos en los que se empleó la técnica Delphi la estimación correcta estaba dentro del rango delimitado por los estimadores.

Estimación de expertos por analogía

Esta técnica consiste en realizar la estimación del proyecto actual comparando dicho proyecto con otros anteriores, de características y ámbito similares. Obviamente, se presupone que la organización mantiene una base de datos con proyectos realizados, o utiliza algún repositorio disponible. Uno de los más importantes es el mantenido por ISBSG (*International Software Benchmarking Standards Group*), un repositorio de métricas de gestión de proyectos de diferentes sectores (banca, telecomunicaciones, etc.), donde se pueden encontrar proyectos similares al que queremos estimar en relación al tamaño, lenguajes, tipo de base de datos, uso de herramientas CASE, etc., y consecuentemente, predecir el esfuerzo necesario total y por fases del ciclo de vida, la productividad, el tiempo de desarrollo, la tasa de defectos, etc.

10.4.2 Puntos de función

Los **puntos de función** –originalmente creados por Albrecht a finales de los años 1970 en IBM– son una de las técnicas más usadas para la estimación de tamaño en proyectos software. Existen variantes a este modelo, siendo las más conocidas los puntos de función IFPUG y COSMIC.

De modo simplificado, y para entender el funcionamiento de los puntos de función en general, diremos que se basan en la cuenta de elementos para medir su funcionalidad. Por ejemplo, en IFPUG se hace uso del concepto de punto de función no ajustado (UPF, *Unadjusted Function Points*) entre los que se consideran:

- *Entradas externas* (entradas de usuario), por ejemplo, selecciones de menú.
- *Salidas externas* (información para el usuario), como mensajes o informes.
- *Consultas*: entradas interactivas que requieren una respuesta del sistema.
- *Ficheros externos*: son las interfaces con otros sistemas.
- *Ficheros internos*: cualquier entidad persistente manejada por el sistema.

Estos elementos se clasifican según su complejidad (simple, media o compleja) por un valor de ajuste acorde a ciertas tablas de ponderación (ver Tabla 10.1). Aunque existen guías

para clasificar cada elemento dentro de la tabla de complejidad, abordar dichas directrices se queda lamentablemente fuera del alcance de este libro.

Tabla 10.1: Factor de ponderación según complejidad

	Simple	Media	Compleja
Entrada externa	3	4	6
Salida externa	4	5	7
Consultas	3	4	6
Ficheros externos	7	10	15
Ficheros internos	5	7	10

Como ejemplo de puntos de función, imaginemos en un sistema los elementos y la complejidad indicados en la Tabla 10.2.

Tabla 10.2: Ejemplo de puntos de función no ajustados

	Simple	Media	Compleja	Suma	
Entrada externa	2×3	+	4×4	+	1×6 = 28
Salida externa	0×4	+	6×2	+	2×7 = 26
Consultas usuario	0×3	+	8×4	+	0×6 = 32
Ficheros externos	0×7	+	2×10	+	0×15 = 20
Ficheros internos	0×5	+	2×7	+	0×10 = 14
<i>Total PF no ajustados:</i>				120	

Los puntos de función no ajustados son la suma de la multiplicación del número de elementos por su peso:

$$PF_{NoAjustados} = \sum_{i=1}^{15} numElementos_i \cdot peso_i$$

Una vez calculados los puntos de función no ajustados, se procede a aplicar un factor de corrección, llamado *ajuste de complejidad técnica* (*PCA*, *Processing Complexity Adjustment*) que depende de los catorce atributos que se muestran en la Tabla 10.3. Estos factores deben ser evaluados en una escala entre 0 y 5, donde 0 significa que el factor es irrelevante para la aplicación y un valor de 5 significa que es un factor esencial. También existen guías para asignar el valor de cada factor. A continuación, el valor de ajuste se calcula con la ecuación:

$$PCA = 0,65 + (0,01 \cdot \sum_{i=1}^{14} F_i)$$

Nótese que la suma de todos los factores tiene un rango entre 0 y 70, por lo que *PCA* está comprendido entre 0,65 y 1,35, es decir, los puntos de función no ajustados pueden

Tabla 10.3: Factores de complejidad técnica

1	Comunicaciones de datos	8	Actualización on-line
2	Datos o procesamiento distribuido	9	Procesamiento complejo
3	Objetivos de rendimiento	10	Reutilización
4	Configuración usada masivamente	11	Facilidad de operación
5	Tasa de transacción	12	Facilidad de instalación y conversión
6	Entrada de datos on-line	13	Puestos múltiples
7	Eficiencia para el usuario	14	Facilidad de cambio

variando un $\pm 35\%$ dependiendo de la complejidad y cada factor influye un 5%. Una vez calculados los puntos de función ajustados, se estiman las líneas de código (*LoC*) mediante tablas que muestran el número medio de líneas de código por punto de función al estilo de la Tabla 10.4. Finalmente, el esfuerzo se puede calcular con la productividad pasada de la organización.

Tabla 10.4: Lenguajes y número medio de líneas de código por punto de función

Lenguaje de programación	Media LoC/PF
Lenguaje ensamblador	320
C	128
Cobol	105
Fortran	105
Pascal	90
Ada	70
Java, C++	53
Generadores de código	15
Hojas de cálculo	6
Lenguajes gráficos	4

Continuando con el ejemplo anterior, para calcular los puntos de función ajustados y suponiendo que tenemos que la suma del factor de complejidad es 52, entonces calculamos el factor de ajuste (*PCA*):

$$PCA = 0,65 + (0,01 \sum_{i=1}^{14} F_i) = 1,17$$

aplicamos ese factor de ajuste obtenido a los puntos de función no ajustados:

$$PF_{ajustados} = PF_{no\,ajustados} \cdot PCA = 372$$

Después, basándonos en la Tabla 10.4, sabemos que aproximadamente 128 líneas de código en C equivalen a un *punto de función*, por lo que tenemos:

$$LoC = 128 \times PF = 47.616 \text{ líneas de código en C}$$

Puesto que el esfuerzo se puede calcular como el tamaño dividido por la productividad, asumiendo una productividad de 12 PF/personas-mes tendremos:

$$\text{esfuerzo} = \text{PF} / \text{productividad} = 372 / 12 = 31 \text{ personas-mes}$$

y finalmente, si la media es de 3.000 euros por persona-mes (250 euros por PF) el coste será de 90.000 euros, aproximadamente.

10.4.3 Modelos algorítmicos o paramétricos

Los modelos algorítmicos o paramétricos representan relaciones –en forma de ecuaciones– entre variables, generalmente el esfuerzo y el tamaño del software, pero pueden incluir otras características del proyecto como la complejidad o los factores relacionados con la organización del proyecto.

Existen multitud de modelos paramétricos, que se pueden agrupar en abiertos –públicos o de caja blanca, si se conocen las ecuaciones que los componen– o propietarios –cerrados o de caja negra, donde una institución comercializa las herramientas de estimación sin que se conozcan las ecuaciones–. Entre los primeros tenemos las técnicas estadísticas de regresión, SLIM, COCOMO, puntos de función Albrecht o sus variantes posteriores, modelos que se analizan más detalladamente a continuación. Entre los modelos propietarios podemos destacar PRICE-S, SEER-SEM o CHECKPOINT, aunque dada su naturaleza no son objeto de estudio en este libro.

Estimación mediante regresión estadística

Si se dispone de un histórico de proyectos, este método relaciona el coste o esfuerzo con otros parámetros del proyecto como el tamaño, la complejidad, etc., utilizando algún tipo de curva de regresión. La Figura 10.2 muestra dos ejemplos de curvas de regresión, donde el esfuerzo (e) se relaciona con el tamaño (t). Hay muchos tipos diferentes de regresiones matemáticas como por ejemplo la regresión lineal simple, multivariable, exponencial, etc., que ajustan los datos disponibles a distintos tipos de curvas. Aunque es el modelo paramétrico más simple, hay estudios que muestran que pueden ofrecer buenas estimaciones.

COCOMO

El modelo paramétrico probablemente más conocido es el de COCOMO (*CO*nstructive *CO*st *M*odel), dado que fue uno de los primeros modelos abiertos desarrollados. Fue propuesto por B. Boehm en 1981 basándose en 63 proyectos –principalmente de la NASA– y actualizado en el año 1995 bajo la denominación COCOMO II, que veremos más adelante. En el modelo inicial, también conocido como COCOMO 81, se distinguen tres tipos de proyectos según su dificultad y entorno de desarrollo:

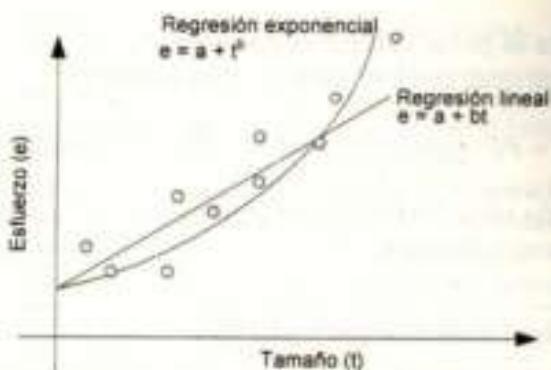


Figura 10.2: Curvas de regresión

- *Orgánico*. El proyecto se desarrolla en un entorno estable, donde los requisitos no cambian o sus cambios son mínimos y el tamaño del proyecto es relativamente pequeño. Por ejemplo, los sistemas de gestión de la información.
- *Empotrado (Embedded)*. El proyecto se desarrolla en un entorno con alta volatilidad de los requisitos, proyectos complejos o innovadores. Por ejemplo, sistemas de tiempo real para los que se desarrolla hardware y software (aviones, telescopios, etc.)
- *Semi-libre (Semidetached)*. Proyectos de desarrollo entre los modelos orgánico y empotrado. Ejemplos de proyecto de este tipo puede ser sistemas para automóviles como aparcamiento automático o *drivers* para hardware.

El modelo COCOMO se compone de 3 submodelos, según la etapa de desarrollo en las que se encuentre el proyecto:

- *Básico*. Las ecuaciones de estimación de esfuerzo y tiempo de desarrollo sólo tienen en cuenta el tamaño del producto para las estimaciones iniciales del proyecto. La medida del tamaño se basa en el número de líneas de código medidas en miles de instrucciones entregadas (KDSI):
 - $Esfuerzo = a \cdot KDSI^b$.
 - $Tiempo = c \cdot esfuerzo^d$.
 - $Personal = esfuerzo/tiempo$.

donde los parámetros a , b y c dependen del tipo de proyecto (orgánico, semilibre o empotrado), como se muestra en la Tabla 10.5. En COCOMO, por defecto, el esfuerzo se mide en horas trabajadas en un mes, considerando que un mes son 152 horas de trabajo. Por tanto, el tiempo se mide en meses. El personal sería el número de empleados a tiempo completo necesarios para llevar a cabo un proyecto.

Tabla 10.5: Parámetros de COCOMO básico e intermedio

	a Básico	a Intermedio	b	c	d
Orgánico	2,4	3,2	1,05	2,5	0,38
Semilibre	3,0	3,0	1,12	2,5	0,35
Empotrado	3,6	2,8	1,2	2,05	0,32

Ejemplo COCOMO Básico

Asumamos que tenemos que desarrollar un sistema de facturación del cual tenemos experiencia (por tanto, lo clasificamos como orgánico). Además, estimamos que el número de líneas de código es de 43.200. Aplicando las ecuaciones:

- $\text{esfuerzo} = a \cdot \text{tamifio}^b = 2,4 \cdot 43,2^{1,05} = 125,16 \text{ personas/mes}$
- $\text{tiempo} = c \cdot \text{esfuerzo}^d = 2,5 \cdot 94,93^{0,38} = 15,67 \text{ meses}$
- $\text{personal} = \text{esfuerzo} \div \text{tiempo} = \sim 8 \text{ personas (a tiempo completo)}$

- **Intermedio.** A la ecuación del modelo básico se incorpora un valor de ajuste compuesto por 15 atributos agrupados en 4 categorías: (i) características del producto, (ii) restricciones relacionadas con el hardware, (iii) niveles de experiencia del personal y (iv) características del proyecto en sí mismo, es decir:

$$\text{esfuerzo}_{int} = \text{esfuerzo}_{nom} \cdot FAE$$

donde FAE (*factor de ajuste del esfuerzo*) es el producto de los 15 factores con los parámetros de la Tabla 10.6, que sirve para estimar el esfuerzo intermedio en función del nominal (N) en una escala que va de muy baja (MB) a extra-alta (EA). En la tabla se indica cómo, por ejemplo, el que la capacidad de los programadores (PCAP) sea muy baja añade un 42% al esfuerzo de desarrollo, mientras que, si por el contrario es muy alta, puede reducir el esfuerzo hasta en un 30% con respecto al nominal.

Ejemplo COCOMO – Continuación

Si se asume un factor de ajuste cuyos atributos toman todos valores altos (A) según la Tabla 10.6 excepto DATA (tamaño de la base de datos) y MODP (prácticas modernas de programación), que toman un valor muy alto (MA) –por una parte–, TURN (tiempo de ejecución completa), VIRT (volatilidad entorno máquina virtual) y TOOL (utilización de herramientas software) que toman un valor bajo (B) –por otra–, y SCED (tiempo de desarrollo requerido) y STOR (restricciones de memoria) que toman un valor nominal (N), entonces $FAE = 0,608$, y en consecuencia:

- $\text{esfuerzo}_{int} = \text{esfuerzo}_{nom} \cdot FAE = (3,2 \cdot 43,2^{1,05}) \cdot 0,608 = 76,09 \text{ personas/mes}$
- $\text{tiempo} = 2,5 \cdot 76,09^{0,38} = 12,97 \text{ meses}$
- $\text{personal} \approx 6 \text{ personas}$

Tabla 10.6: Factores de COCOMO intermedio

Factor	MB	B	N	A	MA	EA
Atributos del producto						
RELY Fiabilidad requerida del software	0,75	0,88	1	1,15	1,4	
DATA Tamaño de la base de datos		0,94	1	1,08	1,16	
CPLX Complejidad del producto	0,7	0,85	1	1,15	1,3	1,65
Atributos hardware						
TIME Restricciones en tiempo de ejecución			1	1,11	1,3	1,66
STOR Restricciones de memoria			1	1,06	1,21	1,56
VIRT Volatilidad entorno máquina virtual		0,87	1	1,15	1,3	
TURN Tiempo de ejecución completa		0,87	1	1,07	1,15	
Atributos del personal						
ACAP Capacidad de los analistas	1,46	1,19	1	0,86	0,71	
AEXP Experiencia en el área de la aplicación	1,29	1,13	1	0,91	0,82	
PCAP Capacidad de los programadores	1,42	1,17	1	0,86	0,7	
VEXP Experiencia en la máquina virtual	1,21	1,1	1	0,9		
LEXP Experiencia con el lenguaje de programación	1,14	1,07	1	0,95		
Atributos del proyecto attributes						
TOOL Utilización de herramientas software	1,24	1,1	1	0,91	0,82	
MODP Prácticas modernas de programación	1,24	1,1	1	0,91	0,83	
SCED Tiempo de desarrollo requerido	1,23	1,08	1	1,04	1,1	

- **Detallado.** Similar al intermedio, pero las estimaciones se realizan en cuatro fases del ciclo de vida: (i) *diseño de producto*, (ii) *diseño detallado*, (iii) *codificación y pruebas unitarias* e (iv) *integración y pruebas de integración*.

Finalmente, como el número de personas necesario en cada fase del ciclo de vida puede variar, se pueden utilizar perfiles predefinidos para la distribución de personal.

COCOMO II

En COCOMO II se reconocen varias cuestiones problemáticas que se estudiaron sobre el COCOMO original. Primero los procesos han evolucionado desde el modelo en cascada a otros modelos: iterativos e incrementales, desarrollo basado en componentes, etc. Segundo, la cantidad de información disponible en las distintas etapas de desarrollo. Por ejemplo, el personal, las características específicas del proyecto y el problema de que la estimación de las líneas de código en las etapas tempranas del desarrollo sea extremadamente difícil de calcular (tanto como el esfuerzo). Por tanto, según la información disponible en cada una de las etapas de desarrollo, las estimaciones se tornarán más acertadas (Figura 10.3).

El nuevo modelo consta de tres submodelos diferentes que tienen en cuenta la información disponible en tres fases distintas del desarrollo:

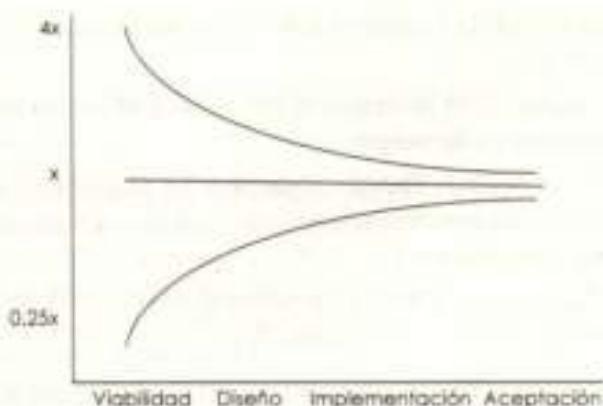


Figura 10.3: Variabilidad en las estimaciones según la fase de desarrollo (Boehm)

- **Fase 1. Composición de la aplicación.** Submodelo que se usa para estimar el esfuerzo y el tiempo de desarrollo en proyectos en los que se utilizan prototipos para disminuir el riesgo en aspectos relacionados con la interfaz gráfica del usuario, o en los que se emplean herramientas CASE para el desarrollo rápido de aplicaciones.

Este modelo utiliza como variable de medida del tamaño del producto los puntos objeto que se basan en la cuenta de artefactos tales como el número de pantallas, informes y módulos. Esta cuenta se pondera mediante un factor de complejidad compuesto de tres niveles: simple, medio y complejo.

- **Fase 2. Diseño preliminar.** Se tiene en cuenta la exploración de diferentes arquitecturas del sistema y conceptos de operación. La estimación de las líneas de código se realiza utilizando puntos de función (que se verán a continuación). Utilizando una ecuación similar a COCOMO 81:

$$PM_{nominal} = A \cdot tamaño^B$$

pero en vez de considerar parámetros constantes para el exponente B como se hacía en COCOMO 81 con orgánico, semi-libre y empotrado, el exponente B se calcula mediante la ecuación (también utilizada en post-arquitectura):

$$B = 0,91 + 0,01 \sum W_i$$

donde cada W_i es la asignación de 0 a 5 (clasificada como *muy bajo, bajo, nominal, alto, muy alto y extra-alto*) a los siguientes factores de escala:

- Precedentes (PREC). Considera la experiencia que tiene la organización en el desarrollo de aplicaciones similares.

- Flexibilidad (FLEX). Considera la rigidez de los requisitos y de las restricciones en el desarrollo.
- Arquitectura/solución de riesgos (RESL). Tiene en cuenta las medidas tomadas para la eliminación de riesgos.
- Cohesión del equipo (TEAM). Considera las dificultades de cohesión y sincronización de las personas involucradas en el proyecto (desarrolladores, equipo de pruebas, usuarios, etc.)
- Madurez de procesos (PMAT). Considera el nivel de madurez de la organización basándose en el CMM (ver Capítulo 9).

De modo similar a COCOMO intermedio, se utilizan 7 factores de ajuste (EM):

$$PM_{ajustados} = PM_{nominal} \left(\prod_{i=1}^7 EM_i \right)$$

a los que les asigna un valor dependiendo de su clasificación en seis niveles (de *mayo bajo* a *extra-alto*). Estos factores de ajuste son la capacidad del personal (PERS), la fiabilidad y complejidad del producto (RELY), la reutilización requerida (RUSE), la dificultad de la plataforma (PDIF), la experiencia del personal (PREX), las facilidades (FCIL) y el calendario (SCED).

- **Fase 3. Post-arquitectura.** Una vez se ha completado el diseño y definido la arquitectura, comienza el desarrollo. En este punto, la estimación puede hacerse, bien mediante puntos de función o bien mediante líneas de código. El proceso es similar al anterior, pero utilizando 17 factores agrupados en cuatro categorías –como en COCOMO 81– tal y como se muestra en la Tabla 10.7.

Putnam – SLIM

Creado inicialmente por Putnam en los años 1970 como modelo abierto, evolucionó posteriormente en un producto comercial bajo el nombre de SLIM (*Software Life Cycle Model*) bajo los auspicios de la compañía SQM. Este modelo asume como principio subyacente que la distribución del personal en los proyectos se ajusta a la distribución de Norden-Rayleigh (Figura 10.4), y a partir de esa distribución se pueden construir las estimaciones de esfuerzo y tiempo.

Putnam definió una *ecuación del software*, basándose en la hipótesis que la cantidad de trabajo de un producto software es el producto del esfuerzo empleado y el tiempo invertido, lo que matemáticamente se representa como:

$$S = C \cdot K \cdot t_d$$

donde S es el tamaño del producto medido en líneas de código, K el esfuerzo medido en personas-mes o personas-año, t_d el tiempo de desarrollo medido en meses o años, y C es

Tabla 10.7: Factores de COCOMO II**Producto**

- Fiabilidad requerida del software (RELY)
- Tamaño de la base de datos (DATA)
- Complejidad del producto (CPLX)
- Reutilización requerida (RUSE)
- Documentación desarrollada (DOCU)

Plataforma de desarrollo

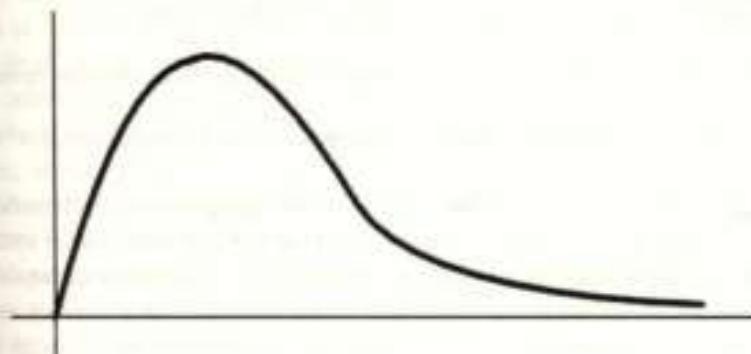
- Restricciones en el tiempo de ejecución (TIME)
- Restricciones en el almacén principal (STOR)
- Volatilidad de la plataforma (PVOL)

Personal

- Capacidad de los analistas (ACAP)
- Capacidad de los programadores (PCAP)
- Experiencia en el desarrollo de aplicaciones similares (AEXP)
- Experiencia con la plataforma de desarrollo (PEXP)
- Experiencia con el lenguaje y herramientas (LEXP)
- Estabilidad del personal (PCON)

Proyecto

- Utilización de herramientas software (TOOL)
- Desarrollo en múltiples localizaciones (SITE)
- Tiempo necesario para el desarrollo (SCED)

**Figura 10.4:** Distribución de Norden-Rayleigh

una constante de productividad definida por un conjunto de prácticas de la organización tales como la gestión del proyecto o la complejidad del producto. Utilizando su base de datos, Putnam estableció la siguiente ecuación:

$$S = C \cdot K^{1/3} \cdot t_d^{4/3}$$

Por otra parte, para Putnam, la curva de utilización del personal en un proyecto, expresada mediante las curvas de Norden-Rayleigh, permite definir una segunda ecuación para el esfuerzo y el tiempo. Basándose en proyectos de su organización, Putnam estableció el parámetro *acumulación de personal (MBI - Manpower Buildup Index)* definido como $MBI = K/r_d^3$, sugiriendo que la contratación de personal tiene implicaciones en el tiempo de desarrollo (t_d) y en el esfuerzo y reflejando que variaciones en los plazos afectan enormemente al esfuerzo.

El *MBI* se puede definir y establecer previamente al desarrollo del proyecto, con los datos que la organización haya recogido. Utilizando las dos ecuaciones se pueden encontrar las diferentes combinaciones de tiempo y esfuerzo para definir la planificación.

Hay estudios que demuestran que SLIM no es un método de estimación adecuado en pequeños proyectos, pero sí cuando se trata de proyectos con tiempos de desarrollo superiores a 6 meses, de tamaño superior a 5.000 líneas de código y esfuerzo por encima de 1.5 personas/año.

10.4.4 Modelos basados en la inteligencia artificial

Recientemente se han aplicado distintas técnicas de la minería de datos a distintos problemas de la Ingeniería del Software en general, y a la estimación en particular. Entre este tipo de técnicas nos encontramos las siguientes:

- **Sistemas basados en reglas:** Un sistema basado en reglas consiste en un conjunto de reglas con la forma *si [...] entonces [...]*. Por ejemplo:

```
si (30 <= CasosUso <= 35) entonces esfuerzo=50 personas/mes
```

```
si (150 <= LoC de la clase) entonces esfuerzoPruebas=alto
```

Las reglas, generadas del histórico de proyectos mediante algoritmos de minería de datos, son una forma sencilla e intuitiva de obtener información y tomar acciones acordes a la información disponible en un momento concreto, pero también tienen el inconveniente de que al incrementar el número de reglas, su manejo y su significado de forma global puede ser difícil. Para entornos no determinísticos, al igual que en la Ingeniería del Software si existe incertidumbre, a las reglas se les ha incorporado lógica difusa.

