

INTRODUCCIÓN A LA INGENIERÍA DEL SOFTWARE

José A. Cerrada Somolinos
(Coordinador)



Editorial universitaria
Ramón Areces

UNED

INTRODUCCIÓN A LA INGENIERÍA DEL SOFTWARE



JOSÉ A. CERRADA SOMOLINOS
Catedrático de Lenguajes y Sistemas Informáticos
Escuela de Informática (UNED)

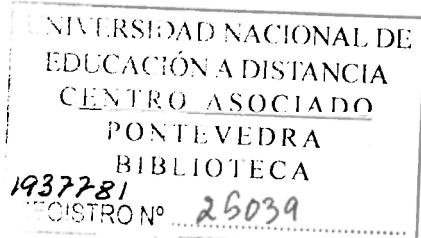
MANUEL E. COLLADO MACHUCA
Catedrático de Lenguajes y Sistemas Informáticos
Facultad de Informática (UPM)

SEBASTIÁN RUBÉN GÓMEZ PALOMO
Profesor Ayudante de Lenguajes y Sistemas Informáticos
Escuela de Informática (UNED)

JOSÉ FELIX ESTIVARIZ LÓPEZ
Profesor Ayudante de Lenguajes y Sistemas Informáticos
Escuela de Informática (UNED)

004.41 - - int

INTRODUCCIÓN A LA INGENIERÍA DEL SOFTWARE



EDITORIAL CENTRO DE ESTUDIOS RAMÓN ARECES, S.A.

Primera edición: junio 2000
Primera reimpresión: enero 2003
Segunda reimpresión: junio 2005
Tercera reimpresión: julio 2006

Reservados todos los derechos.

Ni la totalidad ni parte de este libro puede reproducirse o transmitirse por ningún procedimiento electrónico o mecánico, incluyendo fotocopia, grabación magnética, o cualquier almacenamiento de información y sistema de recuperación, sin permiso escrito de Editorial Centro de Estudios Ramón Areces, S. A.

© EDITORIAL CENTRO DE ESTUDIOS RAMÓN ARECES, S.A.

Tomás Bretón, 21 - 28045 Madrid
Teléfono: 915.398.659
Fax: 914.681.952
Correo: cerala@cerasa.es
Web: www.cerala.es

ISBN-10: 84-8004-417-9
ISBN-13: 978-84-8004-417-2
Depósito legal: SE-3148-2006 Unión Europea

Impreso por Publidisa

CONTENIDO

UNIDAD DIDÁCTICA I

Tema 1

Introducción

1.1 Concepto de Ingeniería de Sistemas	3
1.1.1 Concepto de sistema	3
1.1.2 Sistemas basados en computador	4
1.1.3 Componentes hardware, software y humanos	5
1.2 Características del software	5
1.3 Concepto de Ingeniería de Software	6
1.3.1 Perspectiva histórica	7
1.3.2 La crisis del software	8
1.3.3 Mitos del software	9
1.4 Formalización del proceso de desarrollo	10
1.4.1 El ciclo de vida del software. Modelos clásicos	10
El modelo en cascada	11
El modelo en V	14
1.5 Uso de prototipos	16
1.5.1 Prototipos rápidos	17
1.5.2 Prototipos evolutivos	19
1.5.3 Herramientas para realización de prototipos	20
1.6 El modelo en espiral	21
1.7 Combinación de modelos	23
1.8 Mantenimiento del software	23
1.8.1 Evolución de las aplicaciones	24
1.8.2 Gestión de cambios	25
1.8.3 Reingeniería	26

v1	Introducción a la Ingeniería de Software	
1.9	Garantía de calidad de software	26
1.9.1	Factores de calidad	27
1.9.2	Plan de garantía de calidad	28
1.9.3	Revisiones	29
1.9.4	Pruebas	30
1.9.5	Gestión de configuración	30
1.9.6	Normas y estándares	32
Tema 2		
Especificación de		
Software		35
2.1	Modelado de Sistemas	35
2.1.1	Concepto de modelo	36
2.1.2	Técnicas de modelado	37
	Descomposición. Modelo jerarquizado	37
	Aproximaciones sucesivas	38
	Empleo de diversas notaciones	39
	Considerar distintos puntos de vista	39
	Realizar un análisis del dominio	40
2.2	Análisis de requisitos de software	42
2.2.1	Objetivos del análisis	43
2.2.2	Tareas del análisis	46
2.3	Notaciones para la especificación	49
2.3.1	Lenguaje natural	50
2.3.2	Diagramas de flujo de datos	52
2.3.3	Diagramas de transición de estados	57
2.3.4	Descripciones funcionales. Pseudocódigo	60
2.3.5	Descripción de datos	63
2.3.6	Diagramas de modelo de datos	65
2.4	Documento de especificación de requisitos	68
2.5	Ejemplos de especificaciones	75
2.5.1	Videojuego de las minas	75
2.5.2	Sistema de gestión de biblioteca	83
UNIDAD DIDÁCTICA II		
Tema 3		
Fundamentos del Diseño de Software		103
3.1	Introducción	103

3.2 Conceptos de base	107
3.2.1 Abstracción	108
3.2.2 Modularidad	110
3.2.3 Refinamiento	111
3.2.4 Estructuras de datos	112
3.2.5 Ocultación	113
3.2.6 Genericidad	114
3.2.7 Herencia	116
3.2.8 Polimorfismo	118
3.2.9 Concurrencia	119
3.3 Notaciones para el diseño	121
3.3.1 Notaciones estructurales	122
Diagramas de estructura	124
Diagramas HIPO	126
Diagramas de Jackson	128
3.3.2 Notaciones estáticas	130
Diccionario de datos	130
Diagramas Entidad-Relación	130
3.3.3 Notaciones dinámicas	131
Diagramas de flujo de datos	131
Diagramas de transición de estados	131
Lenguaje de Descripción de Programas (PDL)	131
3.3.4 Notaciones híbridas	132
Diagramas de abstracciones	132
Diagramas de objetos	136
3.4 Documentos de diseño	140
3.4.1 Documento ADD	140
3.4.2 Documento DDD	144

Tema 4**Técnicas Generales de
Diseño de Software** 147

4.1 Descomposición Modular	148
4.1.1 Independencia funcional	150
Acoplamiento	150
Cohesión	154
4.1.2 Comprensibilidad	157
4.1.3 Adaptabilidad	158
4.2 Técnicas de diseño funcional descendente	160
4.2.1 Desarrollo por refinamiento progresivo	160

viii Introducción a la Ingeniería de Software

4.2.2 Programación estructurada de Jackson	162
4.2.3 Diseño estructurado	165
4.2.4 Ejemplo: Sistema de gestión de biblioteca	169
4.3 Técnicas de diseño basado en abstracciones	170
4.3.1 Descomposición modular basada en abstracciones	170
4.3.2 Método de Abbott	172
4.3.3 Ejemplo: Videojuego de las minas	175
4.4 Técnicas de diseño orientadas a objetos	177
4.4.1 Diseño orientado a objetos	178
4.4.2 Ejemplo: Estación meteorológica	180
4.5 Técnicas de diseño de datos	188
4.6 Diseño de bases de datos relacionales	189
4.6.1 Formas normales	189
4.6.2 Diseño de las entidades	191
4.6.3 Tratamiento de las relaciones de asociación	192
4.6.4 Tratamiento de las relaciones de composición	193
4.6.5 Tratamiento de la herencia	194
4.6.6 Diseño de índices	194
4.6.7 Ejemplo: Diseño de datos para la gestión de biblioteca	194
4.7 Diseño de bases de datos de objetos	197
4.8 Ejemplos de diseños	198
4.8.1 Videojuego de las minas	198
4.8.2 Ejemplo: Sistema de gestión de biblioteca	217

UNIDAD DIDÁCTICA III

Tema 5	
Codificación y	
Pruebas	233
5.1 Codificación del diseño	233
5.2 Lenguajes de programación	235
5.3 Desarrollo histórico	235
5.3.1 Primera generación	236
5.3.2 Segunda generación	237
5.3.3 Tercera generación	238

5.3.4 Cuarta generación	241
5.4 Prestaciones de los lenguajes	243
5.4.1 Estructuras de control	243
Programación estructurada	243
Manejo de excepciones	244
Concurrencia	248
5.4.2 Estructuras de datos	252
Datos simples	252
Datos compuestos	253
Constantes	254
Comprobación de tipos	255
5.4.3 Abstracciones y objetos	256
Abstracciones funcionales	257
Tipos abstractos de datos	257
Objetos	258
5.4.4 Modularidad	259
5.5 Criterios de selección del lenguaje	261
5.6 Aspectos metodológicos	263
5.6.1 Normas y estilo de codificación	264
5.6.2 Manejo de errores	265
5.6.3 Aspectos de eficiencia	268
5.6.4 Transportabilidad de software	270
5.7 Técnicas de prueba de unidades	272
5.7.1 Objetivos de las pruebas de software	272
5.7.2 Pruebas de caja negra	274
5.7.3 Pruebas de caja transparente	279
5.7.4 Estimación de errores no detectados	284
5.8 Estrategias de integración	285
5.8.1 Integración <i>Big Bang</i>	286
5.8.2 Integración descendente	286
5.8.3 Integración ascendente	287
5.9 Pruebas de sistema	289
5.9.1 Objetivos de las pruebas	289
5.9.2 Pruebas alfa y beta	290
Tema 6	
Automatización del Proceso	
de Desarrollo	
	293
6.1 Entornos de desarrollo software	293
6.2 Panorámica de las técnicas CASE	294

2	Introducción a la Ingeniería de Software	
6.2.1	Soporte de las fases de desarrollo	294
6.2.2	Formas de organización	295
6.2.3	Objetivo de un entorno de desarrollo	296
6.3	Una clasificación pragmática	297
6.3.1	Entornos asociados a un lenguaje	297
6.3.2	Entornos orientados a estructura	300
6.3.3	Entornos basados en herramientas	301
6.3.4	Entornos asociados a metodología	303
6.3.5	Entornos de 4 ^a generación	305
6.4	Una clasificación por niveles	305
6.5	Herramientas de software	307
6.5.1	Herramientas clásicas	307
6.5.2	Herramientas evolucionadas	308
6.5.3	Herramientas de 4 ^a generación	311
6.6	Entornos integrados	312
6.6.1	Integración de datos	312
6.6.2	Integración del control	313
6.6.3	Integración de la presentación	314
6.6.4	Integración del proceso	314
6.6.5	El repositorio CASE	315
6.7	Bancos o equipos de trabajo	316
6.8	Entornos orientados al proceso	319
	Bibliografía	323
	Glosario de Acrónimos	329
	Índice	331

Tema 1

Introducción

Este Tema se dedica a introducir una serie de conceptos básicos en el campo de la ingeniería de software, con objeto de disponer de una base inicial para las explicaciones que se irán dando en los restantes capítulos del libro. Aunque se presuponen en el lector algunos conocimientos de programación, se realiza un esfuerzo para facilitar la lectura por quienes no tengan conocimientos específicos en ese campo.

1.1 Concepto de Ingeniería de Sistemas

La ingeniería de sistemas es un marco general dentro del cual se pueden situar, con sus características propias, las disciplinas de ingeniería particulares. La ingeniería de sistemas los estudia desde un punto de vista general y global, prescindiendo en parte de su naturaleza concreta; por ello sus técnicas propias son distintas de las técnicas de cada ingeniería particular.

Las siguientes secciones analizan el concepto de sistema y las actividades de la ingeniería de sistemas. Por supuesto, el abordar la ingeniería de sistemas como parte de este libro tiene como finalidad establecer el marco general dentro del cual se habrán de estudiar como técnicas particulares de ingeniería las correspondientes a la concepción, desarrollo y explotación de sistemas informáticos. Por ello se estudian las características propias de estos sistemas, y la naturaleza de los elementos que los componen.

1.1.1 Concepto de sistema

Consultando el diccionario encontramos:

Sistema. (del latín *systēma*, y éste del griego *σύστεμα*) ... Conjunto de cosas que ordenadamente relacionadas entre sí contribuyen a determinado objeto ...

4 Introducción a la Ingeniería de Software

El concepto de sistema es fundamental en los campos científico y técnico. En ingeniería se atiende fundamentalmente a los sistemas artificiales, creados por el hombre. Estos sistemas, cada vez más complejos, han permitido ir dominando progresivamente nuestro entorno natural, que cada vez va siendo más adaptado a la comodidad humana, en detrimento de sus características naturales espontáneas.

La creación de sistemas complejos exige un trabajo colectivo, y por tanto una labor de organización, si se quiere realizar en forma eficaz. La *ingeniería de sistemas* atiende a los aspectos de organización de estos sistemas, tanto en lo referente al sistema en sí como al proceso de su desarrollo. Por ejemplo, para construir un puente los ingenieros de obras públicas deben elegir sus elementos componentes: pilares, plataforma, tirantes, etc., y también decidir en qué forma se irán construyendo y ensamblando dichas partes: primero la cimentación, luego los pilares, etc.

El concepto de sistema puede ser considerado en forma recursiva. Las partes de un sistema pueden, en ciertos casos, ser consideradas como sistemas más sencillos o *subsistemas*, compuestos a su vez, de partes más sencillas.

1.1.2 Sistemas basados en computador

Los sistemas que ha de concebir y construir el ingeniero en informática son sistemas basados en computadores. Estos sistemas contienen uno o más computadores dedicados a tareas de control del conjunto. Entre ellos podemos citar los satélites artificiales y naves espaciales, los modernos aviones, las fábricas de montaje automatizadas con robots, y cada vez con mayor frecuencia sistemas más cercanos a nuestra vida cotidiana tales como automóviles o aparatos electrodomésticos. Algunos de estos sistemas sólo pueden ser desarrollados por equipos multidisciplinares, a base de descomponer el sistema completo en subsistemas separados, desarrollados por especialistas en campos diferentes.

En lo que sigue atenderemos fundamentalmente a *sistemas informáticos* compuestos exclusivamente por computadores y sus elementos periféricos convencionales. En sistemas más complejos siempre podremos delimitar, como subsistemas, conjuntos de sus elementos puramente informáticos. Dentro de un sistema informático podremos distinguir los elementos materiales, que constituyen el *hardware* del sistema, de los programas que gobiernan el funcionamiento del computador, y que constituyen el *software* del mismo.

1.1.3 Componentes hardware, software y humanos

Los sistemas informáticos realizan tareas de tratamiento de la información. Estas tareas consisten fundamentalmente en almacenamiento, elaboración y presentación de datos.

Cuando se concibe un sistema informático para automatizar determinadas acciones se presentan varias posibilidades para la realización de cada operación concreta de tratamiento de información:

- 1 - La operación puede ser realizada directamente por algún elemento físico (hardware) del sistema.
- 2 - La operación puede ser programada, desarrollando el programa (software) apropiado.
- 3 - La operación se realiza manualmente por el usuario del sistema.

Estas tres opciones se derivan del reconocimiento de las diferentes clases de elementos que intervienen en un sistema informático utilizado por personas: los componentes hardware, los componentes software, y los usuarios humanos del sistema.

La última opción puede resultar algo chocante, ya que las personas no son realmente parte del sistema artificial que es objeto de la labor de ingeniería. Sin embargo, la visión de sistema requiere incluir la actividad humana de los usuarios, ya que en la concepción del sistema informático no sólo se decide el trabajo a realizar por el computador, sino también cómo ha de ser usado por las personas que operan dicho sistema.

La concepción de un sistema informático en forma global, que es la actividad de la ingeniería de sistemas basados en computadores, consiste, por tanto, en decidir qué elementos hardware se utilizarán, qué elementos software han de ser adquiridos o desarrollados, y qué personas se necesitan para operar el sistema y, al mismo tiempo, repartir y asignar las actividades de tratamiento de información entre estos diferentes "elementos" componentes.

1.2 Características del software

Tradicionalmente se vienen clasificando los elementos componentes de un sistema informático en dos grandes grupos: el hardware y el software. Los componentes hardware, o equipos físicos, se identifican con relativa

6 Introducción a la Ingeniería de Software

facilidad. El software, sin embargo, es algo más difícil de caracterizar, y a veces se define por exclusión: es software todo lo que no es hardware. El software incluye, por supuesto, los programas que gobiernan el funcionamiento del computador, pero también incluye otros elementos tales como documentos, bases de datos, o incluso algo tan inmaterial como son los procedimientos de operación o de mantenimiento periódico.

El software presenta diferencias importantes respecto al hardware, que afectan profundamente a las actividades de ingeniería de uno y de otro. Comenzaremos por analizar las actividades típicas de la ingeniería de productos hardware, que forman parte de la cultura general del hombre de la calle. Pensemos, por ejemplo, en la ingeniería de automóviles, que son productos bien conocidos.

Los componentes hardware se obtienen mediante un proceso de fabricación en que se obtienen sucesivas copias de un producto patrón diseñado previamente. Esta operación de fabricación es costosa, y conlleva un consumo significativo de energía, materiales y mano de obra. La operación de diseño previo del producto también es costosa, pero su costo repercute sólo parcialmente en cada unidad de producto fabricado.

Por otra parte, la utilización de un elemento hardware implica un proceso de desgaste o envejecimiento que exige reparaciones ocasionales o periódicas para garantizar un servicio aceptable.

La ingeniería de software, sin embargo, presenta características especiales. El proceso de fabricación, consistente en obtener copias sucesivas del producto, es trivial y se puede realizar a un costo muy bajo. La labor importante es la del desarrollo inicial, de manera que el costo total de fabricar miles de unidades de un producto software es similar al de fabricar una sola.

Además, el software no se desgasta. Un programa funcionará al cabo de los años con la misma corrección con que lo hizo el primer día sin necesidad de modificación ninguna. Las tareas de mantenimiento de software son en realidad tareas adicionales de desarrollo, realizadas ocasionalmente durante la vida útil del producto con objeto de mejorarlo.

1.3 Concepto de Ingeniería de Software

El término *Ingeniería de Software* aparece utilizado inicialmente de manera formal con ocasión de un congreso de la OTAN [Buxton76] a finales de los

años 60, aunque hay evidencia de que se venía empleando informalmente en los años anteriores. Con esta denominación se designa el empleo en el desarrollo de productos software de técnicas y procedimientos típicos de la ingeniería en general.

La ingeniería de software amplía la visión del desarrollo de software como una actividad esencialmente de programación, contemplando además otras actividades de análisis y diseño previos, y de integración y verificaciones posteriores. La distribución de todas estas actividades a lo largo del tiempo constituye lo que se ha dado en llamar "ciclo de vida" del desarrollo de software.

1.3.1 Perspectiva histórica

En las décadas iniciales de existencia de la informática, la labor de desarrollo de software se planteaba como una actividad artesanal, basada en la labor de personas habilidosas y más o menos creativas, que actuaban en forma individual y relativamente poco disciplinada.

Al aumentar la capacidad de los computadores gracias a los avances del hardware, aumenta también la complejidad de las aplicaciones a programar, y se aprecia la necesidad de una mejor organización de la producción de software, basada en el trabajo en equipo, con la consiguiente división y organización del trabajo, y el empleo de herramientas apropiadas que automatizan las labores triviales y repetitivas.

La organización del proceso de desarrollo de software da lugar a la aparición de *metodologías de desarrollo* específicas, recomendadas por diversos especialistas en áreas de aplicación más o menos especializadas. En particular aparecen muchas metodologías particulares para el desarrollo de los llamados *sistemas de información*, que constituyen la inmensa mayoría de los desarrollos informáticos.

Estas metodologías se desarrollan a lo largo de los años 70, pero sólo empiezan a aplicarse ampliamente en los años 80, gracias a la aparición de herramientas de soporte apropiadas, denominadas en general *herramientas CASE* (Computer Aided Software Engineering).

Las herramientas CASE tradicionales apoyaban las actividades inmediatamente anteriores a la programación o codificación, para la que se seguían empleando herramientas tradicionales (compilador, etc.) que funcionaban en forma totalmente separada de la herramienta CASE. En los años 90 se amplía el campo de aplicación de las herramientas de ingeniería

8 Introducción a la Ingeniería de Software

de software, reconociendo la necesidad de automatizar aún más la labor de desarrollo mediante la formalización del proceso completo de producción de software y el empleo de herramientas que soporten todo el ciclo de vida de desarrollo. Estas herramientas se designan con las siglas IPSE (Integrated Project Support Environment) o, más recientemente, ICASE (Integrated CASE).

1.3.2 La crisis del software

Una situación de crisis representa un punto importante de evolución. Una crisis se resuelve, para bien o para mal, mediante fuertes cambios.

La crisis del software corresponde a la situación, indicada en la sección anterior, en que los avances en la capacidad de los equipos hardware dió lugar a la necesidad de desarrollar aplicaciones software demasiado complejas para ser llevadas a cabo con las técnicas artesanales del momento. Esta situación aparece con claridad a mediados de la década de los 60.

La evolución forzada por esta crisis se traduce, como ya se ha indicado, en la aparición de metodologías concretas de desarrollo y, en general, en la concepción de la ingeniería de software como disciplina.

Sin embargo la crisis del software se ha diferenciado de lo que normalmente se entiende por crisis en que esta evolución no ha dado lugar a una posterior situación estable, ni siquiera a corto plazo. En lugar de ello nos encontramos con que la evolución del hardware continúa a un fuerte ritmo, abaratando el costo de los equipos informáticos al tiempo que aumenta su capacidad, y forzando un incremento continuo en la complejidad de las aplicaciones software. Con ello se mantiene la situación de crisis que se había producido, y que continúa hasta nuestros días.

Lo que nos encontramos, más que una crisis tradicional, es una situación de fuerte evolución permanente, con avances continuos en las técnicas de ingeniería de software que resultan sin embargo insuficientes en cada momento. De hecho se discute si el término *ingeniería de software* es realmente válido, ya que una disciplina de ingeniería se caracteriza, en general, por haber alcanzado una cierta madurez en las técnicas a aplicar, basadas en un fundamento científico subyacente y no sólo en un pragmatismo artesanal.

1.3.3 Mitos del software

El continuo proceso de evolución en las disciplinas de desarrollo de software, junto con algunas de sus características particulares, tales como una relativa inmaterialidad, hacen difícil obtener una visión serena y justa de la ingeniería de software, provocando que por parte de los usuarios e incluso de algunos profesionales se mantengan opiniones infundadas sobre la importancia o influencia de determinados factores en el éxito o calidad de un producto software.

Algunas de estas opiniones, relativamente generalizadas, constituyen verdaderos mitos, difíciles de erradicar. Podemos mencionar, por ejemplo, las siguientes:

- *El hardware es mucho más importante que el software:* Manifiestamente falso, ya que al usar un computador nuestra interacción es fundamentalmente con el software, y sólo de una manera muy limitada el usuario accede directamente a elementos hardware del equipo. Este menosprecio por el software se evidencia en quienes consideran que la realización de copias "pirata" o ilegales de los programas no es una acción censurable.
- *El software es fácil de desarrollar:* Esto es falso para cualquier aplicación software de cierta importancia. El desarrollo de grandes sistemas es muy complejo y costoso, incluso aunque esos sistemas no empleen ningún material o hardware específico. De hecho el desarrollo de software exige una mayor proporción de mano de obra, frente al empleo de maquinaria, y por ello el progresivo aumento del costo de la mano de obra en los países desarrollados ha llevado a un crecimiento importante en el coste de los productos software.
- *El software consiste exclusivamente en programas ejecutables:* El software no se define de esta manera. Al concebir un sistema informático de manera global hay que pensar en todos los elementos que intervienen: hardware, software y personas. Los procedimientos que han de seguir esas personas para usar correctamente el sistema son también elementos software. Igualmente es software toda la documentación del desarrollo, necesaria para poder mantener el producto después de haber sido puesto en explotación.
- *El desarrollo de software es sólo una labor de programación:* Falso, pues no se puede limitar el trabajo de desarrollo sólo a la fase de codificación. Las tareas de análisis y diseño no son meras actividades

10 Introducción a la Ingeniería de Software

complementarias que puedan ser vistas como un costo añadido, necesario para organizar el trabajo en equipo. Las tareas de análisis y diseño son el fundamento para todo el resto del desarrollo, igual que el proyecto de un arquitecto o de un ingeniero es necesario para acometer la construcción de un edificio u otra obra pública, que no consiste simplemente en colocar materiales uno sobre otro.

- *Es natural que el software contenga errores:* No exactamente. Aunque es cierto que el desarrollo de software, como toda actividad humana, es susceptible de errores, no es admisible que los productos software siempre contengan errores. Si un producto hardware contiene defectos, se rechaza. Con el software debería ocurrir lo mismo. Por desgracia el software erróneo no puede simplemente sustituirse por otro sin defecto, ya que todas las copias del producto software son exactamente iguales. Los errores del software son errores durante su desarrollo inicial, y deben reducirse a un nivel tan bajo como en el diseño de los productos hardware durante la fase de desarrollo de ingeniería.

1.4 Formalización del proceso de desarrollo

Una característica de la ingeniería es la existencia de procedimientos bien establecidos para la realización de las actividades de desarrollo, construcción, fabricación, etc. A continuación se expone cómo se plantean esos procedimientos en el caso de la ingeniería del software.

1.4.1 El ciclo de vida del software. Modelos clásicos

Uno de los primeros logros de la ingeniería del software ha sido identificar de manera precisa la forma que adopta el proceso de desarrollo del software. Este proceso de desarrollo, incluyendo el mantenimiento necesario durante su explotación, se ha dado en llamar *ciclo de vida del software*.

Los modelos clásicos de ciclo de vida plantean el desarrollo y explotación de una aplicación software como una secuencia de actividades sucesivas y diferentes que se van realizando una tras otra. Dos formas bien establecidas de plantear el proceso de desarrollo son:

- El modelo en cascada
- El modelo en V

Ambos modelos identifican actividades similares, y prácticamente sólo se diferencian en la forma de presentación de las mismas.

1.4.1.1 El modelo en cascada

Este es el modelo más primitivo de ciclo de vida, pero que ha resultado fundamental para progresos posteriores, porque en él se identifican ya prácticamente todas las clases de actividades distintas que intervienen en el desarrollo y explotación de software.

En la Figura 1.1 aparece una representación gráfica de este modelo. Se identifican distintas actividades o fases que han de realizarse precisamente en el orden indicado, de manera que el resultado de una de estas fases es el elemento de entrada para la fase siguiente.

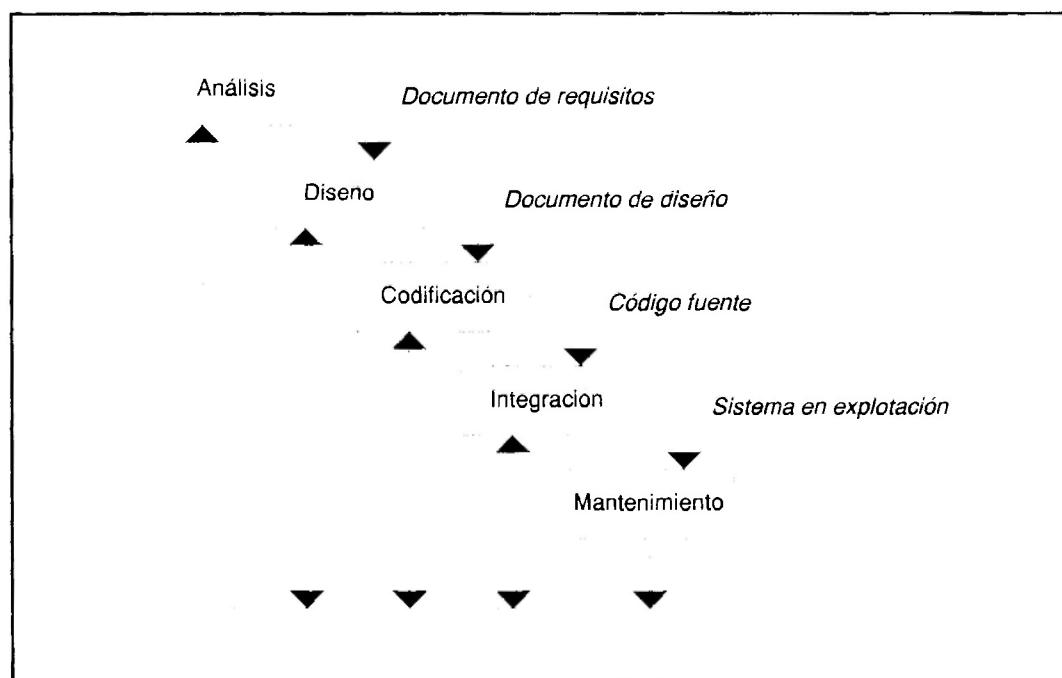


Figura 1.1. Ciclo de vida en cascada

Existen diferentes variantes de este modelo, que se diferencian en el reconocimiento o no como fases separadas de ciertas actividades, de manera que lo que en una variante se plantea globalmente como una sola fase, en otra puede desglosarse en una secuencia de dos o tres fases consecutivas. En la Figura 1.1 se han representado las fases siguientes:

12 Introducción a la Ingeniería de Software

- 1) ANALISIS: Consiste en analizar las necesidades de los usuarios potenciales del software para determinar qué debe hacer el sistema a desarrollar, y de acuerdo con ello escribir una especificación precisa de dicho sistema.
- 2) DISEÑO: Consiste en descomponer y organizar el sistema en elementos componentes que puedan ser desarrollados por separado, para aprovechar las ventajas de la división del trabajo y poder hacer el desarrollo en equipo. El resultado del diseño es la colección de especificaciones de cada elemento componente.
- 3) CODIFICACIÓN: En esta fase se programa cada elemento componente por separado, es decir, se escribe el código fuente de cada uno. Normalmente se harán también algunas pruebas o ensayos para garantizar en lo posible que dicho código funciona correctamente.
- 4) INTEGRACIÓN: A continuación hay que ir combinando todos los elementos componentes del sistema, y probar el sistema completo. Habrá que hacer pruebas exhaustivas para garantizar el funcionamiento correcto del conjunto, antes de ser puesto en explotación.
- 5) MANTENIMIENTO: Durante la explotación del sistema software es necesario realizar cambios ocasionales, bien para corregir errores no detectados con anterioridad, o bien para introducir mejoras. Para ello hay que rehacer parte de los trabajos anteriores.

El modelo de ciclo de vida en cascada trata de aislar cada fase de la siguiente, de manera que las fases sucesivas puedan ser realizadas por grupos de personas diferentes, facilitando la especialización. De esta manera podemos encontrar perfiles profesionales diferentes, tales como analista, diseñador, programador, etc.

Para conseguir esta relativa independencia es fundamental que en cada fase se genere una información de salida precisa y suficiente para que otras personas puedan acometer la fase siguiente. Se insiste así en la necesidad de establecer unos modelos de documentación apropiados. En general se suelen exigir los documentos siguientes:

- A) DOCUMENTO DE REQUISITOS DEL SOFTWARE (en inglés SRD: *Software Requirements Document*): como producto de la fase de análisis. Consiste en una especificación precisa y completa de lo que debe hacer el sistema, prescindiendo de los detalles internos.

- B) DOCUMENTO DE DISEÑO DEL SOFTWARE (en inglés SDD: *Software Design Document*): como producto de la fase de diseño. Consiste en una descripción de la estructura global del sistema, y la especificación de qué debe hacer cada una de sus partes y cómo se combinan unas con otras.
- C) CÓDIGO FUENTE: como producto de la fase de codificación. Contiene los programas fuente, en el lenguaje de programación elegido, y debidamente comentados para conseguir que se entiendan con claridad.
- D) EL SISTEMA SOFTWARE, ejecutable: como producto de la fase de integración. Deben documentarse también las pruebas realizadas sobre el sistema completo.
- E) DOCUMENTOS DE CAMBIOS: tras cada modificación realizada durante el mantenimiento. Los documentos de cambios suelen recopilar información de cada problema detectado, descripción de la solución adoptada, y las modificaciones realizadas sobre el sistema para aplicar dicha solución.

El modelo en cascada hace énfasis también en la necesidad de terminar correctamente cada fase antes de comenzar la siguiente. Esto se debe a que los errores producidos en una fase son muy costosos de corregir si ya se ha realizado el trabajo de las fases siguientes.

Para detectar los errores lo antes posible, y evitar que se propaguen a fases posteriores, se establece un proceso de revisión al completar cada fase, antes de pasar a la siguiente. Esta revisión se realiza fundamentalmente sobre la documentación producida en esa fase, y se hace de una manera formal, siguiendo una lista de comprobaciones establecida de antemano. Si, a pesar de todo, durante la realización de una fase se detectan errores en el resultado de fases anteriores, será necesario rehacer parte del trabajo volviendo a un punto anterior del ciclo de vida, como se indica con línea discontinua en la Figura 1.1.

Como se ha indicado, existen variantes en cuanto a la colección particular de actividades que componen el proceso en cascada. En la Figura 1.2 se representa una variante ampliada de este modelo de ciclo de vida. Este modelo se utilizó en el desarrollo de grandes sistemas que necesitan también una ingeniería de sistemas previa, un diseño arquitectónico, unas pruebas para cada unidad y las pruebas del sistema completo.

14 Introducción a la Ingeniería de Software

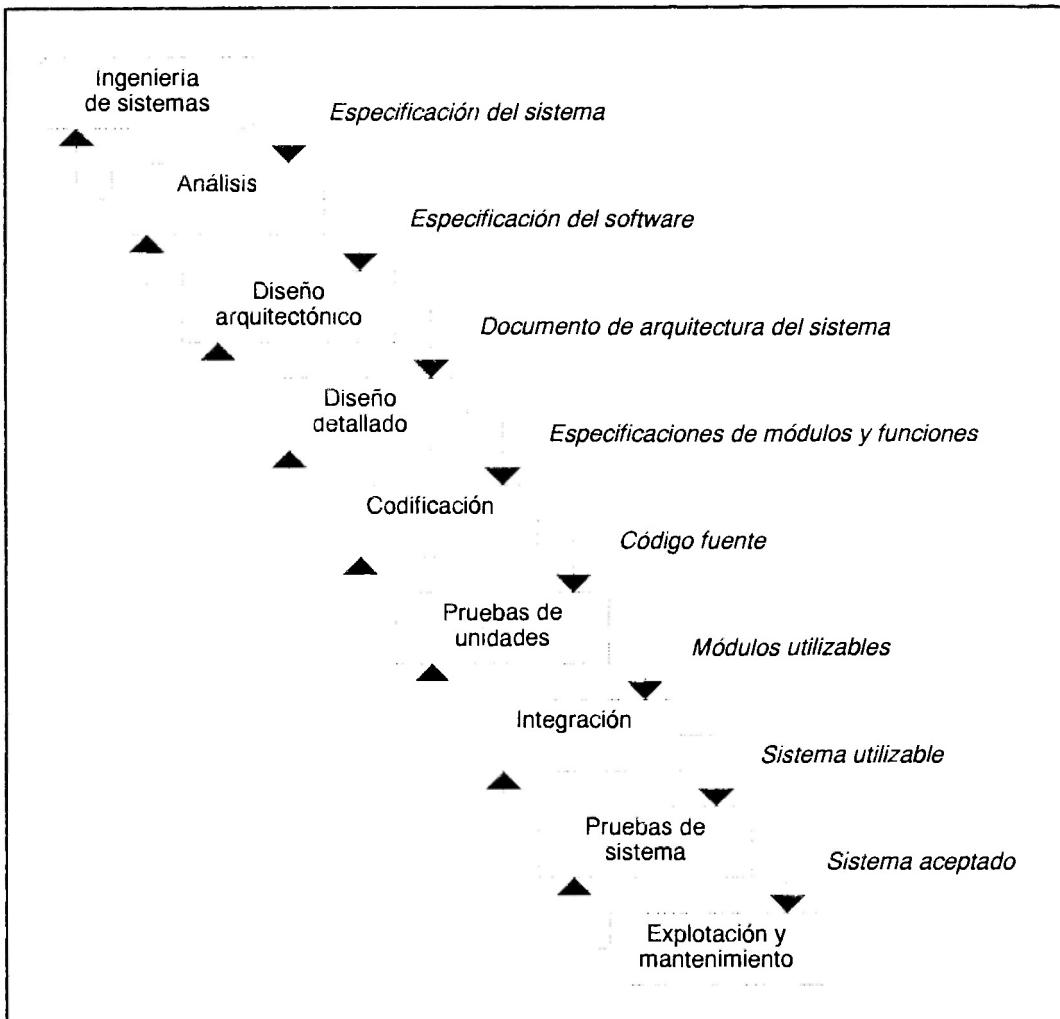


Figura 1.2. *Modelo en cascada, ampliado*

1.4.1.2 El modelo en V

Este modelo se basa en una secuencia de fases análoga a la del modelo en cascada, pero se da especial importancia a la visión jerarquizada que se va teniendo de las distintas partes del sistema a medida que se avanza en el desarrollo. En la Figura 1.3 se recoge esta idea en un diagrama bidimensional, en que el eje horizontal representa avance en el desarrollo y el eje vertical corresponde al nivel de detalle con que se trabaja en cada fase.

En este diagrama vemos cómo en las fases iniciales, en la rama izquierda descendente, el sistema software se va descomponiendo en elementos cada vez más sencillos, hasta llegar a las sentencias del lenguaje de

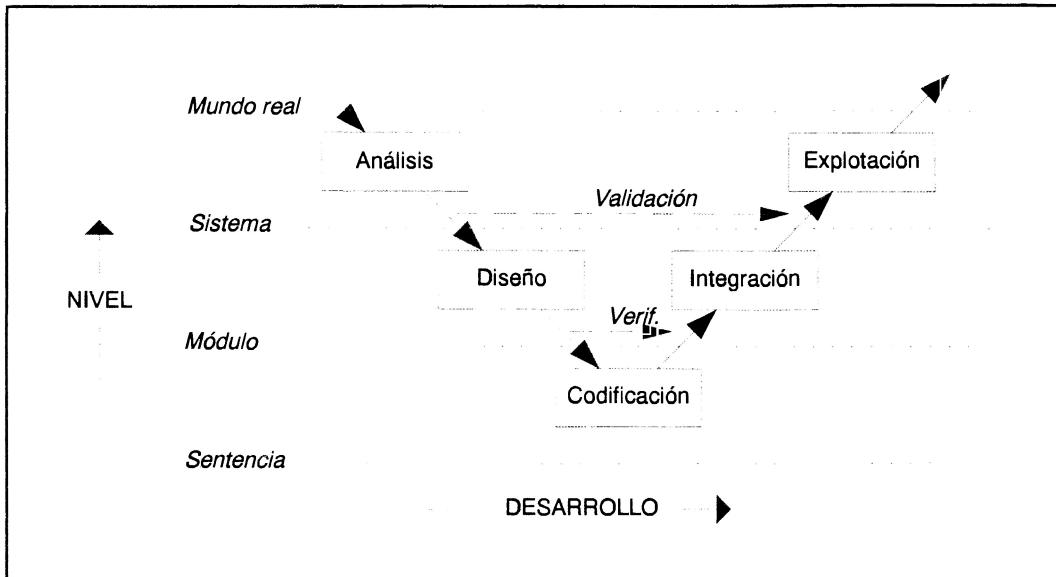


Figura 1.3. *Modelo en V del ciclo de vida*

programación. A partir de ahí el sistema se va construyendo poco a poco a base de integrar los elementos que lo componen, siguiendo la rama derecha ascendente, hasta disponer del sistema completo listo para ser usado.

Al igual que en el modelo en cascada, existen diversas variantes del modelo en V, que se caracterizan por reconocer o no determinados niveles de detalle intermedios. El diagrama de la Figura 1.3 corresponde a un modelo simplificado, similar al modelo en cascada, también simplificado, de la Figura 1.1.

En las actividades situadas en un nivel determinado se trabaja sobre una unidad del nivel de detalle superior, que se organiza en varias unidades del nivel de detalle inferior. Por ejemplo, durante la codificación se trabaja con un módulo, que se organiza y construye con sentencias del lenguaje. Durante las fases de diseño y de integración se trabaja con un sistema software, que se organiza (durante el diseño) o se construye (durante la integración) con varios módulos.

En el diagrama en V se puede poner de manifiesto de manera elegante que el resultado de una fase no sólo sirve como entrada para la fase inmediatamente siguiente, sino que también debe utilizarse en fases posteriores para comprobar que el desarrollo es correcto. En particular la comprobación de que una parte del sistema cumple con sus especificaciones particulares se denomina *verificación*, y en el diagrama simplificado de la

16 Introducción a la Ingeniería de Software

figura 1.3 aparece a nivel de módulo. La comprobación de que un elemento satisface las necesidades del usuario identificadas durante el análisis se denomina *validación*, y en el diagrama aparece a nivel del sistema completo.

Al igual que en el modelo en cascada, se pueden establecer modelos de ciclo en V más elaborados, con un mayor número de fases. En la Figura 1.4 se representa una variante ampliada de este modelo.

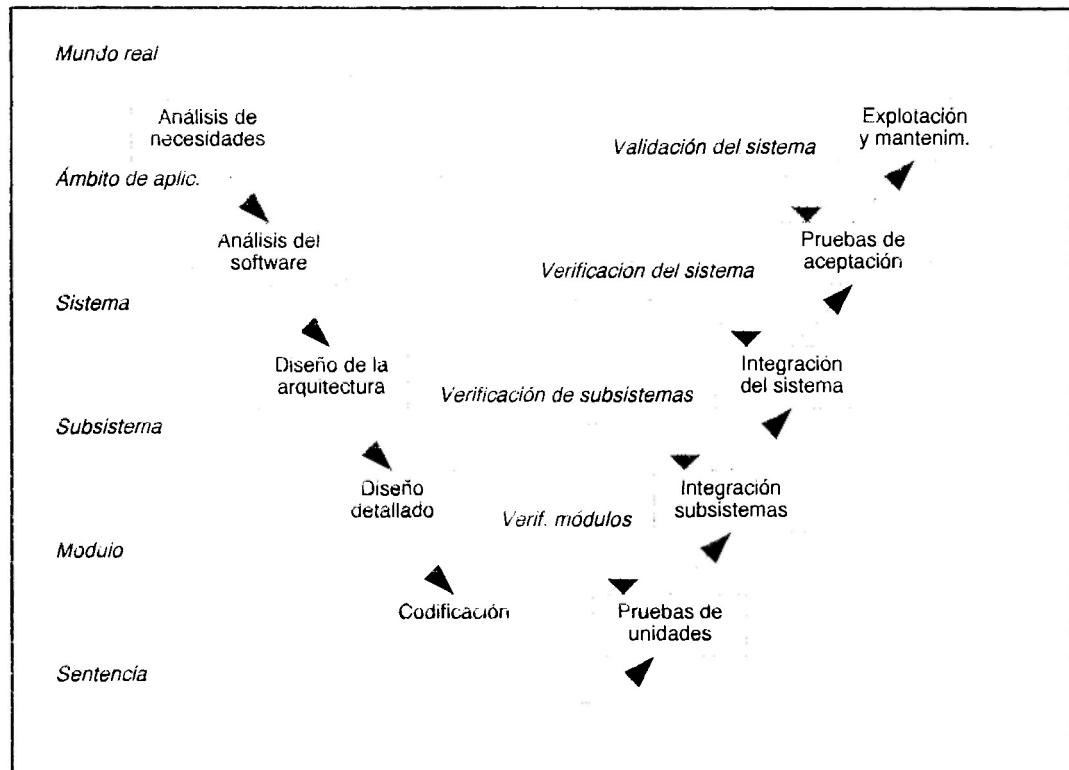


Figura 1.4. Modelo en V, ampliado

1.5 Uso de prototipos

Los modelos clásicos tienen el inconveniente de estar muy orientados hacia una forma de desarrollo lineal, en que cada fase del desarrollo tiene una duración limitada en el tiempo, de forma que una vez terminada una fase pueden dedicarse a otra cosa los recursos humanos o materiales que se han empleado en ella. Esto quiere decir que no se contemplan de manera organizada las vueltas atrás necesarias ocasionalmente al detectar algo inadecuado en una fase anterior durante una fase posterior del desarrollo.

Estas vueltas atrás resultan así bastante costosas, por lo que los modelos clásicos insisten mucho en las actividades de revisión del resultado de cada fase, para evitar los retrocesos, en lo posible.

Desgraciadamente hay situaciones en que no es posible garantizar adecuadamente al concluir una fase que su resultado es correcto. Esto ocurre, por ejemplo, en sistemas innovadores, en que no se dispone de una experiencia previa para contrastar si las decisiones adoptadas durante el análisis y diseño son apropiadas.

El empleo de prototipos puede solucionar, al menos en parte, este problema. Un *prototipo* es un sistema auxiliar que permite probar experimentalmente ciertas soluciones parciales a las necesidades del usuario o a los requisitos del sistema. Si el prototipo se desarrolla con un costo sensiblemente inferior al del sistema final, los errores cometidos en el mismo no resultarán demasiado costosos, ya que su incidencia está limitada por el costo total del desarrollo de dicho prototipo, y normalmente será inferior, ya que siempre habrá algo del prototipo que sea aprovechable para el resto del desarrollo.

Para reducir el costo del desarrollo del prototipo, con respecto al del sistema final, se puede:

- Limitar las funciones, y desarrollar sólo unas pocas
- Limitar su capacidad, permitiendo que sólo se procesen unos pocos datos
- Limitar su eficiencia, permitiendo que opere en forma lenta
- Evitar limitaciones de diseño usando un soporte hardware más potente
- Reducir la parte a desarrollar, usando un apoyo software más potente

Normalmente se distinguen dos clases de prototipos, según se pretenda aprovechar el código del mismo, o sólo la experiencia obtenida con él, tal como se indica a continuación.

1.5.1 Prototipos rápidos

Estos prototipos son aquellos cuya finalidad es sólo adquirir experiencia, sin pretender aprovecharlos como producto. Se denominan también prototipos de *usar y tirar* (en inglés *throw-away*), y *maquetas* (en inglés *mockup*) cuando su funcionalidad o capacidad es muy limitada.

18 Introducción a la Ingeniería de Software

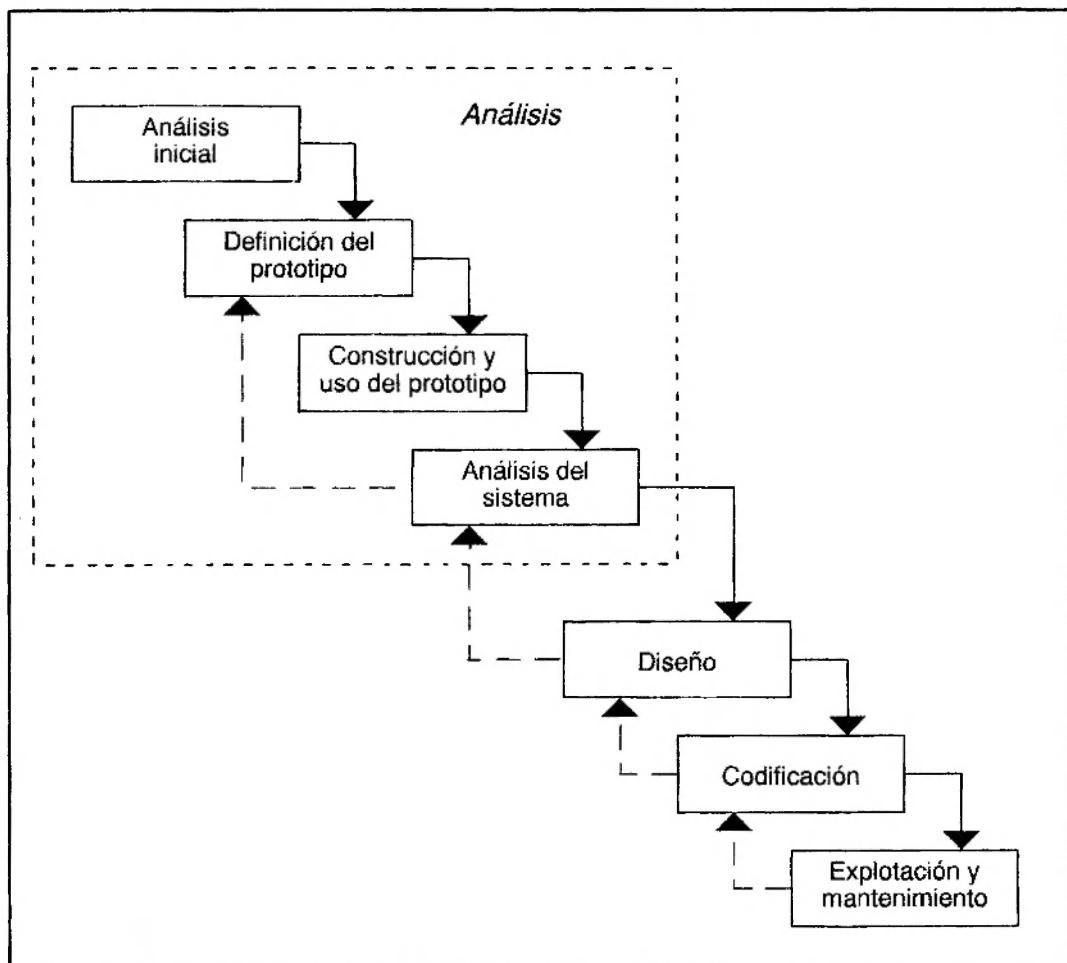


Figura 1.5. Análisis con prototipo rápido

Estos prototipos se aprovechan dentro de las fases de análisis y/o diseño de un sistema, para experimentar algunas alternativas y garantizar en lo posible que las decisiones tomadas son correctas. Una vez completadas estas fases el sistema final se codifica totalmente partiendo de cero, es decir, sin aprovechar el código del prototipo. La Figura 1.5 representa una variante del ciclo de vida en cascada usando un prototipo de usar y tirar durante la fase de análisis.

Lo importante en estos prototipos es desarrollarlos en poco tiempo, y de ahí el nombre de *prototipos rápidos*, para evitar alargar excesivamente la duración de las fases de análisis y diseño. Hay que tener en cuenta que el desarrollo completo y experimentación con el prototipo o prototipos es sólo una parte de alguna fase del ciclo de vida del sistema final.

1.5.2 Prototipos evolutivos

Otra manera de utilizar un prototipo es tratando de aprovechar al máximo su código. En este caso el prototipo se desarrollará sobre el mismo soporte hardware/software que el sistema final, pero sólo realizará algunas de las funciones, o en general, será sólo una realización parcial del sistema deseado.

El prototipo inicial se construirá tras unas fases parciales de análisis y diseño. La experimentación con el prototipo permitirá avanzar en esas fases parciales, y a continuación ampliar el prototipo inicial para irlo convirtiendo en el sistema final mediante adiciones sucesivas.

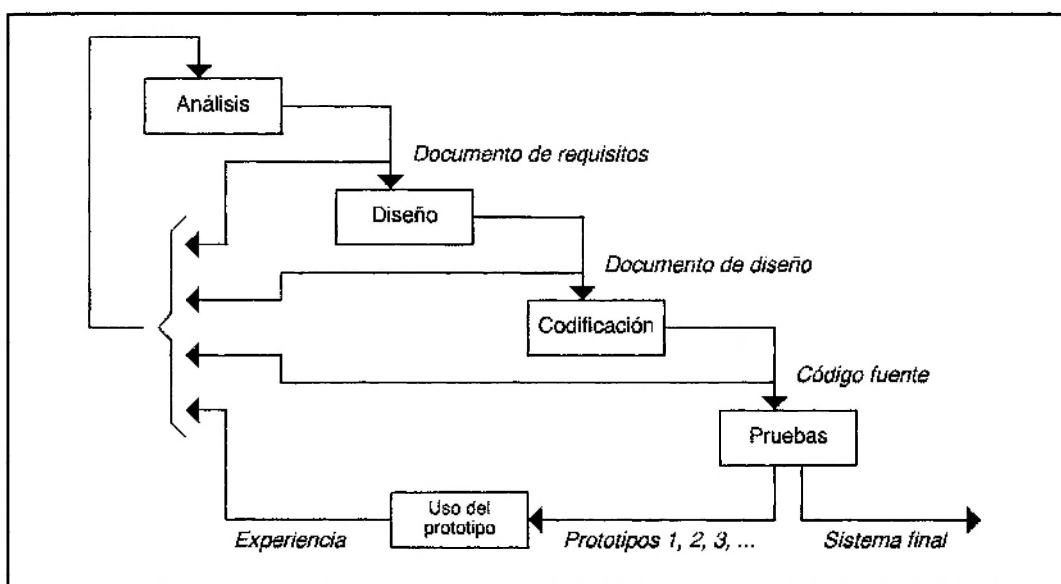


Figura 1.6. *Modelo de ciclo de vida evolutivo*

De esta manera se van construyendo sucesivas versiones del prototipo, cada vez más completas, hasta obtener el sistema deseado. Al mismo tiempo los documentos de especificación, diseño, etc. van siendo también desarrollados progresivamente.

Esta forma de desarrollo puede formalizarse en un modelo de *ciclo de vida evolutivo*, tal como el que se representa en la Figura 1.6. Este modelo puede considerarse como un proceso iterativo en bucle sobre el modelo en cascada, de manera que en cada iteración se hace sólo una parte del desarrollo, avanzando un poco en cada fase. Cada iteración utiliza todo lo que se ha generado en la iteración anterior, y produce un nuevo prototipo, que es una nueva versión parcial del sistema, hasta llegar finalmente al

20 Introducción a la Ingeniería de Software

sistema completo, con lo que se sale del bucle de iteraciones y termina el proceso.

1.5.3 Herramientas para realización de prototipos

Puesto que un prototipo es una versión particular del sistema software a desarrollar, se podrán usar para su construcción herramientas similares a las que se usen en cualquier desarrollo de software. Evidentemente, si se trata de un prototipo evolutivo habrá que usar, en general, las herramientas que se seguirán empleando hasta el sistema definitivo.

En el caso de los prototipos de usar y tirar se podrán emplear herramientas diferentes que para el sistema definitivo. En especial se buscarán herramientas que permitan construir el prototipo en poco tiempo, si se trata de un prototipo rápido, y en todo caso herramientas que permitan hacerlo con el mínimo esfuerzo, para reducir el costo total del desarrollo.

Entre las herramientas que suelen recomendarse para la construcción de prototipos están las denominadas de 4^a generación. Estas herramientas son especialmente adecuadas para la construcción de sistemas de información, que se apoyan en una base de datos de uso general, y utilizan lenguajes especializados para programar la interfaz de usuario, basada en menús, formularios de entrada de datos, y formatos de informes.

Estas herramientas se utilizan muchas veces en la construcción del sistema final, ya que facilitan mucho el trabajo de desarrollo. Si no fuesen seleccionadas para ello por razones de eficiencia o compatibilidad, siempre podrán ser empleadas en la construcción de un prototipo previo, si se considera apropiado.

Los lenguajes de 4^a generación son simplemente un caso particular de *lenguajes de muy alto nivel*, que normalmente siguen un estilo declarativo en lugar de operacional. Los lenguajes declarativos permiten describir el resultado que se desea obtener, en lugar de describir las operaciones para conseguirlo. Los lenguajes de muy alto nivel y sus herramientas asociadas resultarán también apropiadas para el desarrollo de prototipos rápidos. En este sentido se ha recomendado a veces el empleo de lenguajes tales como Prolog o SmallTalk.

Análogos a los lenguajes de muy alto nivel son los *lenguajes de especificación*. Estos lenguajes tienen como objetivo formalizar la especificación de los requisitos de un sistema software, y entre ellos hay

algunos de uso bastante extendido, como VDM y Z. Si especificamos una aplicación con uno de estos lenguajes, y disponemos de un compilador o generador de programas para dicho lenguaje, podremos obtener directamente un prototipo ejecutable.

Otra técnica adecuada para la construcción de prototipos es el uso extensivo de la *reutilización* de software. Combinando varios módulos o subsistemas escritos de antemano podremos construir en poco tiempo un sistema nuevo. Si la colección de software disponible es extensa o especializada en nuestro campo de aplicación, no será difícil encontrar un repertorio de componentes que nos permitan construir con poco esfuerzo un sistema similar al deseado.

1.6 El modelo en espiral

El modelo espiral desarrollado por B. Boehm [Boehm88] puede considerarse como un refinamiento del modelo evolutivo general. Como elemento distintivo respecto a otros modelos de ciclo de vida, introduce la actividad de *análisis de riesgo* como elemento fundamental para guiar la evolución del proceso de desarrollo.

El ciclo de iteración del modelo evolutivo se convierte en una espiral al añadir como dimensión radial una indicación del esfuerzo total realizado hasta cada momento, que será un valor siempre creciente, tal como se indica en la Figura 1.7. Las distintas actividades se representan sobre unos ejes cartesianos, conteniendo cada cuadrante una clase particular de actividades: PLANIFICACIÓN, ANÁLISIS DE RIESGO, INGENIERÍA y EVALUACIÓN, que se suceden a lo largo de cada ciclo de la espiral. La dimensión angular representa el avance relativo en el desarrollo de las actividades de cada cuadrante. En cada ciclo de la espiral se realiza una parte del desarrollo total, siguiendo la secuencia de las cuatro clases de actividades indicadas.

Las actividades de *planificación* sirven para establecer el contexto del desarrollo, y decidir qué parte del mismo se abordará en ese ciclo de la espiral.

Las actividades de *análisis de riesgo* consisten en evaluar diferentes alternativas para la realización de la parte del desarrollo elegida, seleccionando la más ventajosa y tomando precauciones para evitar los inconvenientes previstos.

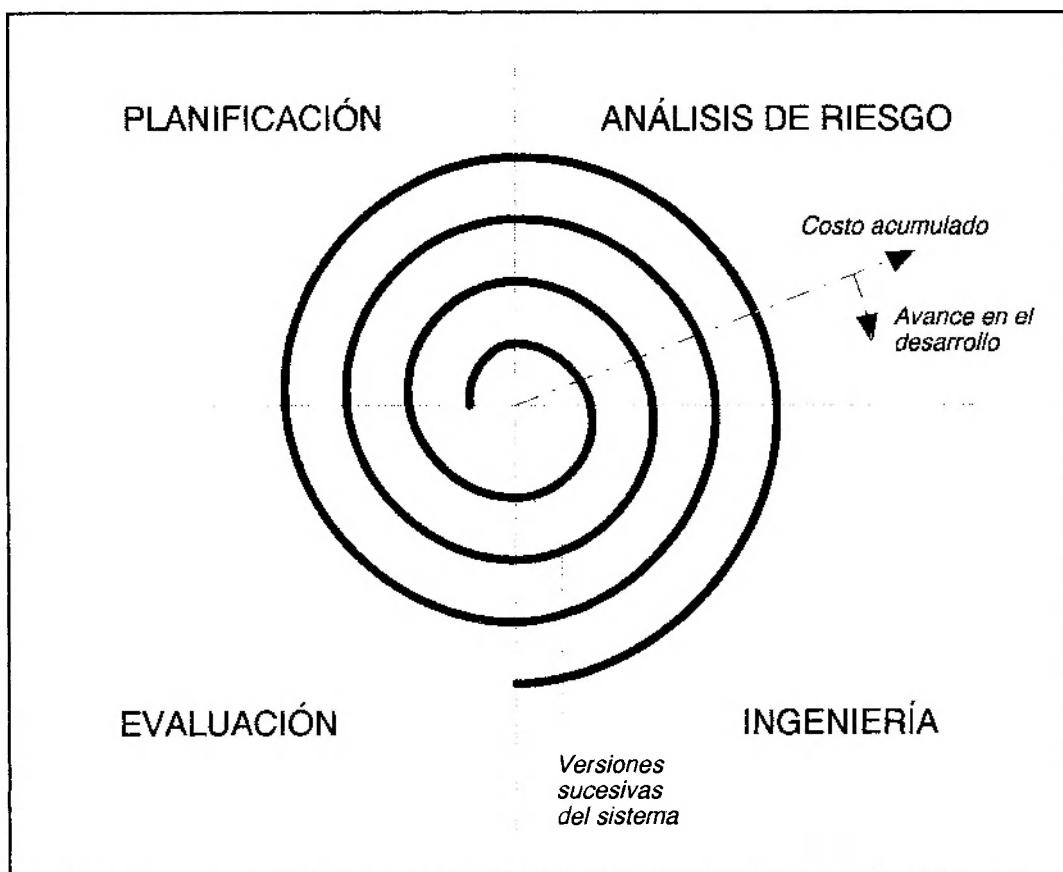


Figura 1.7. *Modelo en espiral del ciclo de vida*

Las actividades de *ingeniería* corresponden a las indicadas en los modelos clásicos: análisis, diseño, codificación, etc. Su resultado será ir obteniendo en cada ciclo una versión más completa del sistema.

Las actividades de *evaluación* analizan los resultados de la fase de ingeniería, habitualmente con la colaboración del "cliente" para quien se realiza el desarrollo. El resultado de esta evaluación se utiliza como información de entrada para la planificación del ciclo siguiente.

Según qué parte del desarrollo se decida hacer en cada ciclo, tendremos distintas variantes del modelo espiral, que podrá así adaptarse a cada proyecto concreto. Por ejemplo, si en cada vuelta se realizase exactamente una de las fases del modelo en cascada, el modelo espiral coincidiría con dicho modelo en cascada en los aspectos de ingeniería. Si en cada vuelta se realiza lo suficiente de cada fase como para obtener una versión parcial ampliada del sistema, el modelo espiral coincidirá con el evolutivo. De

todas formas el modelo espiral siempre se distingue por las actividades de análisis de riesgo, que no aparecen en los otros modelos.

1.7 Combinación de modelos

Cada uno de los diferentes modelos de ciclo de vida presenta ventajas e inconvenientes a la hora de aplicarlo, dependiendo de la clase de sistema a desarrollar y el contexto en que se realiza el desarrollo. A veces puede resultar ventajoso tratar de aprovechar las ventajas de varios modelos diferentes, combinándolos en un modelo mixto de ciclo de vida.

Un ejemplo concreto podría ser el siguiente:

- 1 - Se realiza la fase de análisis empleando la técnica de prototipo rápido.
- 2 - Se evalúan modelos alternativos para el resto del desarrollo
- 3 - Se realiza el resto del desarrollo según el modelo elegido, que podría ser:
 - 3.1 Modelo en cascada
 - 3.2 Modelo evolutivo
 - ... etc. ...

Es fácil darse cuenta de que el modelo espiral es de alguna manera una combinación de modelos, ya que las actividades de ingeniería pueden realizarse según cualquier otro modelo particular de ciclo de vida.

1.8 Mantenimiento del software

Las actividades fundamentales del desarrollo de software, correspondientes a las fases de análisis, diseño, codificación y pruebas se describen con detalle en los siguientes Temas. La fase de mantenimiento es algo particular, y se describe en este primer Tema. Esta etapa final denominada *mantenimiento*, o *explotación y mantenimiento*, no suele representar una actividad específica, sino que consiste normalmente en repetir o rehacer parte de las actividades de las fases anteriores para introducir cambios en una aplicación de software ya entregada al cliente y puesta en explotación.

A continuación se analiza con más detalle en qué consiste el proceso de mantenimiento de una aplicación software, y porqué es necesario realizarlo, en lugar de seguir explotando la aplicación original tal como se desarrolló la primera vez.

24 Introducción a la Ingeniería de Software

1.8.1 Evolución de los aplicaciones

Una de las características del software, que lo diferencian de los productos hardware, es la ausencia de deterioro o envejecimiento durante su utilización. Un sistema software funcionará con la misma corrección al cabo de los años que el primer día de su empleo. ¿Porqué es necesario modificarlo?

Hay varios motivos para ello, que dan lugar a distintos tipos de mantenimiento con diferentes objetivos, entre ellos:

- Mantenimiento correctivo
- Mantenimiento adaptativo
- Mantenimiento perfectivo

El *mantenimiento correctivo* tiene como finalidad corregir errores en el producto software que no han sido detectados y eliminados durante el desarrollo inicial del mismo. Este tipo de mantenimiento no debería existir si el desarrollo inicial se hubiera realizado con las debidas garantías de calidad. Sin embargo es prácticamente imposible evitarlo, ya que el desarrollo de software, como cualquier otra actividad humana, está sujeto a errores.

El *mantenimiento adaptativo* se produce en aplicaciones cuya explotación dura bastantes años, de manera que los elementos básicos hardware y software (máquina + sistema operativo) que constituyen la plataforma o entorno en que se ejecutan evolucionan a lo largo de ese tiempo, modificándose parcialmente la interfaz ofrecida a las aplicaciones que corren sobre ellos. Esto exige modificar una aplicación para adaptarla a las nuevas características del entorno si se quiere seguir utilizándola.

Finalmente el *mantenimiento perfectivo* aparece sobre todo en aplicaciones sujetas a la competencia de mercado. Este tipo de mantenimiento es necesario para ir obteniendo versiones mejoradas del producto que permitan mantener el éxito del mismo. También se realiza sobre aplicaciones en que las necesidades del usuario evolucionan, y por tanto hay que modificar los requisitos iniciales del producto para seguir satisfaciéndolas.

1.8.2 Gestión de cambios

Con independencia del objetivo concreto del mantenimiento, las actividades a realizar durante el mismo son básicamente la realización de cambios sucesivos sobre una determinada aplicación o producto software ya desarrollado. La aplicación de técnicas de ingeniería a la actividad de mantenimiento exige organizar y gestionar apropiadamente el desarrollo de estos cambios.

Podríamos distinguir dos enfoques diferentes en la gestión de cambios, dependiendo del mayor o menor grado de modificación del producto.

Si el cambio a realizar afecta a la mayoría de los componentes del producto, dicho cambio se puede plantear como un *nuevo desarrollo*, y aplicar un nuevo ciclo de vida desde el principio, aunque aprovechando lo ya desarrollado igual que se aprovecha un prototipo.

Si el cambio afecta a una parte localizada del producto, entonces se puede organizar como simple *modificación de algunos elementos*. Es importante darse cuenta de que un cambio en el código del producto software debe dar lugar además a una revisión de los elementos de documentación afectados, es decir, cambiar el código de algunos módulos puede requerir además modificar los documentos de diseño o incluso, en el caso de mantenimiento perfectivo, modificar el documento de especificación de requisitos.

Desde el punto de vista de gestión la realización de cambios se puede controlar mediante dos clases de documentos, que a veces se refunden en uno solo:

- *Informe de problema*: describe una dificultad en la utilización del producto que requiere alguna modificación para subsanarla.
- *Informe de cambio*: describe la solución dada a un problema y el cambio realizado en el producto software.

El informe de problema puede ser originado por los usuarios. Este informe se pasa a un grupo de ingeniería para la comprobación y caracterización del problema, y luego a un grupo de gestión para decidir la solución a adoptar. Este grupo de gestión inicia el informe de cambio, que se pasa de nuevo al grupo de ingeniería para su desarrollo y ejecución.

26 Introducción a la Ingeniería de Software

1.8.3 Reingeniería

Con mucha frecuencia, por desgracia, es necesario mantener productos software que no fueron desarrollados siguiendo las técnicas de ingeniería de software, sino de forma artesanal. En estos casos el desarrollo no suele estar documentado, y se dispone solamente del código fuente, quizá comentado.

Para ayudar al mantenimiento de este tipo de aplicaciones se han desarrollados algunas técnicas particulares que tratan de subsanar estas deficiencias de documentación. Estas técnicas se han denominado inicialmente ingeniería inversa, y más modernamente reingeniería.

La *ingeniería inversa* consiste en tomar el código fuente y a partir de él tratar de construir, si es posible de manera automática, alguna documentación, en particular documentación de diseño, con la estructura modular de la aplicación y las dependencias entre módulos y funciones. Esto facilita la caracterización del ámbito de un posible cambio, es decir, determinar qué módulos y funciones habría que modificar.

Una documentación mecánica de la estructura del sistema software es insuficiente para mantener aplicaciones complejas. La actividad de *reingeniería* se plantea como algo más general, que trata de generar un sistema bien organizado y documentado a partir del sistema inicial deficiente. Este trabajo puede incluir el reconstruir manualmente la documentación inexistente, y modificar el código fuente para reestructurarlo de manera más apropiada.

1.9 Garantía de calidad de software

La calidad de un producto software, como cualquier otro producto de ingeniería, viene determinada fundamentalmente por el proceso seguido en su desarrollo. En el modelo de ciclo de vida encontramos actividades típicamente productivas, tales como las de análisis, diseño y codificación, y otras cuyo objetivo es controlar la calidad del producto, tales como las revisiones y pruebas.

A continuación se analiza cómo se valora la calidad de un producto software, y cómo debe organizarse el proceso de desarrollo para garantizar un nivel de calidad adecuado.

1.9.1 Factores de calidad

La calidad de un producto puede valorarse desde puntos de vista diversos. El software no es una excepción, y existen por tanto diferentes enfoques para la valoración de su calidad.

Lo ideal sería poder medir la calidad del software como se miden ciertos aspectos de calidad de otros productos de ingeniería: pureza de un producto, resistencia de un material, etc. Desgraciadamente ésto no resulta fácil, y las técnicas de aplicación de métricas precisas a los productos software están todavía en evolución.

Existe un esquema general de mediciones de la calidad de software propuesto por McCall y otros [McCall78], basado en valoraciones a tres niveles diferentes, denominados *factores*, *criterios* y *métricas*. Los factores de calidad constituyen el nivel superior, y son la valoración propiamente dicha, significativa, de la calidad. Esta valoración no se hace directamente, sino en función de ciertos criterios o aspectos de nivel intermedio que influyen en los factores de calidad. Las métricas están en el nivel inferior, son mediciones puntuales de determinados atributos o características del producto, y son la base para evaluar los criterios intermedios.

Entre los factores de calidad propuestos encontramos los siguientes:

CORRECCIÓN: Es el grado en que un producto software cumple con sus especificaciones. Podría estimarse como el porcentaje de requisitos que se cumplen adecuadamente.