- **Razonamiento basado en casos:** este método consiste en replicar el proceso que un experto llevaría a cabo para tomar una decisión basándose en su experiencia (ver

método Delphi en la Sección 10.4.1). Teniendo datos históricos de proyectos, una nueva estimación se obtiene encontrando un número proyectos similares conforme a los datos conocidos del proyecto a estimar (basándose en alguna medida de distancia como la distancia euclídea). Después, con ese reducido número de proyectos, se estiman los parámetros desconocidos haciendo, por ejemplo, la media, regresiones u operaciones similares. La Figura 10.5 muestra este proceso de estimación.

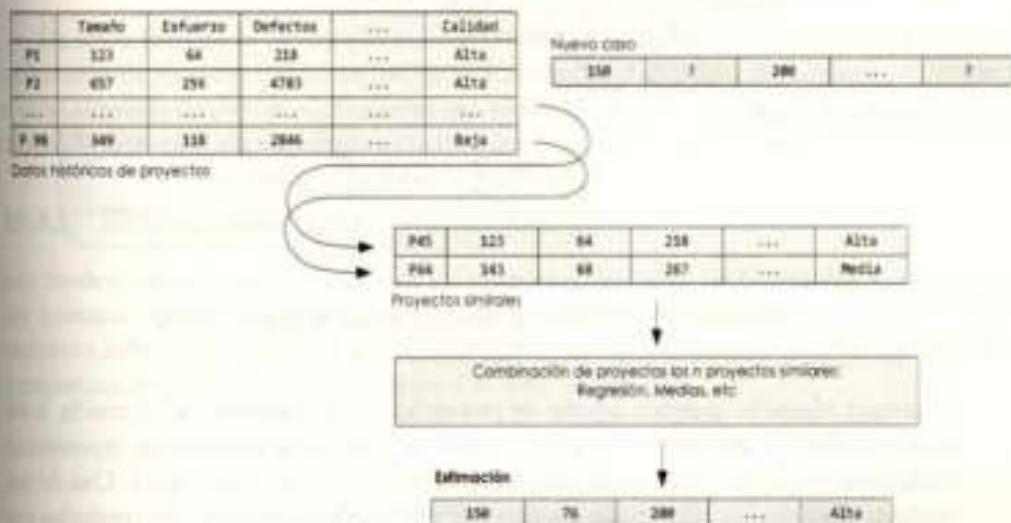


Figura 10.5: Proceso de estimación basado en casos

- **Redes neuronales:** generalmente usadas para el reconocimiento de patrones y clasificación, las redes neuronales también han sido utilizadas en la Ingeniería del Software. Como el ejemplo mostrado en la Figura 10.6, una red neuronal consiste en un conjunto de nodos, generalmente organizados en capas (una capa de entrada, una o dos capas intermedias y una capa de salida) y unas conexiones con sus pesos de las que depende la «información» que se propaga. Existen multitud de tipos de redes neuronales, siendo el más conocido el *perceptrón*, mostrado en la Figura 10.6, en el que la información se va propagando hacia adelante, de la capa de entrada compuesta por atributos tales como el número de casos de uso, experiencia del personal, etc., hacia la capa de salida con los parámetros que se quieren estimar, tales como esfuerzo, tiempo de desarrollo, etc. El mayor inconveniente para la Ingeniería del Software de las redes neuronales, es que una vez entrenadas y con los pesos ajustados con los datos de entrenamiento proporcionados, éstas actúan como cajas negras en las que no se sabe el proceso por el que se ha llegado a dicha estimación.
- **Árboles de decisión:** en general, los árboles de decisión son utilizados para predecir o explicar observaciones. Si las hojas de los árboles de decisión son valores discretos (o

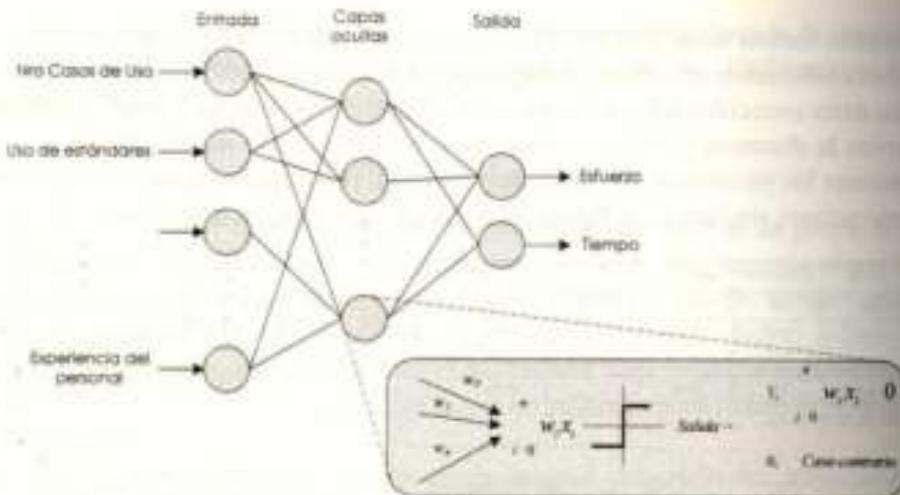


Figura 10.6: Proceso de estimación por analogía

etiquetas), entonces tenemos árboles de clasificación o ecuaciones de regresión, a los cuales también se les llama *árboles de regresión*. Los nodos intermedios representan decisiones, siendo los nodos más altos en la jerarquía los más importantes. Una de sus ventajas es su directa conversión a reglas que pueden ser fácilmente interpretadas por los gestores de proyectos. La Figura 10.7 muestra dos árboles, uno donde las hojas son valores concretos (*árbol de clasificación*) y otro donde las hojas son ecuaciones de regresión para la estimación del esfuerzo. Así, en el árbol de la izquierda el principal atributo es el tamaño, del que depende la estimación del esfuerzo, pues sólo se comprueba el segundo atributo (tipo de base de datos) si el esfuerzo es menor que la cantidad indicada, en cuyo caso, si éste es relacional, la estimación del esfuerzo sería de 16 personas mes. En el árbol de la derecha, se aplicaría una ecuación u otra, dependiendo también del tamaño.

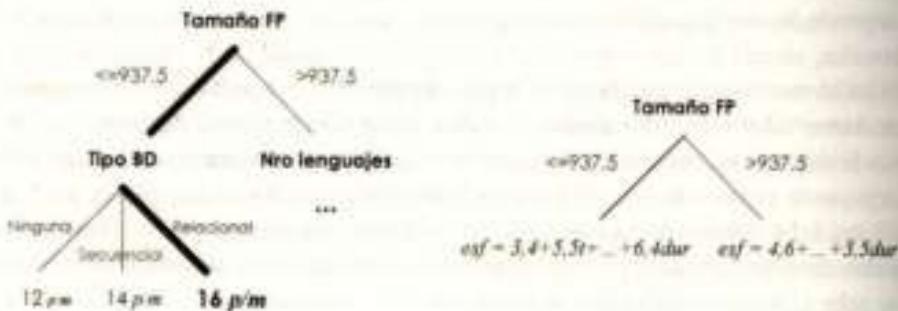


Figura 10.7: Ejemplo de árbol de clasificación y de regresión

- **Otras técnicas de inteligencia artificial:** Además de las técnicas mencionadas, otras técnicas basadas en la inteligencia artificial y minería de datos que han sido aplicadas a la Ingeniería del Software incluyen –entre otros– los algoritmos genéticos (que se han aplicado a la determinación de las ecuaciones de predicción del esfuerzo) y las redes bayesianas (como técnica probabilística). Además de ello, la combinación de técnicas se ha utilizado como forma de mejorar las estimaciones.

De todas formas, la estimación es un problema abierto. Hasta la fecha, parece ser que la utilización de técnicas complejas no mejora sustancialmente las estimaciones con respecto a otras técnicas más simples. Si parece que el uso de expertos humanos, en conjunción con alguna otra técnica, da mejores resultados que el uso individual de una cierta técnica.

10.4.5 Sistemas dinámicos

Los modelos dinámicos iniciados por J. Forrester en los años 1960 permiten estudiar cómo los sistemas complejos evolucionan en el tiempo. Mediante bucles causales donde ciertas variables influyen positiva o negativamente en otras (ver Figura 10.8), permiten simular el comportamiento de sistemas complejos a lo largo del tiempo.

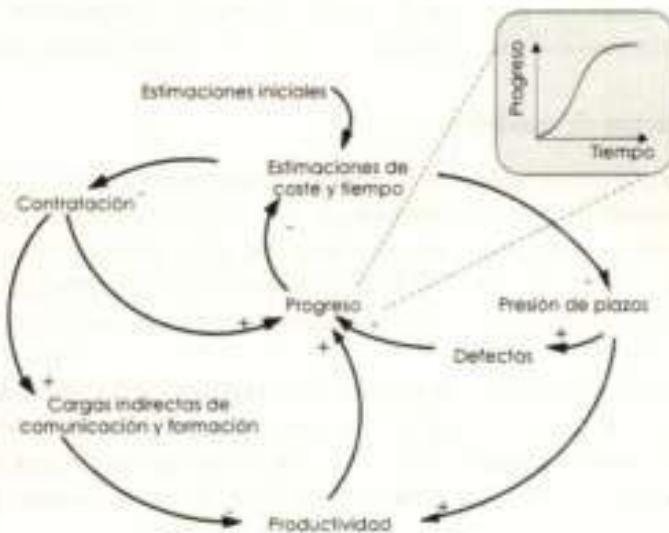


Figura 10.8: Ejemplo de sistema dinámico

En relación a la gestión en la Ingeniería del Software, los sistemas dinámicos permiten la simulación del comportamiento de proyectos software, de tal manera que los gestores de proyectos pueden simular la ejecución de un proyecto con diferentes parámetros, analizando así las políticas de gestión que mejor se adaptan a los objetivos del proyecto. Además, las simulaciones pueden realizarse en distintas etapas en el desarrollo de un proyecto:

- Análisis *a priori* del proyecto para la estimación y prever comportamientos. Por ejemplo, preguntas del estilo: ¿qué ocurriría si se añaden 3 personas a mitad de proyecto?
- Control del proyecto durante su ejecución para adaptar las estimaciones a la evolución del proyecto. Por ejemplo, preguntas como ¿qué está ocurriendo con la calidad con los actuales parámetros?
- Análisis *post-mortem* del proyecto (una vez finalizado) con la idea de mejorar de cara al desarrollo de futuros proyectos y conocer cómo podrían haberse obtenido resultados más favorables. Responde a preguntas tales como ¿qué habría ocurrido de haber incorporado dos personas más al equipo de pruebas?

Al igual que otras técnicas de estimación, los modelos de sistemas dinámicos necesitan validarse y adaptarse a las organizaciones donde se utilizan, ya que la evolución de los mismos depende mucho de las estimaciones iniciales, de las políticas de gestión, de las características del proyecto y de las de la propia organización.

Entre los modelos de sistemas dinámicos más conocidos –pues fue uno de los primeros– se encuentra el modelo de Abdel-Hamid, que ha sido ampliamente validado, utilizado y extendido posteriormente. Este modelo se compone de cuatro submodelos diferentes con las actividades relacionadas con la gestión de recursos humanos, la producción, la planificación y el control de proyectos software, cada una de ellas con un gran número de parámetros.

10.4.6 Evaluación de modelos

No se puede decir que existan métodos de estimación superiores a otros, pero lo cierto es que para una misma entrada, los distintos métodos de estimación –regresión, COCOMO, puntos de función, redes neuronales, programación genética, etc.– generan diferentes resultados. Además, a la hora de utilizarlos puede variar la información disponible, tal como la base de datos de históricos de proyectos, el calibrado realizado, la experiencia del personal con los métodos, etc.

Las técnicas más usadas en la Ingeniería del Software para determinar bondad de la estimación y de la capacidad de predicción de una técnica son el *nivel de predicción* y la *magnitud media del error relativo*. Además, es típico utilizar las técnicas estadísticas del coeficiente de correlación (R) y determinación (R^2) con las curvas de regresión.

- *Magnitud media del error relativo*, $MMER$, se define como:

$$MMER = \frac{1}{n} \sum_{i=1}^n \left| \frac{e_i - \hat{e}_i}{e} \right|$$

donde e es el valor real de la estimación, \hat{e} es el valor estimado y n es el número de proyectos. Las predicciones del modelo evaluado son mejores cuanto menor es $MMER$. Un criterio habitual en la Ingeniería del Software es que $MMER$ sea menor

que 0,25, para que el modelo se considere como bueno. Como ejemplo, la Figura 10.9 muestra gráficamente las distancias entre los valores reales y los estimados (la curva de regresión), distancias utilizadas para el cálculo de *MMER*. Supongamos que, una vez transcurridos 9 proyectos en una organización, obtenemos los resultados de la Tabla 10.8, cuyo valor de *MMER* tal y como muestra la ecuación anterior sería:

$$MMER = \frac{1}{n} \sum_{i=1}^n \left| \frac{e_i - \hat{e}_i}{e} \right| = \frac{1}{9} \left(\frac{|145 - 120|}{120} + \dots + \frac{|320 - 260|}{320} \right) = 0,20$$

lo que implica que el método de estimación satisface el criterio de $MMER \leq 0,25$.

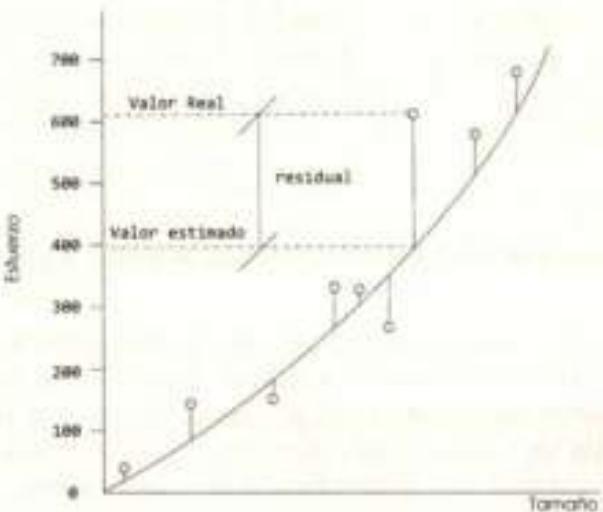


Figura 10.9: La MMER indica la distancia entre los valores reales y estimados

- *Nivel de predicción l*, $Pred(l)$: siendo l un porcentaje, se define como el cociente entre el número de estimaciones que están dentro del porcentaje l en valor absoluto entre el número total de estimaciones. Así, $PRED(0,2) = 0,8$ quiere decir que el 80% de las estimaciones están dentro del 20% de las estimaciones reales. En la Ingeniería del Software, el criterio habitual para aceptar un modelo como bueno suele ser que $Pred(0,25) \geq 0,75$, es decir, que el 75% de las estimaciones se desvien menos del 25% del valor real. La Figura 10.10 representa gráficamente el *nivel de predicción*, donde los trazos verticales junto a los puntos de estimación muestran la variaciones de la estimación al $\pm 25\%$; el valor $Pred(25)$ se calcula sumando el número de veces que la curva corta con los trazos verticales, y después dividiendo esa suma entre el número total de estimaciones. Si continuamos con el ejemplo de la Tabla 10.8, tenemos que 7 de los 9 proyectos están dentro del 25%. Entonces:

$$Pred(25) = \frac{7}{9} = 0,77$$

Por tanto, como $MMER < 0,25$ y $Pred(25) > 0,75$, se podría considerar que el método de estimación es aceptable.

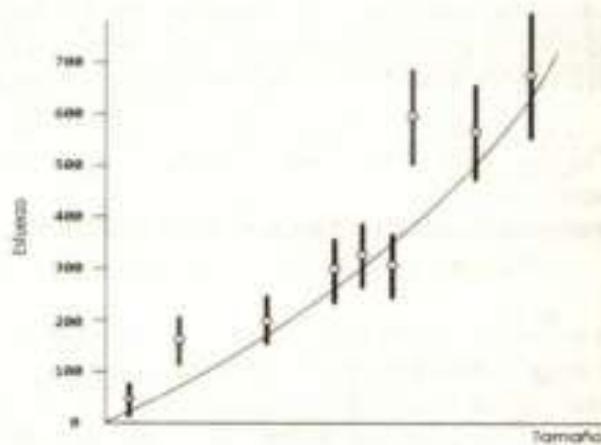


Figura 10.10: Ejemplo de predicción de nivel. 1, $Pred(l)$

- *Coeficiente de determinación múltiple*, R^2 : denota cómo se ajustan los datos a la recta de regresión. Si todos los puntos coinciden con dicha recta, hay una relación perfecta entre las variables dependiente e independiente y en ese caso, el valor de R^2 es 1. Según se alejan los puntos, el valor disminuye; si no hay relación alguna entre las variables, el valor de R^2 es 0. Matemáticamente se define como:

$$R^2 = 1 - \frac{\sum_{i=1}^n (e_i - \hat{e}_i)^2}{\sum_{i=1}^n (e_i - \bar{e})^2}$$

donde dadas n estimaciones, e_i es el valor real, \hat{e}_i el estimado y \bar{e} la media.

Tabla 10.8: Ejemplo de valoración de proyectos

	Estimado	Real	$ Diferencia $	$\pm 25\%$	Estimado	% Error	Error < 25%
A	120	145	25		90-150	0,20	Cierto
B	235	320	85	187,5-312,5		0,26	Falso
C	60	50	10		45-75	0,16	Cierto
D	100	120	20		75-125	0,20	Cierto
E	400	610	210	300-500		0,52	Falso
F	500	580	80		375-625	0,16	Cierto
G	280	310	30		210-350	0,10	Cierto
H	670	690	20	502,5-837,5		0,02	Cierto
I	320	260	60	240-400		0,18	Cierto

10.4.7 Calibración de modelos

Una característica, tanto de los modelos de estimación paramétricos como de los basados en la minería de datos, es que necesitan ser calibrados y ajustados a los entornos en los que se utilizan. Por ejemplo, COCOMO 81 se definió utilizando 63 proyectos, pero es sabido que utilizando los parámetros por defecto, las estimaciones son bastante pobres debido a las diferencias en los entornos, en las tecnologías empleadas, etc. El caso es que, según una organización va recabando datos de los proyectos que realiza, los modelos utilizados en la estimación deben calibrarse con la nueva información, siguiendo los pasos siguientes:

1. Verificar que los nuevos datos, y las métricas, son consistentes con la definición utilizada en el modelo a calibrar. También puede ser necesario quitar datos que han dejado de ser útiles para la organización, por ejemplo, datos de proyectos en los que se utilizaron tecnologías obsoletas y que la organización no volverá a realizar.
2. Calibrar el modelo según los nuevos datos, por ejemplo, afinando los parámetros en el caso de las curvas de regresión, o ejecutando los algoritmos de aprendizaje con los últimos datos. Los nuevos modelos se pueden validar con las medidas mencionadas en la sección anterior.
3. Finalmente, se reemplaza el viejo modelo con el nuevo.

Aunque no se estudia en este libro, también es importante en la construcción de modelos de estimación el utilizar parte de la muestra para la definición del modelo y otra para su evaluación. Así, en minería de datos es una práctica común dividir los datos entre un conjunto de entrenamiento para generar el modelo y otro conjunto de datos de pruebas para validar su bondad y evitar que no generalice correctamente (*overfitting*).

Por otra parte, cuando se utiliza el mismo conjunto de datos para la construcción del modelo de estimación y para su evaluación, existen otras técnicas más complejas como la validación cruzada, que proporcionan una idea más acertada de la bondad de un método, si bien dichas técnicas quedan fuera del alcance de este libro.

10.5 Planificación y seguimiento del proyecto

La **planificación** de un proyecto software, al igual que cualquier otro tipo de proyecto, consiste en identificar las tareas del proyecto, las relaciones entre las tareas (en qué orden deben de ser ejecutadas), asignar los recursos a las tareas y estimar los plazos de ejecución de las tareas.

En la planificación de tareas, el proyecto se divide generalmente en subtareas de forma jerárquica, mediante lo que se conoce como *estructura de descomposición del trabajo*. Una vez definidas las tareas, deben determinarse cuáles son las relaciones entre ellas y el orden en que se deben llevar a cabo. Para ello, frecuentemente se hace uso de técnicas como el

método del camino crítico, PERT y Gantt. A las actividades se les deben asignar los recursos necesarios de personal, espacio, etc., de la manera más eficiente posible, procurando que varias tareas no necesiten los mismos recursos al mismo tiempo para que éstos estén disponibles. Una vez planificado el proyecto y según se vaya ejecutando el mismo, es necesario hacer un **seguimiento** para ir detectando las divergencias, actualizando la planificación (añadiendo tareas olvidadas, actualizando costes y realizando nuevas estimaciones, reasignando recursos, etc.), motivando al personal y solventando todo tipo de inconvenientes que puedan ocurrir durante la realización del proyecto. En las siguientes secciones estudiaremos en más detalle algunas de las técnicas mencionadas más arriba.

10.5.1 Estructura de descomposición del trabajo

La estructura de descomposición del trabajo (EDT) –conocida también por sus siglas en inglés WBS (Work Breakdown Structure)–, es una técnica para descomponer el proyecto en sus distintas tareas o paquetes de trabajo (*work packages*) de forma jerárquica. La organización de las tareas puede hacerse bien mediante la descomposición de los componentes o bien mediante las fases de trabajo. La Figura 10.11 y la Tabla 10.9 muestran un ejemplo de EDT, donde se puede apreciar que la jerarquía permite agrupar las tareas relacionadas.



Figura 10.11: Ejemplo de EDT

Aunque en proyecto de cierta enjundia puede ser difícil, es importante asegurarse de que no se olvida ninguna tarea, ya que las tareas aquí definidas forman parte de la planificación posterior, en la que se representarán las tareas gráficamente considerando el orden entre ellas –relaciones entre las tareas predecesoras y sucesoras– como criterio muy importante.

Tabla 10.9: Actividades en la estructura EDT

<i>Id</i>	<i>Actividad</i>	<i>Responsable</i>
0	Sistema Inteligente CCTV	Roberto
1	Estudio viabilidad	Consuelo
2	Hardware	Luis
2.1	Selección HW	Luis
2.2	Instalación HW	Luis
3	Software	Roberto
3.1	Análisis requisitos	Belén
3.2	Diseño	David
3.3	Codificación	Belén, David
3.4	Pruebas unitarias	Belén, David
4	Integración y pruebas del sistema	Belén, David
5	Documentación y manuales	Consuelo

10.5.2 Los métodos gráficos CPM y PERT

El método del camino crítico (*Critical Path Method*, CPM) y la técnica de evaluación y revisión de proyectos (*Program Evaluation and Review Technique*, PERT) son técnicas de representación de redes de tareas utilizadas para la gestión de proyectos. Proponen descomponer los proyectos en actividades (generalmente representadas mediante EDT), cada una de las cuales tiene una duración y un cierto orden de ejecución. Se trata de técnicas prácticamente equivalentes, desarrolladas por grupos diferentes, pero casi simultáneamente.

PERT fue desarrollado en 1957 por la empresa *Booz Allen Hamilton* para el Departamento de Defensa de los Estados Unidos, mientras que CPM fue desarrollado por la empresa química *du Pont* en 1958. Se diferencian en que CPM considera un valor único (determinista) de estimación de la duración de una tarea, mientras que PERT utiliza una estimación estadística en tres puntos donde para cada estimación se hace una ponderación entre sus valores pesimista, realista y optimista.

Ambos métodos se basan en representar el proyecto mediante un grafo en el cual se representan y ordenan las actividades, estableciendo sus dependencias. Se determina así el *camino crítico*, que denota el camino del grafo cuyas actividades no se pueden retrasar si el proyecto ha de finalizar en el plazo establecido, ya que si alguna de las actividades en este camino se retrasa, también lo hará la finalización del proyecto. Para la representación existen dos alternativas: el *diagrama de dependencias* y el *diagrama de flechas*, dependiendo si son los arcos o los nodos los que representan a las actividades. En ambas representaciones, equivalentes por otra parte, se utiliza el siguiente proceso:

1. Identificar las actividades y dependencias.
2. Realizar el «recorrido hacia adelante» para calcular las fechas más tempranas en las que una actividad puede ser iniciada y completada.

3. Realizar el «recorrido hacia atrás» para calcular las fechas más tardías en las que una actividad puede ser iniciada y completada.
4. Calcular las holguras disponibles de las actividades, que representan el tiempo que una actividad puede retrasarse sin que se retrase por ello la fecha de finalización del proyecto.
5. Identificar el camino crítico, es decir, las actividades sin holgura. Se tendrá en cuenta que si alguna de las actividades del camino crítico se retrase, también se retrasará la fecha de finalización del proyecto.

Imaginemos la estructura EDT de la Tabla 10.10 en la que ya se ha definido qué tareas son predecesoras de otras. A partir de dicha estructura de paquetes se muestra el proceso mediante un ejemplo, utilizando la notación de diagrama de flechas. El primer paso sería construir el grafo con las tareas como arcos, y los hitos (comienzo y fin de tareas), que son los nodos. En su construcción debemos tener en cuenta que sólo puede haber un nodo inicial y un nodo final, y que no puede haber bucles. La Figura 10.12 muestra, en un diagrama de flechas, las tareas como nodos numerados secuencialmente y las actividades como arcos que tienen asociada su duración.

Tabla 10.10: Ejemplo de CPM: actividades con su duración y precedencias

<i>Id</i>	<i>Actividad</i>	<i>Duración</i>	<i>Precedencias</i>
0	Sisterna inteligente de CCTV		
1 (A)	Estudio de viabilidad	5	—
2	Hardware		
2.1 (B)	Selección HW	9	A
2.2 (C)	Instalación HW	11	B
3	Software		
3.1 (D)	Análisis de requisitos	8	A
3.2 (E)	Diseño	10	D
3.3 (F)	Codificación	18	E
3.4 (G)	Pruebas unitarias	7	F
4 (H)	Integración y pruebas del sistema	15	C, G
5 (I)	Documentación y manuales	25	—

Una vez representadas las tareas, se procede al cálculo de las siguientes fechas en dos pasadas (hacia adelante y hacia atrás):

- Inicio pronto (IP): fecha más temprana en la que una actividad puede comenzar, teniendo en cuenta las dependencias entre actividades.
- Terminación más pronta (TP): representa la fecha más temprana en la que puede terminar la actividad.

- Inicio más tardío (IT): fecha más lejana en que una actividad puede comenzar sin que por ello se retrase el proyecto.
- Terminación más tardía (TT): fecha más lejana en que una actividad puede terminar sin que ello implique retrasar la fecha de conclusión del proyecto.

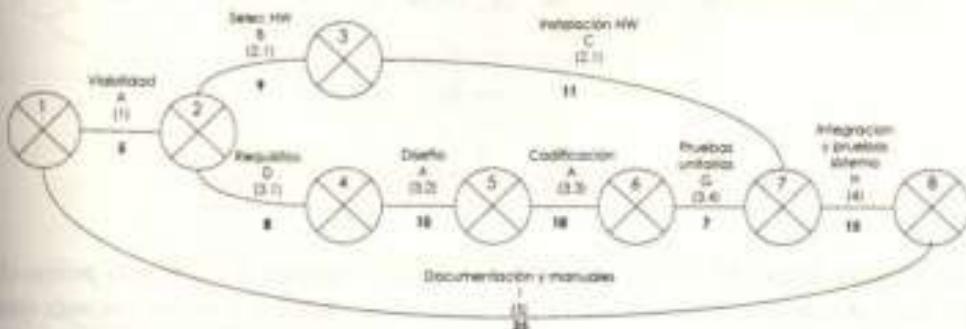


Figura 10.12: Ejemplo de CPM: diagrama de flechas inicial

En el **recorrido hacia adelante** se va calculando de izquierda a derecha la fecha de *inicio más temprano* en la que cada una de las actividades puede comenzar. Para ello, se inserta 0 como fecha inicial y se van sumando los tiempos de las actividades. Cuando hay varias actividades precedentes que convergen en un nodo se escoge el máximo de los tiempos. La Figura 10.13 muestra las fechas de inicio temprano del ejemplo anterior.

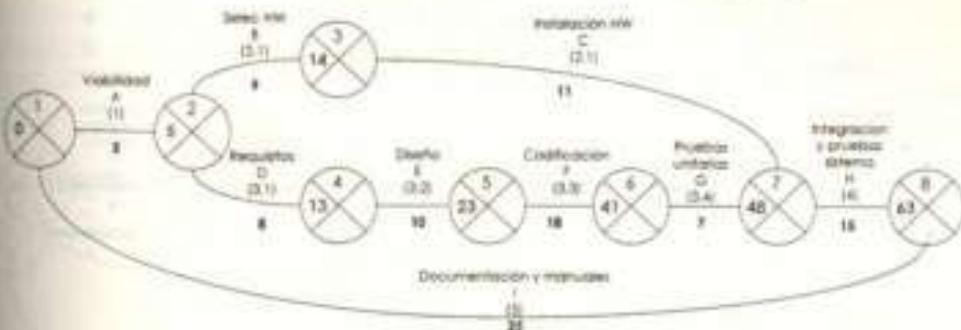


Figura 10.13: Ejemplo de CPM: recorrido hacia adelante

En el **recorrido hacia atrás** se va calculando de derecha a izquierda la fecha de *terminación más tardía* de las actividades. Para ello, primero se copia la última fecha de *inicio más temprano* como *fin más tardío* en el último nodo (derecha) y se va restando la duración de las actividades. Cuando hay varias actividades sucesoras en un nodo, se escoge la de duración mínima. La Figura 10.14 muestra los resultados de la pasada hacia atrás.

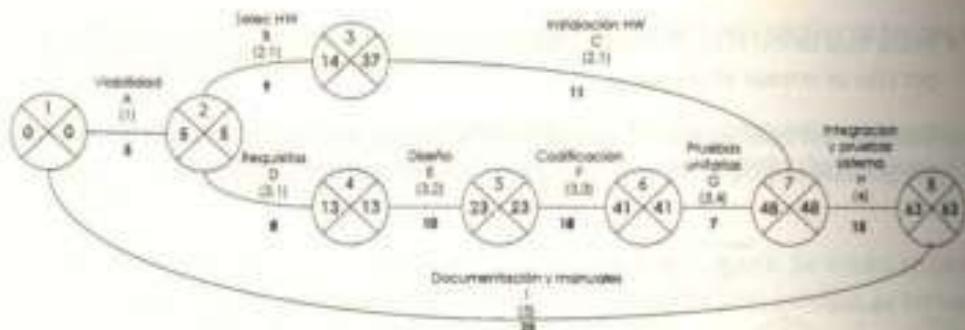


Figura 10.14: Ejemplo de CPM: Recorrido hacia atrás

La **holgura** disponible de las actividades se calcula restando el *inicio más pronto* (IP) y el *inicio más tardío* (IT), o la *terminación más pronta* (TP) y la *terminación más tardía* (TT). Aquellas actividades que no tengan holgura –aquellas en las que las dos fechas sean iguales–, forman el *camino crítico*, es decir, aquellas actividades que no pueden retrasarse para que el proyecto no se demore. Se trata de las actividades que tenemos que vigilar con más detalle. La Figura 10.15 muestra el cálculo de la holgura para el ejemplo, marcando el camino crítico con trazo más grueso. Podremos *jugar* con las actividades que no están en este camino, comenzándolas más tarde, descomponiéndolas rompiéndolas en dos partes, asignando menos recursos, etc., siempre y cuando no excedan el límite de la holgura. La Tabla 10.11 muestra el resumen de todas las variables.

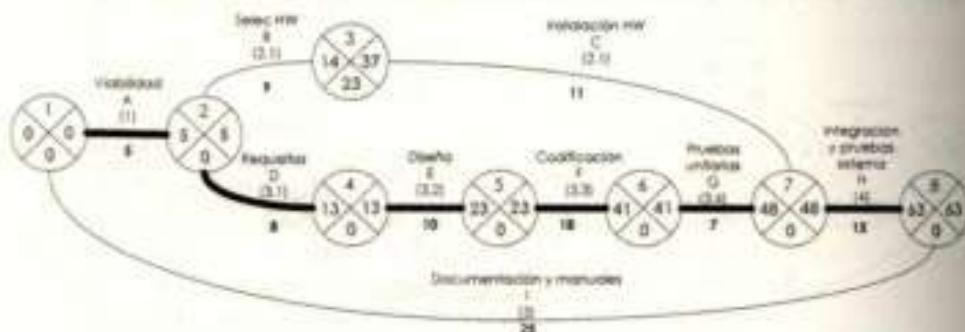


Figura 10.15: Ejemplo de CPM: Holgura y camino crítico

Pueden existir tareas ficticias de duración cero (*dummy*), que se utilizan para asignar precedencias en el orden de otras. En el ejemplo, si fuese necesario haber terminado la tarea «Selección HW», para realizar la tarea «Diseño», tendríamos una precedencia más, tal y como se indica en la Tabla 10.12. El resultado final se muestra en la Figura 10.16: aparece una tarea ficticia entre los nodos 3 y 4 que altera el camino crítico y las fechas.

Tabla 10.11: Valores de CPM

Act	Dur	IP	TP	IT	TT	Hol	¿Caminó crítico?
A	5	0	0	5	5	0	Si
B	9	5	28	14	37	23	No
C	11	14	37	25	46	34	No
D	8	5	5	13	13	0	Si
E	10	13	13	23	23	0	Si
F	18	23	23	41	41	0	Si
G	7	41	41	48	48	0	Si
H	15	48	48	63	63	0	Si
I	25	0	38	25	63	38	No

Tabla 10.12: Actividades incluyendo una nueva precedencia

Id	Actividad	Duración	Precedencias
0	Sistema Inteligente CCTV		
1	(A) Estudio viabilidad	5	—
2	Hardware		
2.1	(B) Selección HW	9	A
2.2	(C) Instalación HW	11	B
3	Software		
3.1	(D) Análisis de requisitos	8	A
3.2	(E) Diseño	10	B, D
3.3	(F) Codificación	18	E
3.4	(G) Pruebas unitarias	7	F
4	(H) Integración y pruebas del sistema	15	C, G
5	(I) Documentación y manuales	25	—

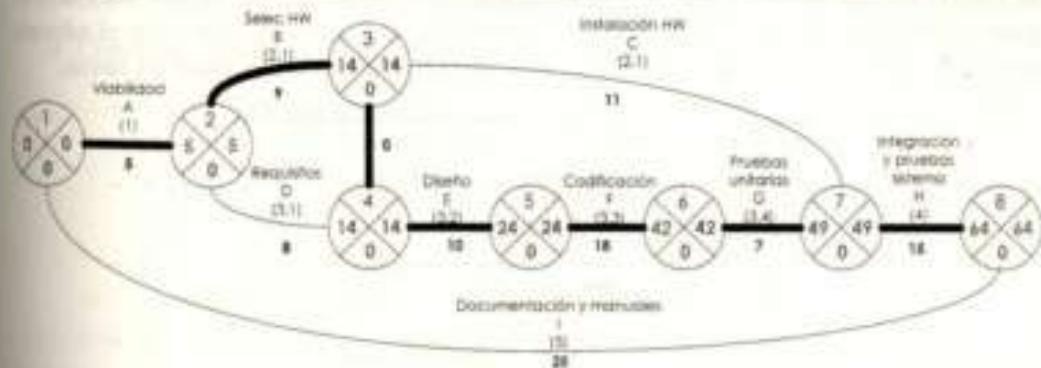


Figura 10.16: Holgura y camino crítico con una tarea ficticia

10.5.3 Diagramas de Gantt

Los diagramas de redes CPM y PERT definen las restricciones entre tareas pero no proporcionan información sobre su calendario. Los **diagramas de Gantt**, originales de Karol Adamiecki en 1896 y reinventados por Henry Gantt en 1910, muestran la evolución de las actividades con sus fechas e hitos. Una de las ventajas de los diagramas de Gantt es que facilitan la visión global del proyecto, mostrando el comienzo y el fin de las actividades, la asignación de recursos a tareas y –durante el desarrollo del mismo– las desviaciones entre los plazos estimados y reales. Como muestra el ejemplo de la Figura 10.17, en el eje vertical se muestran las actividades y en horizontal, el tiempo. Otra ventaja de los diagramas de Gantt es que son una técnica fácil de comprender por todos los miembros del proyecto y además, permite hacer un seguimiento del mismo.

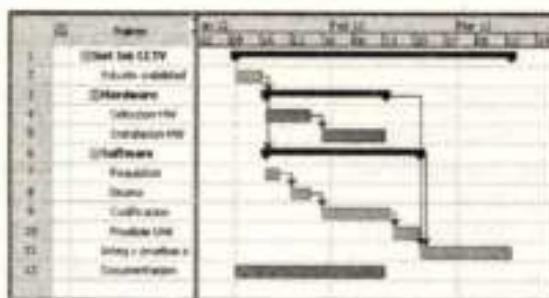


Figura 10.17: Ejemplo de diagrama de Gantt

A menudo, los diagramas de Gantt suelen complementarse asociando información sobre la asignación de recursos a las actividades con ánimo de monitorizar y controlar la carga de trabajo del personal y el progreso de las tareas (ver Figura 10.18). Aunque muy ampliamente utilizados, entre sus desventajas podemos citar que, a medida que crece el número de tareas, resultan complicados de usar y no es tan fácil ver la relación entre las tareas.

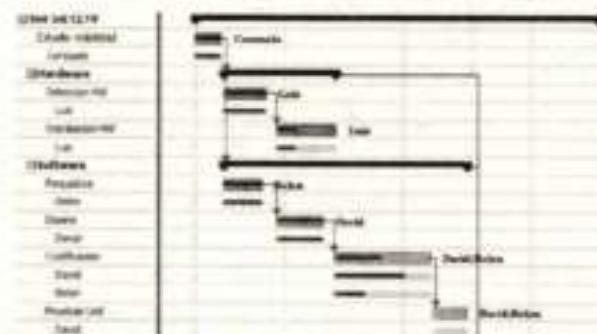


Figura 10-18: Diagrama de Gantt mostrando recursos y estado de las tareas.

10.5.4 Método del valor conseguido

El **método del valor conseguido**, más conocido por sus siglas en inglés, EVM (*Earned Value Management*), se desarrolló en el Departamento de Defensa de los Estados Unidos en la década de 1960 para el control de proyectos. EVM permite hacer un seguimiento del proyecto integrando información de plazos y costes, así como visualizar esta información en una única gráfica mediante tres variables que muestran la evolución de un proyecto en el tiempo:

- Coste presupuestado o coste presupuestado del trabajo planeado (*Budgeted Cost of Work Scheduled*, BCWS): se calcula sumando los costes planificados del proyecto.
- Valor actual o coste real del trabajo realizado (*Actual Cost of Work Performed*, ACWP): representa el esfuerzo invertido hasta la fecha, es decir, los costes realmente gastados en las tareas llevadas a cabo en el proyecto hasta la fecha.
- Valor conseguido (*Earned Value*) o coste presupuestado del trabajo realizado (*Budgeted Cost of Work Performed*, BCWP): representa el grado de finalización de las distintas tareas. Se calcula sumando los costes planificados de las tareas que realmente se han llevado a cabo hasta la fecha.

A partir de estas variables se puede calcular el estado actual del proyecto y predecir cómo irá su ejecución en lo que quede de proyecto a partir de una serie de indicadores. Éstos tienen la finalidad de orientar a los gestores de proyectos a realizar acciones correctivas cuando los proyectos se desvían de la planificación. Además, es posible representar las variables gráficamente, lo que es más importante. La Figura 10.19 muestra cómo dichos indicadores se pueden combinar en una única gráfica, con lo que los gestores de proyectos pueden rápidamente analizar el estado del proyecto.

A continuación, se describen los indicadores más importantes de EVM:

- Constantes del proyecto en relación a los costes y plazos:
 - Valor presupuestado del proyecto (*Budget At Completion*, BAC): simplemente la estimación del coste del proyecto.
 - Reserva de gestión (*Management Reserve*, MR): sobrecoste permitido que el proyecto puede usar de colchón ante cualquier imprevisto.
 - Reserva de plazo (*Slippage Reserve*, SR): tiempo que permitimos que el proyecto se deslice ante cualquier imprevisto que pueda surgir.
 - Ratio de coste:

$$CR = (BAC + MR)/BAC$$

- Ratio de plazo:

$$SR = (BAC + SR)/BAC$$

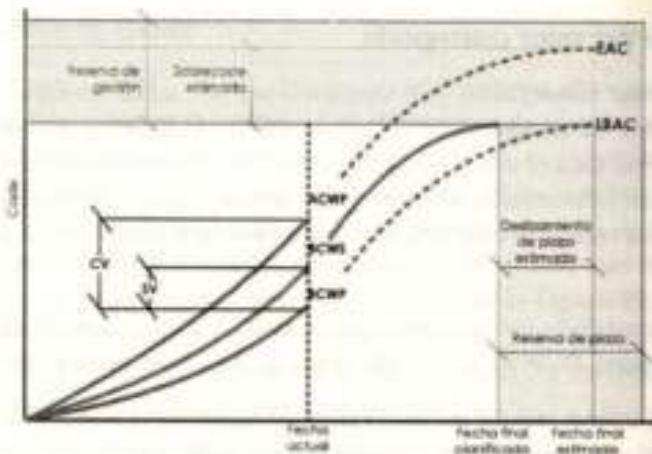


Figura 10.19: Representación gráfica de EVM

- Las varianzas entre las tres variables indican si el proyecto va o no retrasado y si hay o no sobrecoste.
- Varianza de coste (*Cost Variance*, CV): es la diferencia entre el valor conseguido y el valor actual. Si el valor de *CV* es positivo, es que se trabaja con más eficiencia de la planificada, y al revés, si es negativo.:

$$CV = BCWP - ACWP$$

- Varianza de plazo (*Schedule Variance*, SV): es la diferencia entre el valor conseguido y el valor presupuestado. Si esta diferencia es positiva, el proyecto va adelantado, si es negativa el proyecto se está retrasando:

$$SV = BCWP - BCWS$$

- Varianza de coste relativa (*Cost Variance Percentage*, CVP): indica la desviación porcentual con respecto al presupuesto del proyecto:

$$CVP = CV / BCWP$$

- Varianza de plazo relativa (*Schedule Variance Percentage*, SVP): indica el adelanto o retraso acumulado por el proyecto en términos porcentuales. Un valor positivo indica que el proyecto va adelantado, mientras que uno negativo indica retraso con respecto al plazo inicialmente previsto.

$$SVP = SV / BCWS$$

- Índices. En los índices, valores cercanos a uno indican que la ejecución, tanto en costes como en plazos, está muy cerca de la planificada. Cuando más se alejan de uno por abajo, es que el proyecto se está retrasando o costando más de lo planificado, pero si están por encima de uno, el proyecto se está realizando mejor que lo planificado. Los índices tienen la ventaja de que permiten comparar proyectos sin tener en cuenta su tamaño. Algunos de los índices que pueden definirse son:

- Índice de eficiencia de coste (*Cost Performance Index*, CPI):

$$CPI = BCWP / ACWP$$

- Índice de eficiencia de plazo (*Schedule Performance Index*, SPI):

$$SPI = BCWP / BCWS$$

- Índice de coste y plazo (*Cost-Schedule Index*, CSI):

$$CSI = CPI \cdot SPI$$

- Índice de eficacia para la conclusión (*To Complete Performance Index*, TCPI): es el cociente entre el trabajo restante y el presupuesto restante (equivale a CSI):

$$TCPI = (BAC - BCWP) / (EAC - ACWP)$$

La Figura 10.20 muestra gráficamente el uso de los índices. En la parte izquierda, el proyecto va mejor de lo planificado, mientras que en el centro va mucho peor: a pesar de que se está trabajando más, se está consiguiendo menos de lo planificado.

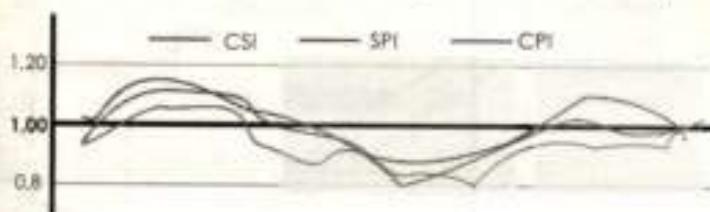


Figura 10.20: Análisis de los índices de EVM

- Previsiones. Son indicadores que ayudan a determinar el estado al final del proyecto, es decir, su progreso. Generalmente se asume que los índices de eficiencia de costes y plazos se mantendrán constantes.
- Valor estimado hasta la conclusión (*Estimate To Complete*, ETC): estimación del coste necesario para la finalización del proyecto. Al final del proyecto será cero, ya que el coste estimado será igual que el coste real:

$$ETC = EAC - ACWP$$

- Valor estimado a la conclusión (*Estimate At Completion*, EAC): se calcula extrapolando las curvas del valor realizado real (ACWP) y el conseguido (BCWP).

$$EAC = ACWP + \frac{(BAC - BCWP)}{CPI}$$

Otra forma de calcular EAC es:

$$EAC = BAC / CPI$$

Ejemplo de aplicación de EVM

Imaginemos un proyecto pequeño en el que marcamos cuatro fechas de revisión, las cuales podemos definir como *hitos* (ver Figura 10.21), y en las que analizaremos el estado de este proyecto.

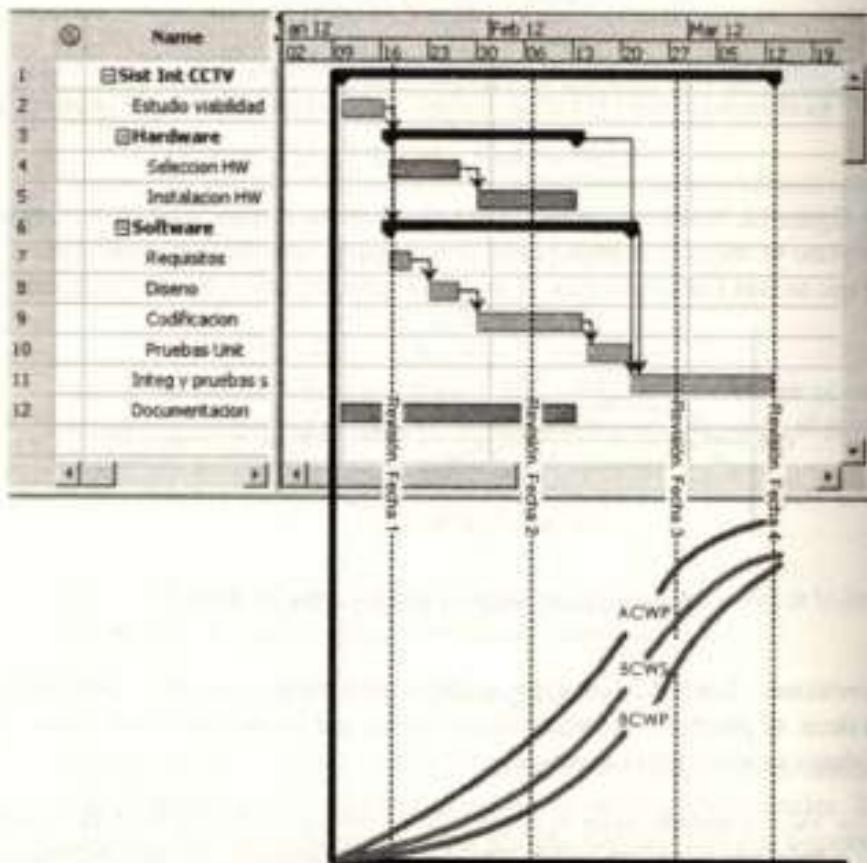


Figura 10.21: Fechas o hitos de análisis del proyecto

En el citado proyecto se conocen los valores de una serie de parámetros, los cuales se listan a continuación:

LBAC: 15.120 euros

Coste hora: 30 euros

Tiempo de conclusión: 504 horas

Ratio coste RC: 1,10

Reserva de gestión: 1.500 euros

Ratio plazo RS: 1,12

Reserva de plazo: 60 horas

Tanto la gráfica del valor conseguido de la Figura 10.22 como la de los índices de la Figura 10.23 muestran que, en general, es un proyecto irregular en el que se cruzan las líneas de los tres valores y se producen desviaciones importantes en relación a lo planificado. Por ejemplo, podemos imaginar que es un proyecto en el que se utiliza tecnología en la que la organización no tiene experiencia o cuenta con personal novel, pero se puede apreciar que los gestores del proyecto han realizado acciones correctivas, terminado el proyecto con un pequeño sobrecoste, pero dentro de la zona de reserva.