FIABILIDAD: Es el grado de ausencia de fallos durante la operación del producto software. Puede estimarse como el número de fallos producidos o el tiempo durante el que permanece inutilizable durante un intervalo de operación dado.

EFICIENCIA: Es la relación entre la cantidad de resultados suministrados y los recursos requeridos durante la operación. Se mediría como la inversa de los recursos consumidos para realizar una operación dada.

SEGURIDAD: Es la dificultad para el acceso a los datos o a las operaciones por parte de personal no autorizado.

FACILIDAD DE USO: Es la inversa del esfuerzo requerido para aprender a usar un producto software y utilizarlo adecuadamente.

MANTENIBILIDAD: Es la facilidad para corregir el producto en caso necesario. Se aplica propiamente al mantenimiento correctivo.

FLEXIBILIDAD: Es la facilidad para modificar el producto software. Se aplica propiamente al mantenimiento adaptativo y al perfectivo.

28 Introducción a la Ingeniería de Software

FACILIDAD DE PRUEBA: Es la inversa del esfuerzo requerido para ensayar un producto software y comprobar su corrección o fiabilidad.

TRANSPORTABILIDAD: Es la facilidad para adaptar el producto software a una plataforma (hardware + sistema operativo) diferente de aquella para la que se desarrolló inicialmente.

REUSABILIDAD: Es la facilidad para emplear partes del producto en otros desarrollos posteriores. Se facilita mediante una adecuada organización de los módulos y funciones durante el diseño.

INTEROPERATIVIDAD: Es la facilidad o capacidad del producto software para trabajar en combinación con otros productos.

Estos factores de calidad se centran en características del producto software. En muchos casos se contemplan también otros aspectos relativos al proceso de desarrollo, ya que la organización de éste influye muy directamente en la calidad del producto obtenido.

1.9.2 Plan de garantía de calidad

Una adecuada calidad del producto es inalcanzable sin una buena organización del desarrollo. Ya se ha indicado anteriormente que las técnicas de ingeniería de software tratan de formalizar el proceso de desarrollo evitando los inconvenientes de una producción artesanal informal.

Esta organización del proceso de desarrollo de software debe materializarse en un documento formal, denominado *Plan de Garantía de Calidad de Software* (en inglés SQAP: Software Quality Assurance Plan). El plan debe contemplar aspectos tales como:

- Organización de los equipos de personas y la dirección y seguimiento del desarrollo.
- Modelo de ciclo de vida a seguir, con detalle de sus fases y actividades.
- Documentación requerida, especificando el contenido de cada documento y un guión del mismo.
- Revisiones y auditorías que se llevarán a cabo durante el desarrollo, para garantizar que las actividades y documentos son correctos y aceptables.

- Organización de las pruebas que se realizarán sobre el producto software a distintos niveles. A veces esta parte se redacta como un plan separado.
- Organización de la etapa de mantenimiento, especificando cómo ha de gestionarse la realización de cambios sobre el producto ya en explotación.

1.9.3 Revisiones

Una revisión consiste en inspeccionar el resultado de una actividad para determinar si es aceptable o, por el contrario, contiene defectos que han de ser subsanados. Las revisiones se aplican, fundamentalmente, a la documentación generada en cada etapa o fase del desarrollo.

De acuerdo con el espíritu de las reglas de ingeniería de software, las revisiones deben ser formalizadas. Esto quiere decir que deben estar contempladas en el modelo de ciclo de vida y que debe existir una normativa para su realización.

Parece interesante enunciar algunas recomendaciones concretas sobre cómo formalizar las revisiones, entre ellas las siguientes:

- Las revisiones deben ser realizadas por un grupo de personas, y no por un solo individuo. Esto facilita descubrir posibles fallos, al existir diversos puntos de vista.
- El grupo que realiza la revisión debe ser reducido, para evitar excesivas discusiones y facilitar la comunicación entre sus miembros. Se recomienda de tres a cinco personas.
- La revisión no debe ser realizada por los autores del producto, sino por otras personas diferentes para garantizar la imparcialidad de criterio.
- Se debe revisar el producto, pero no el productor ni el proceso de producción. El producto permanece, mientras que el cómo se obtuvo influye poco en el uso futuro de ese producto.
- Si la revisión ha de decidir la aceptación o no de un producto, se debe establecer de antemano una lista formal de comprobaciones a realizar, y atenerse a ella.

30 Introducción a la Ingeniería de Software

- Debe levantarse acta de la reunión de revisión, conteniendo los puntos importantes de discusión y las decisiones tomadas. Este documento puede considerarse como el producto de la revisión.

1.9.4 Pruebas

Las pruebas o ensayos consisten en hacer funcionar un producto software o una parte de él en condiciones determinadas, y comprobar si los resultados son los correctos. El objetivo de las pruebas es descubrir los errores que pueda contener el software ensayado.

Las pruebas no permiten garantizar la calidad de un producto. Puede decirse que una prueba tiene éxito si se descubre algún error, con lo que se sabe que el producto no cumple con algún criterio de calidad; por el contrario, si la prueba no descubre ningún error, no se garantiza con ello la calidad del producto, ya que pueden existir otros errores que habrían de descubrirse mediante pruebas diferentes. Esto es así porque nunca puede ensayarse un producto en forma exhaustiva, sino que las pruebas sólo hacen que el producto realice una parte ínfima de la enorme variedad de operaciones concretas que podría realizar.

A pesar de esta limitación, las pruebas son una técnica fundamental de garantía de calidad. En el Tema 5 se describen técnicas concretas para organizar y realizar pruebas de software.

1.9.5 Gestión de configuración

Para precisar el concepto de *configuración* consultaremos, como otras veces, el diccionario:

Configuración. (Del latín *configuratio*) Disposición de las partes que componen una cosa y le dan su peculiar figura. ...

La *configuración de software* hace referencia a la manera en que diversos elementos se combinan para constituir un producto software bien organizado, tanto desde el punto de vista de su explotación por el usuario como de su desarrollo o mantenimiento.

Describiremos aquí brevemente los elementos básicos de la gestión de configuración. Estas ideas se encuentran más desarrolladas en otros textos generales de ingeniería de software, tal como [Pressman98].

Para cubrir la doble visión del software, desde el punto de vista del usuario y del desarrollador, habremos de considerar como elementos componentes de la configuración todos los que intervienen en el desarrollo, y no sólo los que forman parte del producto entregado al cliente. Estos elementos serán, por ejemplo:

- Documentos del desarrollo: Especificaciones, diseño, etc.
- Código fuente de los módulos.
- Programas, datos y resultados de las pruebas.
- Manuales de usuario.
- Documentos de mantenimiento: informes de problemas y cambios.
- Prototipos intermedios
- Normas particulares del proyecto.
- ... etc.

El problema de la gestión de configuración es que estos elementos evolucionan a lo largo del desarrollo y la explotación del producto software, dando lugar a diferentes configuraciones particulares, compuestas de diferentes elementos. Los elementos individuales evolucionan a base de construir sucesivas versiones de los mismos.

Para mantener bajo control la configuración o configuraciones de software hay que apoyarse en técnicas particulares de:

- Control de versiones
- Control de cambios

El *control de versiones* consiste en almacenar de forma organizada las sucesivas versiones de cada elemento de la configuración, de manera que al trabajar sobre una configuración concreta del producto software se pueda acceder cómodamente a las versiones apropiadas de sus elementos.

El *control de cambios* consiste en garantizar que las diferentes configuraciones del software se componen de elementos (y versiones de estos elementos) compatibles entre sí, y que constituyen un conjunto coherente.

El control de cambios se realiza normalmente usando el concepto de *línea base*. Una línea base es una configuración particular del sistema. Cada línea base se construye a partir de otra mediante la inclusión de ciertos cambios, que pueden ser la adición o supresión de elementos, o la sustitución de algunos por versiones nuevas de los mismos. La aceptación de los cambios y la consiguiente creación de la nueva línea base ha de controlarse mediante

32 Introducción a la Ingeniería de Software

pruebas o revisiones apropiadas, para garantizar la corrección de la nueva configuración.

Las líneas base constituyen configuraciones estables, que no pueden ser modificadas (se dice que están "congeladas"). La forma de modificar una línea base es crear otra nueva introduciendo los cambios apropiados. La antigua línea base seguirá existiendo, aunque en algún momento se podrá hacer desaparecer si se está seguro de no necesitarla más.

1.9.6 Normas y estándares

Las recomendaciones de la ingeniería de software se han traducido en ocasiones en normas precisas sobre determinados aspectos del desarrollo de software, desde el enfoque global del proceso de desarrollo hasta normas detalladas de codificación, procedimientos concretos para la realización de ensayos o modelos más o menos precisos de la documentación a redactar.

Algunas de estas normas han sido recogidas por organizaciones internacionales y establecidas como *estándares* a seguir en el desarrollo de determinadas aplicaciones. Entre estas normativas encontramos las siguientes:

IEEE: es la asociación profesional Institute of Electrical and Electronics Engineer establecida en USA. Ha establecido una colección de normas, muchas de ellas admitidas también como normas ANSI (American National Standards Institute) y recogidas globalmente en [IEEE93].

DoD: son las siglas del Departamento de Defensa (Department of Defense) norteamericano. La norma fundamental es la DoD-STD-2167A [DoD88], que se complementa con otras normas adicionales contenido, por ejemplo, modelos de los documentos a redactar. En la actualidad está en revisión y será sustituida por la MIL-STD-SDD, que engloba en una sola tanto la norma principal como las complementarias de documentación.

ESA: es la Agencia Europea del Espacio (European Space Agency). Posee una norma general para el desarrollo de software [ESA91]. Esta norma se apoya en algunos aspectos en las normas del IEEE citadas anteriormente.

ISO: son las siglas del organismo internacional de normalización (International Standards Organization). Está integrado por los organismos nacionales de normalización de un gran número de países (el de España es AENOR). Publica un sinfín de normas para toda la actividad técnico-industrial. Entre sus normas de Ingeniería de Software de mayor nivel se encuentran la ISO-9001, que establece los criterios que han de cumplir las empresas que desarrollen software para obtener certificaciones de determinados niveles de garantía de calidad de su producción.

En muchos países se han desarrollado también normas oficiales similares a éstas, para uso interno. Entre las normas españolas encontramos:

METRICA-2: Desarrollada por el Consejo Superior de Informática del Ministerio para las Administraciones Públicas (MAP). Es una norma para el desarrollo de sistemas de información de las administraciones públicas, basada en la metodología de análisis y diseño estructurado de Yourdon/DeMarco.

Tema 2

Especificación de Software

Este Tema está dedicado a describir la labor de análisis y definición de los requisitos que ha de cumplir un proyecto de software. Esta labor debe dar lugar a la especificación de software, en la que se concretan las necesidades que se desean cubrir y se fijan las restricciones con las que debe trabajar el software a desarrollar. El análisis se realiza dentro de la primera fase (fase de definición) del ciclo de vida y tiene una importancia decisiva para conseguir que el sistema que se construya finalmente sea realmente el que se deseaba.

Primeramente, en este Tema, se hace un breve repaso al modelado de sistemas centrado específicamente en los sistemas desarrollados mediante software. A continuación, se estudia de manera detallada el proceso de análisis de requisitos, estableciendo los objetivos y las tareas de dicho análisis. Seguidamente se describen distintas notaciones que habitualmente se emplean para elaborar la especificación de software. Finalmente, se sugiere un posible formato para el documento que recoge la especificación y que constituye el resultado fundamental del análisis. Con este formato se realizan como ejemplo dos documentos completos de especificación que recogen el análisis de dos aplicaciones que se utilizan a lo largo del libro.

2.1 Modelado de Sistemas

Para la mayoría de los trabajos de ingeniería es bastante habitual utilizar prototipos o maquetas. Por ejemplo en arquitectura es muy común realizar un modelo o maqueta a escala del nuevo edificio. Esto mismo sucede cuando se trata de analizar el comportamiento de un nuevo dispositivo mecánico, eléctrico, hidráulico, etc. Todos estos modelos facilitan al ingeniero la labor de comprensión de los problemas que se plantean en el nuevo sistema a desarrollar.

El modelado de los sistemas realizados mediante software también tiene como objetivo entender mejor el funcionamiento requerido y facilitar la comprensión de los problemas planteados. Sin embargo, para este tipo de

36 Introducción a la Ingeniería de Software

sistemas no se busca, en principio, un modelo físico de su comportamiento. En este caso, el sistema software deberá efectuar de una forma más o menos compleja un determinado tratamiento de la información, y se trata de establecer modelos conceptuales que reflejen por un lado la organización de la información y por otro las diversas transformaciones que se deben llevar a cabo con dicha información.

Hay que tener presente que cuando hablamos de modelo en este Tema nos estamos refiriendo a un modelo completo y preciso del comportamiento u organización del sistema software. No hay que confundir este modelo con el correspondiente a una maqueta o prototipo parcial empleado como ayuda para realizar la actividad de análisis, tal como se indicaba en el Tema anterior.

Existen diversas metodologías para realizar el análisis de los requisitos que debe cumplir un proyecto software. Un aspecto común a todas ellas es que tratan de facilitar la obtención de uno o varios modelos que detallen el comportamiento deseado del sistema. El empleo de una u otra metodología dependerá del tipo de aplicación (gestión, control, cálculo, etc.) o de las preferencias del analista que elabore el modelo. El estudio de metodologías concretas queda fuera del alcance de este libro y serán objeto de estudio en cursos posteriores.

2.1.1 Concepto de modelo

Con carácter general, un modelo conceptual es todo aquello que nos permite conseguir una abstracción lógico-matemática del mundo real y que facilita la comprensión del problema a resolver.

De manera específica, el modelo de un sistema software debe establecer las propiedades y restricciones del sistema. Con el modelo, siempre se tratará de ofrecer una visión de alto nivel, sin descender a explicar detalles concretos del mismo. Por otro lado, el modelo debe explicar QUÉ debe hacer el sistema y no CÓMO lo debe hacer. Después en la etapa de diseño posterior es cuando se debe concretar cómo se deben hacer las cosas.

Así, los objetivos que se deben cubrir con los modelos se pueden concretar en los siguientes:

- 1.- Facilitar la comprensión del problema a resolver.

- 2.- Establecer un marco para la discusión, que simplifique y sistematice la labor tanto del análisis inicial como de las futuras revisiones del mismo
- 3.- Fijar las bases para realizar el diseño
- 4.- Facilitar la verificación del cumplimiento de los objetivos del sistema

2.1.2 Técnicas de modelado

La obtención de un modelo que cubra todos los objetivos anteriores no es una tarea fácil. Las dificultades son de todo tipo. En primer lugar, los sistemas a modelar pueden ser muy complejos. Por otro lado, es relativamente normal que no se disponga de ninguna referencia o experiencia anterior, dado que el sistema que se trata de modelar e implementar no ha sido planteado anteriormente. A continuación se indican algunas técnicas que pueden resultar útiles para realizar el modelado de un sistema software.

2.1.2.1 Descomposición. Modelo jerarquizado

Cuando un problema es complejo, la primera técnica que se debe emplear es descomponerlo en otros más sencillos de acuerdo con el viejo axioma "divide y vencerás". De esta manera, se establece un modelo jerarquizado en el que el problema global queda subdividido en un cierto número de subproblemas. Por ejemplo, si se trata de modelar un sistema para la gestión integral de una empresa, este sistema se puede descomponer en los subsistemas siguientes:

- Sistema de nóminas
- Sistema de contabilidad
- Sistema de facturación
- ... etc. ...

Este tipo de descomposición se denomina horizontal y trata de descomponer funcionalmente el problema. A continuación, en el análisis de cada uno de estos subsistemas se pueden emplear las mismas técnicas que con el sistema global. Por tanto, es posible volver a descomponer cada uno de estos subsistemas a su vez en otros más simples. Por ejemplo, el sistema de nóminas se puede dividir en los siguientes:

- Realización de nóminas
- Pagos a la Seguridad Social

38 Introducción a la Ingeniería de Software

- Pagos del IRPF
- ... etc. ...

Evidentemente, para completar el modelo se tendrán que establecer las interfaces entre las partes o subsistemas en que queda dividido, para posibilitar el funcionamiento del sistema global.

Cuando se descompone un modelo tratando de detallar su estructura, este tipo de descomposición se denomina vertical. Por ejemplo, la realización de nóminas se descompone según la secuencia:

- Entrada de datos del trabajador
- Cálculo de los ingresos brutos
- Cálculo de las retenciones
- Cálculo del pago a la Seguridad Social
- Impresión de la nómina

Esta técnica supone aplicar el método de refinamientos sucesivos al modelado del sistema.

2.12.2 Aproximaciones sucesivas

A pesar de que el sistema a desarrollar nunca será igual a alguno ya en funcionamiento, siempre se podrá tomar como modelo de partida el modelo de otro sistema similar. Por ejemplo, es corriente que el sistema software que se quiere modelar sustituya a un proceso de trabajo ya existente, que se realiza de forma totalmente manual o con un grado de automatización menor del que se pretende lograr con el nuevo sistema. En este caso, se puede crear un modelo de partida basado en la forma de trabajo anterior. Sin embargo, este modelo sólo será preliminar y tendrá que ser depurado mediante aproximaciones sucesivas hasta alcanzar un modelo final.

Como es fácil suponer, la depuración es una labor ardua y difícil que requiere una gran experiencia del analista y en cualquier caso un estudio exhaustivo del problema que se trata de resolver con el sistema software. Para lograr un buen resultado mediante aproximaciones sucesivas, además del analista, es fundamental contar con la colaboración de alguien que conozca muy bien el sistema anterior y que sea capaz de incorporar mejoras dentro del nuevo sistema y discutir las ventajas e inconvenientes de cada uno de los modelos intermedios elaborados.

2.1.2.3 Empleo de diversas notaciones

A veces puede suceder que el modelo resulte muy complejo o incluso imposible de realizar utilizando una única notación para su elaboración. En un apartado posterior se describen distintas notaciones utilizadas habitualmente. En estos casos es importante tratar de utilizar notaciones alternativas o complementarias que simplifiquen el modelo.

Una posible notación para describir el modelo de un sistema es mediante el lenguaje natural. Evidentemente, todo se puede describir mediante una explicación más o menos prolífica empleando el lenguaje natural. Sin embargo, este método puede dar lugar a un modelo difícil de apreciar en su conjunto por lo farragoso de las explicaciones. Además es normal que se produzcan imprecisiones, repeticiones e incluso incorrecciones en el modelo así realizado. Como suele ser habitual, un esquema, una figura, o cualquier método gráfico suele ser más fácil de entender. Hay que recordar que una imagen puede valer más que mil palabras.

Teniendo en cuenta los objetivos, ya citados anteriormente, que debe cubrir un modelo, lo ideal es emplear para su elaboración la notación con la que mejor se cubran dichos objetivos. Incluso es aconsejable emplear varias notaciones juntas cuando sea necesario. Así, existen diversas herramientas de modelado para la ayuda al análisis y diseño de los sistemas, también llamadas herramientas CASE (Computer Aided Software Engineering), que combinan texto, tablas, diagramas, gráficos, etc. Estas herramientas están disponibles en el mercado y están basadas en distintas metodologías de análisis y diseño de sistemas software. Dependiendo del aspecto concreto del sistema que se quiere modelar es más adecuado emplear una u otra notación.

2.1.2.4 Considerar distintos puntos de vista

Para poder concretar la creación de un modelo es necesario adoptar un determinado punto de vista. Después, el modelo que resulte estará necesariamente influido por el punto de vista adoptado. Por ejemplo, cuando un pintor realiza un paisaje, elige el punto de vista que refleje de forma más exacta lo que quiere transmitir con el cuadro: calma, soledad, tristeza, alegría, sosiego, etc. El pintor, con el punto de vista elegido, trata de destacar los rasgos del mundo real que coinciden con el objetivo que se plantea en el cuadro y a la vez trata de ocultar aquellos detalles que no ayudan a perfilar el modelo del mundo real que trata de obtener en el cuadro.

40 Introducción a la Ingeniería de Software

Evidentemente, no se puede comparar la labor de un pintor con la de un ingeniero que trata de obtener el modelo software de una futura aplicación. Sin embargo, también este último debe elegir el punto de vista que permita obtener el modelo más adecuado para representar el comportamiento deseado del sistema. Con este fin, en ocasiones será más adecuado adoptar el punto de vista del futuro usuario del sistema, en otras será más importante considerar el punto de vista del mantenedor del sistema, en algunos modelos será suficiente con perfilarlo desde el punto de vista funcional, etc. Por tanto, en el desarrollo de un modelo se debe elegir convenientemente el punto de vista más adecuado. Para la elección será conveniente esbozar distintas alternativas y elegir aquella que resulte la más idónea.

En ocasiones el modelo no quedará completamente definido con un solo punto de vista. En este caso lo adecuado sería realizar el modelo desde distintos puntos de vista, que muestren todos los aspectos que se quieren destacar del sistema.

2.1.2.5 Realizar un análisis del dominio

Por dominio entenderemos el campo de aplicación en el que se encuadra el sistema a desarrollar. Por ejemplo, supongamos que se trata de desarrollar un sistema para el seguimiento de la evolución de los pacientes de un hospital. Este sistema quedará encuadrado dentro de las aplicaciones para la gestión de hospitales. En este campo, como en cualquier otro, existe desde siempre una cierta manera de realizar las cosas y una terminología ya acuñada que debe ser respetada y tenida en cuenta. Esto es lo que denominaremos realizar un análisis del dominio de la aplicación.

Si bien las peculiaridades de cada aplicación hacen que necesariamente deba ser estudiada como un caso único, es importante analizar el dominio de la aplicación para situarla dentro de un entorno mucho más global. Para realizar este análisis es aconsejable estudiar los siguientes aspectos:

- Normativa que afecte al sistema
- Otros sistemas semejantes
- Estudios recientes en el campo de la aplicación
- Bibliografía clásica y actualizada: Libros y artículos sobre el tema
- ... etc. ...

Este estudio facilitará la creación de un modelo más universal. Como ventajas de este enfoque se puede citar las siguientes:

1.- Facilitar la comunicación entre analista y usuario del sistema:

La creación de un nuevo modelo requiere una colaboración muy estrecha entre el experto que conoce los detalles de la aplicación que se está creando, que denominaremos usuario, y el ingeniero de software que realiza el análisis, que denominaremos analista. Esta colaboración sólo es posible si la comunicación entre ellos resulta fluida y emplean el mismo lenguaje. Por ejemplo, en el sistema de seguimiento de la evolución de los pacientes, para guardar la información de cada paciente desde siempre se emplea el término de "historia clínica" y resultaría confuso cambiar esta denominación por la de "ficha del paciente" que podría sugerir el analista.

2.- Creación de elementos realmente significativos del sistema:

Cuando se ignora el dominio de una aplicación, la solución que se adopta queda particularizada en exceso. Por ejemplo, si se trata de realizar una aplicación de contabilidad para una empresa determinada, es bastante normal que se adapte fielmente a las exigencias del contable de dicha empresa, lo que puede dar lugar a soluciones no válidas para otras empresas. Sin embargo, si se tiene en cuenta el Plan Contable Nacional, la aplicación será válida en cualquier empresa. En este sentido se adoptarán los términos normalizados de balance, cuenta, apunte, transferencia, etc. según un criterio único y universal.

3.- Reutilización posterior del software desarrollado:

Otro de los beneficios importantes del análisis del dominio es la posible reutilización posterior de aquellos elementos que han sido creados dentro de un contexto más globalizado. Este criterio es el que ha sido utilizado desde siempre para dotar a todos los computadores de un paquete de subrutinas matemáticas. Dentro de un dominio de cálculo matemático siempre es necesario disponer de operaciones trigonométricas, logaritmos, matrices, etc. con distintas precisiones y tanto en coma fija como en coma flotante.

Otro ejemplo en este sentido sería el siguiente: supongamos que se trata de realizar un sistema para la gestión de una base de datos en tiempo real que necesita un tiempo de acceso que no se puede lograr

42 Introducción a la Ingeniería de Software

con ninguna de las disponibles en el mercado. Para que el esfuerzo que hay que realizar no sirva sólo para la aplicación concreta, lo que se debe hacer es especificar un conjunto de subrutinas de consulta y modificación que se adapten al estándar SQL (Standard Query Language). Con estas subrutinas cualquier programa que utilice una base de datos mediante SQL podrá utilizar nuestra base de datos con un mejor tiempo de respuesta.

2.2 Análisis de requisitos de software

Como ya se ha comentado anteriormente, la etapa de análisis se encuadra dentro de la primera fase (fase de definición) del ciclo de vida y tiene una importancia decisiva en el desarrollo de todas las etapas posteriores (diseño, codificación, prueba y mantenimiento). Como también ha sido apuntado en el apartado anterior, el ingeniero de software encargado de esta labor lo llamaremos analista.

Con el análisis de requisitos se trata de caracterizar el problema a resolver. Esto se puede concretar en la obtención del modelo global que define por completo el comportamiento requerido del sistema. Esta tarea es bastante compleja. El primer problema que se le presenta al analista es conseguir un interlocutor válido para poderla llevar a cabo. Este interlocutor, que denominaremos de forma genérica *cliente*, puede en algún caso estar constituido por una o varias personas expertas en todo o sólo en una parte del trabajo que se pretende automatizar con el sistema que se trata de especificar, por ejemplo, cuando se trata de automatizar la gestión de una empresa y existe un responsable de la contabilidad, pedidos, facturación, etc. y otro del pago de las nóminas, será necesaria la colaboración de ambos. En otros casos, no existirá nadie capaz de asumir el papel de *cliente*, bien debido a que nadie conoce exactamente lo que se requiere del sistema o bien simplemente por lo novedoso de la aplicación. En estos casos, el analista debe buscar su *cliente* mediante una documentación exhaustiva sobre el tema.

Como ya se ha podido deducir fácilmente, a lo largo de este tema no se asociará al *cliente* con la persona o entidad que financia el proyecto. Aquí, el *cliente* será el encargado o encargados de elaborar junto con el analista las especificaciones del proyecto de software y que posteriormente se encargarán de verificar el cumplimiento de las mismas como si de un contrato se tratara. Por tanto, en algunas ocasiones el *cliente* será el usuario final de la aplicación, en otros coincidirá con el cliente que propiamente financia el proyecto e incluso en otros puede ser simplemente parte de una

especificación de otro proyecto de mayor envergadura en el que está encuadrado el nuestro.

2.2.1 Objetivos del análisis

El objetivo global del análisis es obtener las especificaciones que debe cumplir el sistema a desarrollar. El medio que se emplea para lograr dicho objetivo es obtener un modelo válido, necesario y suficiente para recoger todas las necesidades y exigencias que el *cliente* precisa del sistema y además también todas aquellas restricciones que el analista considere debe verificar el sistema. Las especificaciones se obtendrán basándose en el modelo obtenido.

Evidentemente, en el proceso de análisis quedarán descartadas aquellas exigencias del *cliente* que resulten imposibles de lograr o que simplemente resulten inalcanzables con los recursos puestos a disposición del proyecto. De igual manera, como resultado del diálogo entre el analista y el *cliente* quedarán convenientemente matizadas cada una de las exigencias y necesidades del sistema para adaptarlas a los medios disponibles para el proyecto.

Para lograr una especificación correcta, el modelo global del sistema deberá tener las siguientes propiedades:

1.- Completo y sin omisiones:

Aunque pueda parecer simple, esta propiedad no es sencilla de cumplir dado que a priori no es fácil conocer todos los detalles del sistema que se pretende especificar. Por otro lado, cualquier omisión puede tener una gran incidencia en el diseño posterior e incluso desvirtuar el modelo del sistema. Por ejemplo, supongamos que se omite, por considerarlo sobreentendido, los sistemas operativos (DOS y UNIX) o la configuración mínima que se precisará para la ejecución del sistema a desarrollar. Las consecuencias de esta omisión pueden dar lugar a que se anule o reduzca la utilidad del sistema desarrollado finalmente.

2.- Conciso y sin trivialidades:

En general, uno de los graves errores que se suelen cometer al elaborar una documentación es suponer que será más exhaustiva cuanto más voluminosa resulte. Sin embargo, si el tamaño crece

44 Introducción a la Ingeniería de Software

desmesuradamente suele ser una prueba inequívoca de que no está siendo revisada convenientemente y que muy probablemente tiene trivialidades, repeticiones innecesarias o incluso alguna inexactitud. Por ejemplo, esto puede ocurrir al elaborar un nuevo modelo partiendo de otro anterior semejante. En principio se suele mantener todo aquello que no entra en contradicción con el nuevo modelo. Esto da lugar a que se mantengan párrafos del anterior que no aportan nada al nuevo modelo y que pueden dar lugar a inexactitudes.

Por otro lado, cuando un modelo resulta muy farragoso es muy probable que no se estudie en detalle y que resulte difícil distinguir los aspectos fundamentales de aquellos que no lo son.

3.- Sin ambigüedades

Existe cierta tendencia a pensar que el análisis de requisitos es un mero trámite, que debe ser pasado rápidamente para entrar de lleno en el diseño e implementación del sistema. Esta filosofía, que afortunadamente es cada día menos habitual, da lugar a que en el análisis se dejen ciertos aspectos completamente ambiguos. Las consecuencias de esta actitud son todas negativas:

- Dificultades en el diseño
- Retrasos y errores en la implementación
- Imposibilidad de verificar si el sistema cumple las especificaciones

Si se considera que el modelo resultado del análisis es un contrato con el *cliente*, es evidente que nada debe quedar ambiguo o de lo contrario se producirán inevitablemente fricciones y problemas de consecuencias difíciles de prever.

4.- Sin detalles de diseño o implementación

Como ya se ha indicado anteriormente, el objetivo del análisis es definir el QUÉ debe hacer el sistema y no el CÓMO lo debe de hacer. Por tanto, es claro que no se debe entrar en ningún detalle del diseño o implementación del sistema en la etapa de análisis. Hay que tener en cuenta que el analista puede tener una formación previa muy próxima al diseño y codificación. Esto hace que de una manera involuntaria tenga cierta tentación a buscar la solución en lugar de

exclusivamente plantear el problema. Esta tentación debe ser superada si se quiere realizar un buen análisis.

5.- Fácilmente entendible por el *cliente*

La única forma de conocer si el *cliente* está de acuerdo con el modelo fruto del análisis es que lo entienda y sea capaz de discutir todos sus aspectos. Es importante, por tanto, que el lenguaje que se utilice sea asequible y que facilite la colaboración entre analista y *cliente*. En este sentido es muy interesante el empleo de notaciones gráficas tales como las que se estudiarán en el próximo apartado.

6.- Separando requisitos funcionales y no funcionales

Los requisitos funcionales son los destinados a establecer el modelo de funcionamiento del sistema y serán el fruto fundamental de las discusiones entre analista y *cliente*. Las personas no muy expertas en sistemas software, tales como el usuario, tienen tendencia a creer que los requisitos funcionales son los únicos a tener en cuenta y que sólo en base a ellos se realizarán las pruebas de verificación del sistema después de su implementación. Sin embargo, existen además las restricciones o requisitos no funcionales que están destinados a encuadrar el sistema dentro de un entorno de trabajo y que delimitan:

- Capacidades mínima y máxima
- Interfases con otros sistemas
- Recursos que se necesitan
- Aspectos de seguridad
- Aspectos de fiabilidad
- Aspectos de mantenimiento
- Aspectos de calidad
- ... etc. ...

Estos requisitos no funcionales tienen un origen más técnico y no tienen tanto interés para el *cliente* como lo tienen los funcionales. Por tanto, parece evidente que deban permanecer claramente separados en el modelo del sistema.

7.- Dividiendo y jerarquizando el modelo

La forma más sencilla de simplificar un modelo necesariamente complejo del sistema es dividiéndolo y jerarquizándolo. Esta división

46 Introducción a la Ingeniería de Software

dará lugar a otros submodelos, como ya se indicó anteriormente. En la definición del modelo global del sistema, se deberá ir de lo general a lo particular. Esto facilitará su comprensión y permitirá abordar el análisis de cada parte por separado de una forma racional y sistemática. Ejemplos de este enfoque ya se han mostrado en el apartado anterior de este mismo tema.

8.- Fijando los criterios de validación del sistema:

Es muy importante que en el modelo del sistema queden expresamente indicados cuáles serán los criterios de validación del sistema. No hay que olvidar el aspecto contractual que debe tener el modelo aprobado en la especificación del sistema. En este sentido un buen método para fijar los criterios de validación es realizar con carácter preliminar el Manual de Usuario del sistema. Aunque en el Manual no se recogen todos los aspectos a validar, siempre suele ser un buen punto de partida.

2.2.2 Tareas del análisis

Para la realización correcta de un análisis de requisitos conviene efectuar una serie de pasos o tareas. Dependiendo de las características y complejidad del sistema que se trata de analizar, alguna de las tareas puede resultar trivial o inexistente. A continuación se indican las tareas en el orden en que habitualmente serán desarrolladas:

1.- ESTUDIO DEL SISTEMA EN SU CONTEXTO

Los sistemas realizados mediante software o sistemas software normalmente son subsistemas de otros sistemas más complejos en los que se agrupan subsistemas hardware, subsistemas mecánicos, subsistemas sensoriales, etc. Por tanto, la primera tarea del análisis del sistema software será conocer el medio en el que se va a desenvolver.

Por ejemplo, en un sistema para la supervisión y control del nivel de gases contaminantes en un garaje, se dispondrán sensores de CO, CO₂ y de otros gases situados en diferentes puntos del garaje. Asimismo, para evacuar los gases se dispondrá de varios extractores situados normalmente próximos a alguno de los sensores. Por otro lado, el sistema debe disponer de una consola situada en la cabina de control en la que se muestra periódicamente la situación y en la que se indican mediante alarmas acústicas situaciones que requieran la intervención del operador (avería en un sensor, avería en

un ventilador, nivel de contaminantes alto e incontrolable, etc.). Si se quiere especificar un sistema que pueda ser utilizado en cualquier tipo de garaje y con cualquier tipo de sensor, es importante establecer un contexto global de funcionamiento del sistema. Aparece de inmediato la necesidad de especificar una función de configuración del sistema para el garaje concreto en el que se instala. Esto es, indicar el número y tipo de sensores, su situación, condiciones de alarma, etc. Todos estos detalles sólo se pueden conocer si se analiza el sistema software en su contexto.

Otro aspecto muy importante en este sentido es el análisis del dominio de la aplicación. Este análisis, como ya se indicó en el apartado del modelado de sistemas, incide directamente no sólo en la terminología a emplear en la especificación del sistema, sino también en la creación de sistemas con una visión más globalizadora y que facilita la reutilización posterior de alguna de sus partes.

2.- IDENTIFICACIÓN DE NECESIDADES

Inicialmente, la actitud del *cliente* es solicitar del sistema todas aquellas funciones de las que en algún momento sintió la necesidad, sin pararse a pensar el grado de utilidad que tendrán en el futuro. La labor del analista es concretar las necesidades que se pueden cubrir con los medios disponibles (potencia de cálculo, capacidad de memoria, capacidad de disco, etc.) y dentro del presupuesto y plazos de entrega asignados a la realización del sistema.