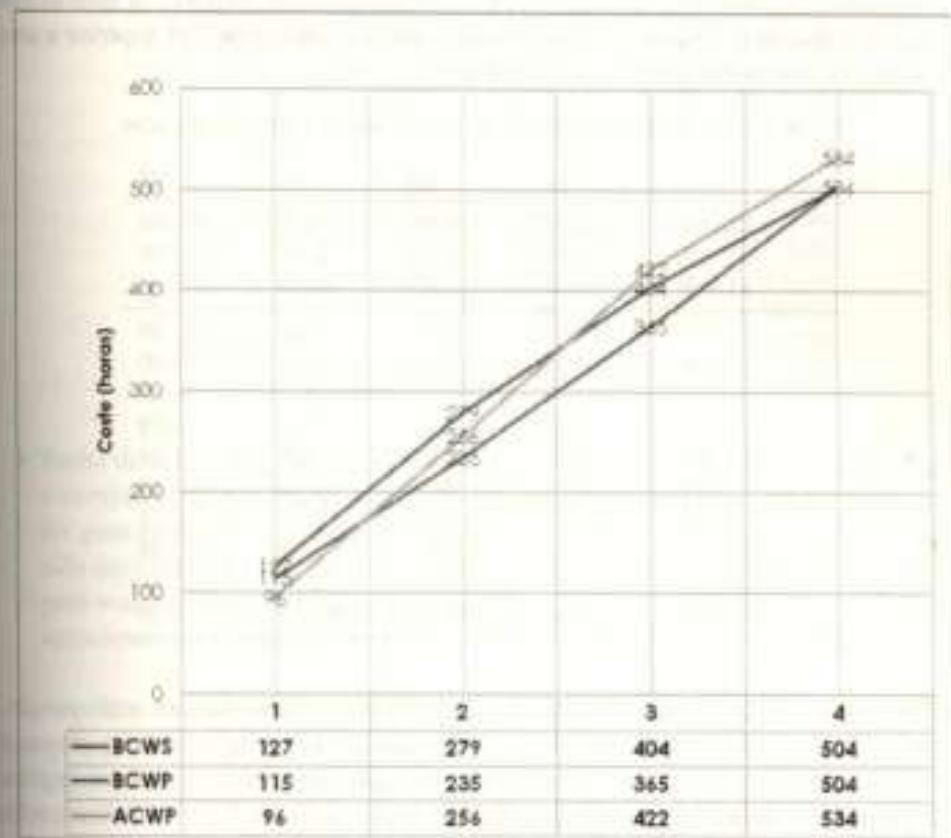


Figura 10.22: Gráfica de EVM



Figura 10.23: Índices del proyecto

Analizando las fechas más en detalle:

- Fecha de revisión 1: la varianza de coste es positiva, por lo que la eficacia con la que se trabaja en este punto es positiva, pues está por debajo de lo que se invierte. En cuanto a la varianza de plazo, se observa un valor inicial negativo, lo que predice una holgura en el plazo. Mirando los índices de eficiencia en el estado del proyecto, los gestores deberían incrementar las horas extra o las personas asignadas para mejorar su evolución temporal y acercar el SPI a uno (actualmente en 0,91), si bien es cierto que el proyecto evoluciona correctamente al tener un índice de CPI superior a uno, y siendo la desviación de SPI aún asumible.

Tabla 10.13: Indicadores EVM en los diferentes hitos del proyecto

	H1	H2	H3	H4
BCWS	127,00	152,00	125,00	100,00
BCWP	115,00	120,00	130,00	139,00
ACWP	96,00	160,00	166,00	112,00
CV	19,00	-40,00	-36,00	27,00
SV	-12,00	-32,00	5,00	39,00
CVP	0,17	-0,33	-0,28	0,13
SVP	-0,09	-0,21	0,04	0,39
CPI	1,20	0,75	0,78	1,24
SPI	0,91	0,79	1,04	1,39
CSI	1,08	0,59	0,81	1,73
% completado	22,82	23,81	25,79	27,58
% gastado	19,05	31,75	32,94	22,22
EAC	12.622,00	20.160,00	19.307,00	12.183,00

- Fecha de revisión 2: en esta revisión el proyecto va francamente mal, ya que se trabaja más de lo esperado, con menor eficiencia de la prevista y con una proyección muy poco alentadora. Tanto la varianza de plazo como la de coste son negativas. Además, los índices de eficiencia están por muy debajo de 1 y prácticamente fuera de una posible rectificación. Los gestores del proyecto deberían, bien renegociar el coste y plazos de entrega, bien incrementar las horas horas extra y el número de personas asignadas a este proyecto, o bien ambas cosas, si se quieren subsanar estas

desviaciones. El caso es que el progreso del desarrollo se ha estancado, mostrándose claramente la falta de experiencia o motivación del personal.

- Fecha de revisión 3: el proyecto aún va mal, pero ha mejorado. En este punto se ha conseguido más trabajo del planificado, pero a su vez, ha costado más que lo que debería (por ejemplo, gracias a muchas horas extras pagadas a un coste superior que la hora normal), lo que ha llevado a que la varianza de plazo sea positiva. Sin embargo, todos los valores acumulados siguen siendo negativos. Al igual que en la revisión anterior, sigue siendo necesaria la mejora de la productividad. Además de horas extra o personal, se podría considerar si es posible la sustitución de personal novel por otro más experimentado.

Tabla 10.14: Indicadores EVM acumulados en los diferentes hitos del proyecto

	H1	H2	H3	H4
BCWS	127,00	279,00	404,00	504,00
BCWP	115,00	235,00	365,00	504,00
ACWP	96,00	256,00	422,00	534,00
CV	19,00	-21,00	-57,00	-30,00
SV	-12,00	-44,00	-39,00	0,00
CVP	0,17	-0,09	-0,16	-0,08
SVP	-0,09	-0,16	-0,10	0,00
CPI	1,20	0,92	0,86	0,94
SPI	0,91	0,84	0,90	1,00
CSI	1,08	0,77	0,78	0,94
% completado	22,82	46,63	72,42	100,00
% gastado	19,05	50,79	83,73	105,95
EAC	12.622,00	16.471,00	17.481,00	16.020,00

- Fecha de revisión 4: En el tramo final, el proyecto se ha recuperado y ha terminado a tiempo, ya que la varianza de plazo es cero. Parece que las medidas correctoras de los gestores del proyecto han dado sus frutos. Además, los índices de eficiencia han sido muy positivos y el único punto negativo es que ha costado más de lo planificado, pero este coste es inferior a la reserva de gestión, un 6% frente a una reserva de aproximadamente el 10% (16.020 euros frente a 15.120 euros planificados).

Los valores de EVM pueden fácilmente realizarse en hojas de cálculo, pero las herramientas de gestión de proyectos como Microsoft Project u Open Project incluyen EVM como técnica de seguimiento de proyectos, permitiendo definir distintos costes por persona, formas de distribuir el trabajo de una actividad (por ejemplo, picos de trabajo al principio o al final, constante, forma de campana, etc.), y suelen incluir funcionalidades para su visualización. Por ejemplo, la Figura 10.24 muestra la variable ACWP para una actividad y tiempo concretos en la realización del proyecto.



Figura 10.24: Cálculo de ACWP con una herramienta de gestión de proyectos

10.6 Revisiones y cierre del proyecto

Com hemos visto en capítulos anteriores, a medida que avanza el trabajo en un proyecto, los trabajos y resultados técnicos (código, documentación, etc.), han de ser revisados. Igualmente, desde la perspectiva de la gestión del proyecto resulta imprescindible realizar revisiones e inspecciones que permitan confirmar si se está cumpliendo con el plan definido, si el proyecto cubre la funcionalidad requerida o si la calidad de los productos y artefactos, tanto intermedios como finales, definidos en el proyecto, es la deseada. Las inspecciones, por ejemplo, dan visibilidad al avance del proyecto, permitiendo dar por concluidos hitos o artefactos cuando se considera que tienen la calidad suficiente.

La guía del conocimiento de la gestión de proyectos (PMBOK) considera los siguientes tipos de revisiones:

- Revisiones del cronograma. Consisten en actualizar fechas de hitos del proyecto aprobado, como respuesta a una solicitud de cambio en el alcance del proyecto por parte del cliente, o a cambios en las estimaciones.
- Revisiones de rendimiento. Analizan actividades con variaciones en los costes planificados, utilizando técnicas como la del valor conseguido.

- Revisiones en la identificación de riesgos, de las que hablaremos más adelante.
- Revisiones de la documentación. Son revisiones formales de la calidad de todo tipo de documentación del proyecto.

Una vez cerrado el proyecto, es conveniente realizar un resumen de lo acontecido en todos sus aspectos, tanto tecnológicos como de gestión, para mejorar de cara futuros esfuerzos. Por ello, además de las revisiones que se lleven a cabo durante la vida del proyecto, es necesario realizar un análisis llamado *post-mortem*, donde se hacen revisiones de los problemas surgidos y las futuras oportunidades abiertas por el proyecto terminado. Entre las actividades a realizar se puede mencionar la inclusión de toda la información posible en el histórico de proyectos, información que, como hemos visto, será usada para realizar estimaciones en futuros proyectos.

10.7 Gestión de los recursos humanos

Para la correcta organización del personal dentro de un proyecto, los gestores del mismo no sólo necesitan conocer el número aproximado de personas que necesitan para llevarlo a cabo, sino también sus características para su asignación a las tareas que mejor se ajusten a cada uno. Entre estas características podemos citar su habilidad e interés para llevar a cabo cierto tipo de trabajos, su experiencia, el conocimiento que poseen de procesos y herramientas, etc. Pero también es necesario conocer sus habilidades personales de gestión, comunicación con el resto del equipo y nivel de responsabilidad.

Pero no sólo por las habilidades de una persona, sino también por su forma de ser, habrá que guiarse a la hora de asignarle tareas en un proyecto. Se suele clasificar a las personas en *extrovertidos* –los que tienden a decir lo que opinan– e *introvertidos* –los que tienden a preguntar opiniones antes de llevar a cabo ciertas tareas–. Por otro lado, también se puede clasificar a las personas en *intuitivos* –que se dejan llevar o se fían de los sentimientos e intuición– y *racionales* –aquellos que buscan más la lógica y los hechos antes de tomar decisiones–. La combinación de estos cuatro tipos de personalidad, además de las características del personal descritas, pueden indicar al gestor de proyectos qué individuos son los más apropiados a cada tipo de tarea.

Aunque en el Capítulo 3 estudiamos alguna métrica relacionada con los recursos y el personal, tales como el grado de complejidad de las comunicaciones y el número de vías de comunicación, es muy importante tener en cuenta la productividad individual del personal. La productividad implica efectividad y eficiencia, tanto a nivel individual como organizativo, entendiéndose por *efectividad* el cumplimiento de los objetivos, y por *eficiencia*, el cumplimiento de los objetivos con la menor cantidad de recursos.

Es conocido que en computación hay personas mucho más productivas que otras. Algunos estudios que han identificado personas hasta cinco veces más productivas que otras con similar rango y salario. Otros estudios consideran que dichas diferencias pueden llegar a ser de 100 a 1. Al hilo de esta situación, algunos autores se han referido al «programador

de productividad negativa neta», esa persona que se identifica en casi todos los proyectos y que es causante de más efectos negativos que positivos. Estos autores destacan que en un equipo de 10 individuos, puede llegar a haber hasta 3 personas que introducen tal tasa de errores que convierten su productividad neta en negativa.

Otro aspecto a tener en cuenta es la organización del personal dentro del equipo de desarrollo. Generalmente, se consideran dos tipos de equipos: *programador jefe*, que es una estructura jerárquica donde el mejor miembro del equipo lo dirige asignando tareas, y la aproximación *no egoísta*, un modelo más matricial donde la responsabilidad se encuentra más repartida entre todos los miembros del equipo.

10.8 Gestión y análisis del riesgo

En la Ingeniería del Software se define el **riesgo** como la posibilidad de que ocurra un evento no deseado que afecte a los objetivos de plazo, coste o calidad del proyecto, e incluso a la supervivencia de una organización. El riesgo está relacionado con la incertidumbre, ya que desconocemos si el evento ocurrirá o no (si no fuera así, se trataría de un hecho).

El **riesgo** se define como la posibilidad de que suceda un evento, un peligro, una amenaza o una determinada situación de consecuencias no deseadas (IEEE, 1990)

Generalmente se definen dos estrategias en la gestión del riesgo. La estrategia *reactiva* consiste en adoptar las medidas necesarias una vez que el evento ha sucedido —se suele utilizar el símil de «ir apagando fuegos»—. Se trata de una estrategia que, por supuesto, no se recomienda. Por otro lado, la estrategia *proactiva* consiste en prevenir los riesgos, siguiendo un proceso para prevenir o mitigar los problemas que puedan acontecer, identificándolos, analizando su probabilidad e impacto, teniendo planes de contingencia para tomar las medidas necesarias y controlando constantemente el proceso de desarrollo.

La guía SWEBOK establece las siguientes actividades dentro de una estrategia de gestión de riesgos:

- Identificación y análisis de riesgos, que consiste en identificar qué puede ir mal, cómo, por qué y cuales serían las posibles consecuencias.
- Evaluación de riesgos críticos, para situar los riesgos más significativos en cuanto a exposición y perspectiva de pérdida.
- Plan de mitigación de riesgos, que contendrá la estrategia para eludirlos o, en todo caso, minimizar su impacto.

La identificación consiste, como hemos indicado, en determinar qué riesgos se van a controlar. Para ello, existen distintas estrategias, tales como tener listas donde se comprueba

si hay riesgos aplicables al proyecto, descomponer el problema y analizarlo, etc. Es importante identificar todos los riesgos potenciales, ya que de otra forma no serían considerados en etapas posteriores.

Generalmente, los riesgos se clasifican en tres categorías distintas:

- Riesgos relacionados con el proyecto, que afectan a los plazos o recursos.
- Riesgos relacionados con el producto, que afectan a la calidad del software que se está desarrollando.
- Riesgos relacionados con el negocio, que afectan a la organización. Un ejemplo de ellos serían las graves consecuencias que para una organización puede tener el fracaso de un proyecto, en términos de acciones legales, reducciones de plantilla, etc.

En el análisis de riesgos se evalúa la probabilidad y gravedad de cada uno, teniendo en cuenta, para ello, los siguientes elementos relacionados con el riesgo:

- El *impacto* de un riesgo es la perdida asociada con el mismo. El impacto generalmente se clasifica como insignificante, tolerable, serio o catastrófico, siendo, a menudo, el coste el criterio para medir las pérdidas.
- La *probabilidad* de un riesgo se suele expresar en rangos discretos, como pudiera ser, por ejemplo, *muy bajo* (<10%), *bajo* (10-25%), *moderado* (25-50%), *alto* (50-75%) o *muy alto* (75-100%).
- La *exposición* a un riesgo se calcula a partir de su impacto y probabilidad.

Generalmente los riesgos se ordenan en una tabla (similar a la Tabla 10.15), donde según avanza el proyecto se deben ir actualizando los datos, pues para cualquier riesgo, tanto el impacto como su probabilidad pueden cambiar.

Tabla 10.15: Ejemplo de gestión y control de riesgos

Orden	Riesgo	Probabilidad	Impacto
1	Cambios en los requisitos ya iniciado el proyecto	Alta	Serio
2	No poder contratar el personal necesario	Alta	Serio
3	El tamaño del software se ha infravalorado	Alta	Tolerable
4	Disponibilidad del cliente menor de la esperada	Moderada	Tolerable
5	Problemas con las licencias de las herramientas	Moderada	Tolerable
6	Formación insuficiente o inadecuada del equipo	Moderada	Serio
7	Incendio de las oficinas	Muy baja	Catastrófico

Dentro de la gestión de riesgos, no es infrecuente hacer un análisis cuantitativo. En la Figura 10.25 se muestra cómo la probabilidad de que un evento ocurra se puede multiplicar por su coste, tal y como se calcula en cada rama del árbol. En esta figura se puede observar cómo el riesgo combinado de no realizar inspecciones (de código o diseño) supera al de realizarlas, por lo que resulta más seguro llevarlas a cabo. Este tipo de análisis cuantitativos puede utilizarse para priorizar riesgos y tener más controlados aquellos con más prioridad.



Figura 10.25: Cálculo cuantitativo de la exposición de riesgos

En cuanto a la planificación de riesgos, el método más general consiste en considerar cada riesgo e identificar para cada uno la estrategia que servirá para gestionarlo. Dependiendo del riesgo y de su efecto, se pueden utilizar estrategias que lo eviten –por ejemplo, si la automatización de un módulo es demasiado compleja, puede hacerse manualmente hasta que mejore la tecnología– o que lo minimicen –por ejemplo, añadiendo personal redundante si hay muchas probabilidades de una alta tasa de abandonos–. Dentro de la planificación de los riesgos es importante detallar los **planes de contingencia**, donde se detallará qué hacer si el evento adverso ocurre. Para ciertos tipos de riesgos, la estrategia consiste en desviárselos a otras fuentes, por ejemplo, contratando un seguro que evite pérdidas económicas.

10.9 Resumen

A lo largo del presente capítulo hemos estudiado los temas básicos de gestión de proyectos: iniciación, planificación, seguimiento y cierre de proyectos.

En el estudio de la planificación hemos considerado la estimación de esfuerzo y plazos con diferentes técnicas. También hemos visto cómo evaluar la bondad de dichas técnicas y la necesidad de calibrarla, por ejemplo, mediante la adaptación de modelos de regresión e inteligencia artificial a la situación actual de una organización. Hemos estudiado cómo una vez definidas y organizadas las tareas –generalmente utilizando la descomposición de paquetes (EDT)–, los modelos de red (CPM y PERT) y los diagramas de Gantt pueden ser útiles para la planificación y el control del proyecto.

Para el seguimiento de proyectos hemos estudiado la técnica del valor conseguido, que permite combinar costes y plazos analizando tanto el estado actual de un proyecto como sus estimaciones futuras, en función de su estado actual.

Otras actividades necesarias para la gestión de proyectos que hemos visto en este capítulo son la gestión de personal y la gestión de riesgos. La gestión de riesgos es una actividad fundamental para evitar o mitigar situaciones adversas al proyecto que pueden afectar sus plazos o calidad, y a su vez a la organización en caso de proyectos vitales para la misma. La gestión del personal, por su parte, se antoja esencial para obtener de los individuos el máximo rendimiento en función, lógicamente, de su capacidad y características personales.

La siguiente nube de palabras muestra la importancia relativa de todos estos conceptos y otros vistos a lo largo de este capítulo.



Figura 10.26: Principales conceptos tratados en el capítulo

10.10 Notas bibliográficas

Hay buena bibliografía sobre la gestión en la Ingeniería del Software, pues existen muchos libros exclusivamente relacionados con el tema. Siempre hemos pensado que uno de los más completos es el de R. Futrell y sus colaboradores (Futrell, Shafer y Shafer, 2002). Entre los estándares internacionales para la gestión de proyectos, cabe resaltar:

- La guía del conocimiento de la gestión de proyectos PMBOK (Project Management Institute, 2004) es el modelo más importante para la gestión de proyectos.
 - El estándar de planes para la gestión de proyectos, ANSI/IEEE 1058 (IEEE, 1987).
 - El estándar de métricas de productividad, IEEE 1045 (IEEE, 1993).

También hay abundante bibliografía sobre estimación de proyectos de software, dentro de la cual se pueden recomendar preferentemente los libros de medición ya mencionados en el Capítulo 3, especialmente el de Fenton y Pfleeger (1997). Otra referencia interesante, más enfocada desde el punto de vista de la gestión de la calidad y que incluye los procesos de medición es el de S. Kan (2002). Un libro que resulta fácil de leer y que es muy recomendable para la aplicación de modelos estadísticos en la Ingeniería del Software es el de K. Maxwell (2002). En cuanto a COCOMO, la referencia original de COCOMO 81 es la de Boehm (1981) y las de COCOMO II la de Boehm et al. (2000). Para los puntos de función, la referencia original es la Albrecht (1979), aunque resulta más accesible el trabajo de Albrecht y Gaffney (1983) y especialmente el libro de Garmus y Herron (2001). El trabajo seminal para los sistemas dinámicos es el de J. Forrester (1961), aunque dentro del ámbito de la Ingeniería del Software, la referencia fundamental es la de Abdel-Hamid (1991).

En cuanto a los repositorios donde pueden encontrarse datos sobre gestión de proyectos, el más conocido es el ISBSG (*International Software Benchmarking Standards Group*) en <http://www.isbsg.org/>, que además de proporcionar un histórico de proyectos de distintas organizaciones, incluye soporte de consultoría y herramientas básicas de medición. Otro repositorio que tiene una conferencia asociada es PROMISE, en el que se albergan conjuntos de datos relacionados con la Ingeniería del Software en general y no sólo de gestión de proyectos. Su dirección Web es <http://www.promisedata.org/>.

Con respecto a la gestión del riesgo, destacan sobre los demás el libro de Hall (1998) y el de McManus (2003).

10.11 Cuestiones de autoevaluación

10.1 ¿Qué tres variables deben los gestores de proyecto balancear en todo proyecto para llevarlo a buen término?

- R. Los gestores de proyectos deben balancear la funcionalidad –dentro de una calidad aceptable por parte del cliente– con los costes y plazos estimados para que el proyecto se considere exitoso.

10.2 Resuma las principales tareas de los gestores de proyectos.

- R. Antes de comenzar el proyecto, la principal tarea consiste en la planificación y estimación. Durante su ejecución, el control del proyecto y la replanificación.

10.3 Razona sobre si la siguiente sentencia es verdadera o falsa: «En un proyecto, basta con realizar una única estimación y un único modelo».

- R. Falso. Las estimaciones deben realizarse varias veces según se adquiera más información sobre el proyecto. Aunque quizás lo más común es utilizar un único tipo de modelo de estimación, si es posible, resulta muy conveniente utilizar varios modelos.

10.4 ¿Por qué es necesario el ajuste empírico (calibración) en los modelos paramétricos?

- R *Ajustando los modelos a parámetros, como por ejemplo, la productividad, una organización generará estimaciones más exactas. Sabemos que a medida que se incrementa la información en la base de datos histórica de proyectos, resulta más necesario realizar una nueva calibración con los nuevos datos e incluso, borrar información sobre proyectos obsoletos.*
- 10.5** En la Ingeniería del Software, ¿cuáles son los criterios de calidad más comunes para valorar la bondad de los modelos de estimación?
- R *Los dos criterios más comunes son la magnitud media del error relativo (MMER), que debería estar por debajo de 0.25, y el nivel de predicción al 25% (Pred(25)) que debería ser superior al 75%.*
- 10.6** ¿Qué tres variables fundamentales nos permiten el seguimiento de un proyecto con la técnica del valor conseguido o EVM?
- R *La técnica del valor conseguido o EVM combina costes y plazos con tres variables denominadas valor presupuestado (BCWS), valor conseguido (BCWP) y valor actual o Earned value (ACWP).*
- 10.7** ¿Cuál es el mayor inconveniente de las técnicas como los sistemas dinámicos, que simulan el comportamiento de los proyectos a partir de los posibles valores de los parámetros de entrada?
- R *El mayor inconveniente está en que estos modelos necesitan ser validados y adaptados a su entorno de organización, lo cual es una tarea iterativa y costosa.*
- 10.8** Razona sobre si la siguiente sentencia es verdadera o falsa: «Las técnicas de diagramas de redes y Gantt son independientes e incompatibles».
- R *Falso. De hecho, ambas son complementarias: las redes favorecen la visualización del orden (tareas predecesoras y sucesoras) y los diagramas Gantt destacan los plazos temporales. Es más, las herramientas de gestión de proyectos suelen mostrar ambas representaciones, simplemente cambiando la vista.*
- 10.9** ¿Cuál es la principal característica de la representación gráfica de la técnica del valor añadido?
- R *La gráfica del valor añadido tiene la característica de que con una sola gráfica se puede ver el estado actual del proyecto y la estimación final, tanto de costes como de plazos.*
- 10.10** Una vez presentado el plan de trabajo al cliente, necesita reducir el plazo de entrega. ¿Por cuáles de las actividades empezaría?
- R *Lo más indicado sería mirar en cuánto se pueden reducir las actividades del camino crítico en un modelo como CPM.*

10.12 Ejercicios y actividades propuestas

10.12.1 Ejercicios resueltos

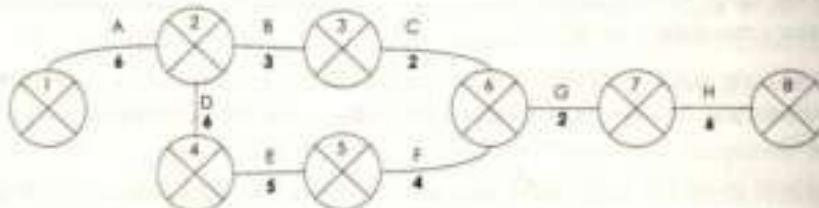
- 10.1** En una empresa de reciente creación, los proyectos poco a poco van creciendo en personal y tamaño. El problema ahora es que empieza a perder dinero con algunos proyectos debido a la falta de estimación. Decide ayudarles pero, ¿cómo empezaría?

Solución propuesta: Puesto que no hay histórico de proyectos, algunas de las posibles soluciones incluirían:

- Utilizar modelos paramétricos con los valores por defecto sin calibrar.
- Adquirir una base de datos de proyectos de otras compañías –como por ejemplo, ISBSG– y utilizar los datos como referencia en la estimación.

A largo plazo, la solución pasa por implantar procesos de medición, guardando en una base datos la experiencia y datos recabados con cada nuevo proyecto con el objetivo de una calibración continua de los métodos.