Para realizar esta tarea de análisis es fundamental que el analista mantenga un diálogo constante y fluido con el *cliente*, esto es, con todos aquellos que puedan aportar algo al sistema en cualquiera de sus facetas. De todos ellos, el analista deberá recoger y estudiar sus sugerencias para elaborar una propuesta que satisfaga a todos. Como resultado de esta labor deben quedar descartadas aquellas funciones muy costosas de desarrollar y que no aportan gran cosa al sistema. Además, para aquellas necesidades que tengan un cierto interés y que requieran más recursos de los disponibles, el analista tendrá que aportar alguna solución aunque sea incompleta, que encaje dentro de los presupuestos del sistema. Por ejemplo, suele ser frecuente la necesidad de acotar la cantidad de información que se necesita guardar para realizar posteriormente estadísticas. Una actitud permisiva en este sentido puede dar lugar a megabytes de información que no sirven para nada.

Desde luego, las necesidades que finalmente se identifiquen deberán estar consensuadas por todos aquellos que junto al analista hayan participado en

48 Introducción a la Ingeniería de Software

el análisis. Es labor del analista convencer a todos de que la solución adoptada es la mejor con los medios disponibles y nunca tratar de imponer su criterio a toda costa.

3.- ANÁLISIS DE ALTERNATIVAS. ESTUDIO DE VIABILIDAD

En esta tarea aparece de forma evidente el enfoque de ingeniero de software del que debe estar impregnada la actividad del analista. En principio existen infinitas soluciones a un mismo problema. La labor del analista se debe centrar en buscar aquella alternativa que cubra las necesidades reales detectadas en la tarea anterior y que tenga en cuenta su viabilidad tanto técnica como económica.

Cuando no es posible con un enfoque determinado encontrar un modelo que cubra todas las necesidades de una forma satisfactoria, se deben buscar soluciones alternativas. Con cada una de las alternativas planteadas es necesario realizar su correspondiente estudio de viabilidad. Estos estudios constituirán una base fundamental para la toma de decisión sobre la alternativa finalmente elegida.

La realización de esta tarea no siempre resulta necesaria. Solamente en aquellos sistemas cuya complejidad, alto presupuesto o dificultad intrínseca así lo aconsejen, se deberá realizar un análisis completo de diversas alternativas.

4.- ESTABLECIMIENTO DEL MODELO DEL SISTEMA

Según se van obteniendo conclusiones de las tareas anteriores, estas se deben ir plasmando en el modelo del sistema. Así, el modelo se irá perfilando a medida que se avanza en el estudio de las necesidades y las soluciones adoptadas. Probablemente, el resultado final será un modelo del sistema global jerarquizado en el que aparecerán subsistemas que a su vez tendrán que ser desarrollados hasta concretar suficientemente todos los detalles del sistema que se quieren especificar.

Para la elaboración del modelo se deberán utilizar todos los medios disponibles, desde procesadores de texto hasta las más sofisticadas herramientas CASE, pasando por las más diversas herramientas gráficas. Todo ello debe contribuir a simplificar la elaboración del modelo y a facilitar la comunicación entre analista, *cliente* y diseñador. En el próximo apartado se detallan la mayoría de las notaciones que se utilizan habitualmente para desarrollar el modelo del sistema.

5.- ELABORACIÓN DEL DOCUMENTO DE ESPECIFICACIÓN DE REQUISITOS

El resultado final del análisis será el documento de especificación de requisitos. Este documento debe recoger absolutamente todas las conclusiones del análisis. Evidentemente, una parte fundamental de este documento será el modelo del sistema y precisamente es en este documento donde se debe ir perfilando su elaboración a lo largo de todo el análisis. Existen distintas propuestas de organización de este documento que serán estudiadas en un próximo apartado de este mismo tema.

Es muy importante tener en cuenta que este documento será el que utilizará el diseñador como punto de partida de su trabajo. Asimismo, este documento también es el encargado de fijar las condiciones de validación del sistema una vez concluido su desarrollo e implementación. Todo ello indica la transcendencia que tiene su elaboración y el cuidado que se debe poner para que sea útil tanto al *cliente* como al diseñador.

6.- REVISIÓN CONTINUADA DEL ANÁLISIS

Desgraciadamente la labor de análisis no acaba al dar por finalizada la redacción del documento de especificación de requisitos. A lo largo del desarrollo del sistema y según aparecen problemas en las etapas de diseño e implementación se tendrán que modificar alguno de los requisitos del sistema. Tampoco resulta muy raro que se modifiquen los requisitos debido a cambios de criterio del *cliente* debidos a los más diversos motivos. Todo ello implica que se debe proceder a una revisión continuada del análisis y de su documento de especificación de requisitos según se producen los cambios.

Si se prescinde de esta última tarea, cosa desgraciadamente bastante habitual, se corre el peligro de realizar un sistema del que no se tenga ninguna especificación concreta y en consecuencia tampoco ningún medio de validar si es o no correcto el sistema finalmente desarrollado.

2.3 Notaciones para la especificación

La especificación será fundamentalmente una descripción del modelo del sistema que se pretende desarrollar. Para un mismo modelo conceptual, dependiendo de la notación o notaciones (texto, gráficos, tablas, etc.) que se empleen para su descripción, se obtendrán distintas especificaciones. A priori, todas las especificaciones deberían ser equivalentes independientemente de la notación empleada. Sin embargo, el empleo de

50 Introducción a la Ingeniería de Software

la notación más adecuada en cada caso redundará en una mejor especificación del sistema. En general, siempre será preferible utilizar una tabla o una notación gráfica que el texto que las pueda describir.

Las diversas metodologías de análisis, desarrolladas a lo largo de años, han ido estableciendo distintas notaciones para describir el modelo global del sistema o ciertos aspectos del mismo. Debido al desarrollo paralelo de las distintas metodologías es frecuente que una misma notación tenga significados distintos en cada metodología (por ejemplo: un círculo puede significar en unos casos una transformación de datos y en otros representar un estado del sistema), o que para un mismo concepto se empleen distintas notaciones (por ejemplo, un estado del sistema puede indicarse mediante un círculo o mediante un rectángulo). Para evitar malas interpretaciones en la especificación, es fundamental conocer el significado concreto de la notación que se utiliza.

En cualquier caso, la notación o notaciones empleadas deberán ser fáciles de entender por el *cliente*, el usuario, y en general por todos aquellos que puedan participar en el análisis y desarrollo del sistema. No obstante, el empleo de una notación u otra dependerá de la complejidad y tipo de sistema a desarrollar (gestión, control, comunicaciones, etc.), de la metodología empleada, de las herramientas disponibles (procesador de textos, software para gráficos, entornos CASE, etc.) e incluso de las preferencias del analista o del *cliente*.

A continuación se hace un repaso a las notaciones que se utilizan más frecuentemente para especificar los sistemas software.

2.3.1 Lenguaje natural

La notación más inmediata que se puede emplear para realizar una especificación es el lenguaje natural que habitualmente empleamos para comunicarnos. Mediante explicaciones más o menos precisas y más o menos exhaustivas es posible describir prácticamente cualquier cosa.

Para sistemas de una complejidad pequeña, es suficiente el lenguaje natural como notación para realizar su especificación y es la manera en que casi siempre se suele realizar. Cuando la complejidad del sistema es mediana o grande, resulta prácticamente imposible utilizar sólo el lenguaje natural. Los principales inconvenientes, como ya ha sido mencionado, son las imprecisiones, repeticiones e incorrecciones que se pueden producir. Además, existen otras notaciones que son mucho más adecuadas para sintetizar aspectos concretos del sistema y que son capaces de ofrecer una

visión más globalizadora, lo que simplifica mucho la especificación del sistema.

El lenguaje natural será la notación que se empleará siempre que sea necesario aclarar cualquier aspecto concreto del sistema, que no sean capaces de reflejar el resto de notaciones. En cualquier caso, siempre que se utilice el lenguaje natural es muy importante organizar y estructurar los requisitos recogidos en la especificación. La mejor manera de lograrlo, es concebir cada uno de los requisitos como una cláusula de un contrato entre el analista y el *cliente*. Así, se pueden agrupar los requisitos según su carácter:

- 1.- Funcionales
- 2.- Calidad
- 3.- Seguridad
- 4.- Fiabilidad
- 5.- ...

y dentro de cada grupo, establecer y numerar correlativamente las cláusulas de distintos niveles y subniveles que estructuren los requisitos:

1.- Funcionales

- R.1.1: Modos de funcionamiento
- R.1.2: Formatos de entrada
- R.1.3: Formatos de salida
- R.1.4.

Para limitar las imprecisiones y ambigüedades propias del lenguaje natural, es conveniente establecer ciertas reglas de uso del lenguaje, que impidan, en la medida de lo posible, un uso retórico o impreciso. Evidentemente, una especificación nunca debe ser una novela y es preferible que se utilicen frases siempre con la misma construcción que tengan siempre la misma interpretación.

El *lenguaje natural estructurado* es una notación más formal que el lenguaje natural. Fundamentalmente se trata de establecer ciertas reglas para la construcción de las frases en las que se especifica una acción de tipo secuencia, condición o iteración. No se trata de obligar a que todas las especificaciones escritas en español utilicen las mismas frases, lo que sería imposible. Se trata de lograr que dentro de una misma especificación todas las frases se construyan de igual manera. Por ejemplo, en lugar de escribir en distintos apartados de la especificación:

52 Introducción a la Ingeniería de Software

Cuando se teclee la clave 3 veces mal, se debe invalidar la tarjeta ...

Cuando el saldo sea menor de cero pesetas se aplicará un interés

Para los clientes mayores de 65 años se establecerá un descuento de

es mejor que todas las frases se construyan de igual manera:

SI se teclea la clave 3 veces mal ENTONCES invalidar tarjeta ...

SI el saldo es menor de cero pesetas ENTONCES el interés será ...

SI el cliente es mayor de 65 años ENTONCES el descuento será ...

Construcciones similares se pueden utilizar para la iteración y la secuencia. No se trata de utilizar siempre una misma palabra clave (SI, REPETIR, etc.), sino emplear la misma construcción en todas las frases, al menos, de una misma especificación.

2.3.2 Diagramas de flujo de datos

Esta notación está asociada a la metodología de análisis estructurado, que fue desarrollada a lo largo de la década de los 70 [DeMarco79]. El enfoque del análisis estructurado es considerar que un sistema software se puede modelar mediante el flujo de datos que entra al sistema, las transformaciones que se realizan con los datos de entrada y el flujo de datos que se produce a la salida como resultado de la transformación. Esta filosofía se puede aplicar con total generalidad para modelar los sistemas de información, en los que efectivamente siempre se producen sucesivas transformaciones de los datos de entrada (salario, ingresos, precios, etc.) para obtener los datos de salida (nómina, liquidaciones, facturas, etc.).

Con los diagramas de flujo de datos (DFD) se pueden modelar de forma muy sencilla las transformaciones y los flujos de datos utilizando una notación gráfica. Como se muestra en la figura 2.1, una flecha indica un flujo de datos en el sentido en el que apunta su cabeza. Con cada flecha de un DFD se tienen que detallar sus Datos mediante un nombre o una descripción. Un proceso o transformación de datos se indica con un círculo o burbuja. Dentro del círculo se nombra o describe el proceso que realiza. Una línea doble indica un almacén de datos. Estos datos almacenados pueden ser guardados por uno o varios procesos y asimismo pueden ser consultados o consumidos por uno o más procesos. Entre las dos líneas se

detalla el nombre o descripción de los datos que guarda el almacén. Finalmente, un rectángulo representa una entidad externa al sistema software (el usuario, otro programa, un dispositivo hardware, etc.), que produce o consume sus datos.

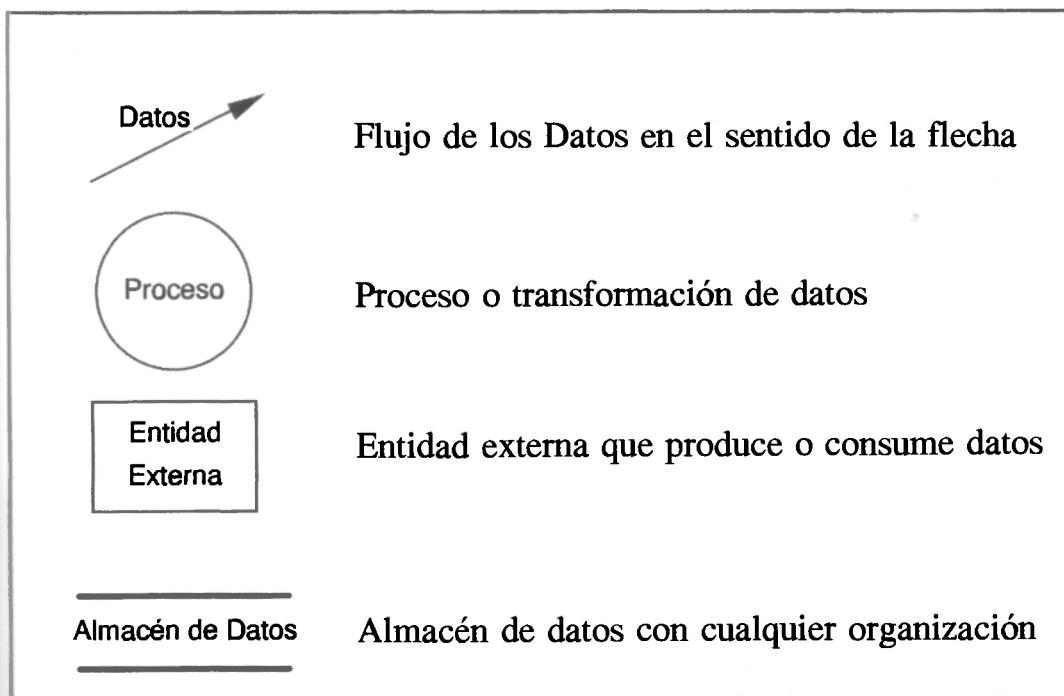


Figura 2.1. Notación DFD básica

Como ejemplo de diagrama de DFD, supongamos el siguiente enunciado de problema: Se trata de especificar un sistema para el control de acceso a un recinto mediante una tarjeta y una clave de acceso. La tarjeta magnética lleva grabada los datos personales del propietario y la clave de acceso. El sistema dispondrá de un teclado numérico para introducir la clave y un display o pantalla para escribir los mensajes. El sistema registrará todos los intentos de accesos aunque sean fallidos por no teclear correctamente la clave o por tratar de utilizar un tipo de tarjeta incorrecta.

En la figura 2.2 se muestra el DFD del sistema global o DFD de contexto. En este ejemplo, las entidades externas son el lector de las tarjetas magnéticas, el teclado para introducir la clave, la pantalla o display y el dispositivo para la apertura de la puerta. El lector de tarjetas suministra a nuestro sistema los datos grabados en las mismas (Nombre y apellidos, clave de acceso, etc.) y mediante el teclado se lee la clave para la correspondiente comprobación. La pantalla recibe de nuestro sistema los

54 Introducción a la Ingeniería de Software

mensajes para poder establecer el diálogo con la persona que trata de acceder al recinto y el dispositivo de apertura de la puerta, las órdenes para abrirla.

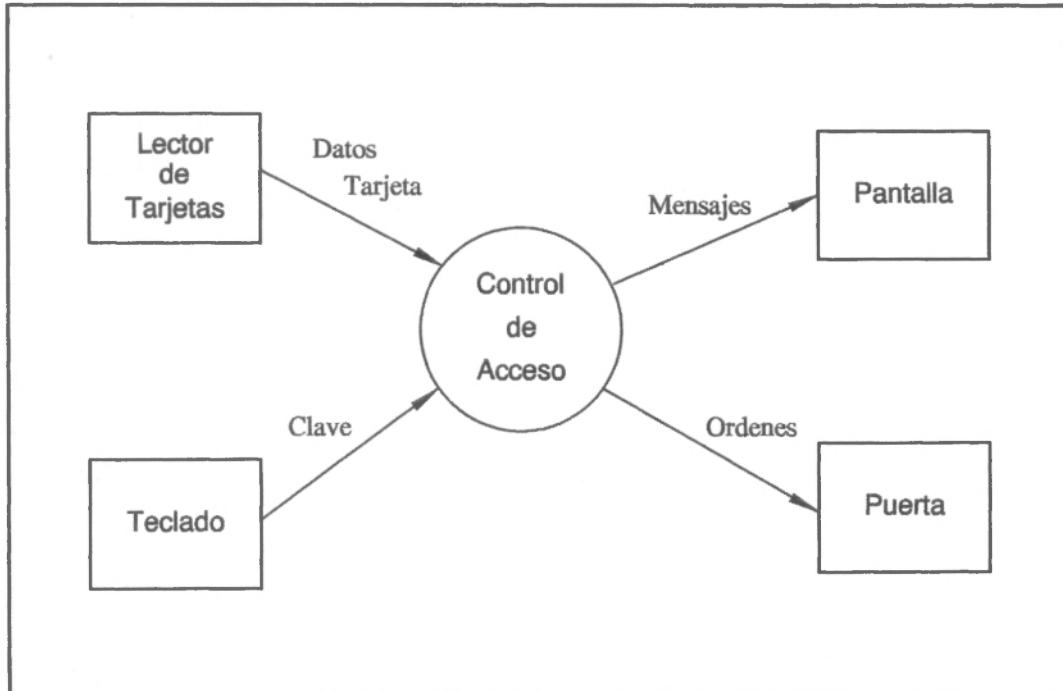


Figura 2.2. DFD de contexto para el sistema de control de acceso

El DFD de contexto nunca es suficiente para describir el modelo del sistema que se trata de especificar. Para refinar el modelo, los DFD se usan de forma jerarquizada por niveles. El DFD de contexto se denomina nivel 0. A continuación, cada proceso o burbuja se puede "explotar" y mostrar cómo se organiza interiormente, mediante otros DFD. En el caso de un diagrama de contexto, sólo se puede explotar un único proceso que es el proceso global del sistema y su explosión dará lugar al único DFD de nivel 1.

En el ejemplo del sistema para el control de acceso, la figura 2.3 muestra el DFD de nivel 1. Como se puede observar, los flujos de entrada y salida al DFD de nivel 1 corresponden exactamente con los que tiene el proceso **Control de Acceso** del diagrama de contexto, esto es: Datos Tarjeta, Clave, Mensajes y Ordenes.

La elaboración del nuevo DFD es el resultado de la labor de análisis del sistema que se está especificando. Teniendo en cuenta las necesidades del *cliente* expresadas en el enunciado, se ve que hace falta utilizar un almacén

para guardar los datos de la tarjeta que intenta acceder, y si finalmente se consigue o no. En la figura 2.3, este almacén se ha denominado **Información de Accesos**.

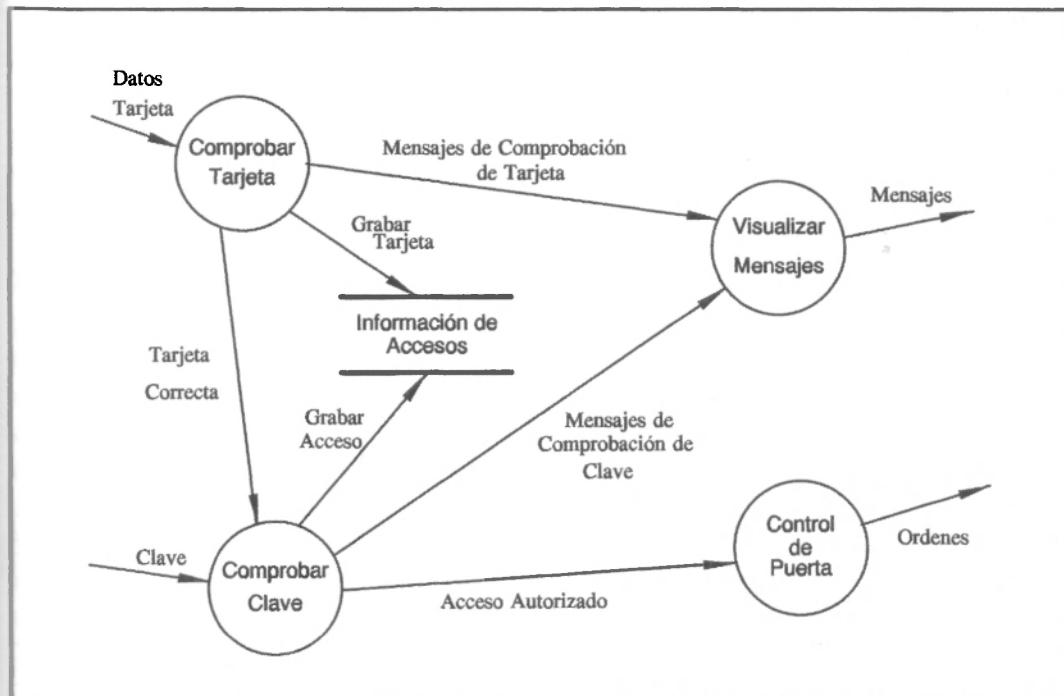


Figura 2.3. DFD de nivel 1 para el sistema de control de acceso

También, de acuerdo con el *cliente*, el proceso **Comprobar Tarjeta** será el encargado de verificar si la tarjeta introducida por el lector es correcta. Esta verificación se reduce a comprobar si los datos que tiene grabados la tarjeta lo están en el orden y formato adecuado y con los controles de seguridad establecidos. En cualquier caso se grabarán todos los datos de la tarjeta leída para el control de acceso.

El proceso **Comprobar Clave** es el encargado de verificar si la clave tecleada es la misma que está grabada en la tarjeta. Si la clave es correcta se autoriza el acceso. En cualquier caso se graba el resultado del intento de acceso. El proceso **Visualizar Mensajes** es el encargado de sacar por pantalla los mensajes que le envían los dos anteriores para establecer el diálogo con la persona que trata de acceder al recinto. Finalmente, el proceso **Control de Puerta** es el encargado de ejecutar y temporizar la orden de apertura de la puerta.

El refinamiento del DFD de nivel 1 puede continuar con cada uno de los procesos que en él aparecen. Una forma de facilitar la identificación de los

56 Introducción a la Ingeniería de Software

sucesivos DFD es numerar de forma correlativa los distintos procesos antes de su refinamiento. El DFD resultante de la explosión de un proceso x tendrá el número x del proceso explotado. Los procesos hijos que aparecen en él se numerarán de 1 en adelante, en la forma $x.1$, $x.2$, ... El único proceso del diagrama de contexto no lleva numeración, y el único diagrama resultante de nivel 1 se designa como DFD 0. De esta manera los sucesivos diagramas se numerarán de la forma:

- Nivel 0 Diagrama de contexto
- Nivel 1 DFD 0
- Nivel 2 DFD 1 hasta DFD n
 de los procesos 1 hasta n del nivel 1
- Nivel 3 DFD 1.1 hasta DFD 1. i
 de los procesos 1.1 hasta 1. i del nivel 2
- DFD 2.1 hasta DFD 2. j
 de los procesos 2.1 hasta 2. j del nivel 2
-
- DFD $n.1$ hasta DFD $n.k$
 de los procesos $n.1$ hasta $n.k$ del nivel 2
- Nivel 4 DFD 1.1.1 hasta DFD 1.1. x
-
- DFD 1.i.1 hasta DFD 1.i. z
-
- ... etc. ...

En todos ellos los flujos de datos de entrada y salida antes de la "explosión" del proceso debe coincidir con los flujos de entrada y salida del DFD resultado de la explosión o refinamiento. La ventaja de esta notación es su simplicidad lo que permite que sea fácilmente entendida por todos: *cliente*, *usuario*, *analista*, etc. En los ejemplos que se desarrollan al final de este capítulo se puede comprobar todo esto.

Aunque se podría continuar refinando de manera indefinida, a partir de un determinado nivel y dependiendo de la complejidad del sistema que se especifica, no tiene sentido subdividir más los procesos. En este punto es conveniente describir cada uno de ellos utilizando otra notación distinta, por ejemplo, el *lenguaje natural estructurado* que ya ha sido presentado o pseudocódigo que se presentará más adelante en este mismo apartado.

Por otro lado, también es necesario describir las estructuras de cada uno de los flujos de datos y las estructuras de los datos guardados en cada uno de los almacenes de datos. Esta descripción también se puede realizar mediante *lenguaje natural estructurado* o bien utilizando alguna de las notaciones para la descripción de datos que se estudiarán en un próximo apartado. En líneas generales, se tratará de describir mediante una tabla, todos los elementos básicos de información que constituyen los diversos datos que se manejan en los distintos DFD del modelo del sistema. Por ejemplo, en la figura 2.3, los datos Clave, Tarjeta Correcta o Información de Accesos.

Para finalizar esta introducción a los diagramas de flujo de datos, conviene precisar algunos aspectos. Principalmente, los DFD sirven para establecer un modelo conceptual del sistema que facilita la estructuración de su especificación. El modelo así desarrollado es fundamentalmente estático dado que refleja los procesos necesarios para su desarrollo y la interrelación que existe entre ellos. Mediante un DFD nunca se puede establecer la dinámica o secuencia en la que se ejecutarán los procesos. Por ejemplo, en la figura 2.3 no se establece ningún orden concreto entre los flujos de salida del proceso **Comprobar Tarjeta** y por tanto, se podrá producir primero **Grabar Tarjeta**, después **Mensaje de Comprobación de Tarjeta** y finalmente **Tarjeta Correcta** o en cualquier otro orden.

La única premisa de carácter dinámico que se puede establecer en un DFD es que en ellos se utiliza un modelo abstracto de computo del tipo flujo de datos [Cerrada00]. Así, los procesos funcionan de forma semejante a una "Red de operadores" y en cada ejecución se utiliza y consume un elemento de cada uno de los flujos de datos de entrada y se generan los elementos de datos previstos para cada uno de los flujos de salida. En el caso de los almacenes de datos, los elementos se utilizan pero no se consumen y pueden ser utilizados posteriormente.

2.3.3 Diagramas de transición de estados

El número de estados posibles que se pueden dar en un sistema software crece de una forma exponencial con su complejidad. Incluso para sistemas bastante simples el número de estados es muy grande. Hay que tener en cuenta que cada vez que se modifica una variable, se evalúa una condición o el término de una expresión se produce un cambio de estado en el sistema. Todos estos estados y las sucesivas transiciones entre ellos configuran la dinámica del sistema que se produce mientras se está ejecutando.

58 Introducción a la Ingeniería de Software

Para realizar la especificación de un sistema, de todos sus estados posibles, sólo es importante resaltar aquellos que tienen cierta transcendencia desde un punto de vista funcional. El diagrama de transición de estados es la notación específica para describir el comportamiento dinámico del sistema a partir de los estados elegidos como más importantes. Por tanto, es una notación que se complementa perfectamente con los diagramas de flujo de datos y que ofrece otro punto de vista del sistema que en la mayoría de los casos resulta imprescindible.

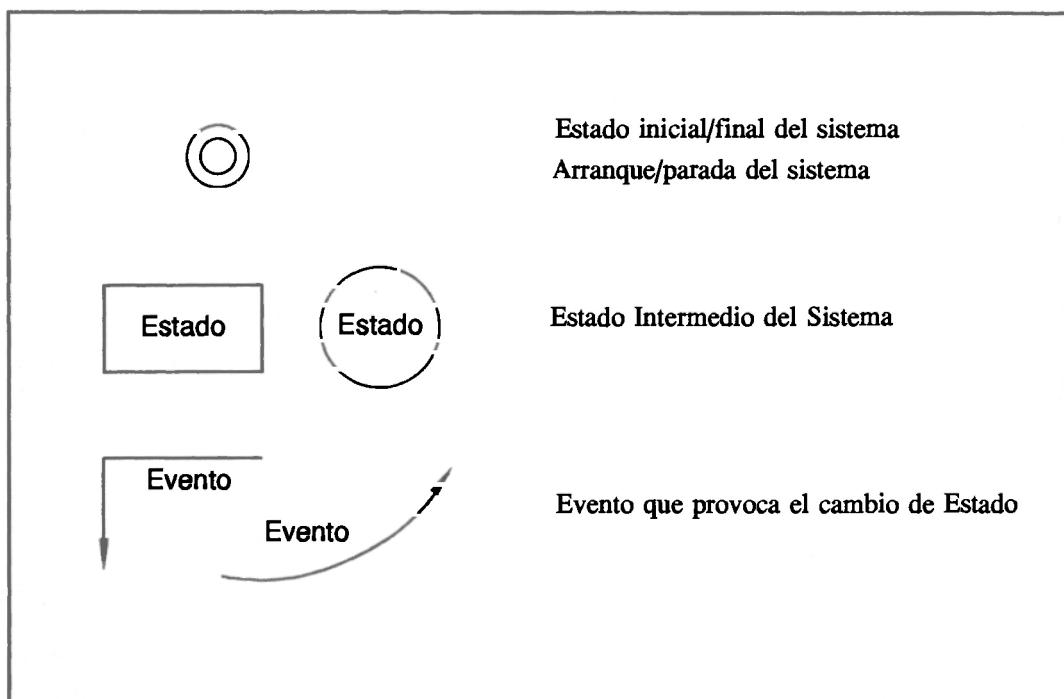


Figura 2.4. Notación básica de Diagramas de Estado

La notación básica para elaborar los diagramas de estados que se emplea habitualmente se muestra en la figura 2.4. Como se puede observar en la figura, se sugieren dos formas distintas para representar un estado: mediante un rectángulo o mediante un círculo. En ambas notaciones se indica mediante el nombre que encierran en su interior el estado concreto que representan. La razón por la que existen estas dos posibilidades es su procedencia. Los diagramas de estado mediante círculos son los más utilizados para representar autómatas de estados finitos (autómatas mecánicos, circuitos eléctricos o electrónicos y automatismos en general). Sin embargo, para evitar la confusión con la notación empleada en la elaboración de los DFD, para la especificación de un sistema software se suele emplear el rectángulo.

Los eventos que provocan el cambio de estado se indican mediante una flecha dirigida desde el estado viejo al nuevo estado. Por razones estéticas se emplean flechas en forma de arco cuando se utilizan círculos y formadas por tramos rectos cuando se utiliza un rectángulo.

Cuando se considere interesante destacar los estados inicial y final del sistema, se puede hacer mediante dos círculos concéntricos.

En la figura 2.5 se muestra el diagrama de estados del sistema de acceso. En este caso el sistema es muy sencillo y los estados posibles son solamente tres: Esperar Tarjeta, Esperar Clave y Permitir Acceso. Respecto a los eventos que provocan los cambios de estado, se deberán utilizar otras notaciones para detallar de manera más precisa el significado concreto de cada uno de ellos. A pesar de todo, este diagrama modela de manera sencilla y precisa la dinámica del sistema y se complementa perfectamente con los DFD del apartado anterior para lograr una especificación lo más completa y exacta posible.

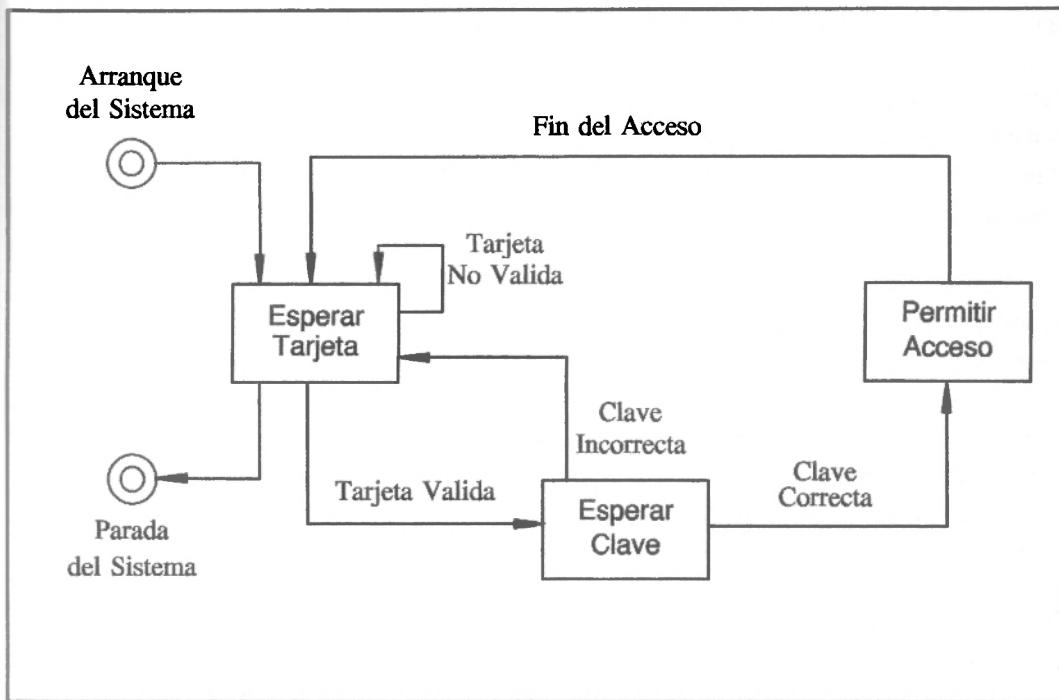


Figura 2.5. Diagrama de Estados del Sistema de Acceso

En los ejemplos del final de este tema se muestra otro diagrama de estado utilizando la notación alternativa.

60 Introducción a la Ingeniería de Software

Cuando la complejidad del sistema así lo requiera, serán necesarios varios diagramas de estados. Además del diagrama de estados global del sistema, pueden ser necesarios otros diagramas adicionales para especificar la dinámica de determinados procesos significativos del sistema. En estos casos, es importante elegir cuidadosamente los estados y las transiciones que faciliten la comprensión del sistema sin profundizar en detalles de diseño que deberán ser concretados posteriormente.

2.3.4 Descripciones funcionales. Pseudocódigo

Los requisitos funcionales son una parte muy importante de la especificación de un sistema. Por tanto, es esencial que las descripciones funcionales se realicen empleando una notación precisa y que no se preste a ambigüedades. Como mínimo se deberá utilizar un *lenguaje natural estructurado* y siempre que sea posible es aconsejable utilizar una notación algo más precisa que denominaremos pseudocódigo.

Entenderemos por pseudocódigo una notación basada en un lenguaje de programación estructurado (Pascal, Modula-2, Ada, etc) del que se excluyen todos los aspectos de declaración de constantes, tipos, variables y subprogramas. El pseudocódigo maneja datos en general, no tiene una sintaxis estricta, como sucede con los lenguajes de programación y se pueden incluir descripciones en lenguaje natural siempre que se considere necesario, como parte del pseudocódigo.

Hay que tener cuidado al emplear pseudocódigo para realizar una especificación software. Si se detalla o "refina" demasiado una descripción funcional mediante pseudocódigo se puede estar realizando el diseño del sistema e incluso su codificación. De hecho, existen notaciones similares al pseudocódigo que están pensadas para realizar el diseño. En este caso, se habla de lenguaje de descripción de programas (PDL - Program Description Language). Estos lenguajes para el diseño, se estudiarán en el tema de la segunda unidad didáctica dedicado a las técnicas generales de diseño.

Es importante destacar que aunque se utilicen notaciones parecidas (pseudocódigo o PDL), los objetivos que se persiguen son muy diferentes. Cuando se especifica se trata de indicar cual es la naturaleza del problema a resolver sin detallar cómo se debe resolver. En este sentido, en una especificación no se debería establecer ninguna forma concreta de organización de la información ni tampoco ninguna propuesta concreta de los procedimientos de resolución de los problemas.

La notación de las estructuras básicas que se pueden emplear en el pseudocódigo tendrán los siguientes formatos:

A.- Selección:

```
SI <Pseudocódigo de la Condición> ENTONCES  
    <Pseudocódigo>  
SI-NO  
    <Pseudocódigo>  
FIN-SI
```

donde <Pseudocódigo de la Condición> es la condición que se debe verificar para efectuar el <Pseudocódigo> indicado después de ENTONCES o en caso contrario efectuar el <Pseudocódigo> indicado después de SI-NO. Por ejemplo, para el sistema de control de acceso tendríamos:

```
SI Datos Tarjeta son Correctos ENTONCES  
    Guardar Datos Tarjeta;  
    Comprobar Clave  
SI-NO  
    Devolver Tarjeta  
FIN-SI
```

Evidentemente, cuando se necesiten se pueden anidar varias selecciones unas dentro de otras.

B.- Selección por casos:

```
CASO <Especificación del elemento selector>  
    SI-ES <Descripción del caso 1> HACER <Pseudocódigo>;  
    SI-ES <Descripción del caso 2> HACER <Pseudocódigo>;  
    ....  
    SI-ES <Descripción del caso N> HACER <Pseudocódigo>;  
    OTROS <Pseudocódigo>  
FIN-CASO
```

donde <Especificación del elemento selector> determina el elemento con el que se efectúa la selección. Este elemento sólo podrá tomar un número acotado de valores distintos y según tome uno u otro se efectuará el <Pseudocódigo> correspondiente. El <Pseudocódigo> indicado después de OTROS se realizará cuando el valor del elemento no coincida con ninguno

62 Introducción a la Ingeniería de Software

de los previstos. Por ejemplo, para seleccionar en un cajero automático el tipo de operaciones que se pueden realizar dependiendo del tipo de tarjeta introducida, se tendría:

```
CASO Tipo de Tarjeta
    SI-ES Visa      HACER Operaciones a crédito;
    SI-ES CuatroB   HACER Operaciones en cuenta;
    SI-ES Express   HACER Operaciones con confirmación;
    OTROS Devolver Tarjeta
FIN-CASO
```

C.- Iteración con pre-condición

```
MIENTRAS <Pseudocódigo de la condición> HACER
    <Pseudocódigo>
FIN-MIENTRAS
```

que efectúa el <Pseudocódigo> mientras que se verifique la condición indicada por <Pseudocódigo de la condición>.

D.- Iteración con post-condición

```
REPETIR
    <Pseudocódigo>
HASTA <Pseudocódigo de la condición>
```

que efectúa el <Pseudocódigo> hasta que se verifique la condición indicada por <Pseudocódigo de la condición>. Por ejemplo, en el sistema de control de acceso, para comprobar la clave se pueden permitir hasta tres intentos:

```
REPETIR
    Leer Clave
HASTA Clave correcta o Tres intentos incorrectos
```

E.- Número de iteraciones conocido

```
PARA-CADA <Especificación del elemento índice> HACER
    <Pseudocódigo>
FIN-PARA
```

donde <Especificación del elemento índice> indica el número de veces que se efectúa el <Pseudocódigo>. Por ejemplo, si se quieren listar todos los accesos realizados en el sistema de control de acceso, se podría indicar:

```
PARA-CADA Acceso registrado HACER
    Escribir datos del acceso;
    Escribir resultado del acceso
FIN-PARA
```

2.3.5 Descripción de datos

La descripción de los datos constituye otro aspecto fundamental de una especificación software. Se trata de detallar la estructura interna de los datos que maneja el sistema. Lo mismo que sucede con la descripción funcional, cuando se realiza una descripción de datos no se debe descender a detalles de diseño o codificación. Sólo se deben describir aquellos datos que resulten relevantes para entender qué debe hacer el sistema.

En principio, también los datos se pueden describir utilizando el lenguaje natural, sin embargo es mejor emplear notaciones más específicas. Una posible solución es utilizar la notación usada para definir tipos de datos en alguno de los lenguajes estructurados habituales (Pascal, Modula-2 o Ada). Sin embargo, esta solución tiene como inconveniente fundamental la gran dependencia de una sintaxis concreta y la necesidad de detallar aspectos propios de la fase de diseño o codificación tales como el tamaño o el tipo concreto de cada elemento.

La notación adoptada en la metodología de análisis estructurado es lo que se conoce como *diccionario de datos* [Yourdon90]. Esta notación es bastante más informal que cualquier definición de tipos de un lenguaje de programación pero con ella se logra una descripción de los datos suficientemente precisa para la especificación de requisitos.

Aunque existen diversos formatos posibles de *diccionario de datos* según los distintos autores o herramientas CASE que lo incorporen, en esencia todos los formatos aconsejan que al menos se pueda describir la siguiente información para cada dato:

64 Introducción a la Ingeniería de Software

A.- Nombre o nombres:

Será la denominación con la que se utilizará este dato en el resto de la especificación. Puede ocurrir que para mayor claridad de la especificación se utilicen distintos nombres para datos que tienen una misma descripción.

B.- Utilidad:

Se indicarán todos los procesos, descripciones funcionales, almacenes de datos, etc. en los que se utilice el dato.

C.- Estructura:

Se indicarán los elementos de los que está constituido el dato, utilizando la siguiente notación:

A + B	Secuencia o concatenación de los elementos A y B
[A B]	Selección entre los distintos elementos A o bien B
{ A } ^N	Repetición N veces del elemento A (se omite N si es indeterminado)
(A)	Opcionalmente se podrá incluir el elemento A
/ descripción /	Descripción en lenguaje natural como comentarios
=	Separador entre el nombre de un elemento y su descripción

Para ilustrar una descripción de datos, a continuación se detallan algunos de los datos del sistema de control de acceso:

Nombres: Datos Tarjeta,
Grabar Tarjeta

Utilidad: Proceso: Comprobar Tarjeta como entrada
Proceso: Comprobar Tarjeta como salida
Almacén de datos: Información de Accesos como entrada
Entidad externa: Lector de Tarjetas como salida

Estructura: Nombre +
 Primer Apellido +
 Segundo Apellido +
 Nivel de Acceso +
 Clave +
 (Código empresa)

Nombre =	/Ristra con 10 caracteres máximo/
Primer Apellido =	/Ristra con 20 caracteres máximo/
Segundo Apellido =	/Ristra con 20 caracteres máximo/
Nivel de Acceso =	[0 1 2]
Código empresa =	[101 102 ... 199]

Nombre: Clave

Utilidad: Proceso: Comprobar Clave como entrada

Estructura: { Dígito } ⁴

Dígito =	/ Carácter numérico del 0 al 9 /
----------	----------------------------------

2.3.6 Diagramas de modelo de datos

Si un sistema maneja una cierta cantidad de datos relacionados entre sí, como sucede habitualmente en los sistemas de información, es necesario establecer una organización que facilite las operaciones que se quieren realizar con ellos. Por ejemplo, si tenemos datos de países, regiones, ciudades, etc. y entre ellos existen ciertas relaciones tales como: una ciudad es capital de un determinado país o un país tiene embajada en una determinada ciudad o una ciudad está en una determinada región, etc., inmediatamente surge la necesidad de establecer una organización entre los datos que simplifique y agilize su utilización. Precisamente dicha organización es la que configura la estructura de la base de datos que utilizará el sistema.

Es fundamental que en la especificación se establezca el modelo de datos que se considera más adecuado para conseguir todos los requisitos del sistema. Este modelo se conoce como modelo entidad-relación (modelo E-R) y permite definir todos los datos que manejará el sistema junto con las relaciones que se desea que existan entre ellos. Aunque el modelo estará sujeto a revisiones durante las fases de diseño y codificación, como sucede con el resto de la especificación, sin embargo, es un punto de partida imprescindible para comenzar cualquier diseño.

66 Introducción a la Ingeniería de Software

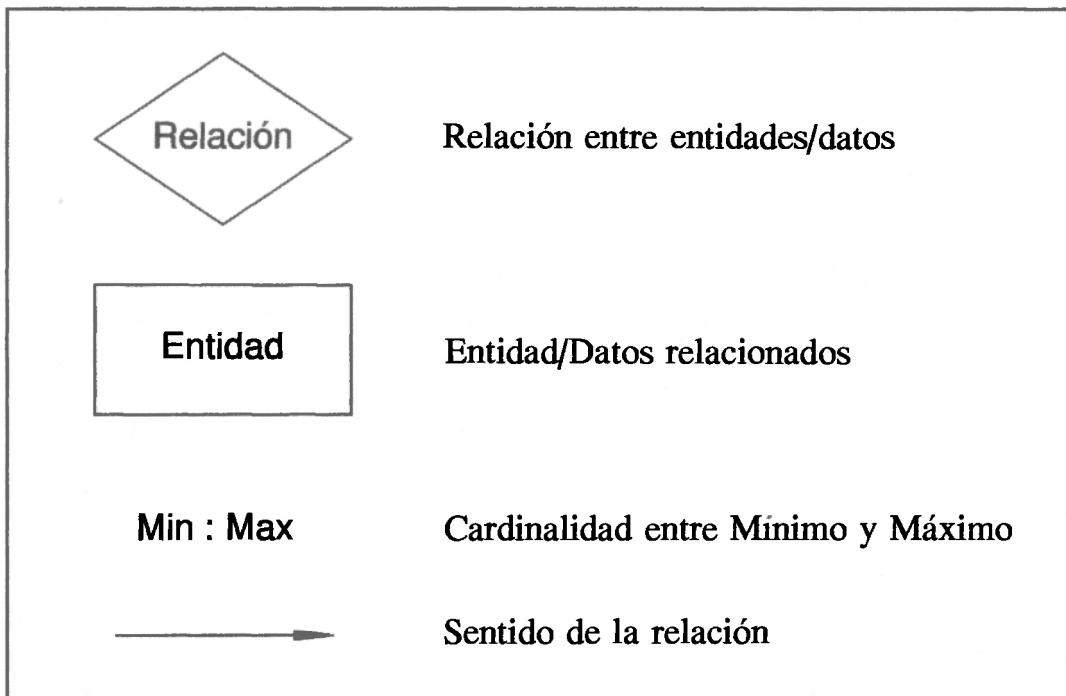


Figura 2.6. Notación básica de diagramas de datos

Existen diversas propuestas de notación para realizar los diagramas E-R, aunque entre todas ellas las diferencias son mínimas. En la figura 2.6 se recogen los elementos de la notación más habituales. Las entidades o datos que se manejan en el diagrama se encierran dentro de un rectángulo, por ejemplo: ciudad, país, región, etc. Las relaciones se indican mediante un rombo que encierra el tipo de relación, por ejemplo: es capital de, tiene embajada en, está en, etc.

Las entidades y la relación que se establece entre ellas se indican uniéndolas todas mediante líneas. Opcionalmente, se puede indicar el sentido de la relación con una flecha. Es conveniente utilizar la flecha sobre todo cuando únicamente con el nombre de la relación encerrado en el rombo no queda claro cual es su sentido.

Otro elemento fundamental es la cardinalidad de la relación, esto es, entre que valores mínimo y máximo se mueve la relación entre entidades. Por ejemplo, un país puede no tener embajada en ninguna ciudad o tenerlas en muchas. En los diagramas E-R, la cardinalidad se indica según se muestra en la figura 2.7, donde cero es un círculo, uno es una raya perpendicular a la línea de relación y muchos o N son tres rayas en bifurcación. La

cardinalidad se dibuja siempre unida a la segunda entidad de la relación con un símbolo para el valor mínimo y otro para el máximo.

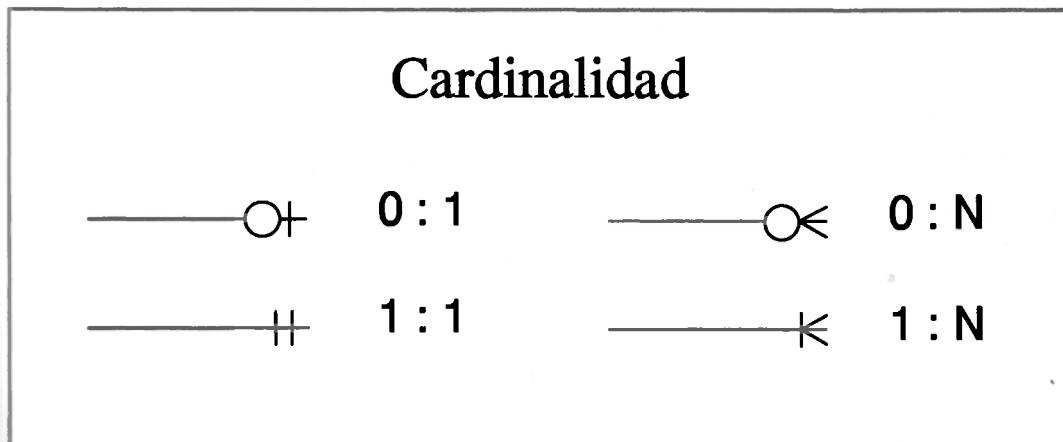


Figura 2.7. Notación para la cardinalidad

Para explicar mejor todo esto, veamos la figura 2.8. Inicialmente la relación "Capital" es ambigua dado que no queda claro si se trata de "es Capital de" o de "tiene como capital". La flecha nos aclara, en este caso, que la relación principal del diagrama es "es Capital". Teniendo en cuenta la cardinalidad unida a la segunda entidad, tenemos la relación: Una ciudad puede ser la capital de ningún país o serlo como máximo de uno.

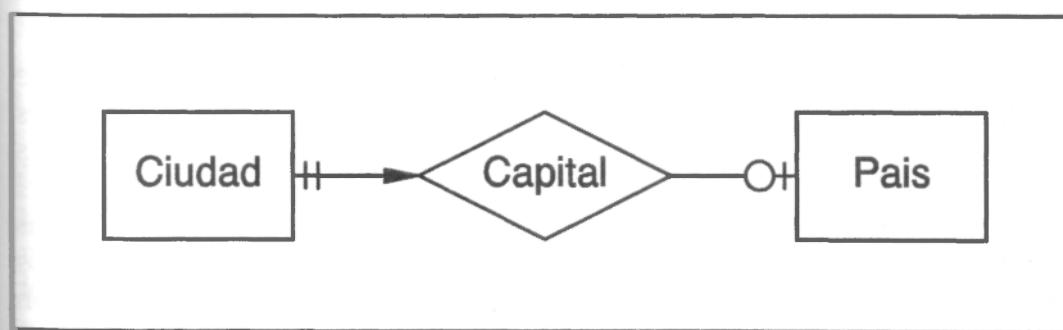


Figura 2.8. Diagrama E-R sencillo

También en el mismo diagrama se indica la relación inversa: Un país tiene como capital una y sólo una ciudad. Esta es la razón de utilizar nombres ambiguos para las relaciones, que se puedan emplear en ambos sentidos y así se prescindirá de la flecha. En la figura 2.9 se tiene el diagrama E-R del sistema completo, cuya interpretación resulta trivial.

68 Introducción a la Ingeniería de Software

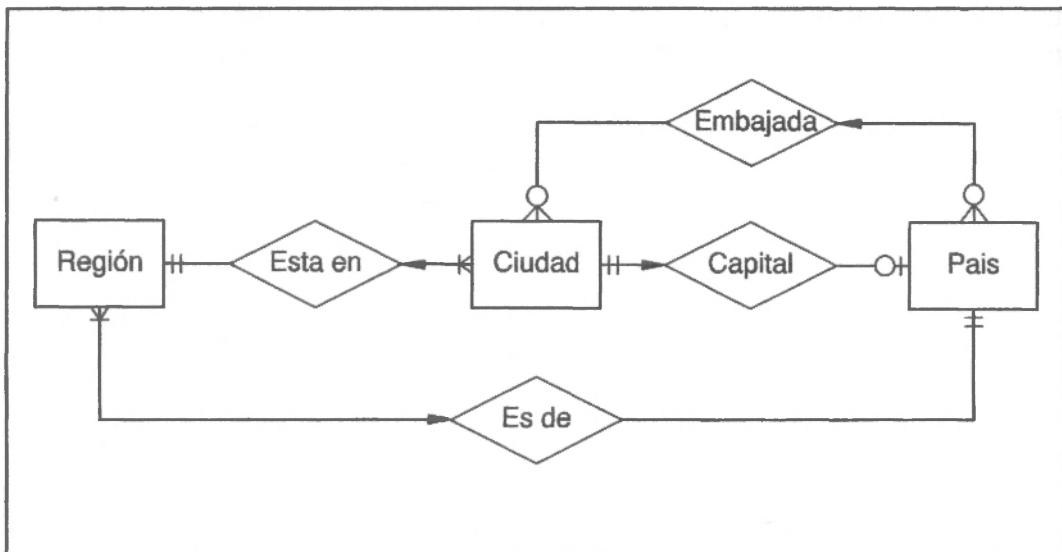


Figura 2.9. Diagrama E-R completo

2.4 Documento de especificación de requisitos

El documento de especificación de requisitos, denominado SOFTWARE REQUIREMENTS DOCUMENT (SRD) o SOFTWARE REQUIREMENTS SPECIFICATION (SRS) en la literatura anglosajona, es el encargado de recoger todo el fruto del trabajo realizado durante la etapa de análisis del sistema de una forma integral en un único documento.

Pueden existir otros documentos previos al SRD, tales como los estudios de viabilidad o los estudios de las diversas alternativas posibles, centrados fundamentalmente en los aspectos de la gestión del proyecto. También es posible, cuando el proyecto es muy amplio y se prolonga mucho en el tiempo, que se elabore un documento con la historia del proyecto y todas las vicisitudes habidas, de las que se podrán sacar enseñanzas en futuros proyectos. Todos estos documentos tienen una utilidad concreta pero no son imprescindibles, sin embargo, el SRD es un documento fundamental y es el punto de partida del desarrollo de cualquier sistema software.

Con carácter general, el SRD debe cubrir todos los objetivos indicados en el apartado 2.2. Además, dado que será un documento que deberá ser revisado con cierta frecuencia a lo largo del desarrollo de la aplicación, es muy conveniente que se redacte de una forma fácil de modificar. Por otro lado, también debe facilitar la labor de verificación del cumplimiento de las especificaciones. Todo esto hace que la mejor manera de redactar este

documento sea en forma de un **contrato** con sus distintas cláusulas organizadas y agrupadas según el carácter de los requisitos.

Son muy numerosas las propuestas de organización del documento SRD. Se puede decir que prácticamente cada autor propone uno distinto. Sin embargo, en líneas generales todas las propuestas son similares. Diversos organismos internacionales tales como IEEE [IEEE84], el Departamento de Defensa norteamericano [DoD88] o la Agencia Espacial Europea [ESA91] han establecido su propia organización de documentos y los imponen a las empresas que contratan para evitar el caos que supondría la coordinación de miles de proyectos cada uno con una organización distinta de la documentación.

A continuación se sugiere un modelo de documento SRD basado en la propuesta de la Agencia Espacial Europea [ESA91]. Este documento tiene un carácter general y en algunos casos no será necesario cumplimentar todos sus apartados. El índice del documento propuesto se recoge en el cuadro 2.1. El contenido de cada apartado debería ser el siguiente:

1. INTRODUCCIÓN

Esta sección debe dar una visión general de todo el documento SRD.

1.1 Objetivo

Debe exponer brevemente el objetivo del proyecto, a quién va dirigido, los participantes y el calendario previsto.

1.2 Ámbito

En esta subsección se identificará y dará nombre al producto o los productos resultantes del proyecto. Asimismo, se explicará qué hace cada uno y si se considera necesario que no será capaz de hacer. También se detallarán de manera precisa las posibles aplicaciones y beneficios del proyecto.

1.3 Definiciones, siglas y abreviaturas

Aquí se incluirá un glosario que contendrá una lista de definiciones de términos, siglas y abreviaturas particulares utilizados en el documento, y que convenga reseñar para facilitar su lectura o evitar ambigüedades. Toda esta información organizada y clasificada convenientemente, se podrá recoger también en uno o varios apéndices al final del documento.

70 Introducción a la Ingeniería de Software

1. INTRODUCCIÓN

- 1.1 Objetivo
- 1.2 Ámbito
- 1.3 Definiciones, siglas y abreviaturas
- 1.4 Referencias
- 1.5 Panorámica del documento

2. DESCRIPCIÓN GENERAL

- 2.1 Relación con otros proyectos
- 2.2 Relación con proyectos anteriores y posteriores
- 2.3 Objetivo y funciones
- 2.4 Consideraciones de entorno
- 2.5 Relaciones con otros sistemas
- 2.6 Restricciones generales
- 2.7 Descripción del modelo

3. REQUISITOS ESPECÍFICOS

- 3.1 Requisitos funcionales
- 3.2 Requisitos de capacidad
- 3.3 Requisitos de interfase
- 3.4 Requisitos de operación
- 3.5 Requisitos de recursos
- 3.6 Requisitos de verificación
- 3.7 Requisitos de pruebas de aceptación
- 3.8 Requisitos de documentación
- 3.9 Requisitos de seguridad
- 3.10 Requisitos de transportabilidad
- 3.11 Requisitos de calidad
- 3.12 Requisitos de fiabilidad
- 3.13 Requisitos de mantenibilidad
- 3.14 Requisitos de salvaguarda

4. APÉNDICES

Cuadro 2.1. Índice del documento SRD

1.4 Referencias

Si el documento contiene referencias concretas a otros, se dará una lista con la descripción bibliográfica de los documentos referenciados y la manera de obtener acceso a dichos documentos. Si fuera necesario, también en este caso se remitirá a un apéndice posterior.

1.5 Panorámica del documento

Esta subsección describirá la organización y el contenido del resto del documento.

2. DESCRIPCIÓN GENERAL

En esta sección se dará una visión general del sistema, ampliando el contenido de la sección de introducción. Constará de los siguientes apartados:

2.1 Relación con otros proyectos

Se describirán las analogías y diferencias de este proyecto con otros similares o complementarios, o con otros sistemas ya existentes o en desarrollo. Si no hay proyectos o sistemas relacionados, se indicará "No aplicable".

2.2 Relación con proyectos anteriores y posteriores

Se indicará si este proyecto es continuación de otro o si se continuará el desarrollo en proyectos posteriores. Si no hay proyectos de esta clase, se indicará "No existen".

2.3 Objetivo y funciones

Se debe describir el sistema en su conjunto con los objetivos y las funciones principales.

2.4 Consideraciones de entorno

En este apartado se describirán las características especiales que debe tener el entorno en que se utilice el sistema a desarrollar. Si no se necesitan características especiales, se indicará "No existen".

2.5 Relaciones con otros sistemas

Se describirán las conexiones del sistema con otros, si debe funcionar integrado con ellos o utilizando entradas o salidas indirectas de información. Si el sistema no necesita intercambiar información con ningún otro, se indicará "No existen".

72 Introducción a la Ingeniería de Software

2.6 Restricciones generales

Se describirán las restricciones generales a tener en cuenta a la hora de diseñar y desarrollar el sistema, tales como el empleo de determinadas metodologías de desarrollo, lenguajes de programación, normas particulares, restricciones de hardware, de sistema operativo, etc.

2.7 Descripción del modelo

Este será probablemente el apartado más extenso de esta sección. Debe describir el modelo conceptual que se propone para desarrollar el sistema en su conjunto y para cada una de sus partes más relevantes. Este modelo se puede realizar utilizando todas las notaciones y herramientas disponibles.

3. REQUISITOS ESPECÍFICOS

Esta sección es la fundamental del documento. Debe contener una lista detallada y completa de los requisitos que debe cumplir el sistema a desarrollar.

Los requisitos deben exponerse en la forma más precisa posible, pero sin que la descripción de un requisito individual resulte demasiado extensa. Si fuera necesario, puede darse una descripción resumida y hacer referencia a un Apéndice con la descripción detallada.

Es importante no incluir en los requisitos aspectos de diseño o desarrollo. Los requisitos son lo mínimo que se impone al sistema, y no hay que describir soluciones particulares que no sea obligatorio utilizar (excepto como aclaración o sugerencia).

Es ventajoso enunciar los requisitos en forma de una lista numerada, para facilitar su seguimiento y la validación del sistema. Cada requisito debe ir acompañado de una indicación del grado de cumplimiento necesario, es decir, si es obligatorio, recomendable o simplemente opcional.

Los requisitos se agruparán en los apartados que se indican a continuación. Si no hay requisitos en alguno de ellos, se indicará "No existen".

3.1 Requisitos funcionales

Son los que describen las funciones o el QUÉ debe hacer el sistema y están muy ligados al modelo conceptual propuesto. Aquí se concretarán las

operaciones de tratamiento de información que realiza el sistema, tales como almacenamiento de información, generación de informes, cálculos, estadísticas, operaciones, etc.

3.2 Requisitos de capacidad

Son los referentes a los volúmenes de información a procesar, tiempo de respuesta, tamaños de ficheros o discos, etc. Estos requisitos deben expresarse mediante valores numéricos e incluso, cuando sea necesario, se darán valores para el peor, el mejor y el caso más habitual.

3.3 Requisitos de interfase

Son los referentes a cualquier conexión a otros sistemas (hardware o software) con los que se debe interactuar o comunicar. Se incluyen, por tanto, bases de datos, protocolos, formatos de ficheros, sistemas operativos, datos, etc. a intercambiar con otros sistemas o aplicaciones.

3.4 Requisitos de operación

Son los referentes al uso del sistema en general e incluyen los requisitos de la interfase de usuario (menús de pantalla, manejo de ratón, teclado, etc.), el arranque y parada, copias de seguridad, requisitos de instalación y configuración.

3.5 Requisitos de recursos

Son los referentes a elementos hardware, software, instalaciones, etc., necesarios para el funcionamiento del sistema. Es muy importante estimar los recursos con cierto coeficiente de seguridad en previsión de necesidades de última hora no previstas inicialmente.

3.6 Requisitos de verificación

Son los que debe cumplir el sistema para que sea posible verificar y certificar que funciona correctamente (funciones de autotest, emulación, simulación, etc.).

3.7 Requisitos de pruebas de aceptación

Son los que deben cumplir las pruebas de aceptación a que se someterá el sistema.

74 Introducción a la Ingeniería de Software

3.8 Requisitos de documentación

Son los referentes a la documentación que debe formar parte del producto a entregar.

3.9 Requisitos de seguridad

Son los referentes a la protección del sistema contra cualquier manipulación o utilización indebida (confidencialidad, integridad, virus, etc.).

3.10 Requisitos de transportabilidad

Son los referentes a la posible utilización del sistema en diversos entornos o sistemas operativos de una forma sencilla y barata.

3.11 Requisitos de calidad

Son los referentes a aspectos de calidad, que no se incluyan en otros apartados.

3.12 Requisitos de fiabilidad

Son los referentes al límite aceptable de fallos o caídas durante la operación del sistema.

3.13 Requisitos de mantenibilidad

Son los que debe cumplir el sistema para que se pueda realizar adecuadamente su mantenimiento durante la fase de explotación.

3.14 Requisitos de salvaguarda

Son los que debe cumplir el sistema para evitar que los errores en el funcionamiento o la operación del sistema tengan consecuencias graves en los equipos o las personas.

APÉNDICES

Se incluirán como apéndices todos aquellos elementos que completen el contenido del documento, y que no estén recogidos en otros documentos accesibles a los que pueda hacerse referencia.

2.5 Ejemplos de especificaciones

En este apartado se recogen las especificaciones completas de dos sistemas, que se irán desarrollando a lo largo del libro.

2.5.1 Videojuego de las minas

El documento SRD para el videojuego de las minas es el siguiente:

DOCUMENTO DE REQUISITOS DEL SOFTWARE (SRD)

Proyecto: JUEGO DE LAS MINAS (Versión Simple)
Autores: J.A. Cerrada y R. Gómez Fecha: Enero 1999
Documento: MINAS-SRD-99

CONTENIDO

1. INTRODUCCIÓN

- 1.1 Objetivo
- 1.2 Ámbito
- 1.3 Definiciones, siglas y abreviaturas
- 1.4 Referencias
- 1.5 Panorámica del documento

2. DESCRIPCIÓN GENERAL

- 2.1 Relación con otros proyectos
- 2.2 Relación con proyectos anteriores y posteriores
- 2.3 Objetivo y funciones
- 2.4 Consideraciones de entorno
- 2.5 Relaciones con otros sistemas
- 2.6 Restricciones generales
- 2.7 Descripción del modelo

3. REQUISITOS ESPECÍFICOS

- 3.1 Requisitos funcionales
- 3.2 Requisitos de capacidad
- 3.3 Requisitos de interfase
- 3.4 Requisitos de operación
- 3.5 Requisitos de recursos
- 3.6 Requisitos de verificación
- 3.7 Requisitos de pruebas de aceptación
- 3.8 Requisitos de documentación
- 3.9 Requisitos de seguridad
- 3.10 Requisitos de transportabilidad
- 3.11 Requisitos de calidad
- 3.12 Requisitos de fiabilidad
- 3.13 Requisitos de mantenibilidad
- 3.14 Requisitos de salvaguarda

76 Introducción a la Ingeniería de Software

1. INTRODUCCIÓN

1.1 Objetivo

Se trata de realizar un videojuego denominado "Juego de las Minas" cuyas reglas y características generales se detallan en los siguientes apartados de este documento.

La utilización de este videojuego deberá ser fácil de aprender sin necesidad de la lectura previa de ningún manual. En todo caso, el juego dispondrá de las correspondientes ayudas para facilitar su aprendizaje.

1.2 Ámbito

En este proyecto se realizará una versión simple del videojuego que solamente utilizará pantalla alfanumérica y teclado.

1.3 Definiciones, siglas y abreviaturas

Tablero: Elemento gráfico que se muestra en la pantalla del computador con forma de cuadrícula y en el que se desarrolla el juego.

Casilla: Cada uno de los elementos de los que está formada la cuadrícula del tablero.

Mina: Elemento oculto en una casilla del tablero y que si se destapa provoca la finalización del juego de manera infructuosa para el jugador.

1.4 Referencias

No aplicable

1.5 Panorámica del documento

En el resto de este documento se recogen el modelo conceptual, las reglas del juego y los requisitos que debe cumplir el sistema a desarrollar.

2. DESCRIPCIÓN GENERAL

2.1 Relación con otros proyectos

Ninguna.

2.2 Relación con proyectos anteriores y posteriores

En este desarrollo se abordará una versión simple que utilizará una interfase de usuario para pantalla alfanumérica y teclado. En una fase posterior, se desarrollará una versión más elaborada que utilizará pantalla gráfica y ratón. El desarrollo de esta versión simple y la posterior deberá diferir solamente en los módulos específicos dedicados a la interfase hombre-máquina.

2.3 Objetivo y funciones

El objetivo es realizar una versión simplificada del juego de las minas. En este juego se trata de descubrir dentro del tablero la situación de las minas sin que ninguna de ellas explote y en el menor tiempo posible.

Las funciones básicas serán:

- Selección del nivel de dificultad del juego (bajo, medio, alto)
- Desarrollo de la partida según las reglas de juego
- Elaboración y mantenimiento de la tabla de mejores resultados
- Ayudas al jugador

2.4 Consideraciones de entorno

No existen

2.5 Relación con otros sistemas

No existen

2.6 Restricciones generales

Para la realización de este proyecto se deberán utilizar las siguientes herramientas y entornos:

Lenguaje de programación:	Modula-2
Metodología:	Desarrollo modular basado en abstracciones
Sistema operativo:	DOS versión 3.0 o posterior
Máximo personal:	1 persona

2.7 Descripción del modelo

En el juego se emplea un tablero cuadrado de N casillas de lado semejante al que se muestra en la figura M.1. En este tablero están ocultas un número determinado de minas que pueden estar situadas en cualquier casilla. Inicialmente se muestra el tablero con todas las casillas tapadas. En el ejemplo de la figura M.1, las casillas tapadas están en blanco. El jugador tiene que ir destapando las casillas para descubrir la situación de las minas. Cuando se destapa una casilla que tiene una mina, esta mina explota y se acaba el juego infructuosamente. El objetivo del juego es encontrar la situación de todas las minas sin que explote ninguna de ellas.

Para facilitar la búsqueda de todas las minas, el jugador podrá marcar una casilla cuando tenga la certeza de que en ella existe una mina. En la figura M.1 la marca se indica con el símbolo "!!". Esta marca impide que pueda ser destapada la casilla por error. El jugador también podrá quitar la marca de una casilla. En todo momento se mostrará en pantalla el número de minas ocultas y todavía no marcadas.

El jugador podrá seleccionar el grado de dificultad del juego entre tres niveles: bajo, medio y alto. En cada nivel se empleará un tamaño distinto de tablero y el número de minas a descubrir también será distinto. La situación de las minas en el tablero se realizará de forma aleatoria.

78 Introducción a la Ingeniería de Software

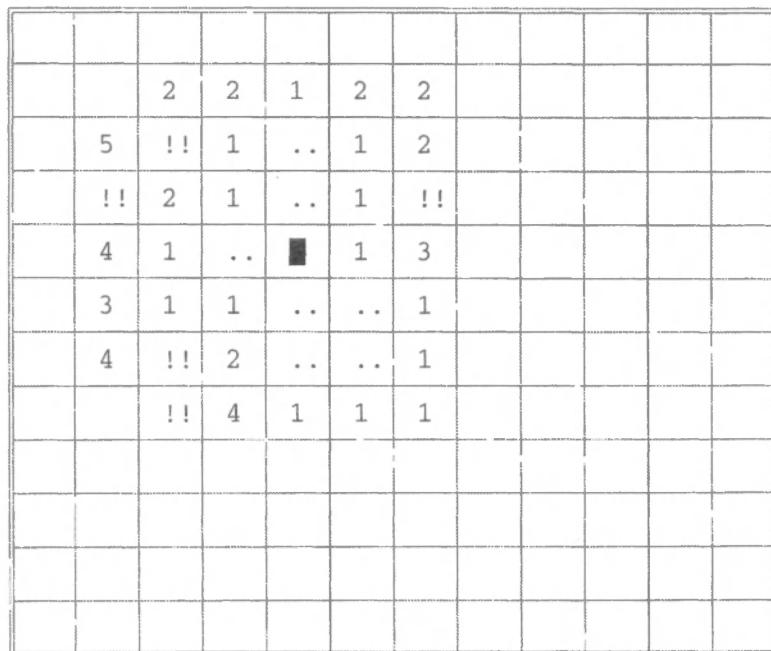


Figura M.1. Tablero del juego de las minas

Cada vez que el jugador destapa una casilla, se deberá indicar si tenía una mina, en cuyo caso finaliza el juego, o el número de minas que rodean la casilla que ha sido destapada. En la figura M.1 las casillas con el simbolo "..." indican que el número de minas que las rodean son cero. El número de minas que pueden rodear una casilla pueden ir desde 0 hasta 8.