10.2 Dada la siguiente red CPM:



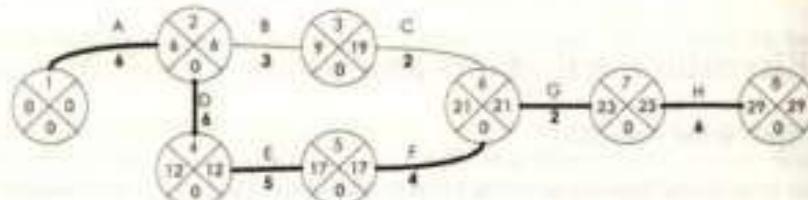
1. Dibuje la red e identifique el camino crítico.
2. Una vez iniciado el proyecto, se modifican ciertos requisitos y aunque las actividades actuales no necesitan ser modificadas, se añaden dos actividades nuevas:

Actividad	Predecesor	Duración
I	D	9
J	I	4

- (a) Dibuje la nueva red reflejando las nuevas actividades.
- (b) Identifique el camino crítico de la nueva red.
- (c) ¿Cuántos días puede retrasarse la actividad F sin que esto afecte al proyecto?
3. ¿Qué otra alternativa gráfica puede utilizarse para representar el proyecto?

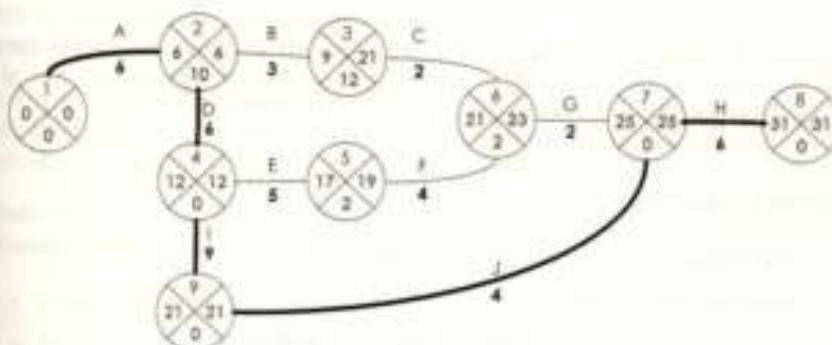
Respuesta:

1. Camino crítico: A-D-E-F-G-H.



2. En cuanto a la nueva red:

(a) Su representación gráfica sería la siguiente:



(b) Camino crítico de la nueva red: A-D-I-J-H, 31 días.

(c) Tiene una holgura de 2 días.

3. Como alternativa podemos emplear diagramas de Gantt.

10.3 El conjunto de datos de Kemerer (1987) es ampliamente utilizado en actividades de investigación. Calcule su regresión lineal para los siguientes datos y estime el esfuerzo para un proyecto en el que se han estimado 800 puntos de función.

Personas/mes	Puntos de función
287,0	1217,1
82,5	507,3
1107,3	2306,8
86,9	788,5
336,3	1337,6
84,0	421,3
23,2	99,9
130,3	993,0
116,0	1592,9
72,0	240,0
258,7	1611,0
230,7	789,0
157,0	690,9
246,9	1347,5
69,9	1044,3

Solución propuesta: La regresión lineal del esfuerzo, con un coeficiente de correlación de 0,76 es (expresado en *personas/mes*):

$$\text{esfuerzo} = 0,34 \cdot PF - 121,57$$

Para calcular el esfuerzo de un proyecto de 800 *KDSI*, sería:

$$\text{Esfuerzo} = 0,34 \cdot 800 - 121,57 = 150,43 \text{ personas/mes}$$

- 10.4** Utilizando COCOMO 81 y considerando un proyecto orgánico de aproximadamente 3.000 líneas de código entregado (3 *KDSI*), calcule el esfuerzo, el tiempo de desarrollo, el personal necesario y la productividad del mismo.

Solución propuesta:

- Esfuerzo = $2,4 \cdot 3^{1,05} = 7,6$ personas/mes.
- Tiempo de desarrollo = $2,5 \cdot 7,6^{0,38} = 5,4$ meses.
- Personal: $7,6 / 5,4 = 1,4$ personas a tiempo completo.
- Productividad: $3.000 / 7,6 = 395$ líneas de código/persona-mes.

- 10.5** Dada la siguiente tabla con valores estimados y reales calcule, utilizando *MMER* y *Pred(25)*, si podemos clasificar el modelo como bueno.

Proyecto	1	2	3	4	5	6	7	8	9	10
Estimado	50	75	135	164	345	260	320	450	500	600
Real	67	90	210	160	320	310	451	610	580	690

Solución propuesta: Completando la tabla, obtenemos:

Proyecto	1	2	3	4	5	6	7	8	9	10
Estimado	50	75	135	164	345	260	320	450	500	600
Real	67	90	210	160	320	310	451	610	580	690
Diferencia	17	15	75	4	25	50	131	160	80	90
% diferencia	0,25	0,16	0,35	0,03	0,08	0,16	0,29	0,26	0,13	0,13
>0,25%	sí	no	sí	no	no	no	sí	sí	no	no

y calculando *MMER* = 0,18 y *Pred(25)* = 60%, por lo que no podríamos considerar como bueno el método usado para la estimación, pues *MMER* está por debajo de 0,25 y el valor de *Pred(25)* no llega al 75%.

10.12.2 Actividades propuestas

- 10.1** Introduzca en una hoja de cálculo las ecuaciones de COCOMO y los factores de COCOMO intermedio y analice gráficamente cómo afectan al esfuerzo y tiempos de desarrollo las variaciones en estos parámetros para diferentes tamaños de producto. ¿Cuáles son los factores que más afectan al esfuerzo nominal?

- 10.2** Obtenga datos del repositorio PROMISE (<http://promisedata.org/>) relacionados con la estimación del esfuerzo. Utilizando la herramienta Explorer del paquete de minería de datos Weka (<http://www.cs.waikato.ac.nz/~ml/weka/>), cree un árbol de regresión con el algoritmo *M5P* incluido entre los clasificadores proporcionados y compare los resultados con la *curva de regresión* que también puede ser generada con la herramienta *Explorer*.
- 10.3** Utilizando alguna herramienta de gestión de proyectos u hoja de cálculo que incorpore el método del valor conseguido, cree un proyecto ficticio añadiéndole recursos y costes a las actividades para simular diferentes situaciones en diferentes etapas del desarrollo.
- 10.4** Analice las cuatro fechas de revisión del proyecto con los parámetros descritos a continuación utilizando la técnica de valor conseguido:
- LBAC: 21.180 euros
 - Fecha conclusión: 706 horas
 - Reserva de gestión: 2.500 euros
 - Reserva plazo: 72 horas
 - Coste hora: 30 euros
 - Ratio coste (*RC*): 1,12
 - Ratio plazo (*RS*): 1,10

	H1	H2	H3	H4
BCWS	92	167	202	245
BCWP	106	167	178	255
ACWP	100	175	160	252
CV	6,00	-8,00	18,00	3,00
SV	14,00	0,00	-24,00	10,00
CVP	0,06	-0,05	0,10	0,01
SVP	0,15	0,00	-0,12	0,04
CPI	1,06	0,95	1,11	1,01
SPI	1,15	1,00	0,88	1,04
CSI	1,22	0,95	0,98	1,05
% realizado	15,01	23,65	25,21	36,12
% gastado	14,16	24,79	22,66	35,69
EAC	19.981	22.195	19.038	20.931

- 10.5** Para la realización de un proyecto que consiste en un juego interactivo por internet, se ha estimado que serán necesarias 10 personas. Describa cómo seleccionaría a los miembros del equipo con respecto a los roles y conocimientos necesarios para poder completar el proyecto.
- 10.6** Busque información en internet o en artículos de investigación que expliquen lo que se entiende por economía o deseconomía de escala, y cómo esta consideración afecta a los modelos de estimación. ¿Son todos los modelos de estimación consistentes con uno de los dos tipos de economías?

- 10.7 En la guía del conocimiento de la gestión de proyectos (PMBOK) hay algunos aspectos que no han sido cubiertos en este capítulo –como por ejemplo, la adquisición– y que pueden ser relevantes a la Ingeniería del Software. Investigue éste y otros aspectos que complementan lo estudiado en este capítulo.
- 10.8 En relación con la estimación, hay una tendencia creciente a reconocer la experiencia humana para conseguir estimaciones precisas. Algunos estudios sugieren que las personas con experiencia realizan estimaciones más precisas que algunos de los modelos más avanzados vistos en el capítulo. Busque algunos de estos artículos (por ejemplo, a través de la búsqueda académica de Google Scholar) y debata en grupo sobre ello. Una buena base de partida puede ser el artículo «*A controlled experiment to assess the benefits of estimating with analogy and regression models*» publicado por Stensrud y Myrtveit en la revista *IEEE Transactions on Software Engineering* en 1999.
- 10.9 Se han publicado muchos artículos de estimación en la revista *IEEE Transactions on Software Engineering* a lo largo de los años. Haga una lista de los artículos más relevantes, tomando notas sobre cómo han ido evolucionando las técnicas de estimación desde los años 1970 con técnicas como las de Wolverton o Walston y Felix, hasta técnicas más avanzadas con minería de datos y redes Bayesianas.
- 10.10 Introduzca en algún paquete estadístico que sepa utilizar los datos publicados en algún artículo de investigación sobre estimación. Realice diferentes tipos de regresión y analice sus coeficientes de correlación y de determinación.

Gestión de la configuración del software

La gestión de la configuración del software es como cepillarse los dientes o pasar la revisión del automóvil, no es algo que disfrutes, pero sabes que las consecuencias de no hacerlo pueden ser terribles algún día.

— Tom Welsh

11.1 La importancia de poner las cosas en su sitio

Desmontar parte del motor de un coche antiguo para repararlo es una tarea realmente entretenida. Sobre todo porque habitualmente no conservamos ningún tipo de manual o descripción del mismo. En ocasiones, la avería es más o menos evidente (por ejemplo, un manguito pierde agua), pero se requiere desmontar un gran montón de piezas para poder acceder al lugar donde se encuentra e intentar arreglarla. A veces, uno termina reparando la avería. Es un momento de satisfacción. El problema es que, también en ciertas ocasiones, después de cerrar el capó del coche, nos percatamos de que ha quedado una pieza fuera que supuestamente formaba parte del motor antes de la reparación. Estas cosas suceden porque no hemos tenido cuidado en gestionar la configuración del motor de nuestro vehículo.

La escena descrita tiene un paralelo en la Ingeniería del Software. Un producto software de mediana envergadura está formado por miles de *elementos de configuración*, incluyendo no sólo el código sino también otros artefactos como bibliotecas externas y documentos. Además, las relaciones entre esos elementos para formar versiones funcionales del producto completo son más complejas de lo que la intuición o la memoria de los profesionales puede llegar a controlar. Por ejemplo, ciertos programas funcionan con la versión 1.1 de una

biblioteca determinada, pero no funcionan con versiones 1.2 y posteriores de la misma biblioteca, porque cambió —pongamos por caso— algún pequeño detalle en una interfaz. También en un software en desarrollo, el cambio que hizo ayer un programador a última hora de la tarde hace que a la mañana siguiente otros de sus compañeros comiencen la mañana con errores de compilación que ¡el día anterior no sucedían!

Por ejemplo, muchos desarrolladores de aplicaciones web en Java se han enfrentado a errores como el siguiente cuando trataban de configurar una aplicación web existente:

```
javax.xml.transform.TransformerFactoryConfigurationError: Provider  
org.apache.xalan.processor.TransformerFactoryImpl not found  
    javax.xml.transform.TransformerFactory.newInstance(Unknown Source)
```

Este es uno de los errores que se producen simplemente por tener accesible una versión incorrecta de la biblioteca Xerces para procesamiento de XML. Desgraciadamente, el error no informa del lugar en que se encuentra la causa del problema, y sólo la intuición o experiencia del programador (o la suerte en encontrar una solución en los foros de internet) permitirán dar con la solución. Problemas de configuración como éste tienen fácil solución; basta con especificar claramente las versiones con las que el software se ha probado. Un pequeño cuidado en la documentación de la configuración evita horas de esfuerzo en el intento de depurar este tipo de errores.

La complejidad del ensamblaje de ciertas versiones de ciertos elementos para hacer que el software funcione hace que la denominada *gestión de la configuración* sea una actividad esencial de soporte al proceso de ingeniería. Si no sabemos qué piezas forman el software y cómo encajan para que la cosa funcione, sólo podemos augurar problemas en el cumplimiento de los plazos.

Lógicamente, la gestión de la configuración no sólo abarca el producto final entregado, sino también las innumerables versiones de los diferentes elementos que se van produciendo como resultado de las actividades de ingeniería. Además, la dinámica de cambio y evolución del software hace que las piezas o elementos continúen cambiando una vez el software está en uso. Por tanto, el control debe extenderse de forma muy especial a los cambios que se hagan una vez entregado el software, es decir, a lo que se suele denominar *actividades de mantenimiento*. De hecho, introducir cambios en un software después de la entrega requiere habitualmente una decisión más meditada y justificada que antes de que haya clientes utilizando el software.

El objeto de este capítulo es toda la mecánica del control de los elementos, sus cambios y cómo se combinan para tener el producto software funcional. Esta parte de la Ingeniería del Software es realmente una actividad de soporte a otras actividades de desarrollo y no requiere del tipo de esfuerzo intelectual de carácter creativo que se debe emplear en el diseño del software. Pero esto es más bien un problema que una ventaja, ya que, como nos indica la cita que encabeza este capítulo, su aparente simplicidad hace que en muchas ocasiones se pase por alto o se descuide, lo cual en ocasiones lleva a perder el control de los elementos del software, que en un cierto momento dejan de «encajar» entre sí.

11.2 Objetivos

El objetivo general de este capítulo es conocer la función, los conceptos y las principales técnicas y herramientas de la gestión de la configuración del software, como función de soporte de otras actividades de Ingeniería del Software. Este objetivo principal se desglosa en los siguientes objetivos específicos:

- Comprender el concepto de configuración del software como la disposición de los elementos que conforman un software, y su evolución en el tiempo en forma de versiones de los elementos de configuración.
- Comprender la importancia de las líneas base para dar estabilidad a la gestión de la configuración del software, gracias al control formal de los cambios.
- Saber establecer y gestionar el proceso fundamental de la gestión de cambios.
- Conocer los fundamentos del control de versiones de elementos software y su papel dentro de la gestión de la configuración.
- Conocer los aspectos más relevantes de la planificación, organización y auditoría de la gestión de la configuración del software.

En el ámbito profesional, la gestión de la configuración requiere, entre otras, las siguientes competencias:

- Saber instalar, configurar y utilizar herramientas de control de versiones, entregas y construcciones. Aunque éstas se utilizan más en las actividades de codificación, pueden utilizarse para cualquier tipo de artefacto de la Ingeniería del Software.
- Diseñar procedimientos organizativos de control de cambios, teniendo en cuenta las características de la organización.
- Integrar los artefactos de diferentes ingenieros o incluso de diferentes equipos en líneas base estables que tengan funcionalidades determinadas.

Las actividades de gestión de la configuración requieren una cierta actitud de orden en los profesionales, con una orientación al control de los elementos que son producto de las diferentes actividades.

11.3 La configuración del software

Durante las diferentes fases de la Ingeniería del Software se genera una gran cantidad de *artefactos* de diferentes tipos, incluyendo los archivos con el código fuente de los programas, pero también muchos otros elementos tales como la documentación de los mismos, sus especificaciones, notas técnicas, etc.

Teniendo en cuenta lo anterior, llegamos a una primera definición de *elemento de configuración*, que más adelante reformularímos como un concepto de gestión:

Un elemento de configuración es cada uno de los *artefactos de Ingeniería del Software* que evolucionan durante el desarrollo y progreso de un sistema software

Son ejemplos notables de elementos de configuración los archivos de código fuente y los documentos, pero pueden serlo también elementos hardware, notas, bibliotecas y cualquier otra pieza que forma parte del sistema software final o en desarrollo.

Los elementos de configuración evolucionan durante el desarrollo, y tienen dependencias entre ellos. Por ejemplo, la especificación de un requisito software en un documento de requisitos da lugar a ciertos elementos en un diagrama de diseño, los cuales dan lugar, a su vez, a elementos de código fuente, y éstos, a los correspondientes elementos de código objeto. El código fuente tiene dependencias de otros archivos fuente o de bibliotecas preexistentes. Las dependencias se definen además sobre ciertas versiones de los otros elementos, es decir, sobre un cierto estado de un elemento que puede cambiar, siendo importante el darse cuenta que diferentes versiones de un mismo elemento pueden tener diferentes dependencias con otros elementos. El concepto de *versión* de un elemento de configuración puede definirse de la siguiente forma.

Una **versión** (de un elemento de configuración) es cada uno de los siguientes estados diferentes en los que un elemento de configuración se encuentra durante el desarrollo o durante la evolución del software

Este concepto de versión es muy conocido. Por ejemplo, todos entendemos que la versión 8.1 del navegador Internet Explorer de Microsoft es una variante menor de la versión 8.0 del mismo software. En ejemplos como éste, las versiones hacen referencia a un compuesto de muchos elementos software, que se entrega como un producto completo. En general, cada elemento o conjunto de elementos dentro de un software puede *versionarse*, si bien estas versiones son internas al desarrollo y no se exponen al conocimiento de los usuarios finales. Por otro lado, cabe preguntarse qué naturaleza o alcance deben tener los cambios en un elemento de configuración para que se considere que se ha creado una nueva versión. Por ejemplo, si un desarrollador está trabajando en un archivo fuente en un proyecto, cada pequeño cambio que hace (como por ejemplo, añadir un nuevo método en una clase, o añadir un comentario), ¿representa una versión nueva? Es importante tener en cuenta las siguientes aclaraciones respecto al concepto de versión.

- Las versiones pueden determinar diferentes estados de elementos de configuración atómicos, pero también de elementos compuestos, como puede ser un software entregable completo (por ejemplo, Mozilla Firefox 3.6.8), o una biblioteca (por ejemplo,

Xerces 2.6.2), o de subsistemas o conjuntos de código que tienen sentido sólo internamente dentro de una organización.

- No todo cambio acaba siendo considerado una versión. Durante el trabajo diario se hacen cambios a los elementos que no desembocan en versiones nuevas: si se registrase cada mínimo cambio como una versión diferente, pronto la configuración tendría un volumen de versiones inmanejable.

En general, podemos imaginar que los elementos están en el instante t_1 en un cierto estado (una cierta versión) dentro del sistema de gestión de la configuración. Entonces, un miembro del equipo de desarrollo toma uno de esos elementos para trabajar sobre él, y en un cierto instante t_2 registra en el sistema de gestión de la configuración el elemento modificado. Entre los instantes t_1 y t_2 (que puede ser un intervalo de días), pueden haberse hecho muchos cambios, pero desde el punto de vista de la gestión de la configuración, sólo el estado final que se registra en t_2 constituye una versión diferente a la anterior.

El gran volumen de elementos de configuración y la enorme cantidad de interrelaciones entre ellos hacen necesario establecer procedimientos de control y usar herramientas que permitan hacer seguimientos y reconstruir versiones intermedias. Es necesario controlar el tipo de cambios que pueden hacerse cuando un software ha llegado a una configuración estable, y quién debe aprobar dichos cambios. Todo esto es *gestión de la configuración*.

Antes de seguir hay que comprender el concepto general de *configuración*, que no es exclusivo del software. Todo sistema como tal tiene una configuración, que no es otra cosa que la disposición y las interrelaciones de sus partes. La configuración de un modelo de automóvil es la combinación de ciertas piezas de determinada forma, que dan lugar a unas prestaciones determinadas. Aunque otro modelo puede compartir piezas (por ejemplo, tener los mismos amortiguadores), diferirá en otras, que dan lugar a productos diferenciados. En el desarrollo de sistemas informáticos, la configuración es la disposición concreta de ciertos elementos de hardware y software combinados dentro de un sistema o producto determinado, y así se define en el glosario IEEE de Términos de Ingeniería del Software:

Configuración es la disposición de un sistema informático o un componente, definida por el número, naturaleza e interconexiones de sus partes constituyentes

A esta visión de la configuración como disposición de las partes y sus relaciones, la guía SWEBOK añade lo siguiente:

La **configuración** puede verse como una colección de versiones concretas de elementos hardware, *firmware* o software, combinados según ciertos procedimientos de construcción para servir a un propósito determinado

De este modo, se puede considerar la configuración de un software como una sucesión de configuraciones en el tiempo. Por lo tanto, hay dos maneras de ver la configuración: en un instante concreto del tiempo, como se considera en la primera acepción, o a través del tiempo, según la historia de los cambios. Lógicamente, no es necesario en todas las ocasiones registrar cada uno de los cambios que se producen en los elementos que conforman el software, sino sólo en aquellos que se consideren relevantes o definitivos.

De las definiciones anteriores se obtiene la siguiente (IEEE, 1990):

La gestión de la configuración es la disciplina que aplica dirección y control técnico y administrativo para: identificar y documentar las características físicas y funcionales de los elementos de configuración, controlar los cambios de esas características, registrar e informar del procesamiento de los cambios y el estado de la implementación, y verificar la conformidad con los requisitos especificados

siendo el elemento de configuración la unidad en gestión de la configuración.

Un elemento de configuración es un agregado de hardware, software o ambos, al que se da una designación y se trata como una entidad independiente en el proceso de gestión de la configuración

Ahora bien, si restringimos los elementos de configuración solamente a aquellos que se consideran parte del software, hablaremos de *gestión de configuración del software*, cuyos elementos de configuración llamaremos *elementos de configuración software* o *elementos software*. De ello se deduce que la gestión de la configuración del software consiste en un conjunto de actividades de *soporte* a otras actividades de Ingeniería del Software, pues todas las actividades –desde las de obtención de requisitos hasta las pruebas– generan artefactos (de muy diversa índole), que pueden considerarse elementos software. Teniendo en cuenta el valor de las actividades de gestión de la configuración del software como actividades de soporte, se puede dar una definición alternativa de gestión de la configuración:

La gestión de la configuración también puede definirse como el control de las diferencias en el sistema para minimizar el riesgo y el error

A continuación, pasamos a examinar las actividades de la gestión de la configuración del software, junto con conceptos adicionales que son importantes para el ingeniero implicado en estas actividades.

11.4 Actividades de gestión de la configuración del software

Las siguientes son cuatro preguntas típicas de la gestión de la configuración del software, que nos indican el tipo de servicios que esta actividad debe proporcionar al resto de las actividades de ingeniería:

- Dame la versión actual del documento X, incluyendo una lista con todos los cambios desde la fecha Y.
- Dame el último ejecutable, con una lista de los problemas pendientes y todas las características aún no implementadas.
- Dame un listado del módulo Z, en su versión W.
- Dime quién aprobó el cambio A al elemento de configuración B, quién lo realizó, y por qué se aprobó.

La primera cuestión tiene que ver con el control de cambios y versiones de documentos. De igual forma, los cambios al código que se piden en la tercera pregunta reflejan la necesidad de almacenar la historia completa de versiones. La segunda pregunta refleja también cómo la gestión de la configuración del software trata de productos construidos (un ejecutable) por diferentes partes. Esto da lugar a los conceptos de construcción (*build*) y entrega (*release*). Por último, todo el proceso de gestión y control debe estar registrado (incluyendo las responsabilidades de los individuos), como refleja la cuarta pregunta.

La Figura 11.1 resume las principales actividades que se engloban en la gestión de la configuración. La figura muestra las actividades de planificación y gestión de la configuración como el marco general de control y establecimiento de procedimientos en esta área de la Ingeniería del Software.

El documento denominado *plan de gestión de la configuración* (del que se hablará más adelante) recoge las políticas, procedimientos y planes relativos a la configuración dentro de la organización. En algunas organizaciones, la gestión de la configuración del software es una función independiente, que se encarga del control de los artefactos que se producen en el desarrollo, pero es muy frecuente también que la organización se limite a una persona responsable, o incluso que la responsabilidad esté distribuida entre los desarrolladores. De hecho, en muchos casos las mismas herramientas de desarrollo integran como funcionalidad propia elementos de gestión de la configuración del software como el control de versiones, delegando así funciones a los programadores. En cualquier caso, el plan de gestión de la configuración del software debe existir bajo alguna forma, bien sea como un plan detallado o al menos, como un conjunto de políticas comunes para el desarrollo.