El juego dispondrá de un cronómetro que actualiza en pantalla cada segundo el tiempo transcurrido desde el comienzo de la última partida. Si el jugador consigue descubrir todas las minas, el programa registrará el tiempo invertido junto con el texto que deseé poner el jugador y actualizará la lista de los mejores tiempos registrados en el nivel correspondiente: bajo, medio o alto.

2.7.1 Descripción de datos

Los principales datos que se utilizan son los siguientes:

Nombre: Tablero

Estructura: tamaño del tablero + nº de minas + { información de casilla }

información de casilla = si/no mina +
 si/no destapada +
 si/no marcada

Nombre: Mejores resultados

Estructura: { información resultado }

información resultado = Texto del ganador +
nivel dificultad +
tiempo invertido

2.7.2 Diagrama de transición de estados

El diagrama de estados del sistema se muestra en la figura M.2

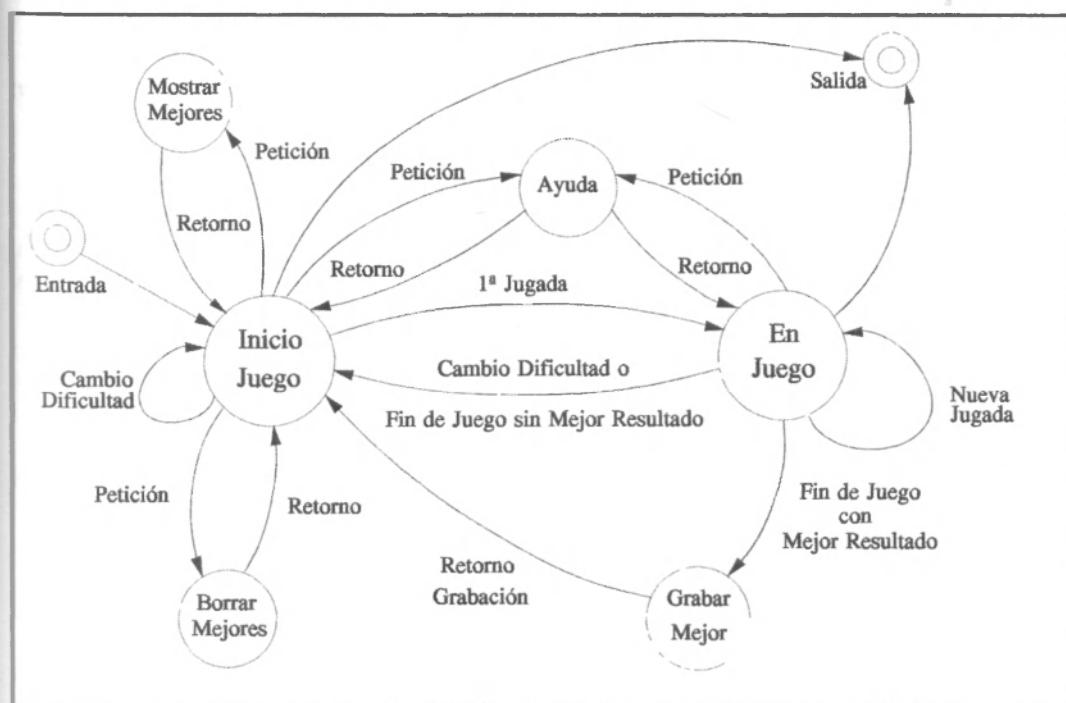


Figura M.2. Diagrama de estados

3. REQUISITOS ESPECÍFICOS

3.1 Requisitos funcionales

R.1.1: Las reglas del juego son las siguientes:

R.1.1.1: El juego se iniciará al destapar la primera casilla.

R.1.1.2: Al iniciarse el juego se pondrá el cronómetro en marcha y se situarán de forma aleatoria las minas en el tablero

R.1.1.3: El cronómetro se actualizará cada segundo

80 Introducción a la Ingeniería de Software

- R.1.1.4: En cada paso del juego, el jugador puede destapar una casilla, marcar en una casilla la presencia de una mina o quitar la marca puesta en cualquier paso anterior.
- R.1.1.5: Al destapar una casilla que tiene una mina se acaba el juego de forma infructuosa para el jugador y se muestra la situación de todas las minas.
- R.1.1.6: Al destapar una casilla que no tiene mina se informa al jugador del número de minas que rodean la casilla destapada.
- R.1.1.7: Cuando se destapa una casilla que no tiene ninguna mina a su alrededor, automáticamente también se destapan todas las casillas que la rodean. Este requisito se cumplirá de forma recursiva con las nuevas casillas destapadas.
- R.1.1.8: Cuando se marca la presencia de una mina en una casilla se impide que dicha casilla pueda ser destapada mientras no se quite la marca.
- R.1.1.9: A una casilla marcada se le puede quitar la marca
- R.1.1.10: El jugador consigue su objetivo cuando todas las casillas que no tienen mina han sido destapadas. En este momento se para el cronómetro indicando el tiempo invertido.
- R.1.2: En todo momento el jugador estará informado de los segundos transcurridos y de las minas que todavía quedan por marcar del total de minas ocultas inicialmente.
- R.1.3: El jugador en cualquier momento podrá cambiar el nivel de dificultad del juego entre los tres siguientes: bajo, medio y alto. Por defecto estará seleccionado el nivel medio. Estos niveles representarán distintos tamaños del tablero y un número diferente de minas ocultas.
- R.1.4: El jugador puede registrar el texto que quiera, junto al tiempo invertido, en la tabla de los mejores resultados y dentro del nivel de dificultad en el que se realizó el juego.
- R.1.5: El jugador en cualquier momento podrá finalizar el juego.
- R.1.6: La tabla con los mejores resultados no se borrará nunca al apagar el computador.
- R.1.7: Para reinicializar la tabla de resultados existirá una opción especial no accesible a cualquier jugador.
- R.1.8: La situación inicial de las minas en el tablero será aleatoria y deberá dar lugar a un reparto aproximadamente uniforme en todo el tablero.
- R.1.9: (Requisito deseable) Se dispondrá de una ayuda con las reglas del juego que podrá ser consultada solamente antes de comenzar o al finalizar un juego.
- R.1.10: El juego dispondrá de una ayuda simplificada que podrá ser consultada en cualquier momento y que explicará las teclas que se pueden utilizar durante el juego y la función de cada una.

3.2 Requisitos de capacidad

R.2.1: Precisión del cronómetro < 0.1 segundo

R.2.2: Respuesta al pulsar una tecla < 0.1 segundo

R.2.3: Tiempo para situar inicialmente las minas < 1 segundo

3.3 Requisitos de interfase

R.3.1: Para la presentación del tablero en pantalla será necesario al menos disponer de una pantalla de tipo alfanumérico.

3.4 Requisitos de operación

Además de los indicados en el apartado de descripción del modelo respecto a la estructura del tablero, casillas, etc., se deberán cumplir los siguientes:

R.4.1: En todo momento y dentro del tablero deberá señalarse claramente la casilla en la que está situado el jugador.

R.4.2: Para moverse de una casilla a otra de las que la rodean sólo será necesario pulsar una tecla una sola vez.

R.4.3: Para destapar, marcar o desmarcar una casilla sólo será necesario pulsar una tecla una sola vez.

R.4.4: El cambio de nivel de dificultad se realizará de forma rotativa (bajo -> medio -> alto -> bajo) con sólo pulsar una tecla una sola vez. El cambio de nivel reiniciará el nuevo tablero de juego.

R.4.5: Para finalizar el juego o mostrar las ayudas del juego sólo será necesario pulsar una tecla una vez.

3.5 Requisitos de recursos

R.5.1: Este sistema no requiere recursos especiales para su utilización y podrá ser ejecutado en cualquier instalación básica basada en un PC compatible con la configuración mínima siguiente:

- CPU: 486 o posterior
- Memoria: 640 Kbytes o más
- Pantalla: cualquiera con modo texto de 80 × 25 caracteres
- Disquete: 3½ pulgadas o 5¼ pulgadas
- No necesita disco duro

82 Introducción a la Ingeniería de Software

3.6 Requisitos de verificación

R.6.1: (Requisito deseable) Se dispondrá de un modo auxiliar para depuración en el que el tablero será "transparente" y en el que el jugador conoce la situación de todas las minas a priori.

3.7 Requisitos de pruebas de aceptación

Ninguno.

3.8 Requisitos de documentación

R.8.1: El videojuego estará autodocumentado con las funciones de ayuda previstas

3.9 Requisitos de seguridad

R.9.1: (Requisito deseable) Se dispondrá de algún mecanismo de protección contra la copia indiscriminada del programa.

3.10 Requisitos de transportabilidad

Ninguno.

3.11 Requisitos de calidad

No aplicable.

3.12 Requisitos de fiabilidad

No aplicable.

3.13 Requisitos de mantenibilidad

No aplicable.

3.14 Requisitos de salvaguarda

No aplicable.

2.5.2 Sistema de gestión de biblioteca

Se trata de informatizar la gestión del préstamo de libros en una biblioteca relativamente pequeña, atendida por una sola persona. En esta biblioteca se pueden consultar libros en la sala de lectura, sin ningún control especial, y también se pueden sacar libros en préstamo, en forma controlada.

Para sacar libros en préstamo el usuario debe estar dado de alta en un registro de lectores, en el que habrá una ficha por cada usuario, conteniendo sus datos personales.

Los libros estarán catalogados. Habrá un registro de libros con una ficha por cada uno, conteniendo los datos fundamentales: título, autor, materias de que trata, etc.

Cuando un libro se saque en préstamo, se anotará en la ficha del libro a quién está prestado. Un mismo lector puede tener a la vez varios libros en préstamo, hasta un límite fijo. También hay un límite fijo para el número de días que se puede tener prestado un libro. Pasado ese plazo sin que se haya devuelto el libro, el bibliotecario avisará por teléfono al lector moroso para que realice la devolución.

El sistema informático se usará también como ayuda para localizar los libros solicitados por los usuarios, permitiendo realizar búsquedas en el catálogo de libros, bien por autor, título o materia.

A continuación se presenta un ejemplo de lo que podría ser el SRD de este sistema, recogiendo el análisis realizado con la metodología de ANÁLISIS ESTRUCTURADO.

84 Introducción a la Ingeniería de Software

DOCUMENTO DE REQUISITOS DEL SOFTWARE (SRD)

Proyecto: SISTEMA DE GESTIÓN DE BIBLIOTECA
Autores: M. Collado y J.F. Estivariz
Fecha: Mayo 1999
Documento: BIBLIO-SRD-99

CONTENIDO

1. INTRODUCCIÓN

- 1.1 Objetivo
- 1.2 Ámbito
- 1.3 Definiciones, siglas y abreviaturas
- 1.4 Referencias
- 1.5 Panorámica del documento

2. DESCRIPCIÓN GENERAL

- 2.1 Relación con otros proyectos
- 2.2 Relación con proyectos anteriores y posteriores
- 2.3 Objetivo y funciones
- 2.4 Consideraciones de entorno
- 2.5 Relaciones con otros sistemas
- 2.6 Restricciones generales
- 2.7 Descripción del modelo

3. REQUISITOS ESPECÍFICOS

- 3.1 Requisitos funcionales
- 3.2 Requisitos de capacidad
- 3.3 Requisitos de interfase
- 3.4 Requisitos de operación
- 3.5 Requisitos de recursos
- 3.6 Requisitos de verificación
- 3.7 Requisitos de pruebas de aceptación
- 3.8 Requisitos de documentación
- 3.9 Requisitos de seguridad
- 3.10 Requisitos de transportabilidad
- 3.11 Requisitos de calidad
- 3.12 Requisitos de fiabilidad
- 3.13 Requisitos de mantenibilidad
- 3.14 Requisitos de salvaguarda

1. INTRODUCCIÓN

1.1 Objetivo

El objetivo del sistema es facilitar la gestión de una biblioteca mediana o pequeña, en lo referente a la atención directa a los usuarios. Esto incluye, fundamentalmente, el préstamo de libros, así como la consulta bibliográfica.

1.2 Ámbito

El sistema a desarrollar se denominará BIBLIO-1, y consistirá en un único programa que realizará todas las funciones necesarias. En particular, deberá facilitar las siguientes:

- Gestión del préstamo y devolución de libros
- Consulta bibliográfica por título, autor o materia

El sistema no ha de soportar, sin embargo, la gestión económica de la biblioteca, ni el control de adquisición de libros, ni otras funciones no relacionadas directamente con la atención a los usuarios.

El sistema facilitará la atención al público por parte de un único bibliotecario, que podrá atender a todas las funciones.

1.3 Definiciones, siglas y abreviaturas

Ninguna.

1.4 Referencias

Ninguna.

1.5 Panorámica del documento

El resto de este documento contiene una descripción del modelo del sistema, en la Sección 2, y la lista de requisitos específicos en la Sección 3.

Para confeccionar el modelo se ha aplicado la metodología de ANÁLISIS ESTRUCTURADO. Los diagramas de flujo de datos se han incluido en la Sección 2, así como el diccionario de datos y el diagrama Entidad-Relación correspondiente al modelo de datos. Las especificaciones de procesos (funciones) se han incluido en el apartado *3.1 Requisitos funcionales*.

2. DESCRIPCIÓN GENERAL

2.1 Relación con otros proyectos

Ninguna.

2.2 Relación con proyectos anteriores y posteriores

Ninguna.

86 Introducción a la Ingeniería de Software

2.3 Objetivo y funciones

La biblioteca dispone de una colección de libros a disposición del público. Cualquier persona puede consultar los libros en la sala de lectura, sin necesidad de realizar ningún registro de esta actividad.

Los libros pueden ser también sacados en préstamo por un plazo limitado, fijado por la organización de la biblioteca, y siempre el mismo. En este caso es necesario mantener anotados los libros prestados y qué lector los tiene en su poder. Para que un lector pueda sacar un libro en préstamo debe disponer previamente de una ficha de lector con sus datos, y en particular con indicación de su teléfono, para facilitar la reclamación del libro en caso de demora en su devolución.

Un lector puede tener a la vez varios libros en préstamo, hasta un máximo fijado por la organización de la biblioteca, igual para todos los usuarios.

Los usuarios deben disponer de algunas facilidades para localizar el libro que desean, tanto si es para consultarlos en la sala como si es para sacarlos en préstamo. Se considera razonable poder localizar un libro por su autor, su título (o parte de él) o por la materia de que trata. Para la búsqueda por materias, existirá una lista de materias establecida por el bibliotecario. Cada libro podrá tratar de una o varias materias.

El objetivo del sistema es facilitar las funciones más directamente relacionadas con la atención directa a los usuarios de la biblioteca. Las funciones principales serán:

- Anotar los préstamos y devoluciones
- Indicar los préstamos que hayan sobrepasado el plazo establecido
- Búsquedas por autor, título o materia

Como complemento serán necesarias otras funciones, en concreto:

- Mantener un registro de los libros
- Mantener un registro de los usuarios (lectores)

2.4 Consideraciones de entorno

Ninguna.

2.5 Relaciones con otros sistemas

Ninguna.

2.6 Restricciones generales

- El sistema se desarrollará y documentará empleando la metodología de análisis y diseño estructurado.
- La implementación se hará sobre una base de datos relacional.
- El programa deberá poder ejecutarse en máquinas de poca capacidad y con sistemas operativos sencillos, sin soporte de multitarea, tal como MS-DOS en PCs.

2.7 Descripción del modelo

El sistema de gestión de biblioteca será operado por una sola persona, que dispondrá de un terminal con pantalla y teclado. También existirá una impresora por la que podrán obtenerse listados y fichas de los libros y de los lectores. Esta organización se refleja en el Diagrama de Contexto que se muestra en la figura B.1.

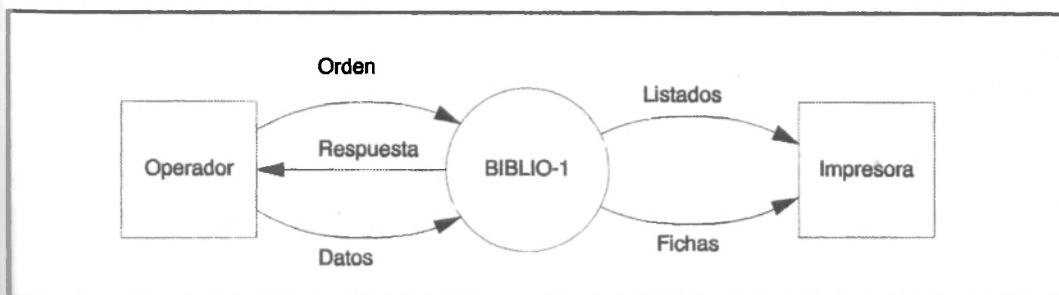


Figura B.1. DC. Diagrama de contexto

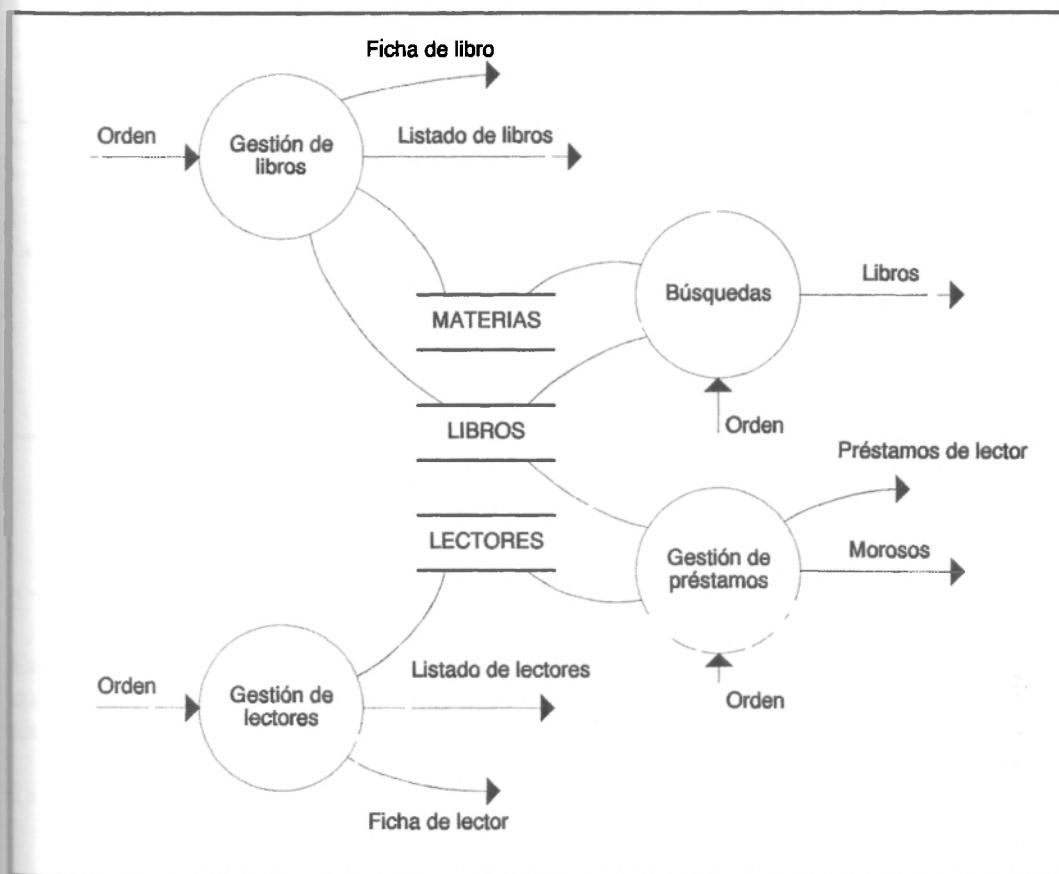


Figura B.2. DFD.0 Gestión de la biblioteca

88 Introducción a la Ingeniería de Software

La operación del sistema se hará mediante un sistema de menús para seleccionar la operación deseada, y con formularios en pantalla para la entrada y presentación de datos. Las funciones a realizar se pueden organizar en varios grupos principales, tal como se indica en el diagrama de flujo de datos de la figura B.2. Estos grupos de funciones se describen a continuación. La descripción precisa de cada función se incluye en la *Sección 3: Requisitos Específicos*.

2.7.1 Gestión de libros

Estas funciones permiten mantener actualizado el registro (fichero) de libros existentes en la biblioteca. El desglose de las mismas se refleja en el diagrama de flujo de datos de la figura B.3.

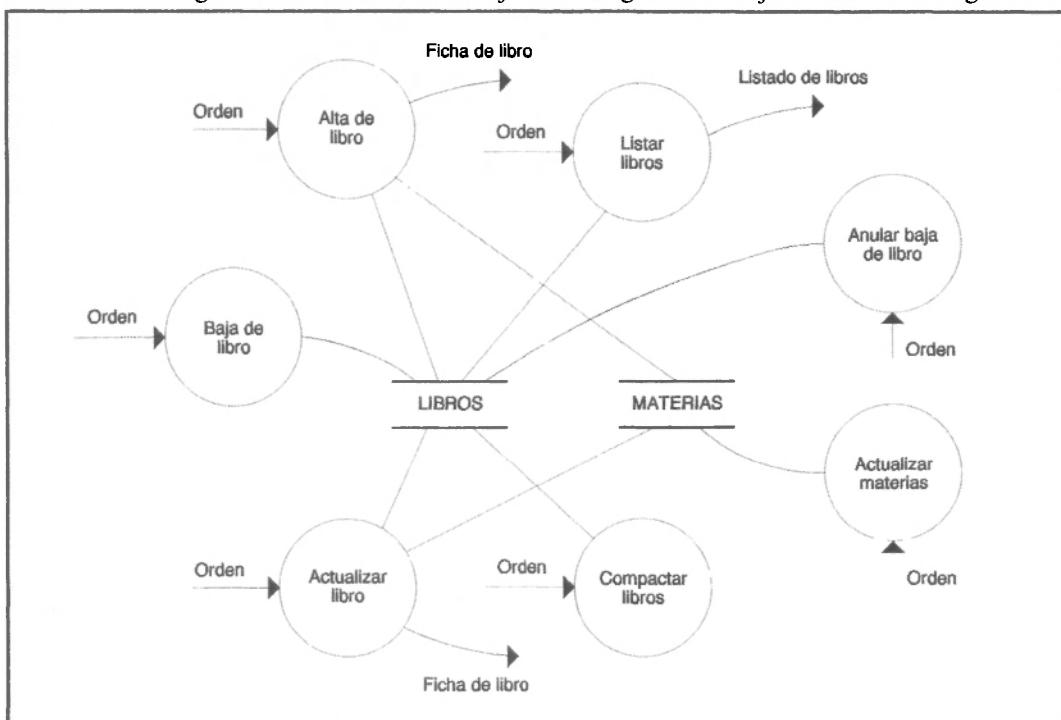


Figura B.3. DFD.1 Gestión de Libros

Las funciones de gestión de libros son las siguientes:

- Función 1.1 Alta de libro: registra un nuevo libro
- Función 1.2 Baja de libro: marca un libro como dado de baja
- Función 1.3 Anular baja de libro: suprime la marca de baja
- Función 1.4 Actualizar libro: modifica los datos del libro
- Función 1.5 Listar libros: lista todos los libros registrados
- Función 1.6 Compactar registro de libros: elimina los libros dados de baja
- Función 1.7 Actualizar lista de materias: actualiza la lista de materias consideradas

2.7.2 Gestión de lectores

Estas funciones permiten mantener actualizado el registro (fichero) de usuarios (lectores). El desglose de las mismas se refleja en el diagrama de flujo de datos de la figura B.4.

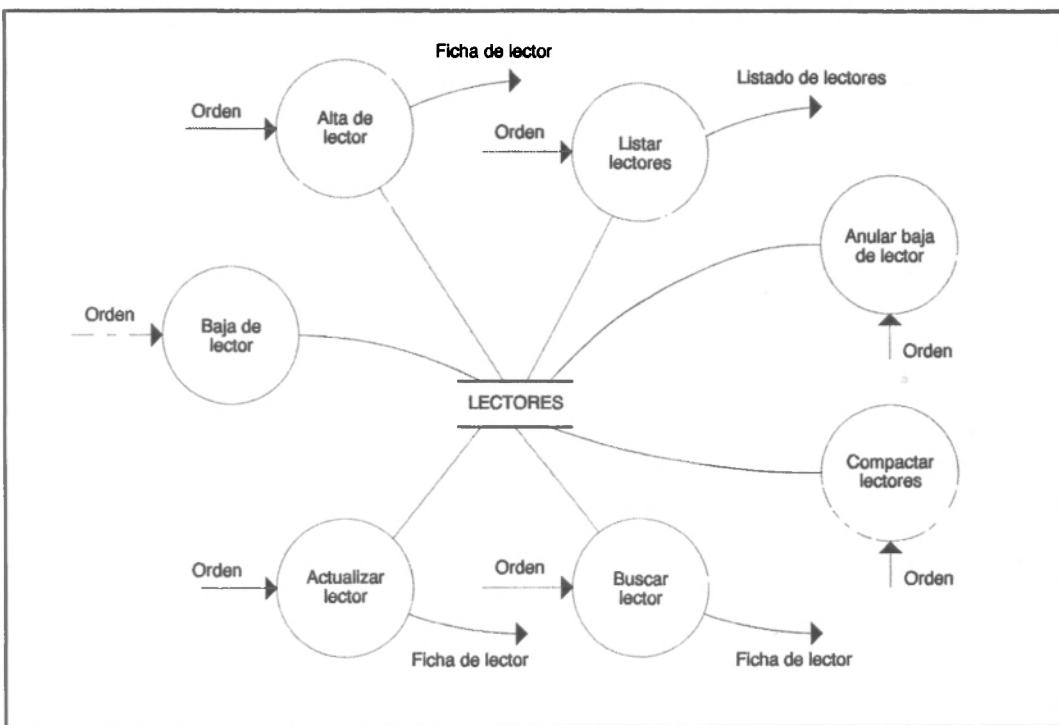


Figura B.4. DFD.2 Gestión de lectores

Las funciones de gestión de lectores son las siguientes:

- Función 2.1 Alta de lector: registra un nuevo usuario (lector)
- Función 2.2 Baja de lector: marca un lector como dado de baja
- Función 2.3 Anular baja de lector: suprime la marca de baja
- Función 2.4 Actualizar lector: modifica los datos del lector
- Función 2.5 Listar lectores: lista todos los lectores registrados
- Función 2.6 Compactar registro de lectores: elimina los lectores dados de baja
- Función 2.7 Buscar lector: localiza un lector por nombre o apellido

2.7.3 Gestión de préstamos

Estas funciones permiten tener anotados los libros en préstamo, y conocer en un momento dado los que han sido retenidos más tiempo del permitido. El desglose de las mismas se refleja en el diagrama de flujo de datos de la figura B.5.

Las funciones de gestión de préstamos son las siguientes:

- Función 3.1 Anotar préstamo: anota los datos del préstamo
- Función 3.2 Anotar devolución: elimina los datos del préstamo
- Función 3.3 Lista de morosos: lista todos los préstamos no devueltos en el plazo fijado
- Función 3.4 Préstamos de lector: lista los libros que tiene en préstamo un lector dado

90 Introducción a la Ingeniería de Software

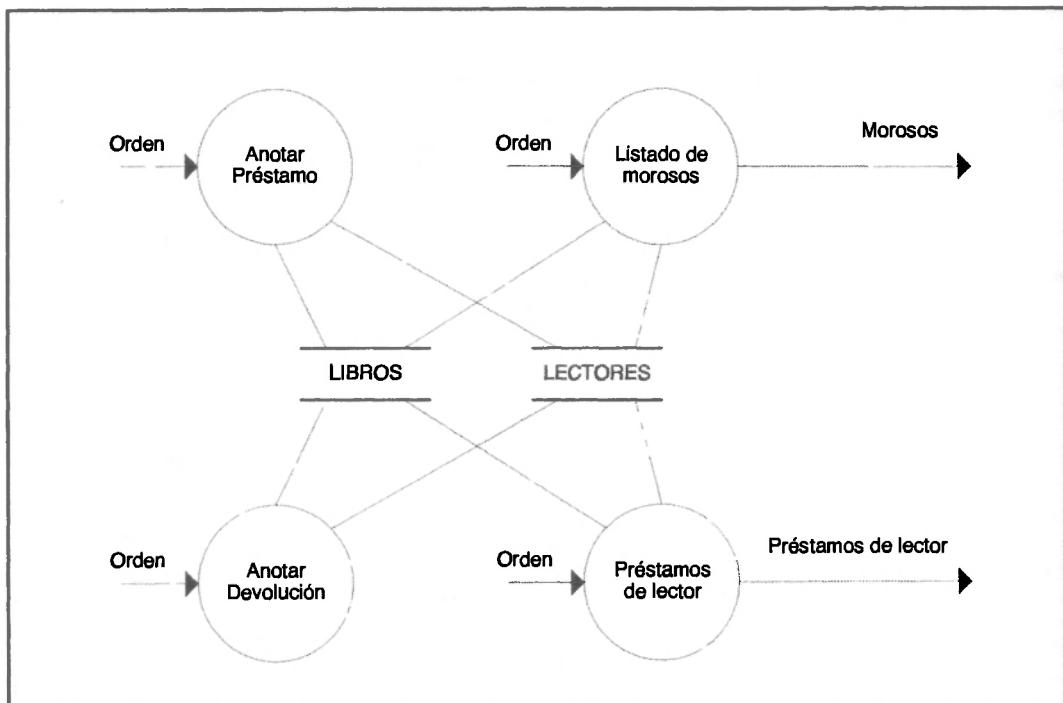


Figura B.5. DFD.3 Gestión de préstamos

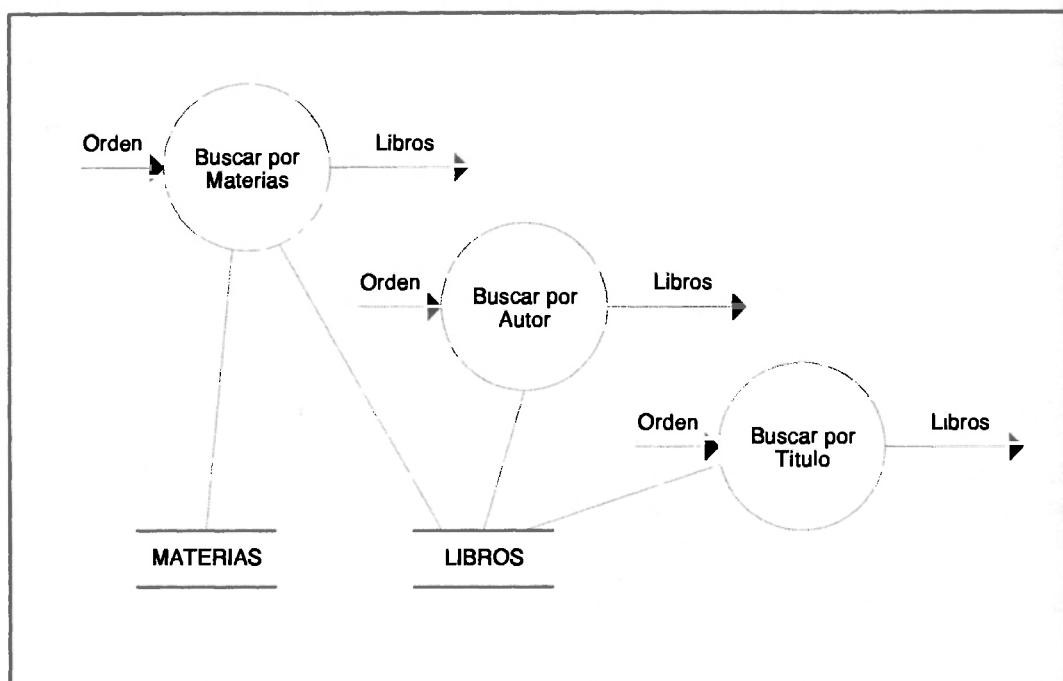


Figura B.6. DFD.4 Búsquedas

2.7.4 Búsquedas

Estas funciones permiten localizar libros por autor, título o materia. El desglose de las mismas se refleja en el diagrama de flujo de datos de la figura B.6.

Las funciones de búsqueda son las siguientes:

- Función 4.1 Buscar por autor: localiza los libros de un autor dado
- Función 4.2 Buscar por título: localiza los libros cuyo título contiene un texto dado
- Función 4.3 Buscar por materia: localiza los libros que tratan de una materia dada

2.7.5 Modelo de datos

El diagrama Entidad-Relación correspondiente a los datos principales del sistema se recoge en la figura B.7. El significado de cada elemento es el siguiente:

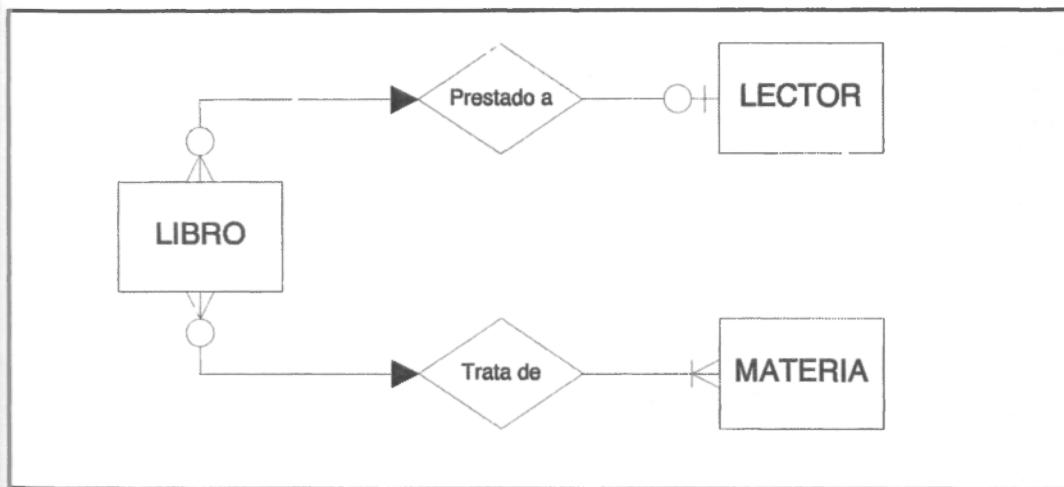


Figura B.7. Modelo de datos del sistema

- Entidad Libro: Contiene los datos de identificación del libro
- Entidad Lector: Contiene los datos personales del lector
- Entidad Materia: Contiene el término clave que identifica la materia
- Relación Prestado-a: Enlaza un libro con el lector que lo ha sacado en préstamo. Contiene la fecha del préstamo
- Relación Trata-de: Enlaza un libro con cada materia de la que trata

Las estructuras de datos principales se recogen en el diccionario de datos del cuadro B.1

92 Introducción a la Ingeniería de Software

DATO	DESCRIPCIÓN
LECTORES	{ FICHA-DE-LECTOR }
LIBROS	{ FICHA-DE-LIBRO }
MATERIAS	{ MATERIA }
FICHA-DE-LECTOR	CÓDIGO-DE-LECTOR + DATOS-DE-LECTOR
FICHA-DE-LIBRO	CÓDIGO-DE-LIBRO + DATOS-DE-LIBRO + DATOS-DE-PRÉSTAMO
MATERIA	/Nombre de la materia/
CÓDIGO-DE-LECTOR	/Número del lector/
CÓDIGO-DE-LIBRO	/Número del libro/
DATOS-DE-PRÉSTAMO	FECHA + CÓDIGO-DE-LECTOR
DATOS-DE-LECTOR	/Nombre, apellidos, DNI, domicilio, teléfono, .../
DATOS-DE-LIBRO	/Autor, título, materias, .../
ORDEN	/Datos de cada orden/
RESPUESTA	/Respuesta en pantalla/
LISTADO-DE-LIBROS	LIBROS
LISTADO-DE-LECTORES	LECTORES
MOROSOS	{ FECHA + /Datos del libro/ + /Datos del lector/ }
PRÉSTAMOS-DE-LECTOR	{ FECHA + /Datos del libro / }

Cuadro B.1. Diccionario de datos del sistema

3. REQUISITOS ESPECÍFICOS

Todos los requisitos se consideran obligatorios, salvo que se indique lo contrario.

3.1 Requisitos funcionales

3.1.1 Almacenamiento de datos

R.1.1 El sistema mantendrá almacenados en forma permanente al menos los datos indicados en LIBROS, LECTORES y MATERIAS, en el Diccionario de Datos descrito en la Sección 2.7.5.

3.1.2 Funciones principales

R.1.2 El sistema realizará al menos las funciones que se describen a continuación, bajo petición del operador.

3.1.2.1 Gestión de libros

Función 1.1 Alta de libro: registra un nuevo libro

Entrada: DATOS-DE-LIBRO

Salida: FICHA-DE-LIBRO

Usa:

Actualiza: LIBROS

Efecto: Compone una nueva ficha de libro asignando código automáticamente y leyendo los datos del libro por pantalla (las materias se seleccionarán de entre la lista de materias registradas). Registra la ficha en el fichero de libros, y además la imprime.

Excepciones:

Función 1.2 Baja de libro: marca un libro como dado de baja

Entrada: CÓDIGO-DE-LIBRO

Salida:

Usa:

Actualiza: LIBROS

Efecto: Lee el código del libro por pantalla, y marca la ficha de ese libro en el fichero de libros como dada de baja.

Excepciones: Si no existe el libro con ese código, o ya estaba dado de baja, da un aviso.

Función 1.3 Anular baja de libro: suprime la marca de baja

Entrada: CÓDIGO-DE-LIBRO

Salida:

Usa:

Actualiza: LIBROS

Efecto: Lee el código del libro por pantalla, y anula la marca de dado de baja en la ficha de ese libro en el fichero de libros.

Excepciones: Si no existe el libro con ese código, o no estaba dado de baja, da un aviso.

Función 1.4 Actualizar libro: modifica los datos del libro

Entrada: CÓDIGO-DE-LIBRO, DATOS-DE-LIBRO

Salida: FICHA-DE-LIBRO

Usa:

Actualiza: LIBROS

Efecto: Lee el código del libro por pantalla, localiza la ficha de ese libro, y actualiza los datos de ese libro, también por pantalla (las materias se seleccionarán de entre la lista de materias registradas). Registra la ficha actualizada en el fichero de libros, y la imprime.

Excepciones: Si no existe el libro con el código indicado, da un aviso.

Función 1.5 Listar libros: lista todos los libros registrados

Entrada:

Salida: LISTADO-DE-LIBROS

Usa: LIBROS

Actualiza:

Efecto: Imprime un listado con todos los datos de todos los libros, incluso los que estén dados de baja.

Excepciones:

94 Introducción a la Ingeniería de Software

Función 1.6 Compactar registro de libros: elimina los libros dados de baja

Entrada:

Salida:

Usa:

Actualiza: LIBROS

Efecto: Suprime del fichero de libros las fichas de libros que estén dados de baja, liberando el espacio que ocupaban.

Excepciones:

Función 1.7 Actualizar lista de materias: actualiza la lista de materias consideradas

Entrada:

Salida:

Usa:

Actualiza: MATERIAS

Efecto: Se edita por pantalla la lista de materias a considerar, pudiendo añadir, suprimir materias, o modificar sus nombres.

Excepciones: Si se intenta suprimir una materia habiendo libros registrados que tratan de ella, se pedirá confirmación especial. Si se fuerza la eliminación, se suprimirá también la referencia a esa materia de los libros que traten de ella.

3.1.2.2 Gestión de lectores

Función 2.1 Alta de lector: registra un nuevo usuario (lector)

Entrada: DATOS-DE-LECTOR

Salida: FICHA-DE-LECTOR

Usa:

Actualiza: LECTORES

Efecto: Compone una nueva ficha de lector asignando código automáticamente y leyendo los datos del lector por pantalla. Registra la ficha en el fichero de lectores, y además la imprime.

Excepciones:

Función 2.2 Baja de lector: marca un lector como dado de baja

Entrada: CÓDIGO-DE-LECTOR

Salida:

Usa:

Actualiza: LECTORES

Efecto: Lee el código del lector por pantalla, y marca la ficha de ese lector en el fichero de lectores como dada de baja.

Excepciones: Si no existe el lector con el código indicado, o ya estaba dado de baja, da un aviso.

Función 2.3 Anular baja de lector: suprime la marca de baja

Entrada: CÓDIGO-DE-LECTOR

Salida:

Usa:

Actualiza: LECTORES

Efecto: Lee el código del lector por pantalla, y suprime la marca de dado de baja de la ficha de ese lector en el fichero de lectores.

Especificación de Software 95

Excepciones: Si no existe el lector con el código indicado, o no estaba dado de baja, da un aviso.

Función 2.4 Actualizar lector: modifica los datos del lector

Entrada: CÓDIGO-DE-LECTOR, DATOS-DE-LECTOR

Salida: FICHA-DE-LECTOR

Usa:

Actualiza: LECTORES

Efecto: Lee el código del lector por pantalla, localiza la ficha de ese lector y actualiza los datos de ese lector, también por pantalla. Registra la ficha actualizada en el fichero de lectores, y la imprime.

Excepciones: Si no existe el lector con el código indicado, da un aviso.

Función 2.5 Listar lectores: lista todos los lectores registrados

Entrada:

Salida: LISTADO-DE-LECTORES

Usa: LECTORES

Actualiza:

Efecto: Imprime un listado con todos los datos de todos los lectores, incluso los que estén dados de baja.

Excepciones:

Función 2.6 Compactar registro de lectores: elimina los lectores dados de baja

Entrada:

Salida:

Usa:

Actualiza: LECTORES

Efecto: Suprime del fichero de lectores las fichas de lectores que estén dados de baja, liberando el espacio que ocupaban.

Excepciones:

Función 2.7 Buscar lector: localiza un lector por nombre o apellido

Entrada: NOMBRE-O-APELLIDO

Salida: FICHA-DE-LECTOR

Usa: LECTORES

Actualiza:

Efecto: Presenta en pantalla un listado con los datos de cada lector que contenga el nombre o apellido indicado.

Excepciones:

3.1.2.3 Gestión de préstamos

Función 3.1 Anotar préstamo: anota los datos del préstamo

Entrada: CÓDIGO-DE-LIBRO, CÓDIGO-DE-LECTOR

Salida:

Usa: LECTORES

Actualiza: LIBROS

Efecto: Lee por pantalla los códigos del libro y del lector. Anota en la ficha del libro los datos del préstamo: código del lector y la fecha del día, tomada del reloj del sistema. Actualiza la ficha en el fichero de libros.

96 Introducción a la Ingeniería de Software

Excepciones: Si no existe el libro o el lector, o el libro ya estaba prestado, o el lector tiene ya el máximo de libros en préstamo, da un aviso.

Función 3.2 Anotar devolución: elimina los datos del préstamo

Entrada: CÓDIGO-DE-LIBRO

Salida:

Usa: LECTORES

Actualiza: LIBROS

Efecto: Lee por pantalla el código del libro, y elimina los datos de préstamo de su ficha.
Actualiza esa ficha en el fichero de libros.

Excepciones: Si el libro no existe, o no está prestado, da un aviso.

Función 3.3 Lista de morosos: lista todos los préstamos no devueltos en el plazo fijado

Entrada:

Salida: MOROSOS

Usa: LIBROS, LECTORES

Actualiza:

Efecto:

Imprime un listado con los datos de libros y lectores de todos los préstamos para los que el plazo desde que se realizaron hasta hoy sea superior el plazo límite establecido.

Excepciones:

Función 3.4 Préstamos de lector: lista los libros que tiene en préstamo un lector dado

Entrada: CÓDIGO-DE-LECTOR

Salida: PRÉSTAMOS-DE-LECTOR

Usa: LIBROS, LECTORES

Actualiza:

Efecto: Presenta en pantalla un listado con los datos abreviados de cada libro que tenga en préstamo ese lector, y la fecha de cada préstamo.

Excepciones: Si el lector no existe, da un aviso.

3.1.2.4 Búsquedas

Función 4.1 Buscar por autor: localiza los libros de un autor dado

Entrada: NOMBRE-O-APELLIDO

Salida: FICHA-DE-LIBRO

Usa: LIBROS

Actualiza:

Efecto: Presenta en pantalla un listado con los datos de cada libro cuyo autor contenga el nombre o apellido indicado.

Excepciones:

Función 4.2 Buscar por título: localiza los libros cuyo título contiene un texto dado

Entrada: TEXTO

Salida: FICHA-DE-LIBRO

Usa: LIBROS

Actualiza:

Efecto: Presenta en pantalla un listado con los datos de cada libro cuyo título contenga el texto indicado.

Excepciones:

Función 4.3 Buscar por materia: localiza los libros que tratan de una materia dada

Entrada: MATERIA

Salida: FICHA-DE-LIBRO

Usa: LIBROS

Actualiza:

Efecto: Presenta en pantalla la lista de materias y permite seleccionar la materia objeto de la búsqueda. Presenta en pantalla un listado con los datos de cada libro que trate de la materia indicada.

Excepciones:

3.2 Requisitos de capacidad

R.2.1 El software debe soportar al menos 9.000 libros, 9.000 lectores y 250 materias.

R.2.2 No obstante lo anterior, la capacidad de almacenamiento puede estar limitada por la capacidad del hardware de cada instalación.

R.2.3 La ejecución de cualquiera de las funciones principales (excepto listados) deberá durar 5 segundos, como máximo, en un PC con procesador 80486. Esto incluye las búsquedas con registros de 9000 libros o lectores. El tiempo puede ser proporcionalmente mayor con registros de mayor capacidad.

R.2.4 En el caso de los listados, el límite anterior se incrementará en el tiempo físico de impresión, dependiente de la impresora.

3.3 Requisitos de interfase

No aplicable.

3.4 Requisitos de operación

R.4.1 La selección de una función se hará mediante un sistema de menús

R.4.1.1 (Deseable) La selección de una función principal no debería exigir más de dos niveles de menú

R.4.2 Los códigos de libro y lector deberán ser fáciles de teclear, exigiendo pocas pulsaciones.

R.4.3 (Deseable) Sobre un libro o lector seleccionado mediante una función se podrá invocar otra sin necesidad de volver a seleccionar manualmente dicho libro o lector.

R.4.4 (Deseable) La función de actualizar la lista de materias debería ser accesible desde las de alta de libro y actualizar libro.

98 Introducción a la Ingeniería de Software

- R.4.5 El sistema podrá configurarse para que el programa entre en funcionamiento automáticamente al arrancar el equipo.
- R.4.6 Existirá la posibilidad de sacar y restablecer copias de seguridad (backups). Para ello bastará que se puedan salvar y restaurar los ficheros con los registros permanentes del sistema.
- R.4.7 En las funciones que actualizan los registros permanentes, se pedirá confirmación antes de hacerlo, presentando los datos que se van a actualizar.
- R.4.8 Al arrancar la aplicación (o durante la operación), se podrán modificar los límites de préstamo, tanto el plazo como el número de libros para un mismo lector.

3.5 Requisitos de recursos

- R.5.1 El programa deberá poder ejecutarse en equipos tipo PC (o similares) de gama baja, con la configuración mínima equivalente a la siguiente:
- CPU: 486 o posterior
 - Memoria: 640 Kb
 - Pantalla: alfanumérica 80 x 25 caracteres
 - Disquete: 3½" 720Kb o 5¼" 1,2Mb, para backup
 - Disco fijo: con capacidad para los programas y los registros permanentes

3.6 Requisitos de verificación

- R.6.1 (Deseable) Deberá disponerse de un sistema de acceso a los registros de libros, lectores y materias, independiente de la aplicación, y que permita examinar su contenido, preferiblemente obteniendo un listado del mismo, para comprobar que la información almacenada es la correcta.

3.7 Requisitos de pruebas de aceptación

- R.7.1 Se deben probar al menos una vez todas y cada una de las funciones, tanto con entradas normales como con datos que provoquen errores, en su caso.

3.8 Requisitos de documentación

- R.8.1 Existirá un manual de operación, sencillo, que describa el uso del sistema.
- R.8.2 (Deseable) Un resumen del manual de operación estará disponible como ayuda en línea, durante la operación del sistema.

3.9 Requisitos de seguridad

- R.9.1 (Deseable) La compactación de los registros sólo debería realizarse si previamente se ha sacado una copia de "backup" de los mismos.
- R.9.2 Aunque se llene el disco fijo y no se puedan registrar nuevos libros, lectores, materias o préstamos, no deberá perderse información anterior de los registros permanentes.

3.10 Requisitos de transportabilidad

- R.10.1 (Deseable) El programa se codificará en un lenguaje de programación de alto nivel, para el que exista una definición normalizada.

3.11 Requisitos de calidad

Ninguno en especial.

3.12 Requisitos de fiabilidad

Ninguno en especial.

3.13 Requisitos de mantenibilidad

Ninguno en especial.

3.14 Requisitos de salvaguarda

Ninguno en especial.

Tema 3

Fundamentos del Diseño de Software

Con este Tema se inicia el estudio de las etapas de desarrollo. Después de haber especificado QUÉ se quiere resolver durante la especificación, las etapas de desarrollo se dedican a determinar CÓMO se debe resolver el proyecto. La primera de estas etapas es la de diseño, se continúa con la de codificación y se finaliza con las etapas de pruebas del software realizado. Toda esta segunda unidad didáctica está dedicada a la etapa de diseño.

Concretamente, en este Tema se abordan los fundamentos de la etapa de diseño. Primeramente se concreta qué se entiende por diseño y se analizan las actividades que se deben realizar para llevarlo a cabo. A continuación se introducen algunos conceptos fundamentales que deben tenerse en cuenta para realizar cualquier diseño. Seguidamente se describen distintas notaciones que se pueden emplear para la descripción de un diseño software. Algunas de estas notaciones son versiones ampliadas de las ya estudiadas para la especificación y otras son totalmente específicas para el diseño. Finalmente, se proponen unos formatos concretos para la elaboración de los documentos del diseño arquitectónico y del diseño detallado en los que se recogen todos los resultados del diseño.

3.1 Introducción

Comenzaremos esta unidad didáctica, dedicada íntegramente al diseño, tratando de establecer qué se entiende por diseño de software. Si consultamos el diccionario, encontramos la siguiente definición:

Diseño. (del italiano *disegno*) ... || 2. Descripción o bosquejo de alguna cosa, hecho por palabras.

Lo que trasladado al contexto del diseño de sistemas software sería la descripción o bosquejo del sistema a desarrollar. En nuestro caso, la descripción se puede y se debe hacer mediante otras notaciones más formales y no sólo empleando palabras. Se trata, por tanto, de definir y formalizar la estructura del sistema con el suficiente detalle como para permitir su realización física.

104 Introducción a la Ingeniería de Software

El punto de partida principal para abordar el diseño es el documento de especificación de requisitos (SRD). La realización del diseño de un sistema es un proceso creativo que no es nada trivial y que casi siempre se lleva a cabo de una forma iterativa mediante prueba y error. En este proceso es muy importante la experiencia previa y siempre que sea posible se tratará de reutilizar el mayor número de módulos o elementos ya desarrollados. Cuando esto último no sea posible, por tratarse del diseño de un sistema completamente original y sin ningún antecedente previo, es seguro que para comenzar el diseño, al menos se podrá aprovechar el enfoque dado a algún otro proyecto anterior, lo que se conoce como aprovechar el *know-how* (saber hacer). Después, en sucesivas iteraciones se perfilará el enfoque más adecuado para el nuevo diseño. Esta forma de trabajo es la misma que se utiliza para diseñar nuevos coches, aviones, edificios, puentes o cualquier otra obra de ingeniería. Está claro que los expertos en la construcción de aviones son más adecuados para diseñar uno nuevo que los encargados de diseñar puentes.

En el próximo Tema se estudiarán ciertas técnicas de carácter general para abordar el diseño de cualquier sistema. Desgraciadamente, estas técnicas son empíricas y no están suficientemente formalizadas como para que se puedan realizar de una manera automática. Para cierto tipo de aplicaciones sencillas o muy específicas (nóminas, cálculos científicos, control de stocks, etc.) existen herramientas tales como lenguajes de cuarta generación, hojas de cálculo, paquetes de cálculo científico, etc., que simplifican mucho la labor de diseño y codificación, pero esto no es la situación general. En los restantes casos, la aplicación de las técnicas no es tan sistemática y es muy importante la habilidad del diseñador.

El método más eficaz para adquirir experiencia en el diseño es participar en alguno de ellos y aprender de los otros diseñadores sus técnicas de trabajo. También es aconsejable estudiar diseños ya realizados y analizar pormenorizadamente las razones por las que se adopta una u otra solución. Estas razones normalmente no se suelen explicar en los libros ya que dependen de cada caso concreto y de las preferencias de cada diseñador, lo que hace muy difícil establecer ninguna regla general. Por tanto, es muy complicado aprender a diseñar exclusivamente leyendo libros.

En los desarrollos de sistemas software de los años 60, el objetivo del diseño era conseguir la mayor eficiencia posible, dado que la memoria de los computadores era escasa y su potencia de cálculo no muy grande. Actualmente y debido al tremendo aumento de los costos de desarrollo, el objetivo fundamental es conseguir que el software sea fácil de mantener y si es posible que se pueda reutilizar en futuras aplicaciones. Para conseguir

ambos objetivos, la etapa de diseño es la más importante de la fase de desarrollo de software.

Durante la etapa de diseño se tiene que pasar de una forma gradual del QUÉ debe hacer el sistema, según se detalla en el documento de requisitos, al CÓMO lo debe hacer, estableciendo la organización y estructura física del software. En esta transformación gradual no siempre está claramente determinado dónde acaba el análisis y dónde empieza el diseño propiamente dicho. En algunos casos se llega a considerar que los requisitos forman parte del diseño externo o funcional del sistema. De cualquier manera, durante el diseño se tiene que pasar de las ideas informales recogidas en el SRD a definiciones detalladas y precisas para la realización del software mediante refinamientos sucesivos.

A lo largo del proceso de diseño se realiza un conjunto de actividades. Estas actividades tienen como objetivo sistematizar cada uno de los aspectos o elementos de la estructura del sistema. Aunque no existe consenso ni en el número ni en la denominación de las actividades, con ellas se persigue un refinamiento progresivo del diseño. Dependiendo de la complejidad del sistema a diseñar algunas de las actividades quedarán englobadas dentro de otras o simplemente desaparecerán. A continuación se enumeran y describen actividades habituales en el diseño de un sistema:

- 1.- **DISEÑO ARQUITECTÓNICO:** Ésta es la primera actividad a realizar y en ella se deben abordar los aspectos estructurales y de **organización del sistema** y su posible división en subsistemas o módulos. Además, se tienen que establecer las relaciones entre los subsistemas creados y definir las interfaces entre ellos. Esta actividad proporcionará una visión global del sistema y por tanto resulta esencial para poder avanzar en las actividades siguientes. Por ejemplo, para el sistema de control de acceso del tema anterior, no tiene ningún sentido abordar el diseño de la visualización de mensajes sin tener absolutamente establecidos todos los módulos del sistema, los tipos de mensajes posibles que cada uno necesitará, la interfase con la que se interrelacionarán los distintos módulos, etc.
- 2.- **DISEÑO DETALLADO:** Con esta actividad se aborda la **organización de los módulos**. Se trata de determinar cuál es la estructura más adecuada para cada uno de los módulos en los que quedó subdividido el sistema global. Simplificando, podemos decir que si se utilizara el lenguaje Modula-2 como notación de diseño esta actividad sería la encargada de la elaboración de los *módulos de definición* para cada

uno de los módulos del programa global. Así, estos módulos de definición serían el resultado fundamental de esta actividad.

También como resultado del diseño detallado, aparecen nuevos módulos que se deben incorporar al diseño global, componentes que es razonable agrupar en un único módulo, o módulos que desaparecen por estar vacíos de contenido. Esto refleja la interrelación entre el diseño arquitectónico y el detallado a pesar de que pudieran parecer actividades puramente secuenciales. Cuando se trata de diseñar sistemas no muy complejos, el diseño detallado no existe como tal actividad y es simplemente un refinamiento del diseño arquitectónico.

- 3.- **DISEÑO PROCEDIMENTAL:** Esta actividad es la encargada de abordar la **organización de las operaciones** o servicios que ofrecerá cada uno de los módulos. Siguiendo con la simplificación anterior del empleo de Modula-2, en esta actividad se diseñan los algoritmos que se emplean en el desarrollo de los *módulos de implementación* para los procedimientos más importantes de cada uno de los módulos del sistema. Normalmente, en esta actividad se detallan en pseudocódigo o PDL solamente los aspectos más relevantes de cada algoritmo (Ver apartado 2.3.4 del Tema 2). Posteriormente en la etapa de codificación se dará forma definitiva a los algoritmos.
- 4.- **DISEÑO DE DATOS:** En esta actividad se aborda la **organización de la base de datos** del sistema y se puede realizar en paralelo con el diseño detallado y procedimental. El punto de partida para esta actividad es el diccionario de datos y los diagramas E-R de la especificación del sistema. Con esta actividad se trata de concretar el formato exacto para cada dato y la organización que debe existir entre ellos. El diseño de datos es de una gran importancia para conseguir el objetivo de que el sistema sea reutilizable y fácilmente mantenible. Sin embargo, una parte muy importante de esta actividad se suele realizar en el momento que se elabora la especificación de requisitos del sistema.
- 5.- **DISEÑO DE LA INTERFAZ DE USUARIO:** En cualquier aplicación software, cada día resulta más importante el esfuerzo de diseño destinado a conseguir un diálogo más ergonómico entre el usuario y el computador. Por ejemplo, la facilidad de aprendizaje y manejo de una aplicación aumenta considerablemente cuando en la interfaz de usuario se emplean ventanas desplegables y ratón en lugar de menús y teclado. Precisamente esta actividad es la encargada de la

organización de la interfaz de usuario. La importancia de esta actividad ha propiciado el desarrollo de técnicas y herramientas específicas que facilitan mucho el diseño. Sin embargo, para conseguir una buena interfaz de usuario hay que tener en cuenta incluso factores psicológicos.

El resultado de todas estas actividades de diseño debe dar lugar a una especificación lo más formal posible de la estructura global del sistema y de cada uno de los elementos del mismo. Esto constituye el "producto del diseño" y se recogerá en el *Documento de Diseño de Software* (en inglés SDD: *Software Design Document*, o también *Software Design Description*). Si la complejidad del sistema así lo aconseja se podrán utilizar varios documentos para describir de forma jerarquizada la estructura global del sistema, la estructura de los elementos en un primer nivel de detalle y en los sucesivos niveles de detalle hasta alcanzar el nivel de codificación con los listados de los programas. Las normas ESA establecen un *Documento de Diseño Arquitectónico* (ADD: *Architectural Design Document*) y un *Documento de Diseño Detallado* (DDD: *Detailed Design Document*) al que se pueden añadir como apéndices los listados de los programas una vez completado el desarrollo. El formato de estos documentos será el objeto del último apartado de este mismo Tema.

3.2 Conceptos de base

La experiencia acumulada a lo largo de los años por los diseñadores de sistemas software ha permitido identificar una serie de conceptos o características de los sistemas y sus estructuras que tienen especial influencia en la calidad de un diseño de software.

Muchos de estos conceptos aparecen en un campo bastante amplio de la actividad de desarrollo de software, y su utilidad no se limita exclusivamente a la tarea de diseño. Algunos de ellos se pueden aplicar de manera natural en el análisis e incluso en otro tipo de actividades dentro y fuera del ámbito del desarrollo de sistemas software. Todas las técnicas que se estudiarán en el próximo Tema están basadas en uno o varios de los conceptos que aquí se recogen.

La aplicación particular de determinadas técnicas da lugar a metodologías específicas, que podrán cambiar o mejorar en el futuro. Sin embargo, los conceptos de base permanecen y se seguirán utilizando probablemente sin grandes variaciones. Lamentablemente, los conceptos por sí mismos no constituyen una técnica o metodología de diseño. La importancia relativa de cada concepto en las diferentes técnicas de diseño varía según la opinión

108 Introducción a la Ingeniería de Software

o preferencias personales de los distintos expertos. A continuación se recoge una lista bastante amplia de los conceptos más interesantes a tener en cuenta en cualquier diseño.

3.2.1 Abstracción

El concepto de abstracción se utiliza en un gran número de actividades humanas. Por ejemplo, en el lenguaje con el que nos comunicamos se utilizan esencialmente un conjunto amplio de abstracciones a las que nos referimos a través de las palabras. Cada nueva palabra sirve para referirnos a un nuevo concepto abstracto aceptado por todos sin necesidad de aclarar cada vez su significado.

Cuando se diseña un nuevo sistema software es importante identificar los elementos realmente significativos de los que consta y además abstraer la utilidad específica de cada uno, incluso más allá del sistema software para el que se está diseñando. Este esfuerzo dará como resultado que el elemento diseñado podrá ser sustituido en el futuro por otro con mejores prestaciones y también podrá ser reutilizado en otros proyectos similares. Estos son dos de los principales objetivos del diseño: conseguir elementos fácilmente mantenibles y reutilizables.

Durante el proceso de diseño se debe aplicar el concepto de abstracción en todos los niveles de diseño. Tomando como ejemplo el sistema para el control de acceso enunciado en el tema anterior, en un primer nivel aparecen abstracciones tales como: Tarjeta, Mensajes, Ordenes, etc. Inicialmente, todos ellos son elementos abstractos y su diseño se puede realizar sin tener en cuenta al sistema de control de acceso concreto en el que se utilizarán. Esto facilitará los cambios futuros y la posibilidad de reutilizarlos en diversos sistemas. En un segundo nivel aparecen nuevas abstracciones como: Clave, Control de Puerta, Comprobar Clave, etc. a los que se aplicarán los mismos criterios. Este proceso de abstracción se puede y se debe continuar con cada elemento software que tengamos que diseñar. El mismo procedimiento es el que se utiliza para diseñar un coche, un avión, una casa, etc. Inicialmente tenemos como elementos abstractos el motor, el chasis, las ruedas, etc. y para diseñar estos elementos a su vez necesitamos filtros, bujías, correas, etc. que si se diseñan convenientemente se podrán utilizar en cualquier modelo de coche.

En el diseño de los elementos software se pueden utilizar fundamentalmente tres formas de abstracción:

- Abstracciones funcionales
- Tipos abstractos
- Máquinas abstractas

Como se explica en el libro de Fundamentos de Programación [Cerrada00], una abstracción funcional sirve para crear expresiones parametrizadas o acciones parametrizadas mediante el empleo de funciones o procedimientos. Para diseñar una abstracción funcional es necesario fijar los parámetros o argumentos que se le deben pasar, el resultado que devolverá en el caso de una expresión parametrizada, lo que se pretende que resuelva y como lo debe resolver (el algoritmo que se debe emplear). Por ejemplo, se puede diseñar una abstracción funcional para Comprobar Clave cuyo parámetro es la clave a comprobar y que devuelva como resultado si/no la clave es correcta.

En cuanto a los tipos abstractos, estos sirven para crear los nuevos tipos de datos que se necesitan para abordar el diseño del sistema. Por ejemplo, un tipo Tarjeta para guardar toda la información relacionada con la tarjeta utilizada: clase de tarjeta, identificador, clave de acceso, datos personales, etc. Además, junto al nuevo tipo de dato se deben diseñar todos los métodos u operaciones que se pueden realizar con él. Por ejemplo, Leer Tarjeta, Grabar Tarjeta, Comprobar Tarjeta, etc.

Una máquina abstracta permite establecer un nivel de abstracción superior a los dos anteriores y en él se define de una manera formal el comportamiento de una *máquina*. Un ejemplo de este tipo de abstracción sería un intérprete de órdenes SQL (Standard Query Language) para la gestión de una base de datos. El lenguaje SQL establece formalmente el comportamiento perfectamente definido para la *máquina abstracta* encargada de manejar la base de datos: construcción, búsqueda e inserción de elementos en la base de datos. Otro ejemplo clásico de máquina abstracta es la definición de un protocolo de comunicaciones en el que se establecen los posibles estados (espera, envío, recepción, inactivo,), las transiciones entre ellos, los tipos de mensajes, etc. que configuran la máquina abstracta como un autómata. También, el sistema para control de acceso utilizado como ejemplo se puede ver como una máquina abstracta si se consigue formalizar todo su comportamiento. Para todos estos casos la tarea de diseño consiste en definir formalmente la máquina abstracta que se necesita.

3.2.2 Modularidad

Casi siempre los proyectos requieren el trabajo simultáneo y coordinado de varias personas. Una forma clásica de conseguir la coordinación es mediante el empleo de un diseño modular. Uno de los primeros pasos que intuitivamente parece claro que se debe dar al abordar un diseño es dividir el sistema en sus correspondientes *módulos* o partes claramente diferenciadas. Esta división permite encargar a personas diferentes el desarrollo de cada módulo y que todas ellas puedan trabajar simultáneamente. Sin embargo, para conseguir la adecuada coordinación, la división en módulos no resulta trivial y además es necesario que las interfaces entre todos ellos queden completamente definidas y correctamente diseñadas. En el próximo Tema se estudiará la técnica de descomposición modular que permite lograr este objetivo.

Además de posibilitar la división del trabajo, las ventajas de utilizar un diseño modular son de todo tipo:

CLARIDAD: Siempre es más fácil de entender y manejar cada una de las partes o módulos de un sistema que tratar de entenderlo como un todo compacto.

REDUCCIÓN DE COSTOS: Resulta más barato desarrollar, depurar, documentar, probar y mantener un sistema modular que otro que no lo es. Hay que tener en cuenta que si el número de módulos crece innecesariamente esta afirmación puede no ser correcta. Cuando hay demasiados módulos se aumentan también las relaciones entre ellos y las interfaces necesarias.

REUTILIZACIÓN: Si los módulos se diseñan teniendo en cuenta otras posibles aplicaciones resultará inmediata su reutilización.

El concepto de modularidad se debe aplicar en el diseño de cualquier sistema, incluso en aquellos para los que existan limitaciones de tiempo o memoria que impidan que en su codificación se puedan emplear módulos compilados por separado. Por ejemplo, si ciertas operaciones son críticas y se deben realizar en un tiempo muy corto, no se pueden emplear subrutinas con las que se gastaría un tiempo precioso en cada llamada. En este caso se pueden emplear macros para sustituir a las subrutinas sin que esto afecte para nada al diseño.

La modularidad es un concepto de diseño que no debe estar ligado a la etapa de codificación y mucho menos al empleo de un determinado

lenguaje de programación. Sin embargo, históricamente se ha asociado el concepto de módulo a determinadas estructuras de los lenguajes de programación. En lenguajes como FORTRAN, Pascal, etc. un módulo sólo se puede asociar a una subrutina, procedimiento o función. Estas estructuras resultan insuficientes cuando se quiere conseguir que cada miembro del equipo de trabajo pueda desarrollar su actividad por separado y de forma coordinada y tampoco permiten agrupar en una misma entidad sólo datos o datos y las operaciones que los manejan. Otros lenguajes posteriores sí que están dotados de estructuras específicas para estos fines. Los package de ADA o los MODULE de Modula-2 son los ejemplos más claros. La ventaja que supone la utilización de estos últimos lenguajes es que resulta inmediato trasladar el diseño a estructuras del lenguaje.

3.2.3 Refinamiento

El concepto de refinamiento resulta imprescindible para poder realizar cualquier tipo de diseño. En un diseño, sea del tipo que sea, siempre se parte inicialmente de una idea no muy concreta que se va refinando en sucesivas aproximaciones hasta perfilar el más mínimo detalle. Fue Niklaus Wirth [Wirth71] el primero que recomendó la aplicación de refinamientos sucesivos para el diseño de programas.

Las especificaciones recogidas en el documento SRD siempre terminan siendo expresadas en un lenguaje de programación de alto nivel. Sin embargo, el alto nivel de los lenguajes de programación es sólo relativo, y esos lenguajes están más próximos al lenguaje del computador que al nuestro propio. La forma natural de acercar ambos lenguajes (el natural y el de programación) es utilizando el refinamiento: el objetivo global de un nuevo sistema software expresado en su especificación se debe refinar en sucesivos pasos hasta que todo quede expresado en el lenguaje de programación del computador. Además, todos los lenguajes tienen la posibilidad de declarar subprogramas (funciones y procedimientos) capaces de expresar los pasos del refinamiento como acciones o expresiones parametrizadas. Por tanto, el proceso de refinamiento se puede dar por concluido cuando todas las acciones y expresiones quedan refinadas en función de otras acciones o expresiones o bien en función de las instrucciones básicas del lenguaje empleado.

El uso directo e irrestringido de refinamientos sucesivos conduciría a programas monolíticos. El refinamiento directo debe dejar de aplicarse si el tamaño del programa resultante para una sola acción global excede de un límite razonable, establecido típicamente en unas dos páginas de texto. Si

112 Introducción a la Ingeniería de Software

se excede este límite, convendrá aplicar las ideas de abstracción y modularidad para fragmentar una operación global en varias separadas.

En el próximo Tema se estudiará la técnica de diseño descendente en la que se emplean los refinamientos sucesivos como mecanismo básico de diseño.