Las actividades de *identificación de la configuración* son la base del resto de las actividades, determinando qué debe incluirse en la configuración y de qué forma. Para hacer efectivo el control de la configuración, deben establecerse procedimientos formales de *control de cambios*, incluyendo la toma de decisiones sobre qué cambios deben hacerse, quién

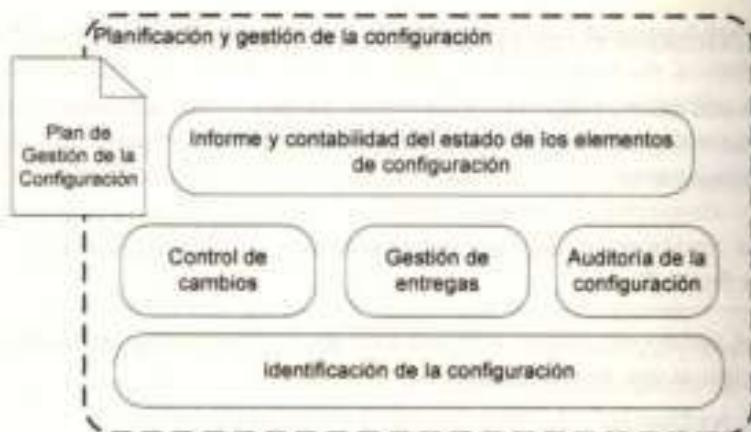


Figura 11.1: Proceso típico de gestión de cambios

los hará, y cómo esos cambios se controlan dentro del desarrollo. La *gestión de entregas* se ocupa de manera específica de las configuraciones estables del producto que se entregarán a los clientes, o servirán como punto de referencia en el desarrollo. Las tres actividades juntas, identificación, control de cambios y gestión de entregas, cubren los procesos técnicos y organizativos de la evolución de los artefactos que conforman el software, y por su importancia se tratarán individualmente a continuación.

Además de las actividades anteriores, la actividad de *auditoría de la configuración* cumple la función de garantía de calidad dentro de esta área de la Ingeniería del Software, y las actividades de *informe y contabilidad del estado de la configuración* cumplen la función de informar a la dirección y al equipo de desarrollo del estado de los elementos, permitiendo la medición y el control del progreso. Estas dos actividades relacionadas con el seguimiento y evaluación general del proceso de configuración se tratarán más adelante.

La Figura 11.2 muestra –en forma de diagrama de transición de estados– un resumen conceptual del ciclo de vida de un artefacto de ingeniería. Distingue en primer lugar la vida previa del artefacto antes de ser identificado, en la cual los cambios suceden sin ningún tipo de control. Pero una vez se identifica el elemento, los cambios siempre tendrán lugar bajo el marco del control de versiones. Además, siempre que un elemento haya entrado en una línea base, el proceso de cambios requiere de aprobación a la entrada y a la salida. Aunque se trata de una simplificación con carácter ilustrativo, refleja los elementos esenciales que discutiremos a continuación.

11.4.1 Identificación de la configuración del software

El primer problema que debe considerarse en la gestión de la configuración del software es cómo **identificar** únicamente a cada uno de los *elementos software*. Para ello, es primordial comenzar por decidir lo siguiente:



Figura 11.2: El ciclo de vida de un artefacto en relación con la gestión de la configuración

- Qué elementos software se controlarán. El código fuente es el más habitual, pero también pueden incluirse los planes, los documentos de diseño, el código ejecutable o los archivos de configuración, entre otros.
- Qué relaciones o dependencias entre elementos software se controlarán, que en el caso del código afectan, incluso, al proceso de compilación. Es importante definir qué tipo de relaciones se van a examinar y proveer los mecanismos para ello.

Lo habitual es incluir como elementos software los documentos formales del desarrollo, el código fuente y su documentación –en sentido amplio-. En ocasiones sólo se aplica gestión de la configuración al código fuente, por ejemplo, en casos en los que el desarrollo es corto y es poco probable que los documentos formales cambien con frecuencia.

El concepto de versión surge porque los elementos software evolucionan en el tiempo:

Una **versión** de un elemento software es un *ítem* particular e identificado, que puede verse como un estado en la evolución de un elemento software. Cuando la versión está pensada para reemplazar al estado anterior, se suele denominar **revisión**, mientras que si es una versión nueva que no reemplaza a la anterior, se suele denominar **variante**.

Desde el punto de vista de la gestión, hay cambios que pueden realizarse a ciertos elementos software sin un proceso de gestión del cambio formal. Por ejemplo, cuando un grupo de programadores está trabajando en un subsistema, los programadores introducen cambios de manera ordenada y planificada, pero sin requerir autorización o un proceso de evaluación de los cambios. Hacerlo de otra manera sería colapsar el proceso de desarrollo. No obstante, hay otros momentos en los que cambiar un software requiere de ese tipo de procesos formales de gestión del cambio. Por ejemplo, cuando se ha entregado un software a un cliente.

Las configuraciones fijas de un software que requieren de procesos formales para cambiarse se denominan *líneas base*. Una definición más formal es la siguiente:

Una **línea base** es una especificación o producto que ha sido revisado y consensuado formalmente, que servirá de base para futuros desarrollos, y que sólo puede cambiarse mediante procedimientos de control de cambios formales

El aspecto fundamental de las líneas base es que antes de entrar en alguna, el cambio de un elemento software puede ser rápido e informal, pero cuando entra en la línea base, los cambios requieren un control formal con procedimientos definidos. Estas líneas base sirven para controlar determinados puntos en el desarrollo, tales como los siguientes:

- La línea base funcional, con los requisitos revisados. Esta línea base contiene la primera versión estable de la funcionalidad requerida y por tanto, puede utilizarse como base para la estimación del esfuerzo de desarrollo mediante métodos que miden función, como las técnicas de conteo de *puntos de función*.
- La línea base asignada, con los requisitos revisados y las especificaciones de interfaces. Esta segunda línea base contiene ya un primer diseño de alto nivel, incluyendo interfaces entre partes del software que son relevantes para la configuración.
- Las líneas base de desarrollo son determinados estados de la configuración durante el proceso de desarrollo. Podría ser una línea base conteniendo un prototipo arquitectónico, o una primera versión funcional, pero no entregable, del sistema.
- La línea base de producto se corresponde con la configuración del producto que se ha entregado al cliente.

Es importante resaltar que una vez se ha establecido una línea base, la entrada de un elemento en la misma tiene lugar siempre después de algún evento de aceptación formal. En el caso del código puede ser una revisión formal o una fase de prueba, y en el caso de un documento, puede ser algún tipo de revisión del mismo.

Un concepto asociado a la identificación es *Software Library*¹, lugar donde se centraliza el almacenamiento de todos los elementos de configuración que están controlados.

¹Dado que la traducción «biblioteca software» podía llevar a confusión, se ha mantenido el término original.

11.4.2 Control de los cambios en el software

El control de los cambios es esencial para la gestión de la configuración. Ese control consiste en la definición de los procesos de inclusión de cambios en una línea base. Esa inclusión tiene que estar necesariamente controlada, es decir, debe contar con un proceso en el que el cambio sea valorado, y se apruebe o rechace de acuerdo a los criterios de una cierta *autoridad de control de cambios*. Esta autoridad de control de cambios puede ser una sola persona –en proyectos pequeños, el gerente del proyecto o el arquitecto de software podrían desempeñar dicho papel–, si bien es más habitual que sea un grupo de personas representativas, lo que habitualmente se llama *panel de gestión de la configuración*.

El panel de gestión de la configuración es un comité que controla el proceso de cambio y ejerce de autoridad de control. Suele estar formado por representantes de todas las partes interesadas, incluyendo clientes, desarrolladores y usuarios

El *proceso de gestión de cambios* es un procedimiento establecido para la aprobación formal de cambios relativos a una cierta línea base. La Figura 11.3 muestra un esquema genérico de un proceso de cambios típico.

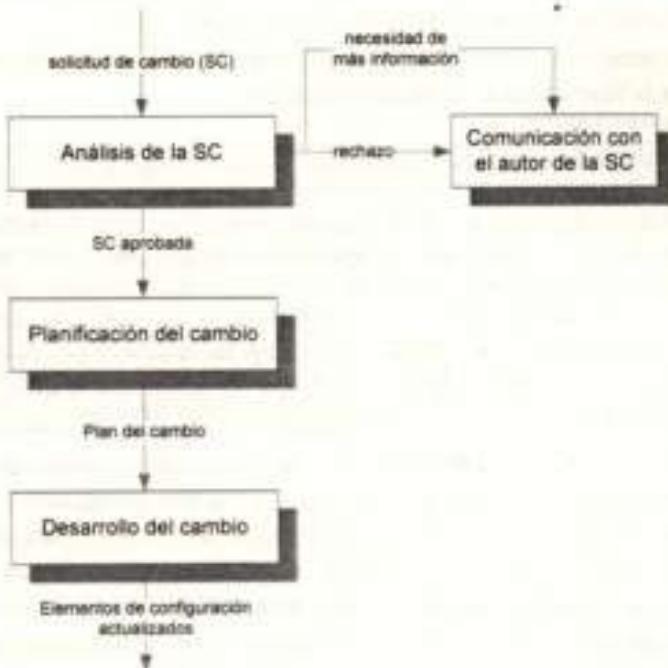


Figura 11.3: Proceso típico de gestión de cambios

El proceso comienza siempre con una *solicitud de cambio*. El cambio puede originarse porque el cliente ha encontrado un fallo en un software entregado, pero también puede provenir del equipo de desarrollo o de otros actores, debido a necesidades diversas. La solicitud de cambio llega a la autoridad de control de cambios competente, y ésta comienza un proceso de *análisis de la solicitud de cambio*. El análisis debe terminar con una aprobación o un rechazo formal y documentado de la solicitud de cambio, pero puede constar de varias iteraciones, en cada una de las cuales la autoridad de control de cambios solicita información adicional al autor.

La autoridad de control de cambios puede también requerir información de otras fuentes. Como los cambios implican un gasto de esfuerzo de recursos humanos y de recursos de desarrollo, una fuente de información importante es el plan del proyecto y la correspondiente unidad de gestión financiera y/o de recursos humanos.

La solicitud de cambio eventualmente se aprobará. En caso de que eso suceda, lo que se desencadena a continuación es un ciclo de Ingeniería del Software completo. En la Figura 11.3 estas actividades se dividen en una primera fase de planificación, y después, el desarrollo del cambio, que incluirá las actividades de Ingeniería del Software necesarias, incluyendo, en todo caso, las pruebas, siempre que se haya cambiado el código fuente o algún elemento que afecte a la creación del programa ejecutable. Como el cambio o los cambios introducidos pueden tener efectos no esperados y ocultos en el funcionamiento de otras partes del software, se deben repetir todas las pruebas del mismo para detectar posibles fallos laterales. A esto se le denomina *prueba de regresión*.

Una vez que termina el desarrollo, la configuración del software ha cambiado, y así debe reflejarse en la base de datos de la configuración.

Ejemplo de solicitud de cambio

Una solicitud de cambio es un documento que contiene una solicitud de modificación de un sistema. Estas solicitudes son de carácter declarativo, es decir, indican lo que hay que cambiar, pero no cómo debe hacerse el cambio. Los siguientes son elementos típicos de una solicitud de cambio: un identificador de la solicitud de cambio; la referencia o identificador de la persona o unidad que ha originado la solicitud; una indicación de la optionalidad o importancia del cambio propuesto; los plazos, en su caso; el tipo de cambio y una descripción textual del cambio a realizar.

La Figura 11.4 muestra un ejemplo de solicitud de cambios típica. En ella podemos apreciar la identificación, el origen de la solicitud y la manera de codificar el estado y en su caso, algunos datos de la planificación de la realización del cambio. Este tipo de formularios deberían ser –siempre que esto sea posible– electrónicos, de modo que las solicitudes quedan registradas junto al resto de la información de contabilidad de la configuración.

El proceso de gestión de cambios descrito requiere una evaluación de cada solicitud de cambio. No obstante, durante el desarrollo del software, antes de la entrega o de la creación de una línea base estable, los cambios o adiciones al software también deben aprobarse de manera formal. Pero en este caso, se aprueba el cambio final, ya que la necesidad del cambio

Solicitud de Cambio Software (SCS)	# requisito _____	# SCS _____
Origen: _____	Fecha: _____	# entrega _____
Tipo:		
<input type="checkbox"/> Nuevo requisito <input type="checkbox"/> Cambio en requisito <input type="checkbox"/> Cambio en el diseño <input type="checkbox"/> Problema con el software <input type="checkbox"/> Problema de interfaz de usuario <input type="checkbox"/> Error en la documentación <input type="checkbox"/> Sugerencia de mejora <input type="checkbox"/> Otra: _____		
Prioridad:		
<input type="checkbox"/> Alta <input type="checkbox"/> Media <input type="checkbox"/> Baja		
Descripción:		
Adjúntese toda la documentación adicional necesaria.		
Estado		
Revisado & Estimado / En Espera / Cancelado / Aprobado / Terminado		
Fecha de aprobación		
Comentarios		
# Nueva versión _____		
Adjunte los documentos necesarios del análisis, estimación y planificación del cambio.		

Figura 11.4: Solicitud de cambios típica

está derivada de los requisitos y el plan del proyecto, y por tanto, no es necesario que haya solicitudes de cambio. La Figura 11.5 muestra una secuencia de actividades típicas de un programador o varios programadores durante el desarrollo, y cómo la autoridad de control de cambios tiene un rol de aprobación final del producto.

Es importante resaltar que la aprobación de la autoridad de control de cambios no es el único evento que genera *control de versiones*. Típicamente, los desarrolladores utilizan el control de versiones durante su tarea de codificación y la prueba unitaria, sin necesidad de pasar por el proceso de aprobación formal, que sólo se da cuando el elemento software desarrollado ya ha pasado la prueba unitaria y la eventual revisión técnica. Finalmente, debemos enfatizar que el control de versiones es una actividad que está inserta como una parte en un flujo de trabajo de gestión de cambios más amplio.



Figura 11.5: Actividades típicas de desarrollo y aprobación

Un ejemplo más complejo de gestión de cambios

El establecimiento de un proceso de gestión de cambios es un elemento crítico en la gestión de la configuración del software, ya que de algún modo sintetiza las prioridades de la organización y por ello, debe dar cabida a la participación de todos los responsables.

Como ejemplo, la Figura 11.6 resume los actores implicados en el proceso de gestión de cambios en una institución con un esquema más complejo. Allí podemos apreciar, en primer lugar, una división de la función en un Comité de Control, que es el que toma las decisiones, y un equipo de revisión, que adopta funciones de apoyo al anterior. Es importante destacar, además, cómo en este esquema se le otorga un papel a los comités de control de otros proyectos, y también (aunque de manera más general), un papel de control a la alta gestión o dirección de la organización.

11.4.3 Gestión de entregas

El término **entrega** (*release*) hace referencia a un estado de la configuración que se utiliza para distribuirla, fuera del contexto del desarrollo. Esta distribución puede ser a un cliente del software, o también una entrega interna en algún punto determinado. Las entregas a las que todos estamos acostumbrados son las versiones de los productos, por ejemplo la versión 4 de Mozilla Firefox. Pero dentro de un proyecto, hay entregas parciales, por ejemplo, la creación de una línea base a partir de un prototipo arquitectónico del sistema.

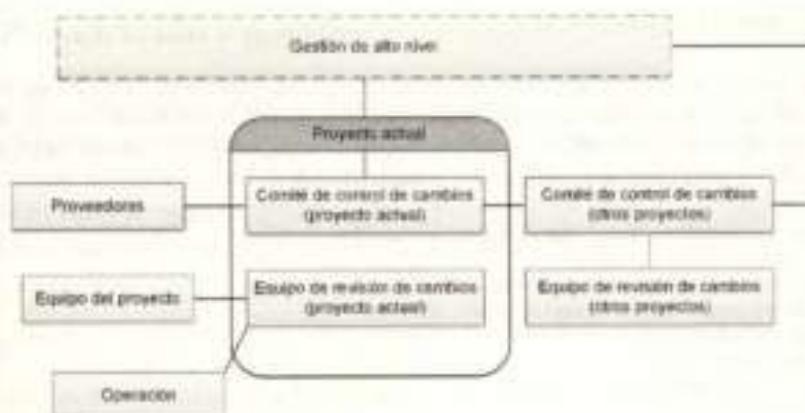


Figura 11.6: European Association for Railway Interoperability: Gestión de cambios

Las entregas requieren de un proceso denominado creación del software (*software building*), que consiste en la combinación de los componentes necesarios en el orden apropiado para obtener la versión ejecutable del software. Es habitual que el proceso de creación se realice mediante una aplicación software especializada, como los instaladores automáticos o los archivos de comandos –que automatizan el proceso de combinación de diferentes elementos en un software ejecutable-. Los instaladores automáticos, por ejemplo, permiten instalar una aplicación de manera interactiva, incluyendo las versiones correctas de los elementos de configuración, los archivos ejecutables, las bases de datos necesarias, etc., teniendo en cuenta distintas configuraciones según el sistema operativo.

Pueden identificarse al menos los siguientes tipos de entregas (*releases*):

- Entregas operativas. Son las entregas que están siendo utilizadas por clientes usuarios, y de las cuales se hace una gestión de cambios activa. Estos cambios, si responden a defectos encontrados, se suelen actualizar mediante parches (*patches*) o versiones que incluyen resolución de problemas (*bug fixes*). Cuando se decide no mantener una entrega, se considera como *fuerza de soporte*.
- Entregas de mantenimiento. Es una nueva entrega orientada a reparar errores, consistente en la suma a una entrega operativa de una colección de resoluciones o parches.
- Entregas en planificación. Es una entrega para la cual ya se han planificado o incluso, se han comenzado a desarrollar incrementos funcionales o de atributos de calidad del software (no simples parches).

La aprobación de qué es una entrega y qué no lo es, en ocasiones, se encuentra definida contractualmente, pero lo habitual es que la autoridad de control de cambios disponga de un procedimiento para la aprobación y control de nuevas entregas, que pasarán a planificarse.

La herramienta de creación ant

Apache Ant es una herramienta de creación de programas basada en Java, donde las instrucciones para construir una determinada entrega o versión de un software se especifican en un archivo XML. El siguiente ejemplo –adaptado de la documentación de Ant– sirve para entender sus conceptos fundamentales.

```
<project name="MiProyecto" default="dist" basedir=".">
    <!-- Propiedades globales -->
    <property name="src" location="src"/>
    <property name="build" location="build"/>
    <property name="dist" location="dist"/>
    <target name="init">
        <!-- Crear una marca de tiempo -->
        <tstamp/>
        <!-- Crear el directorio donde se generará el bytecode -->
        <mkdir dir="${build}"/>
    </target>

    <target name="compile" depends="init" description="Compilar fuentes">
        <!-- Compilar el código en ${src} y colocar bytecode en ${build} -->
        <javac srcdir="${src}" destdir="${build}"/>
    </target>
    <target name="dist" depends="compile" description="Generar entrega">
        <!-- Create the distribution directory -->
        <mkdir dir="${dist}/lib"/>
        <!-- Colocar todo en el directorio
            ${build} en el archivo MiProy-${DSTAMP}.jar -->
        <jar jarfile="${dist}/lib/MiProy-${DSTAMP}.jar" basedir="${build}"/>
    </target>
    <target name="clean" description="Borrar todo">
        <!-- Borrar los directorios ${build} y ${dist} -->
        <delete dir="${build}"/>
        <delete dir="${dist}"/>
    </target>
</project>
```

La organización general de un archivo ant se estructura en diferentes objetivos (**target**). Los objetivos típicos son la inicialización (preparar directorios o copiar archivos necesarios para el resto del proceso de creación), el borrado de los elementos generados, la compilación y la creación de distribuciones, entre otros. Cada objetivo puede tener dependencias con otros objetivos. Por ejemplo, hay que compilar antes de generar el archivo con la distribución final. En nuestro ejemplo, el target de nombre **dist** requiere (**depends**) que se haya ejecutado correctamente el target denominado **compile**.

Un archivo ant puede contener también propiedades globales, que se utilizan como variables en el resto del archivo. Dentro de cada objetivo se pueden utilizar comandos diferentes, incluyendo manipulación del sistema de archivos (como **mkdir** o **delete**), o bien invocaciones a programas externos que participan en la construcción (como **javac**, el compilador de Java o **jar**, la herramienta de creación de paquetes).

11.5 Planificación y gestión

La gestión de la configuración del software requiere una planificación y una organización al principio del proyecto, adaptada al contexto de la organización. El documento que recoge la planificación se denomina *plan de gestión de la configuración*. El estándar IEEE 828-2005 (IEEE, 2005) proporciona una guía para los contenidos del plan de gestión de la configuración. A continuación se muestra el ejemplo de contenidos del estándar:

1. *Introducción.*

2. *Gestión de la configuración del software:*

- *Organización.*
- *Responsabilidades en la configuración.*
- *Políticas, directivas y procedimientos aplicables.*

3. *Actividades de gestión de configuración del software:*

- *Identificación de la configuración.*
- *Control de la configuración.*
- *Contabilidad del estado de la configuración.*
- *Auditorías y revisiones de la configuración.*
- *Control de las interfaces.*
- *Control de los subcontratistas/proveedores.*

4. *Calendarios de gestión de configuración del software.*

5. *Recursos de gestión de configuración del software.*

6. *Mantenimiento del plan de gestión de la configuración del software.*

El plan de gestión de la configuración tiene, en primer lugar, que especificar las responsabilidades de la gestión de configuración, y los procedimientos documentados que deben seguirse, incluyendo, en su caso, el uso de herramientas específicas. Las actividades incluidas en el plan cubren las que ya se han visto en este capítulo, pero también pueden incluir aspectos más específicos, como el control de la configuración del software que proviene de terceros, sean subcontratistas o proveedores externos de software.

La justificación económica y la asignación de recursos para la configuración es a menudo problemática, dado que es un aspecto de soporte, transparente al desarrollo. No obstante, es un elemento clave para un proceso de ingeniería sólido; por ejemplo, es una de las áreas clave de proceso necesarias para pasar al Nivel 2 (*repetible*) en el modelo de madurez CMM.

11.5.1 Contabilidad y medición en gestión de la configuración

La **contabilidad de la configuración** tiene como objeto proporcionar información precisa sobre los elementos de configuración y la historia de sus cambios para diferentes propósitos de gestión. Esta información debe actualizarse continuamente y estar disponible desde el establecimiento de la primera línea base, que suele ser la de los requisitos funcionales.

El conjunto de informaciones mínimas que se debe registrar es el siguiente:

- La identificación de cada elemento de configuración.
- El estado de todas las solicitudes de cambio.
- El estado de la implementación de los cambios que han sido aprobados, incluyendo no sólo los elementos técnicos, sino los de carácter contractual.
- El resultado y el estado de todas las auditorías de configuración.
- La historia completa de todas las versiones de cada elemento de configuración.

Además del mantenimiento de la información actualizada, se debe diseñar una política de información con, al menos, dos elementos:

- La forma en que se informará. Básicamente, hay dos aproximaciones: notificación (envío selectivo de información en el momento en que se produce una actualización) o disponibilidad (la información está disponible, pero son sus usuarios los que tienen que acceder a ella).
- Las políticas de acceso, es decir, qué roles profesionales pueden acceder a qué tipos de información de configuración, y la política de registro de los accesos, en caso de que se considere necesaria.

Gracias al sistema de información que requiere la contabilidad de la configuración, se pueden obtener mediciones sobre diferentes aspectos, entre ellas:

- Uso de las herramientas de gestión de la configuración. En este caso, la utilidad de las mediciones está en analizar el rendimiento y dimensionamiento del sistema de soporte a la configuración. Las mediciones típicas afectan al número de transacciones con este sistema. En ocasiones, la actualización de las versiones implica muchos elementos (archivos) y podría darse una degradación del rendimiento.
- Resolución de problemas. El sistema de gestión de la configuración permite obtener mediciones de las tasas de defectos encontrados y la velocidad de desarrollo, información crítica para la planificación de las actividades de mantenimiento.
- Gestión del cambio. Por ejemplo, las mediciones de partes del código implicadas en más solicitudes de cambio pueden indicar algún tipo de problema funcional.

El sistema de gestión de configuración es, por otro lado, la fuente de datos fundamental para las mediciones y el cálculo de métricas del resto de las actividades de Ingeniería del Software. Por ejemplo, las métricas de diseño que se centran en aspectos de la estructura del software deben tomarse de la línea base, y puede estudiarse cuantitativamente la evolución de ese tipo de métricas gracias al registro histórico completo que proporciona la contabilidad de la configuración.

11.5.2 Auditoría de la configuración software

La **auditoría de la configuración** tiene como objetivo principal la evaluación de la integridad de cada elemento de configuración antes de que éste entre en la línea base. También hay procesos de auditoría relacionados con la configuración que atañen a los procedimientos y la documentación de la misma, pero aquí nos centraremos en los aspectos específicos de la integridad de la configuración.

En esencia, la auditoría de los elementos de configuración debe comprobar que se han cumplido los requisitos (funcionales y no funcionales) y que el diseño de los elementos está documentado de manera apropiada. Este tipo de auditoría, en ocasiones, se divide en dos actividades diferenciadas: auditorías de configuración funcionales y auditorias de configuración físicas.

Las *auditorías de configuración funcionales* consisten en la comparación sistemática de los requisitos especificados con los resultados de las pruebas, las inspecciones o los análisis. Representa un paso previo a la consolidación de la línea base. Las *auditorías de configuración físicas* tienen como objetivo verificar que el software es consistente con sus especificaciones de diseño, y que la realización de ese diseño está documentada de manera apropiada. Este tipo de auditoría no puede aprobarse sin que se haya aprobado la auditoría funcional correspondiente. Algunos ejemplos de preguntas de comprobación (*checklist*) típicas para una auditoría de configuración física podrían ser: ¿Se ha encontrado en la línea base la especificación para cada elemento de configuración? La documentación del diseño para los elementos cambiados. ¿se ha encontrado actualizada? ¿Se ha realizado un análisis o una inspección sobre cada uno de los elementos cambiados?

11.6 Técnicas y herramientas para el control de versiones

Lo que popularmente se entiende como *versión* de un producto software (por ejemplo, la versión 9.0 del navegador Microsoft Internet Explorer) son realmente entregas (*releases*) de ese software. Toda entrega es una versión del software, pero también hay muchas versiones de carácter interno o intermedio que se producen en el proceso de desarrollo o evolución, pero que no se comercializan, entregan o distribuyen como nuevas versiones al exterior.

En cualquier caso, las versiones de un producto software no son otra cosa que la combinación concreta de ciertas versiones de sus componentes (incluyendo código fuente y/o objeto, documentación y cualquier otro artefacto de Ingeniería del Software). Para gestionar

las versiones de los componentes existen un gran número de herramientas comerciales y abiertas. En todas ellas existe un *repositorio de versiones*, donde se guardan las diferentes versiones de cada elemento. En el resto de esta sección describimos los conceptos fundamentales del control de versiones y el ejemplo de la herramienta *subversión*.

Gestión de la configuración en métodos ágiles

Los procedimientos de gestión de configuración requieren un proceso de aprobación formal en el que una autoridad de control de cambios monitoriza los cambios. En el caso de los ciclos de vida en cascada, esta organización es adecuada, ya que hay una separación y secuencia temporal entre las actividades de requisitos y las actividades de diseño y desarrollo. No obstante, esos sistemas de control no se ajustan del todo bien a métodos incrementales que han aparecido posteriormente, tales como los métodos *ágiles*, si no se hace un énfasis en la rapidez en el análisis de los cambios. En esos métodos se planifican y desarrollan *incrementos*, es decir, subconjuntos de los requisitos de la aplicación. En lugar de hacer el desarrollo de todos los requisitos a la vez, se toman pequeños conjuntos de requisitos. Esto permite crear mucho más tempranamente entregas con partes de la funcionalidad, lo cual lógicamente multiplica el trabajo de gestión de configuración.

En el extremo de esta filosofía se hacen creaciones (*builds*) diarias del código, de modo que hay una presión en los desarrolladores por hacer cambios en unidades que proporcionen funcionalidad significativa, y que hayan pasado las pruebas unitarias de manera casi simultánea a la codificación.

Este tipo de aproximaciones a la configuración requieren de una notable agilidad en los procesos de flujo de trabajo para el control de las versiones y también de las revisiones y la generación de líneas base de desarrollo, si bien la filosofía y los conceptos son los mismos que en los métodos tradicionales.

11.6.1 Versiones, divisiones y deltas

Los sistemas de control de versiones permiten el almacenamiento y la compartición fiable de la historia de versiones de diferentes artefactos de desarrollo. Aunque se utilizan principalmente para el código fuente, pueden emplearse para otros tipos de archivos también.

Estos sistemas proporcionan identificación a las versiones sucesivas, generalmente mediante una identificación numérica. La Figura 11.7 muestra un ejemplo típico para un archivo, donde se muestra cómo comenzó con una primera versión 1.0 del código, que después evolucionó a una versión 1.1. Podemos suponer que el cambio no fue una modificación mayor, ya que se incrementó el número menor y no el mayor, pero esto es una convención que depende del equipo de desarrollo. En la figura se puede también apreciar cómo la versión 1.1, evolucionó en tres direcciones diferentes, incorporando cambios en diferentes aspectos. De hecho, una de las divisiones (o variantes) —la 1.1a concretamente— se integró después con la versión 2.0 de la línea principal.

En definitiva, la historia de versiones no es un esquema lineal, y los sistemas de control de versiones permiten reconstruir cualquiera de las versiones sucesivas que se han almac-

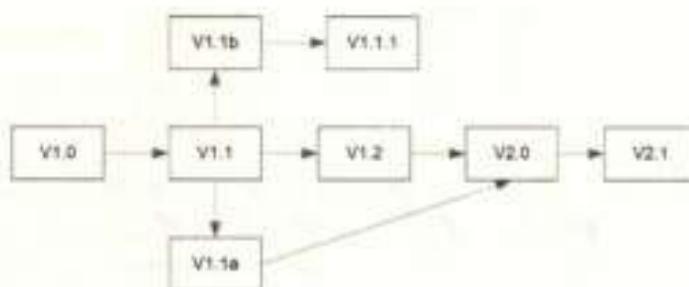


Figura 11.7: Ejemplo de historia de versiones para un archivo de código fuente

cenado en el repositorio de versiones. Estos repositorios no almacenan todas las versiones completas, sino solamente los cambios entre una versión y otra. Al conjunto de cambios necesarios para pasar de una versión a otra sucesiva se suele denominar *delta*.

11.6.2 Políticas de control de versiones en grupos de trabajo

Todos los sistemas de control de versiones tienen que resolver un problema fundamental: cómo permitir a los desarrolladores compartir información (archivos) y al mismo tiempo, impedir que por una actualización simultánea se pierda parte del trabajo.

Problemática de la actualización

La Figura 11.8 esquematiza el problema de la actualización, cuyas consecuencias es necesario evitar en el control de versiones. Inicialmente, dos programadores descargan una copia local del archivo F desde el repositorio. A continuación, ambos comienzan a trabajar en él, generando dos versiones locales: F' y F''. Cuando el primero actualiza el archivo en el repositorio, éste pasa a tener la versión modificada F'. Cuando el segundo programador realice la misma actualización, sobreescibirá los cambios del anterior a menos que se use un repositorio con soporte especial para el control de versiones. Si no es el caso, el trabajo contenido en F'' no se habrá actualizado en el repositorio; se habrá perdido.

La solución pesimista

Algunos sistemas utilizan el modelo *bloqueo-modificación-desbloqueo* para solventar el problema recién descrito. Según este modelo, sólo una persona puede modificar un archivo al mismo tiempo. Así, Luis debe bloquear primero el archivo antes de modificarlo y, durante la modificación de Luis, Ricardo tiene que esperar si quiere trabajar sobre el mismo archivo. El problema es que se bloquea el archivo entero, incluso aunque los cambios que quieran hacer los distintos programadores no sean incompatibles –podrían querer añadir métodos a una clase que no interfieran entre sí–. La Figura 11.9 ilustra esta solución, denominada *enfoque pesimista*.

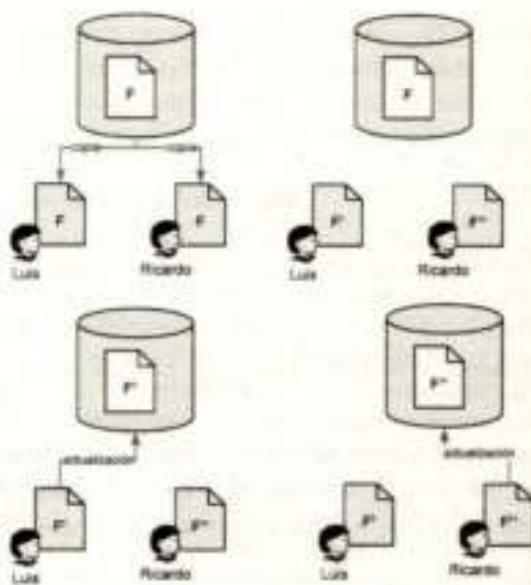


Figura 11.8: Ejemplo de problema de actualización del archivo F.java

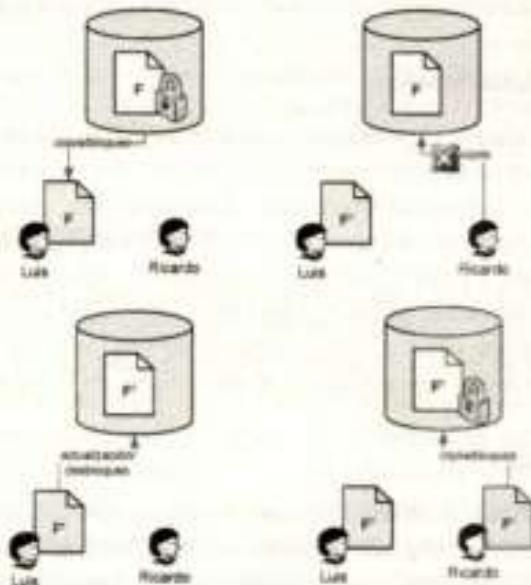


Figura 11.9: Bloqueo de archivos en un sistema de control de versiones

La solución optimista

Una alternativa para evitar que los bloqueos impidan el trabajo simultáneo es un modelo del tipo *copiar-modificar-mezclar*. Cada usuario se conecta al repositorio y crea una *copia de trabajo personal* –una réplica local de los archivos y directorios del repositorio– y comienza a trabajar en paralelo, modificando sus copias privadas. Finalmente, todas las copias privadas se combinan en una nueva versión final. El sistema de control de versiones a menudo proporciona alguna herramienta para facilitar la mezcla, pero en última instancia, es un ser humano el responsable de hacer que ésta se lleve a cabo correctamente. La Figura 11.10 presenta esquemáticamente un posible escenario de este tipo.

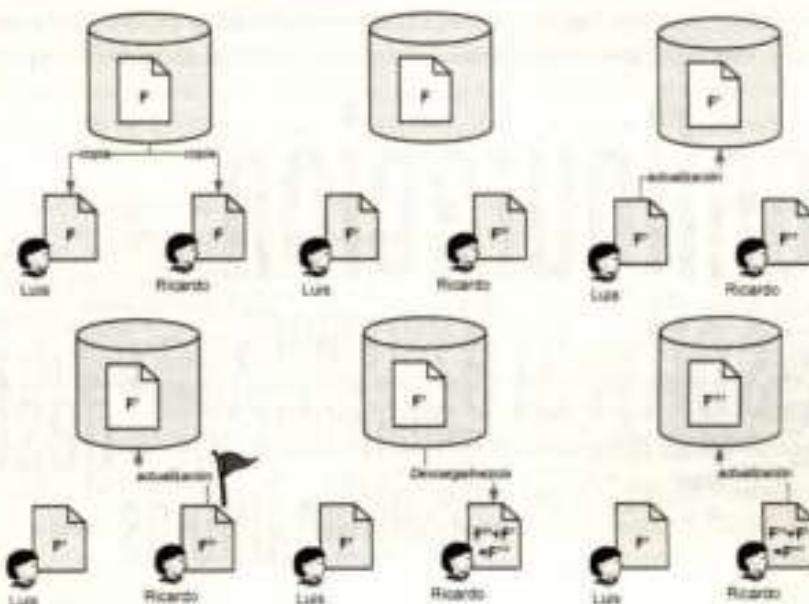


Figura 11.10: Mezcla de archivos en un sistema de control de versiones

Esta solución, conocida como *enfoque optimista*, permite que varios usuarios trabajen en copias locales. Una vez que uno de ellos actualiza la copia del repositorio, en todas las actualizaciones posteriores, en lugar de reemplazarse directamente la copia del repositorio (lo que nos llevaría al problema inicial), se produce una advertencia. El usuario que intenta actualizar cuando ya existe una (o más) actualizaciones anteriores, es advertido de que otro usuario ha actualizado la copia mientras él hacía modificaciones. El sistema de control de versiones permite, entonces, descargar la copia actualizada por el otro para que el segundo usuario pueda fusionar las dos versiones antes de hacer la actualización definitiva.

Algunas herramientas proporcionan medios para la mezcla automática, pero si los cambios han afectado a los mismos fragmentos de código, la herramienta no puede –en todos los

casos— hacer una mezcla con sentido. Es entonces responsabilidad del desarrollador hacer una mezcla «manual» que tenga en cuenta todos los cambios y no cree conflictos entre ellos.

Aunque el sistema de mezcla pueda parecer menos seguro a primera vista que el de bloqueo, en la práctica no hay tantos conflictos como podría suponerse, dado que los desarrolladores suelen tener partes de responsabilidad asignada en las que se solapan pocos archivos fuente, por eso resulta una alternativa interesante a los sistemas de bloqueo.

11.7 Resumen

Como en capítulos anteriores, la siguiente nube de palabras resume los principales conceptos del capítulo y su importancia relativa dentro del tema de la gestión de la configuración del software, una actividad –como hemos visto– de soporte al resto de las actividades.



Figura 11.11: Principales conceptos tratados en el capítulo.

En este capítulo hemos tratado las actividades fundamentales de la configuración: informe o contabilidad, control de cambios, identificación, auditoría y gestión de entregas. Todas ellas están sujetas a una actividad de planificación específica de la configuración.

Existen numerosas herramientas para la gestión de la configuración y entre ellas, hemos tratado una de las más habituales: las herramientas de control de versiones, que habitualmente se utilizan para la identificación y almacenamiento histórico del código fuente que se desarrolla. Al final del capítulo hemos analizado la problemática de la actualización de un repositorio de versiones en equipos de trabajo, donde varias personas acceden, descargan, crean copias locales, modifican y actualizan archivos del repositorio. En entornos de trabajo como éstos –el caso habitual en una compañía de desarrollo, por cierto– resulta esencial incorporar herramientas de control de versiones que permitan gestionar las actualizaciones con el menor número de errores y retrasos posible.

11.8 Notas biobiográficas

Una de las referencias clásicas en el área es el libro «*Implementing Configuration Management: Hardware, Software and Firmware*» (Buckley, 1995) aún actual a pesar del tiempo transcurrido tras su publicación. Otras referencias más actuales para profundizar en el estudio y la práctica de la gestión de la configuración del software son el manual de gestión de configuración «*Software Configuration Management Handbook*» (Leon, 2004), un volumen de amena lectura donde el lector encontrará una evaluación completa de la gestión de la configuración del software desde un punto de vista teórico –con explicaciones detalladas de las actividades– así como los procedimientos de aplicación en cualquier organización; la guía práctica «*Software Configuration Management Implementation Roadmap*» (Moreira, 2004), que propone un conjunto de tareas personalizables de fácil seguimiento para ayudar en la aplicación de la gestión de configuración del software. El libro describe un proceso paso a paso en la aplicación de las técnicas de gestión de la configuración, describiendo las actividades típicas a realizar en un proyecto.

Brown y sus colaboradores (1998) proporcionan una visión muy interesante de la gestión de la configuración a través del concepto de *anti-patrón*. Un anti-patrón es una mala práctica, es decir, un modo incorrecto de hacer alguna cosa, pero que se da con frecuencia en la práctica. Los anti-patrones que se describen en el libro son un buen complemento al material de este capítulo, ya que muestran de forma negativa los aspectos fundamentales de la gestión de configuración. Otra referencia muy recomendable es «*Software Configuration Management Patterns*» (Berczuk y Appleton, 2003), que incluye guías de buenas prácticas en la gestión de la configuración y para la producción de software de mayor calidad.

Existen varios estándares relacionados con la gestión de la configuración del software o relevantes para la misma. En cuanto a los publicados por IEEE, el estándar IEEE 1042-1987 «*Guía para la gestión de configuraciones del software*» es un estándar de carácter general que incluye los conceptos, actividades y herramientas, así como guías para la elaboración del plan de gestión de la configuración. Incluye guías específicas para la elaboración de estos planes en el caso de sistemas en tiempo real, sistemas experimentales y líneas de producto. También es relevante el estándar IEEE 828 «*Estándar para los planes de gestión de la configuración*» que describe los contenidos mínimos de uno de estos planes.

En cuanto a las especificaciones publicadas por otros organismos, la recomendación ISO/IEC TR 15846 «*Software Engineering-Software Life Cycle Process-Configuration Management for Software*» describe la gestión de la configuración del software en el marco de los demás procesos de ciclo de vida. También el documento ESA PSS-05-09 (*Guide to Software Configuration Management*) publicado por la Agencia Espacial Europea resulta una interesante guía para la adecuada gestión de la configuración en entornos de desarrollo.

Para quienes quieran conocer cómo se gestiona la configuración en otras disciplinas de ingeniería, resultarán de interés las monografías «*Practical CM III: Best Configuration Management Practices for the 21st Century*» (Lyon, 2008) y «*Engineering Documentation Control Handbook, 2nd Ed.: Configuration Management for Industry*» (Watts, 2001).

11.9 Cuestiones de autoevaluación

- 11.1 Se está desarrollando una aplicación para la empresa ACME. Un archivo con el logotipo de ACME, que se muestra en la aplicación, ¿puede considerarse un elemento de configuración?
- R *El logotipo puede y debe considerarse un elemento de configuración, ya que es un elemento más del software en desarrollo que puede tener diferentes versiones a lo largo del mismo.*
- 11.2 Indique la diferencia entre una versión y una entrega en el contexto de la gestión de la configuración del software.
- R *El concepto de versión está asociado a los diferentes estados de los elementos de configuración, sean éstos simples o agregados de elementos. Una entrega es, por supuesto, un estado determinado de esos elementos, pero lo que le da el carácter de entrega es la decisión de que se entregará a un cliente o usuario, o formará una entrega interna.*
- 11.3 Reflexionar sobre la verdad o falsedad de la siguiente afirmación: «En un control de versiones optimista, en el que *n* usuarios modifican simultáneamente el mismo archivo, una herramienta de mezcla (*merge*) automática siempre puede decidir sobre la versión correcta que incorpora todos los cambios».
- R *Las herramientas de mezcla no pueden resolver cualquier tipo de conflictos en los cambios de diferentes desarrolladores. Entre otras razones, por el simple motivo de que no son suficientemente inteligentes para decidir cuál de los cambios en conflicto es el correcto. Esto sólo lo pueden decidir los desarrolladores.*
- 11.4 ¿Cuál es la tarea fundamental de la autoridad de control de cambios?
- R *Su tarea fundamental es gestionar el proceso de análisis y ejecución de los cambios sobre las líneas base, y decidir qué cambios darán lugar a actividades de desarrollo y cuáles no.*
- 11.5 Reflexionar sobre la verdad o falsedad de la siguiente afirmación: «Sólo las solicitudes de cambio aceptadas deben registrarse en el sistema de contabilidad de la configuración».
- R *La afirmación es falsa, ya que se deben registrar todas las solicitudes de cambio, ya que proporcionan información valiosa para la gestión. Por ejemplo, es posible que una solicitud de cambio se deniegue por razón de que no hay recursos de desarrollo disponibles en un momento dado, pero quizás en el futuro se reconsideren esas peticiones como posibles ampliaciones funcionales.*
- 11.6 ¿Por qué una auditoría de configuración física requiere que la auditoría de configuración funcional haya sido aprobada?
- R *La auditoría de configuración funcional comprueba los elementos que contrastan el software con las especificaciones funcionales. Debido a que el diseño debe responder a las especificaciones funcionales, éste no puede considerarse válido si hay una discordancia previamente detectada de carácter funcional.*
- 11.7 En un control de versiones optimista, ¿cabe la posibilidad de que dos desarrolladores implementen una misma operación en una clase Java, por ejemplo, resultando en un conflicto?
- R *La posibilidad existe, ya que podrían estar trabajando en funcionalidades diferentes, pero que afectasen a clases comunes, y que ambos desarrolladores codificaran métodos similares en una clase que serían redundantes. Si la división del trabajo entre los desarrolladores es razonable, normalmente no se dará este tipo de casos.*

11.8 Un plan de gestión de la configuración ¿tiene una influencia en el coste de un proyecto?

R. *Por supuesto que la tiene, ya que los roles, procedimientos y herramientas que se especifican en ella representan elementos de coste: recursos humanos, recursos hardware y quizás licencias de software. Los requisitos de esfuerzo de lo especificado en el Plan influirán en el coste del proyecto, o en último caso, en los costes de operación de la organización de desarrollo.*

11.9 Si utilizamos una herramienta como ant para la construcción de un software en desarrollo, los scripts de ant ¿serían también elementos de configuración?

R. *Por supuesto que lo serían, ya que son parte integrante del software en su estado de entregable. Además, la configuración debe registrar sus versiones e incluso las dependencias con la versión concreta de la herramienta ant (y de las versiones de las herramientas utilizadas en la construcción) que se utilizan.*

11.10 La corrección de un defecto en el software, encontrado por el cliente tras la entrega ¿debe registrarse en la línea base?

R. *Es crítico que se registre, y de hecho, deberá pasar algún tipo de inspección o análisis antes de que los elementos de configuración que se han cambiado entren a formar parte de la línea base para no afectar a la estabilidad de la misma.*

11.10 Ejercicios y actividades propuestas

11.10.1 Ejercicios resueltos

El objetivo de los siguientes ejercicios resueltos es adquirir práctica con un sistema de control de versiones abierto, mediante ejemplos típicos de uso. Para ello, supondremos que tenemos una primera versión de una biblioteca de código Java sobre la cual irán trabajando tres desarrolladores en paralelo.

11.1: Políticas de gestión de la configuración del software

La empresa ACME SA, está especializada en el desarrollo con tecnologías de fuente abierto (*open source*) sobre la tecnología Java, que cuenta con 50 desarrolladores en plantilla y que desarrolla directamente sobre los requisitos expresados como casos de uso UML, y creando para cada nuevo proyecto un prototipo arquitectónico antes de proceder al desarrollo completo. El equipo de dirección de desarrollo ha examinado las prácticas actuales y ha encontrado que la práctica de la gestión de la configuración del software en la organización actualmente se lleva a cabo de forma autónoma por los propios programadores, que comparten el código de cada proyecto en un sistema de control de versiones CVS.

Usted está a cargo de comenzar a establecer una política más adecuada de gestión de la configuración del software. El primer paso es diseñar un procedimiento para la identificación de la configuración dentro de la empresa. Diseñe un breve documento con ese procedimiento, incluyendo los roles, las actividades y el esquema de nombrado de los elementos de configuración. Puede basarse en procedimientos que encuentre en la web que hayan utilizado otras organizaciones de características similares.

Solución propuesta: Obviamente, no existe una solución única a este ejercicio, y realmente tendríamos que conocer más sobre la empresa. No obstante, el siguiente esquema recoge los elementos fundamentales. Hay que tener en cuenta que el objetivo fundamental es la identificación sin ambigüedades de los elementos.

- Responsabilidades: se definen las dos siguientes responsabilidades que tienen una influencia en la identificación:
 - Gestor de la configuración software. Es el rol encargado de la supervisión de la identificación y del mantenimiento de los sistemas elegidos para la configuración.
 - Panel de gestión de la configuración, grupo que actúa como autoridad de configuración en la gestión de cambios. Su participación en la identificación es indirecta, ya que entre los aspectos que analizarán estará la propia identificación de los cambios.
- Actividades: para cada proyecto nuevo o cambio a un proyecto existente se tendrán como entrada los requisitos, y en caso de que sea un cambio a un producto ya existente, la línea base que se actualizará. Los pasos genéricos del procedimiento son los siguientes:
 - Seleccionar, identificar y clasificar los elementos que serán objeto de gestión de la configuración. Estos elementos incluirán todos los módulos independientes en la programación Java (clases, interfaces) y también los casos de uso UML que se utilizan.
 - Crear un esquema de identificación que refleje la estructura del producto.
 - Crear un esquema de etiquetado para cada uno de los elementos.
 - Determinar las líneas base. Un procedimiento simple podría ser el de tener una línea base de requisitos y otra para la primera entrega del código. No obstante, dependiendo del proyecto podría haber más líneas base para el código, por ejemplo, para acomodar la creación de los prototipos arquitectónicos.
 - Determinar las entregas y sus contenidos.
 - Establecer la *Software Library*.
- Un posible esquema de etiquetado es el siguiente:

`Proyecto.Entrega.Tipo.<parte dependiente del tipo>`

La combinación `Proyecto.Entrega` identifica las entregas (*releases*) para los usuarios con dos dígitos. Para los casos de uso tendremos una parte dependiente del tipo como la siguiente:

`Código.version.<atributos>`

El código seguirá la habitual convención de versiones mayores y menores separadas por punto. Los atributos incluirán al menos el responsable de la identificación del caso de uso, y la referencia al documento con la descripción del mismo. En artefactos de código tendremos una parte dependiente del tipo como la siguiente:

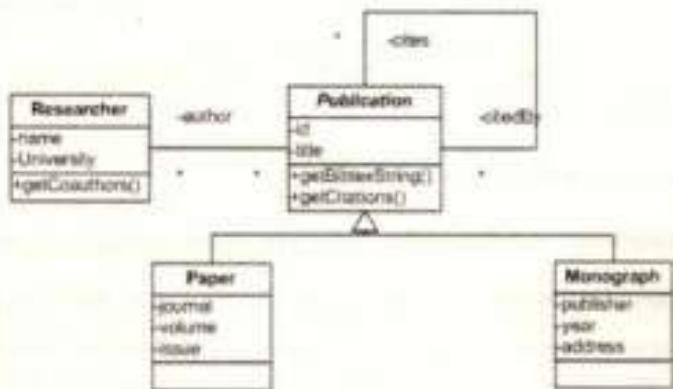
`Nombre_archivo.version.<atributos>`

El nombre de archivo seguirá las convenciones de Java en cuanto a nombres completamente cualificados y se establecerá una organización de paquetes según las convenciones habituales en Java. Los atributos deben incluir el responsable. Se ubicará todo el código Java en un sistema de control de versiones (de hecho, ya se estaba haciendo) de manera centralizada, y por tanto, la historia de cambios estará ya incluida en ese sistema.

11.2: Acceso a servidores Subversion

Subversion es un sistema de control de versiones de código fuente abierto, que se encarga de mantener un árbol histórico de archivos y directorios a lo largo del tiempo. En este ejercicio de carácter práctico y en los dos siguientes se trata de adquirir experiencia en el manejo de un sistema de control de versiones.

Suponga que está usted en un grupo de tres programadores desarrollando una pequeña aplicación que responde al diagrama de clases² mostrado. En el contexto descrito, se deben completar una serie de pasos descritos en [1], [2] y [3].



- 1 Instalar un servidor Subversion.
- 2 Crear un repositorio con `svnadmin create`. Decida sobre el tipo de tecnología de almacenamiento a utilizar ¿Berkeley DB o FSFS?
- 3 Incluir en el repositorio los contenidos del ejemplo con el comando `svn import`.

Solución propuesta: No hay en la descripción del contexto del problema razones sólidas para elegir una u otra tecnología. Utilizaremos FSFS por ser tener mejores características según la documentación del software. En la máquina en la que tenemos instalado el servidor, utilizaremos una sentencia como la siguiente:

```
svnadmin create --fs-type fsfs C:\svn\cites
```

²Para realizar este ejercicio debe descargar los archivos fuente proporcionados en la página web de material complementario. Lógicamente, el código de ejemplo es muy simple y sólo tiene propósitos didácticos.

```
C:\>svn import C:\backup\codigo file:///svn/cites -m "Importacion inicial"

Añadiendo C:\backup\codigo\Researcher.java
Añadiendo C:\backup\codigo\in.txt
Añadiendo C:\backup\codigo\Main.java
Añadiendo C:\backup\codigo\out.txt
Añadiendo C:\backup\codigo\Paper.java
Añadiendo C:\backup\codigo\Publication.java
Añadiendo C:\backup\codigo\Monograph.java
Añadiendo C:\backup\codigo\ScientificCitations.java

Commit de la revisión 1.
```

Si a continuación utilizamos el comando para inspeccionar el repositorio `svnlook`, tendremos la siguiente salida, donde se indica la fecha de la última actualización:

```
svn look info C:\svn\cites

Ricardo 2009-04-23 23:38:18 +0200 (lun, 23 abr 2009)
19
Importacion inicial
```

11.3: Actualización de código en la línea base

Lleve a cabo las siguientes acciones sobre el repositorio creado en el ejercicio anterior:

- 1 Crear dos copias locales para los desarrolladores *Pablo* y *Rafael*, respectivamente.
- 2 Pablo cambia en el método de conversión a cadena de `Publication` (método `toString()`), que retorna una cadena con el identificador de publicación, un separador y el título de la publicación) el separador «::» por un separador «->», y actualiza el repositorio.
- 3 Rafael cambia en su copia la misma clase, pero lo que hace es eliminar de esa conversión a cadena el uso del identificador de la publicación.

Solución propuesta: Para tomar la copia local [1] utilizaremos el comando `checkout`.

```
svn checkout file:///svn/cites

A cites\Researcher.java
A cites\Main.java
A cites\in.txt
A cites\out.txt
A cites\Paper.java
A cites\Publication.java
A cites\Monograph.java
A cites\ScientificCitations.java
```

Revisión obtenida: 1

Después se hará lo mismo desde el directorio del segundo desarrollador. Una vez que Pablo haya terminado, las buenas prácticas le llevarán a que compruebe qué ha cambiado con respecto al estado del repositorio.

```
svn status
M  Publication.java
```

Pablo ve de este modo que lo único cambiado es su copia local, por lo que es seguro hacer una actualización del repositorio.

```
svn commit -m "cambio en Publication:toString"
```

```
Enviando Publication.java
Transmitiendo contenido de archivos...
Commit de la revisión 2.
```

11.4: Fusión de cambios

Continuando con el ejemplo anterior, ahora Rafael comprueba los cambios de su copia local con respecto al repositorio (que ya tiene el cambio de Pablo), esta vez actualizando su copia local.

- 1 Indicar el comando para esa actualización.
- 2 ¿Se ha actualizado correctamente el repositorio?
- 3 En caso negativo, indicar las posibles soluciones.

Solución propuesta: Para realizar [1] Rafael utilizará:

```
svn update
C  Publication.java
Actualizado a la revisión 2.
```

La C indica que hay una situación de conflicto. De hecho, la actualización de Rafael tendrá el siguiente fragmento de código:

```
public String toString(){
    StringBuffer sb= new StringBuffer();
    <<<<< .mine
    append(title);
    *****
    sb.append(id).append("->").append(title);
    >>>>> .r2
    return sb.toString();
}
```

Donde se contrasta la versión local que existía (.mine) con la de la última versión del repositorio (.r2). Además de esto, Subversion coloca archivos con las versiones originales en conflicto. Aquí se pueden tomar varias acciones:

- Reemplazar el archivo por una de las copias de los archivos originales.
- En el propio archivo, resolver «a mano» el conflicto.
- Utilizar `svn revert` para descartar los cambios locales.

En nuestro caso, se puede corregir a mano quedando el siguiente código:

```
public String toString(){  
    StringBuffer sb= new StringBuffer();  
    "->".append(title);  
    return sb.toString();  
}
```

Habrá que informar a Subversión de que el conflicto está arreglado, utilizando para ello el comando `svn resolved Publication.java`. A partir de ese momento, ya se puede utilizar `svn commit` y se generará la tercera revisión del archivo.

11.10.2 Actividades propuestas

11.1 En la primera de las actividades propuestas vamos a ejercitarnos sobre control de versiones mediante la simulación de varias actividades de desarrollo de software, tal y como se llevarían a cabo en un proyecto real. Para ello, hemos de crear un repositorio-ejercicio en un servidor CVS vacío (en <http://www.nongnu.org/cvs/> pueden descargarse versiones del software) para, a continuación, proceder a la creación de los archivos originales:

1. Crear un módulo en el repositorio, siguiendo los pasos siguientes:
 - Crear un directorio `dir-create` y poner dentro de él un simple archivo de texto `README.TXT` conteniendo nuestro nombre.
 - Crear un módulo CVS con el comando apropiado.

El directorio estará ya identificado como repositorio CVS y creado en la carpeta del servidor CVS, y sin embargo, el archivo que habíamos creado dentro aún no está en el repositorio. Para añadirlo usaremos el comando `CVS add`. La adición del archivo no implica que sus contenidos se hayan guardado, pues para ello hace falta una operación de `Commit` –la verdadera operación de `check-in` donde se le asignará una versión al archivo–. En la carpeta del repositorio ya se podrá ver el archivo que va a contener las versiones (hasta aquí hemos creado un módulo e introducido un contenido en él).

2. Desde un directorio diferente al utilizado para albergar el repositorio ejercicio, realizar un `CVS checkout` del repositorio recién creado en dos directorios diferentes, que imaginaremos que son los directorios de dos programadores que trabajan en el mismo proyecto:
 - El primero será `dir - Luis`
 - El segundo será `dir - Nuria`
3. Actualizar la copia de Luis introduciendo una línea nueva y modificando alguna que ya estuviese. Nótese que Nuria no es consciente de este cambio, ya que es local a Luis.

En ambos casos, el mensaje CVS será `U dir-create/README.TXT`, indicando que ese archivo se actualiza localmente.

4. Ahora, Luis hará un CVS Commit, y se le asignará una nueva versión. La copia de Luis está actualizada, pero Nuria aún no puede darse cuenta.
5. Entonces Nuria quiere comprobar que su copia está actualizada para empezar a hacer cambios. Esta operación es un CVS Update. Como Nuria no había hecho cambios, simplemente el archivo se actualiza con los cambios de Luis. Al observar las propiedades del archivo de Nuria, veremos que la versión ha pasado a ser la 1.2, generada por Luis.
6. Ahora, Nuria añade otra línea y cambia una de las que había, y después realiza un commit, generando la versión 1.3.
7. Entonces, Luis, sin actualizar, realiza cambios locales añadiendo una línea a su archivo. Al intentar hacer un update, se detecta el conflicto.

Una vez llegado a la situación de conflicto, las herramientas de control de versiones pueden sugerir una mezcla (*merge*) y mostrarla al usuario que actualiza. Si la considera razonable, se guardará en el repositorio. En caso contrario, el usuario que actualiza debe decidir cómo afrontar la situación.

- 11.2-11.5** Vamos ahora a ejercitarnos en el control de cambios. *Aegis*³ es un software de fuente abierta que permite el control de cambios sobre un control de versiones. El concepto sobre el que se fundamenta es que la línea base siempre debe funcionar, entendiendo por *funcionar* que haya pasado todos los tests. Los sistemas de control de versiones como CVS permiten que un desarrollador que ha tomado una copia de trabajo de unos archivos, los actualice en el repositorio mediante una operación de check-in. El problema es que esta operación de actualización es *incondicional*, lo que puede resultar en que la línea base que se tenía deje de funcionar. La idea de *Aegis* es que no se pueda introducir nada en la línea base que no esté controlado, es decir, que no pueda construirse y funcionar por algún motivo (compilación, prueba, conflictos con otras actualizaciones, etc.).

La propuesta de esta actividad práctica se estructura en los siguientes puntos:

11.2 Instalar y configurar *Aegis*.

11.3 Crear un nuevo proyecto.

11.4 Incluir un archivo fuente en el proyecto (por ejemplo, un pequeño programa en Java) e intentar almacenarlo en el nuevo proyecto.

11.5 *Aegis* fuerza al desarrollador a dar una serie de pasos antes de conseguir llevar a buen término esa sencilla secuencia de pasos. ¿Cuáles son esos pasos?

- 11.6-7** Como ya hemos dicho en otras partes de este libro, la programación extrema (*Extreme Programming*, XP) es un método de desarrollo de software ágil que tiene una visión muy particular de la integración del código y la gestión de la configuración. La siguiente es un fragmento de una descripción de la práctica de integración secuencial tomada de las reglas de XP publicadas en la página web oficial (<http://www.extremeprogramming.org/rules/>):

[...] because of parallel integration of source code modules there is a combination of source code which has not been tested together before. Numerous integration problems arise without detection. Further problems arise when there is no clear cut latest version. [...] If you can not lay your hands on a complete, correct,

³<http://aegis.sourceforge.net/>

and consistent test suite you will be chasing bugs that do not exist and passing up bugs that do. Some projects try to have developers own specific classes. The class owners then ensure that code for each class is integrated and released properly. This reduces the problem but interclass dependencies can still be wrong. It does not solve the whole problem. Yet another way is to appoint an integrator or integration team. Integrating code from multiple developers is more than a single person can handle. And a team of people is too big a resource to integrate more than once a week. In this environment developers work with obsolete versions which are then erroneously re-integrated into the code base.

These solutions do not address the root problem. You want developers to be able to proceed in parallel, courageously making changes to any part of the system required, but you also want an error free integration of those efforts. Like a dozen steaming locomotives headed for the switch house all at the same time, there is going to be trouble. Instead of restricting development to being sequential, or requiring complex integration procedures let's rethink the problem. Our locomotives can all get into the switching house without a crash if they just take turns. We need to do this with code integration as well. Strictly sequential (or single threaded) integration by the developers themselves in combination with collective code ownership is a simple solution to this problem. All source code is released to the source code safe or repository by taking turns. That is, only one development pair integrates, tests and releases changes to the source code repository at any given moment. [...] This is not to imply that you can not integrate your own changes with the latest version at your own workstation any time you want. You just can't release your changes to the team without waiting for your turn. Some sort of lock mechanism is required. The simplest thing is a physical token passed from developer to developer. [...]

A la luz de este texto, responda a las siguientes cuestiones:

- 11.6 La aproximación de integración secuencial ¿cómo se compara al control de versiones optimista y pesimista?
- 11.7 ¿Entra esta práctica en conflicto con los principios y técnicas vistos en el capítulo?
- 11.8 Busque en la guía SWEBOK bibliografía sobre la gestión de la configuración del software y compruebe si se han publicado ediciones más recientes de las referencias allí incluidas.
- 11.9 En grupos, investiguen sobre herramientas que proporcionen soporte a las diferentes actividades relacionadas con la gestión de la configuración del software. Cada grupo elegirá una actividad y buscará herramientas relacionadas con la misma, dando preferencia a las de código fuente abierto. Se recomienda utilizar como documento guía la Sección «Selección e implementación de herramientas» del capítulo dedicado a la gestión de la configuración del software en la guía SWEBOK.
- 11.10 Acceda a un buscador de literatura científica –como Google Scholar o CiteSeerX– y recupere algún compendio reciente de publicaciones sobre gestión de la configuración del software.

12

Herramientas

Ni flaquearemos ni fallaremos; no cederemos ni nos cansaremos...

Dadnos las herramientas y nosotros terminaremos el trabajo.

— Winston Churchill

12.1 Las herramientas nos diferencian

La especie humana no está adecuadamente adaptada para vivir en el medio natural, pues sus defensas físicas son inferiores a las de la mayor parte de los animales. La piel del hombre no es tan buen aislante térmico como la de la foca polar o el reno. Sus extremidades no están tan adaptadas para la huida como las de la liebre, ni su cuerpo tan bien preparado para la defensa como el del armadillo o el del erizo. Con las manos desnudas apenas puede atacar a animales pequeños, y le es difícil defenderse de los ataques de otros animales. Ni siquiera le resulta fácil camuflarse para evitar la lucha, como hace el camaleón. Sin embargo, todas estas desventajas se compensan con un órgano insustituible: un cerebro grande y complejo.

Lo que permitió dominar la Tierra al hombre moderno, un ser tan deficientemente preparado para competir con otras especies por la supervivencia, fue su capacidad para aprovechar y transmitir a sus descendientes información gracias a su inteligencia, algo a lo que hemos llamado *cultura*. Hace unos 40.000 años, los seres humanos ya habían evolucionado lo suficiente como para ser capaces de razonar y ser conscientes tanto de su existencia como del entorno que los rodeaba. Fue posiblemente esa conciencia de inferioridad física lo que llevó a nuestros antepasados a crear herramientas, al principio muy primitivas y toscas, luego más y más sofisticadas.

Las herramientas son una de las características definitorias de la capacidad superior de los seres humanos. Si bien otras especies también hacen uso de algunas herramientas rudimentarias, han sido incapaces de razonar acerca de la necesidad de construir y perfeccionar

herramientas o de incorporarlas decididamente a todos los ámbitos de sus «culturas», si es que puede llamárseles así. En el caso de los seres humanos, su uso generalizado ha sido el factor determinante y diferenciador que permitió a la especie llegar a dominar el planeta.

A lo largo de la historia, los seres humanos han desarrollado herramientas diferentes para los fines más diversos, desde las primitivas herramientas de caza hasta el sofisticado instrumental de la telemedicina. En las primeras etapas de la evolución se desarrollaron exclusivamente herramientas mecánicas que hacían más fácil el trabajo físico: el hacha, la rueda y la palanca, por poner tres ejemplos, supusieron enormes avances. La revolución industrial añadió fuentes de energía artificiales, lo que supuso un salto cualitativo muy importante en la evolución humana. La electricidad primero y después la electrónica, que desembocaron en la revolución de las tecnologías de la información y las comunicaciones, han multiplicado las funcionalidades y aumentado, en varios órdenes de magnitud, la velocidad de las herramientas que el ser humano tiene a su disposición. Así, las computadoras actuales son capaces de realizar tareas a velocidades inimaginables anteriormente, y por supuesto, muy superiores a las que otras herramientas menos sofisticadas permiten.

En este libro se han descrito diferentes actividades y tareas a realizar como parte de las diferentes técnicas de Ingeniería del Software estudiadas. Muchas de esas tareas son repetitivas, o necesitan de una gran capacidad para recordar datos, informaciones diversas o detalles, o simplemente, se trata de aburridas (pero importantes) tareas de gestión a las que los ingenieros del software no les gusta dedicarse. Por supuesto, es posible llevar a cabo manualmente las actividades y no utilizar herramientas, pero esto seguramente obligaría a dedicar más recursos a cada actividad y, lo que es aún peor, introduciría incertidumbre sobre la exactitud con la que se está llevando a cabo cada tarea.

Este capítulo muestra cómo las herramientas –un tipo particular de ellas, las herramientas CASE–, facilitan las diferentes tareas a las que se enfrentan los gestores de proyectos, los ingenieros del software y los desarrolladores. Estudiaremos cómo las herramientas reducen su carga de trabajo, especialmente en aquellos aspectos más repetitivos y por ende, más sencillos de automatizar, permitiéndoles el centrarse en las facetas más puramente creativas del desarrollo. En definitiva, veremos cómo el uso de herramientas CASE ha sido determinante para mejorar la productividad de los equipos de desarrollo y la calidad del software.

12.2 Objetivos

El presente capítulo tiene como objetivo fundamental presentar el concepto de herramienta desde la perspectiva de la Ingeniería del Software y revisar las ventajas que su uso proporciona a los desarrolladores.

Más detalladamente, se pretende que el lector sea capaz de:

- Valorar el papel de las herramientas en el proceso de desarrollo de software.
- Clasificar e identificar los diferentes tipos de herramientas de Ingeniería del Software.

- Evaluar una herramienta según diferentes criterios.
- Seleccionar, entre las diferentes herramientas disponibles, la que se adapta mejor a unas necesidades específicas siguiendo para ello un proceso organizado.

12.3 Introducción

Desde los inicios de la programación de computadoras existió la necesidad de automatizar parte de las tareas a realizar. En un primer momento, el objetivo era simplemente facilitar la tarea al programador, por lo que las primeras herramientas fueron herramientas de programación: compiladores, intérpretes, editores, etc. Más tarde, fue necesario probar el software y se vio cómo muchas de las tareas relacionadas con las pruebas podían hacerse de manera más eficaz utilizando herramientas. A medida que el desarrollo de software se fue haciendo más complejo, comenzaron a aparecer herramientas para dar soporte a las nuevas actividades a llevar a cabo. Hoy en día, hay literalmente cientos de herramientas a disposición de los desarrolladores, algunas de las cuales ya se han tratado en la sección correspondiente.

En esta sección se van a definir los términos de los que se hace uso durante el resto del capítulo, así que comenzaremos por definir herramienta software:

Una herramienta software es un programa de computadora que ayuda a realizar un determinado proceso o lo automatiza completamente

Cuando las herramientas de software se aplican a la propia tarea de desarrollar software, se habla entonces de herramientas CASE. Este término, acrónimo del inglés *Computer Aided Software Engineering*, puede traducirse por Ingeniería del Software asistida por computadora.

Una herramienta CASE es una herramienta software que se utiliza en una o más fases del desarrollo de un producto software para apoyo de alguna tarea específica de Ingeniería del Software

Existen múltiples definiciones de herramienta CASE, pero todas ellas coinciden en sus objetivos: desarrollar, probar, analizar, diseñar o mantener un software o su documentación. Puede afirmarse que las herramientas CASE facilitan a los ingenieros del software, a los gestores de proyecto, y a los desarrolladores, en general, la realización de actividades tanto de gestión (control del proyecto, elaboración de informes, planificación de las pruebas, etc.), como técnicas (facilidades para construcción de interfaces, elaboración de prototipos, evaluación y seguimiento de los requisitos, construcción de un ejecutable listo para ser

utilizado, etc.) No es menos importante resaltar el hecho de que las herramientas CASE proporcionan una muy valiosa información acerca del software en desarrollo, y que sin ellas, sería muy difícil, si no imposible, recopilar, utilizar y mantener.

Rara vez se hace uso de una única herramienta CASE durante el desarrollo. Al contrario, suelen utilizarse diferentes herramientas, por lo que es importante la capacidad de integración de las mismas, esto es, de trabajar conjuntamente con el fin de completar trabajos que no podrían realizar individualmente. De este modo, se define el término *integración de una herramienta CASE* (I-CASE) como:

Se denomina **integración CASE (I-CASE)** a la capacidad que tiene una herramienta CASE para compartir datos y cooperar con otras herramientas

Finalmente, puede utilizarse el término *meta-herramienta* para designar herramientas que permiten crear otras, por ejemplo, un generador de analizadores lexicográficos (Lex, JLex, Flex, etc.), un generador de compiladores (tales como Yacc, CUP o javacc), u otros.

12.3.1 Justificación de las herramientas CASE

La necesidad de utilizar herramientas en la Ingeniería del Software no se discute actualmente. Sin embargo, las herramientas CASE son relativamente recientes, pues las primeras aparecieron en la década de los años 1980. Hoy en día, las herramientas CASE son una tecnología madura y ampliamente utilizada pero, ¿cuáles son las necesidades que tienen los desarrollos actuales que hacen necesarias las herramientas CASE? A continuación se enumeran algunas de las razones fundamentales para su utilización:

- La dificultad inherente al propio desarrollo de software, tal y como ya se estudió en el capítulo de introducción.
- La existencia de tareas tediosas, repetitivas o simplemente automatizables, que pueden facilitarse introduciendo un software que las mecanice parcial o completamente.
- El volumen de información que se genera a lo largo del desarrollo, algo difícilmente controlable y que no cabe en una, o en unas pocas, «mentes» de desarrolladores.
- La necesidad de una perspectiva global a la hora de tomar ciertas decisiones durante el desarrollo, algo difícil de obtener cuando los datos sobre los diferentes aspectos del desarrollo no se encuentran centralizados.
- La posibilidad real de trabajar sobre los mismos datos, incluso aunque los diferentes actores involucrados en el desarrollo (clientes, usuarios, desarrolladores, etc.), se encuentren en localizaciones físicamente dispersas.

La inversión en herramientas CASE es, salvo raras excepciones, beneficiosa para las organizaciones, pues conlleva un importante aumento de la productividad. Genéricamente, el aumento de la productividad se mide en términos de aumento del rendimiento originado por la variación de cualquiera de los factores que intervienen en la producción. En el caso de las herramientas CASE, la automatización de muchas tareas de diseño y desarrollo reduce considerablemente el esfuerzo en recursos humanos, lo cual hace que aumente el ratio entre el valor producido y los productos obtenidos, que se refleja en un beneficio directo en términos de productividad. Esta mejora de la productividad justifica, por sí misma, el empleo de herramientas CASE en una organización.

Veamos un ejemplo concreto. En la generación de escenarios de prueba, una tarea costosa en términos de esfuerzo, el uso de herramientas CASE que permitan generarlos semiautomáticamente a partir de la interfaz de usuario supone una importante mejora de la productividad, cercana al 50%. Así, en un sistema donde fueran necesarios –de media– 20 escenarios de prueba por cada diálogo de usuario, con la generación consiguiente de guiones automatizados para probar la interfaz, y asumiendo que el desarrollo manual de un escenario de prueba completo llevase unos 30 minutos, un ahorro del 50% implicaría miles de horas de trabajo, y por supuesto, la obtención de guiones con una mucho menor propensión a errores.

Otro ejemplo concreto es el que describen Chmura y Crockett (1995) en un interesante artículo publicado en la revista IEEE Software. En este trabajo se mide el éxito de la utilización de herramientas CASE por el número de sistemas complejos que una cierta agencia gubernamental norteamericana era capaz de producir. Antes de la introducción de herramientas CASE, el personal de la agencia era capaz de desarrollar un sistema complejo al año. Con el uso de herramientas CASE, el mismo equipo produce cuatro sistemas al año, que además tienen un mayor grado de funcionalidad y complejidad que los desarrollados en el pasado.

12.3.2 Ventajas e inconvenientes del uso de herramientas CASE

Los beneficios del empleo de herramientas CASE en el desarrollo de software son patentes. Algunos de los más importantes son:

- Aumento de la productividad, pues cuando se utilizan herramientas CASE son necesarios menos recursos para realizar el mismo trabajo.
- Software de mayor calidad, consecuencia directa del aumento de la fiabilidad.
- Posibilidad de elaborar informes u obtener datos del desarrollo que no sería posible realizar de otro modo.
- Facilita la aplicación sistemática de un proceso, así como la fidelidad a un estándar u otras restricciones, pues la herramienta implementa mecanismos de control que no permiten realizar acciones no autorizadas o fuera del estándar.