3.2.4 Estructuras de datos

La organización de la información es una parte esencial del diseño de un sistema software. Las decisiones respecto a que datos se manejan y la estructura de cada uno de ellos afectan de una forma decisiva al resto del diseño: abstracciones, módulos, algoritmos, etc. Por ejemplo, una decisión errónea respecto a si un determinado resultado intermedio se guarda o se calcula nuevamente cada vez que se necesite, puede hacer tan lento el sistema que resulte inservible. Esto mismo puede suceder si la estructura de datos elegida complica de manera excesiva la búsqueda de un determinado dato.

De la importancia de las estructuras de datos en el diseño son buena prueba las metodologías de diseño orientadas a los datos, tales como las propuestas Jackson [Jackson75], [Jackson83] y Warnier [Warnier81]. En estas metodologías es precisamente la estructura de datos el punto de partida del que se arranca para realizar el diseño.

El estudio de la gran diversidad de estructuras de datos posibles queda fuera del alcance e interés de este libro. Además, existen muchos y buenos libros dedicados al estudio de las estructuras de datos más adecuadas para resolver los problemas clásicos que se presentan en la mayoría de los casos [Aho83], [Wirth80], [Collado87]. Para el diseño basta tener en cuenta, en general, las estructuras fundamentales, tales como:

- Registros
- Conjuntos
- Formaciones
- Listas, Pilas, Colas
- Árboles
- Grafos
- Tablas
- Ficheros

Es labor del diseñador la búsqueda, a partir de estas estructuras básicas, de la combinación más adecuada para lograr aquella estructura o estructuras que den respuesta a las necesidades del sistema planteadas en su

especificación. Como siempre, resulta trascendental la experiencia acumulada de diseños anteriores.

3.2.5 Ocultación

Para poder utilizar correctamente un programa no es necesario conocer cuál es su estructura interna. Por ejemplo, el usuario de un procesador de textos puede ser cualquier persona que no posea absolutamente ningún conocimiento de informática y mucho menos de la estructura interna del programa. De forma semejante se puede pensar respecto a cada uno de los módulos en los que se divide el programa en su diseño. Al programador "usuario" de un módulo desarrollado por otro programador del equipo puede quedarle completamente *oculta* la organización de los datos internos que maneja y el detalle de los algoritmos que emplea.

El concepto de ocultación aplicado al desarrollo de software fue propuesto por Parnas [Parnas72]. Cuando se diseña la estructura de cada uno de los módulos de un sistema, se debe hacer de tal manera que dentro de él queden ocultos todos los detalles que resultan irrelevantes para su utilización. Tomando como ejemplo el sistema de control de acceso, en el módulo para el control de puerta deben permanecer ocultos todos los detalles específicos del manejo de la puerta concreta utilizada (eléctrica, hidráulica, neumática, etc.) y por tanto del modo de actuar para conseguir su apertura o cierre (tipo de señal y temporizaciones necesarias). Al "usuario" de este módulo sólo le interesa conocer cómo se le pasa la orden de que la puerta se abra o se cierre.

Con carácter general, se trata de ocultar al "usuario" todo lo que pueda ser susceptible de cambio en el futuro y además es irrelevante para el uso (hardware, sistema operativo, compilador, idioma, etc.). Sin embargo, se muestra en la interfaz sólo aquello que resultará invariable con cualquier cambio. Las ventajas de aplicar este concepto son las siguientes:

DEPURACIÓN: Resulta más sencillo detectar qué módulo concreto no funciona correctamente. También es más fácil establecer estrategias y programas de prueba que verifiquen y depuren cada módulo por separado en base a lo que hacen sin tener en cuenta cómo lo hacen.

MANTENIMIENTO: Cualquier modificación u operación de mantenimiento que se necesite en un módulo concreto no afectará al resto de los módulos del sistema. Si se cambian el hardware, el sistema operativo, etc. sólo se tiene que modificar el módulo encargado de

114 Introducción a la Ingeniería de Software

ocultar estos detalles. El resto de módulos que usen el módulo modificado permanecerán sin cambios.

Aunque el concepto de ocultación se puede y se debe aplicar con cualquier metodología o lenguaje de programación, estructuras como los packages de ADA o los MODULE de Modula-2 y, por supuesto, los lenguajes de programación orientados a objetos, tienen la ventaja de facilitar de manera considerable la labor de diseño cuando se aplica el concepto de ocultación.

3.2.6 Genericidad

Un posible enfoque de diseño es agrupar aquellos elementos del sistema que utilizan estructuras semejantes o que necesitan de un tratamiento similar. Si se prescinde de los aspectos específicos de cada elemento concreto es bastante razonable diseñar un elemento genérico con las características comunes a todos los elementos agrupados. Posteriormente, cada uno de los elementos agrupados se pueden diseñar como un caso particular del elemento genérico. Esta es la idea básica que aporta el concepto de genericidad.

Para ilustrar este concepto con un ejemplo, supongamos que se trata de diseñar un sistema operativo multitarea que necesita atender las órdenes que le llegan de los distintos terminales. Una de las órdenes habituales será imprimir por una cualquiera de las impresoras disponibles. Por otro lado, es normal que desde varios terminales simultáneamente se necesite acceder a ficheros o bases de datos compartidas. Cada una de estas actividades necesita de un módulo gestor para decidir en qué orden se atienden las peticiones. La forma más sencilla de gestión es poner en cola las órdenes, peticiones de impresión o peticiones de acceso a ficheros o bases de datos compartidas. Inmediatamente surge la necesidad de una estructura genérica en forma de cola para la que se necesitan también unas operaciones genéricas tales como poner en cola, sacar el primero, iniciar la cola, ver cuántos hay en cola, etc. Evidentemente en cada caso el tipo de información que se guarda en la cola es diferente: tipo de orden a ejecutar, texto a imprimir, dato a consultar, etc. A partir de la cola genérica y con un diseño posterior más detallado se tendrá que decidir la estructura concreta de las distintas colas necesarias y si para alguna de ellas es conveniente utilizar prioridades, lo que daría lugar a operaciones específicas.

Indudablemente el concepto de genericidad es de gran utilidad en el diseño y da lugar a soluciones simples y fáciles de mantener. Sin embargo, la implementación de los elementos genéricos obtenidos como resultado del

diseño puede resultar bastante compleja e incluso desvirtuar el propio concepto de genericidad. Por ejemplo, un lenguaje de programación, tal como Pascal o Modula-2, impone unas restricciones muy fuertes en el manejo de datos de distinto tipo y las posibles operaciones entre ellos. Con estos lenguajes es necesario definir un tipo distinto de cola para cada tipo de elemento a almacenar, y aunque las operaciones sean esencialmente idénticas para los distintos datos almacenados, también es necesario implementar como operaciones distintas las destinadas a manejar cada tipo de cola. Esto es, si tenemos los tipos:

```
Cola_de_enteros
Cola_de_reales
Cola_de_caracteres
```

es necesario implementar las operaciones:

Poner_entero	(en la cola de enteros)
Poner_real	(en la cola de reales)
Poner_caracter	(en la cola de caracteres)
Sacar_entero	...
Sacar_real	...

....

esta solución, que es la única posible en estos lenguajes, desvirtúa de forma evidente la genericidad propuesta en el diseño.

El lenguaje de programación ADA tiene la posibilidad de declarar elementos genéricos (**generic**) tales como tipos, constantes, subprogramas y objetos, que se pueden utilizar para parametrizar un procedure o un package. En este caso, tendríamos los siguientes elementos genéricos:

```
generic
    type COLA is private;
    procedure Poner_en_Cola( X: in COLA );

```

....

Ahora, es posible obtener los procedimientos para cada tipo de dato almacenado en la cola de la siguiente forma:

```
procedure Poner_Entero is new Poner_en_Cola( INTEGER );
procedure Poner_Real is new Poner_en_Cola( REAL );
procedure Poner_Caracter is new Poner_en_Cola( CHARACTER );
```

116 Introducción a la Ingeniería de Software

```
procedure Poner_OtroTipo is new Poner_en_Cola( OtroTipo );
```

.....

sin necesidad de desarrollar implementaciones distintas para cada uno de ellos.

3.2.7 Herencia

Otro enfoque posible del diseño cuando hay elementos con características comunes es establecer una clasificación o jerarquía entre esos elementos del sistema partiendo de un elemento "padre" que posee una estructura y operaciones básicas. Los elementos "hijos" heredan del "padre" su estructura y operaciones para ampliarlos, mejorarlos o simplemente adaptarlos a sus necesidades. A su vez los elementos "hijos" pueden tener otros "hijos" que hereden de ellos de una forma semejante. De manera consecutiva se puede continuar con los siguientes descendientes hasta donde sea necesario. Esta es la idea fundamental en la que se basa el concepto de herencia.

El mecanismo de herencia permite reutilizar una gran cantidad de software ya desarrollado. Habitualmente, en un nuevo proyecto siempre se parte de otros proyectos ya realizados de los que se toman todos aquellos elementos que nos pueden resultar útiles bien directamente, o con pequeñas modificaciones, o de forma combinada entre varios, etc. Facilitar esta labor es uno de los objetivos fundamentales del concepto de herencia.

Si tratamos de realizar un software para dibujo asistido por computador, las figuras que se manejan se podrían clasificar del siguiente modo:

FIGURAS:

ABIERTAS:

TRAZO RECTO:

LÍNEA RECTA

TRAZO CURVO:

SEGMENTO DE CIRCUNFERENCIA

CERRADAS:

ELIPSES:

CÍRCULOS

POLÍGONOS:

TRIÁNGULOS:

EQUILÁTEROS

RECTÁNGULOS:

CUADRADOS

Aquí, el elemento "padre" será FIGURAS. Las operaciones básicas con cualquier tipo de figura podrían ser:

- *Desplazar*
- *Rotar*
- *Pintar*

Aunque estas operaciones las heredan todos los tipos de figura, normalmente deberán ser adaptadas en cada caso. Así, *Rotar* los CÍRCULOS significa dejarlos tal cual están.

Por otro lado, al concretar los elementos "hijos" aparecen nuevas operaciones que no tenían sentido en el elemento "padre". Así, la operación de calcular el *Perímetro* solo tiene sentido para figuras CERRADAS. De forma semejante sólo los POLÍGONOS tendrán una operación específica para determinar la posición de sus Vértices, sólo en los RECTÁNGULOS se puede hablar de las longitudes de sus lados *LadoUno* y *LadoDos* y de su *Diagonal*, etc.

El concepto de herencia está muy ligado a las metodologías de análisis y diseño de software orientadas a objetos. Aunque en este apartado sólo se ha mencionado la herencia simple entre un "hijo" y un único "padre", es posible realizar diseños en los que se tengan en cuenta herencias múltiples de varios "padres". En esencia se trata de aprovechar elementos ya desarrollados para producir otro nuevo que combine simultáneamente las estructuras y propiedades de más de un elemento anterior.

La aplicación del concepto de herencia en la fase de diseño es posible sin mayor problema. Sin embargo, para trasladar de una manera directa y sencilla los resultados del diseño a la codificación es aconsejable utilizar un lenguaje de programación orientado a objetos. Algunos de estos lenguajes son los siguientes:

- Smalltalk
- Object Pascal
- Objetive-C
- Eiffel
- C++

Todos ellos disponen del mecanismo de herencia. En [Budd94] se detallan las particularidades del mecanismo de herencia propuesto por cada lenguaje.

3.2.8 Polimorfismo

La definición de polimorfismo que encontramos en el diccionario es la siguiente:

- **Polimorfismo.** Propiedad de los cuerpos que pueden cambiar de forma sin variar su naturaleza.

En nuestro caso, el concepto de polimorfismo engloba distintas posibilidades utilizadas habitualmente para conseguir que un mismo elemento software adquiera varias formas simultáneamente:

1.- El concepto de genericidad introducido anteriormente es una manera de lograr que un elemento genérico pueda adquirir distintas formas cuando se particulariza su utilización.

2.- El concepto de polimorfismo está muy unido al concepto de herencia. Como se indicó anteriormente, las estructuras y operaciones heredadas se pueden adaptar a las necesidades concretas del elemento "hijo": no es lo mismo *Rotar ELIPSES* que *CÍRCULOS*. Por tanto, la operación de rotar tiene distintas formas según el tipo de figura a la que se destina y es en el momento de la ejecución del programa cuando se utiliza una u otra forma de rotación. Este tipo de polimorfismo se conoce como de **Anulación**, dado que la rotación específica para los círculos anula la más general para las elipses.

En algunos casos, no hay anulación real dado que no tiene sentido diseñar e implementar una operación general que abarque todas las posibles situaciones que se puedan plantear con todos los posibles descendientes. Para estos casos se plantea un polimorfismo **Diferido** en el cual se plantea la necesidad de la operación para el elemento "padre", pero su concreción se deja diferida para que cada uno de los elementos "hijos" concrete su forma específica. Este es el caso de la operación *Rotar FIGURAS* que resultaría muy compleja y además probablemente inútil.

3.- Por último, existe otra posibilidad de polimorfismo que se conoce como **Sobrecarga**. En este caso quienes adquieren múltiples formas son los operadores, funciones o procedimientos. El ejemplo clásico de polimorfismo por sobrecarga lo constituyen los operadores matemáticos: +, -, * y /. Estos operadores son similares para operaciones entre enteros, reales, conjuntos o matrices. Sin embargo, en cada caso el tipo de operación que se invoca es distinto: la suma

entre enteros es mucho más sencilla y rápida que la suma entre reales y por otro lado con el mismo operador + se indica la operación suma entre valores numéricos o la operación de unión entre dos conjuntos o la suma de matrices. En el diseño de esta **Sobrecarga** de operadores, a pesar de que nos resulta tan familiar, se ha utilizado el concepto de polimorfismo. Este mismo concepto se debe aplicar cuando se diseñan las operaciones a realizar entre elementos del sistema que se trata de desarrollar. Por ejemplo, se puede utilizar el operador + para unir ristas de caracteres o realizar resúmenes de ventas.

El polimorfismo por **Sobrecarga** también se puede utilizar con funciones o procedimientos. Así, podemos tener la función *Pintar* para FIGURAS y diseñar también una función *Pintar* para representar en una gráfica los valores de una tabla o llenar con distintos trazos una región de un dibujo, etc.

Todas estas posibilidades de polimorfismo redundan en una mayor facilidad para realizar software reutilizable y mantenible. Si se quiere diseñar un módulo que utilice mucha gente, los elementos que se propongan deberán resultar familiares al mayor número posible de usuarios potenciales.

Al igual que la herencia, el concepto de polimorfismo está ligado a las metodologías orientadas a objetos y para simplificar la implementación es conveniente utilizar algún lenguaje orientado a objetos de los indicados en el apartado anterior.

3.2.9 Concurrencia

Los computadores disponen de una gran capacidad de proceso que no se debe desaprovechar. Mientras el operador decide la siguiente tecla que debe pulsar o durante el tiempo en que se está imprimiendo, el computador puede realizar de manera concurrente otras tareas. Sobre todo en los sistemas de tiempo real existe la necesidad de aprovechar toda la capacidad de proceso del computador para atender a todos los eventos que se producen y en el mismo instante en que se producen. En estos casos normalmente se establecen tiempos máximos de respuesta ante determinados eventos (alarmas, fallos, errores, etc.).

Cuando se trata de diseñar un sistema con restricciones de tiempo se debe tener en cuenta lo siguiente:

120 Introducción a la Ingeniería de Software

- 1.- TAREAS CONCURRENTES: Determinar qué tareas se deben ejecutar en paralelo para cumplir con las restricciones impuestas. Se deberá prestar especial atención a aquellas tareas con tiempos de respuesta más críticos y aquellas otras que se ejecutarán con mayor frecuencia. Al diseño y codificación de ambos grupos de tareas se debe prestar especial cuidado, pues de ello depende que se cumplan finalmente las restricciones.
- 2.- SINCRONIZACIÓN DE TAREAS: Determinar los puntos de sincronización entre las distintas tareas. Normalmente las tareas nunca funcionan cada una por separado sin tener en cuenta a las demás. Cuando una tarea T1 se encarga de obtener un resultado que debe utilizar otra T2, ambas se deben sincronizar. La tarea T2 esperará hasta que T1 tenga disponible un nuevo resultado y en el caso de que T2 no haya utilizado el anterior resultado es T1 la que debe esperar. Para la sincronización entre tareas se pueden utilizar semáforos o mecanismos de tipo *rendezvous* disponibles en los sistemas operativos o en lenguajes de programación concurrentes (Ada, Pascal Concurrente, etc.).
- 3.- COMUNICACIÓN ENTRE TAREAS: Determinar si la cooperación se basa en el empleo de datos compartidos o mediante el paso de mensajes entre las tareas. En cualquier caso, el diseñador debe concretar la estructura de los datos compartidos o de los mensajes que se intercambian. También se debe concretar qué tareas serán las productoras de los datos y qué otras serán las consumidoras. En el caso de utilizar datos compartidos se tendrá que evitar que los datos puedan ser modificados en el momento de la consulta, lo que daría lugar a errores muy difíciles de detectar. En este caso, es necesario utilizar mecanismos como semáforos, monitores, regiones críticas, etc. para garantizar la exclusión mutua entre las distintas tareas que modifican y consultan los datos compartidos. Estos mecanismos están disponibles en los sistemas operativos o en lenguajes de programación concurrente.
- 4.- INTERBLOQUEOS (*deadlock*): Estudiar los posibles interbloqueos entre tareas. Un interbloqueo se produce cuando una o varias tareas permanecen esperando por tiempo indefinido una situación que no se puede producir nunca. Por ejemplo, el caso más sencillo de interbloqueo entre dos tareas se produce cuando ambas simultáneamente se quedan esperando algún resultado que se deben proporcionar mutuamente la una a la otra y ninguna puede avanzar.

El concepto de concurrencia introduce una complejidad adicional al sistema y por tanto sólo se debe utilizar cuando no exista una solución de tipo secuencial sencilla que cumpla con los requisitos especificados en el documento SRD. En general, se requieren conocimientos específicos para el diseño de sistemas concurrentes o de tiempo real [BenAri89] y en gran número de casos la solución finalmente adoptada resulta ser completamente a la medida.

3.3 Notaciones para el diseño

Para concretar y precisar las descripciones resultantes de todo el proceso de diseño se pueden utilizar múltiples notaciones. Como sucedía con la especificación, cada notación de diseño ha sido propuesta dentro de una metodología concreta, y un mismo símbolo (rectángulo, círculo, etc.) puede tener significados distintos según los casos. Además, debido a la evolución de las metodologías y según las propuestas de distintos expertos o fabricantes de herramientas de diseño, se utilizan distintos símbolos para representar una misma idea. De las decenas de notaciones que existen, en este apartado se ha tratado de recoger aquellas que disfrutan de mayor aceptación, empleando para ello los símbolos utilizados más comúnmente. A pesar de todo, si se utiliza una nueva herramienta o metodología de diseño, es muy aconsejable cerciorarse antes del significado de cada símbolo. Con este repaso panorámico se trata de facilitar el aprendizaje de cualquier notación nueva situándola dentro del contexto adecuado.

El objetivo fundamental de cualquier notación es resultar precisa, clara y sencilla de interpretar en el aspecto concreto que describe. Se trata sobre todo de evitar ambigüedades e interpretaciones erróneas del diseño. En este sentido lo más adecuado sería emplear notaciones formales de tipo quasi matemático. Sin embargo, sólo una parte muy pequeña del resultado de un diseño se suele describir utilizando únicamente notaciones formales.

Resulta muy difícil establecer dónde queda la frontera entre la especificación y el diseño de las características externas o estructurales de un sistema. Así, resulta normal que para ciertos aspectos de diseño se empleen notaciones semejantes e incluso las mismas del análisis. Para estas notaciones nos remitiremos a lo dicho en el apartado del Tema anterior dedicado a las notaciones para la especificación.

Según el aspecto del sistema que se trata de describir es aconsejable emplear una u otra clase de notación. De acuerdo con este criterio, clasificaremos las notaciones en los siguientes grupos:

122 Introducción a la Ingeniería de Software

- Notaciones estructurales
- Notaciones estáticas o de organización
- Notaciones dinámicas o de comportamiento
- Notaciones híbridas

El lenguaje natural como notación queda fuera de esta clasificación puesto que se puede utilizar para cubrir cualquier aspecto del sistema. Siempre que se utilice, se deberán tener en cuenta las recomendaciones dadas en el Tema anterior y en todo caso, se procurará utilizar solamente como complemento a otras notaciones.

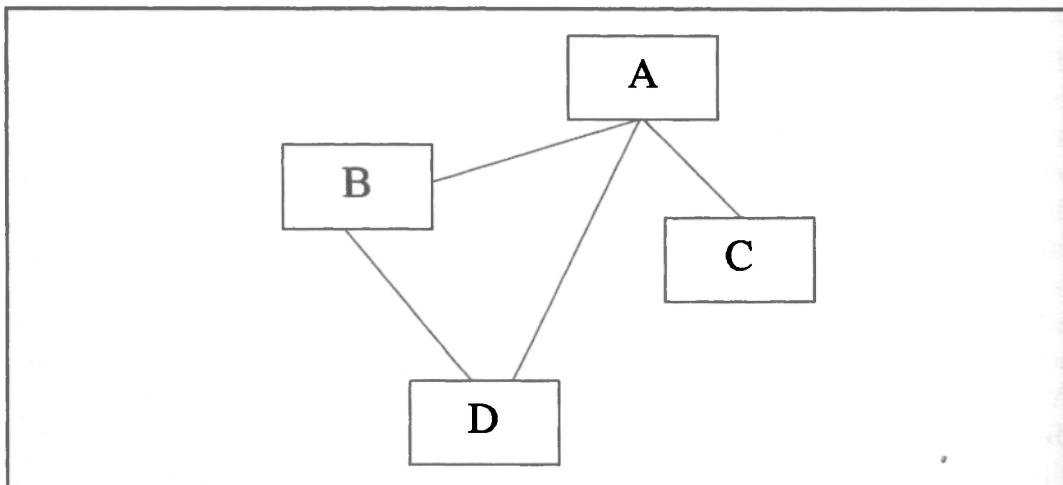


Figura 3.1 Diagrama de Bloques

3.3.1 Notaciones estructurales

Estas notaciones sirven para cubrir un primer nivel del diseño arquitectónico y con ellas se trata de desglosar y estructurar el sistema en sus partes fundamentales. En principio, cualquier notación informal utilizada en otras actividades de ingeniería y que permita describir la estructura global de un sistema, puede ser válida. Una notación habitual para desglosar un sistema en sus partes es el empleo de diagramas de bloques.

En la figura 3.1 se muestra el diagrama de bloques de un sistema que está dividido en cuatro bloques y en el que además se indican las conexiones que existen entre ellos. En algunos casos estas conexiones también pueden reflejar una cierta clasificación o jerarquía de los bloques. Cuando la

notación es informal, es necesario concretar explícitamente si con el diagrama se indica algo más que la simple conexión existente entre los bloques.

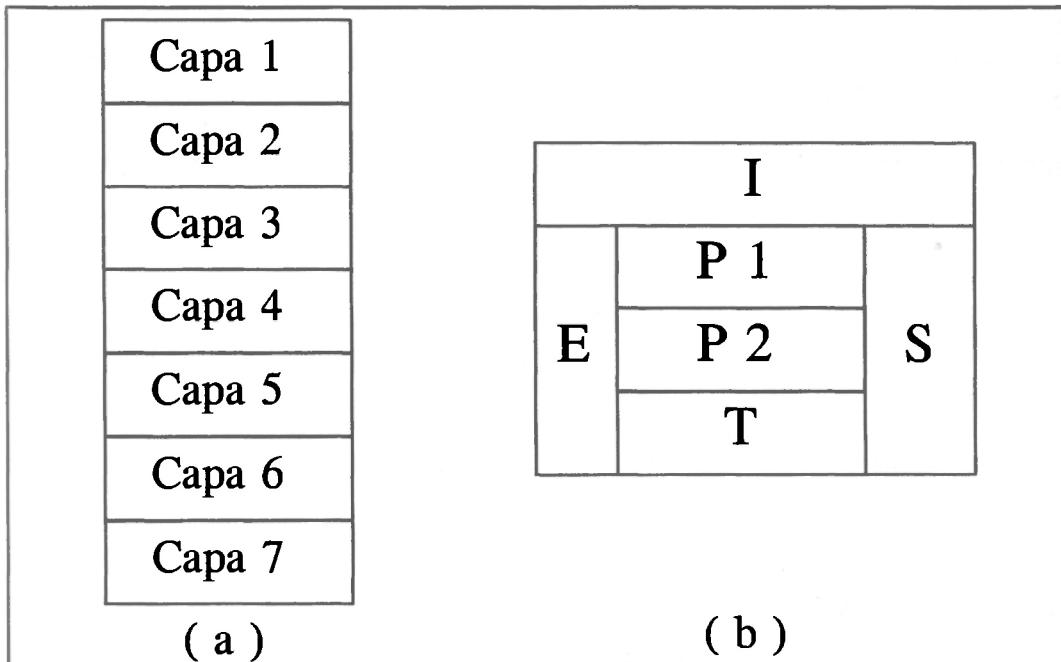


Figura 3.2 Diagramas de cajas adosadas

Otra notación informal para estructurar un sistema se muestra en la figura 3.2. En este caso se emplean "cajas adosadas" para delimitar los bloques. La conexión entre los bloques se pone de manifiesto cuando entre dos cajas existe una frontera común.

En el diagrama (a) de la figura 3.2 se muestra un sistema organizado por capas. Esta estructura se emplea, por ejemplo, en el estándar OSI de comunicaciones. En él se establecen 7 capas o niveles: Físico, Enlace, Red, Transporte, Sesión, Presentación y Aplicación. A grandes rasgos, cada capa es un subsistema hardware o software encargado de incorporar a los mensajes que se envían cierta información redundante adicional. La misma capa al recibir un mensaje se encarga de eliminar las redundancias introducidas en el envío y comprobar si la información recibida es correcta. Entre capas adyacentes está definida una interfaz completamente estándar. Todo esto posibilita que las modificaciones en la implementación de una capa nunca afecten al resto y que el software de comunicaciones de distintos fabricantes sea compatible.

124 Introducción a la Ingeniería de Software

Otro ejemplo es la propuesta de Hatley [Hatley87], que sugiere como plantilla de arquitectura para estructurar los sistemas la que se muestra en la figura 3.2 (b). Los subsistemas generales propuestos en la plantilla son: E (Entrada), S (Salida), T (Test para la autocomprobación y el mantenimiento), I (Interfase de usuario) y P1, P2, (Procesado y control del sistema). A continuación se presentan algunas notaciones más formales para describir la estructura de un sistema.

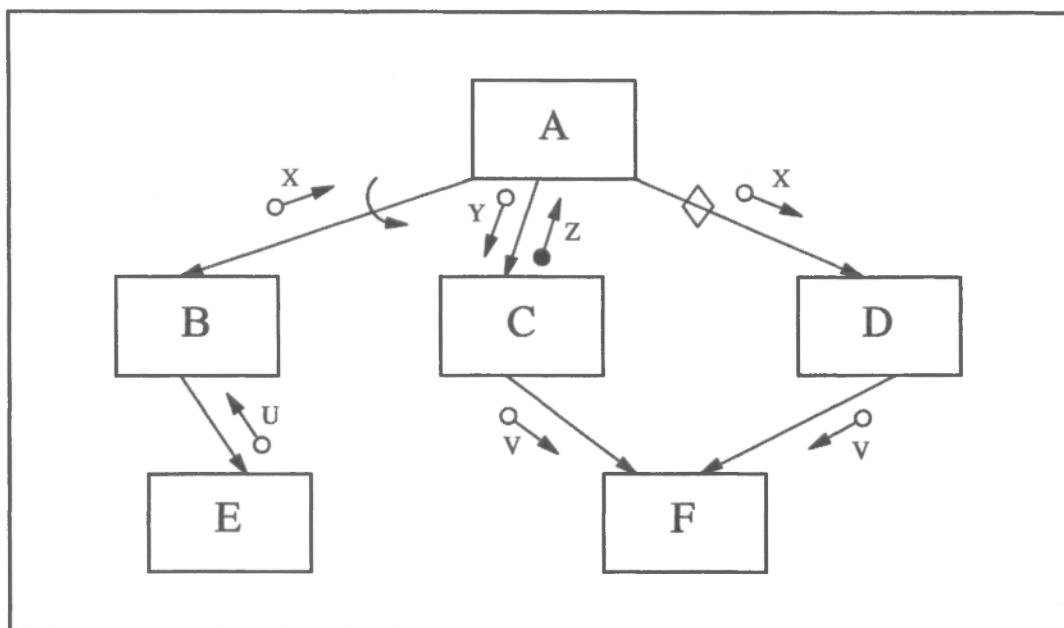


Figura 3.3 Diagrama de estructura

3.3.1.1 Diagramas de estructura

Estos diagramas fueron propuestos por Yourdon [Yourdon79] y Myers [Myers78] para describir la estructura de los sistemas software como una jerarquía de subprogramas o módulos en general. En la figura 3.3 se muestra un diagrama de estructura. El significado de los símbolos utilizados es el siguiente:

RECTÁNGULO: representa un módulo o subprograma cuyo nombre se indica en su interior

LÍNEA: une a dos rectángulos e indica que el módulo superior llama o utiliza al módulo inferior. A veces la línea acaba en una flecha junto al módulo inferior al que apunta.

ROMBO: se sitúa sobre una Línea e indica que esa llamada o utilización es opcional. Se puede prescindir de este símbolo si en la posterior descripción del módulo superior, mediante pseudocódigo u otra notación, se indica que el módulo inferior se utiliza sólo opcionalmente.

ARCO: se sitúa sobre una Línea e indica que esa llamada o utilización se efectúa de manera repetitiva. Se puede prescindir de este símbolo si en la posterior descripción del módulo superior, mediante pseudocódigo u otra notación, se indica que el módulo inferior se utiliza de forma repetitiva.

CÍRCULO CON FLECHA: se sitúa en paralelo a una Línea y representa el envío de los datos, cuyo nombre acompaña al símbolo, desde un módulo a otro. El sentido del envío lo marca la flecha que acompaña al círculo. Para indicar que los datos son una información de control se utiliza un círculo relleno. Una información de control sirve para indicar SI/NO, o bien un estado: Correcto / Repetir / Error / Espera / Desconectado /

En el Tema siguiente se verán técnicas de diseño que permiten obtener el diagrama de estructura a partir de la especificación. Cuando el diseño realizado es razonablemente correcto, el resultado es un diagrama en forma de árbol mostrando la jerarquización de los módulos. Como sucede en la figura 3.3, es normal que varios módulos superiores utilicen un mismo módulo inferior, lo que significa que ese módulo se reutiliza en varios puntos del sistema.

El diagrama de estructura no establece ninguna secuencia concreta de utilización de los módulos y tan sólo refleja una organización estática de los mismos. Por ejemplo, en la figura 3.3 el módulo A utiliza a los módulos B, C y D en cualquier orden sin tener en cuenta su situación a la izquierda, a la derecha o en el centro.

Observando el diagrama de la figura 3.3 se puede decir lo siguiente:

- El módulo E proporciona el dato U al módulo B. Este dato es una entrada al sistema (teclado, ratón, generación por cálculo, etc.).
- El módulo B transforma el dato U y obtiene el dato X que proporciona al módulo A. El módulo A utiliza al módulo B de forma repetitiva.

126 Introducción a la Ingeniería de Software

- El módulo A proporciona al módulo C el dato Y y este último le devuelve al módulo A la información de control Z.
- El módulo C proporciona al módulo F el dato V. También, el módulo D proporciona al módulo F el mismo dato V.
- El módulo F no devuelve nada a ninguno de sus módulos superiores. Eso ocurre, por ejemplo, si se encarga de realizar la salida de los datos que se le envían (impresora, pantalla, etc.).
- El módulo D se utiliza opcionalmente por A y recibe de este último el dato X que a su vez A recibió de B.
- El diagrama es estrictamente jerarquizado y no existen Líneas entre módulos de un mismo nivel.

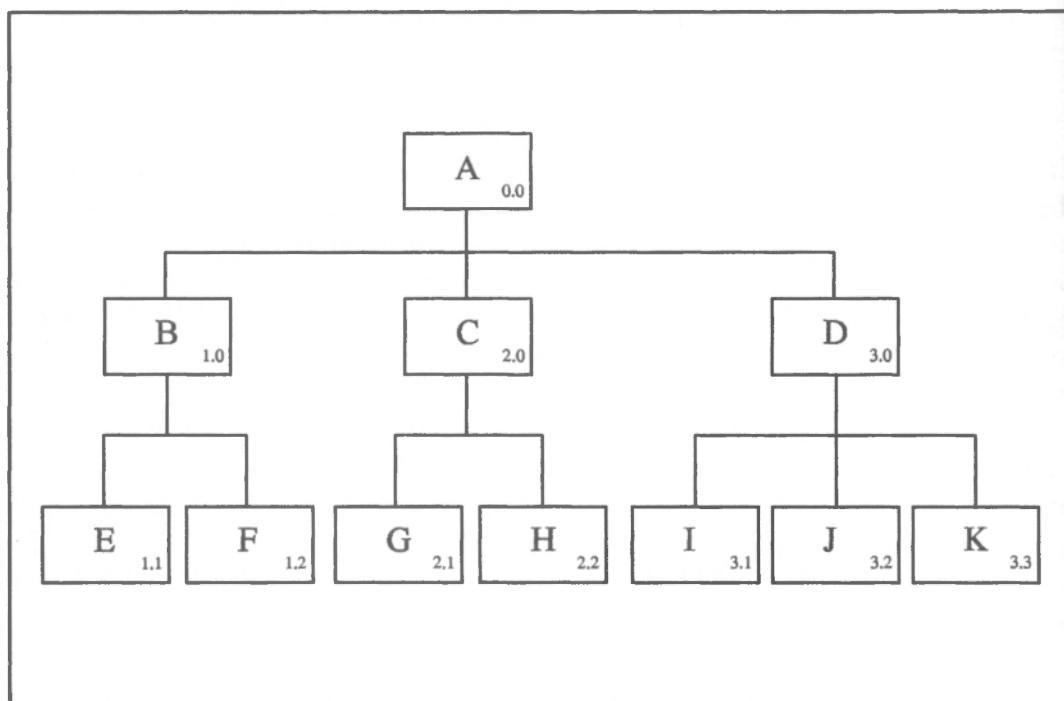


Figura 3.4 Diagrama HIPO de contenidos

3.3.1.2 Diagramas HIPO

Los diagramas HIPO (Hierarchy-Input-Process-Output) son una notación propuesta por IBM para facilitar y simplificar el diseño y desarrollo de sistemas software, destinados fundamentalmente a gestión (facturación,

contabilidad, nóminas, inventario, etc.). La mayoría de estos sistemas se pueden diseñar como una estructura jerarquizada (Hierarchy) de subprogramas o módulos. Además, el formato de todos los módulos se puede adaptar a un mismo patrón caracterizado por los datos de entrada (Input), el tipo de proceso (Process) que se realiza con los datos y los resultados de salida que proporciona (Output).

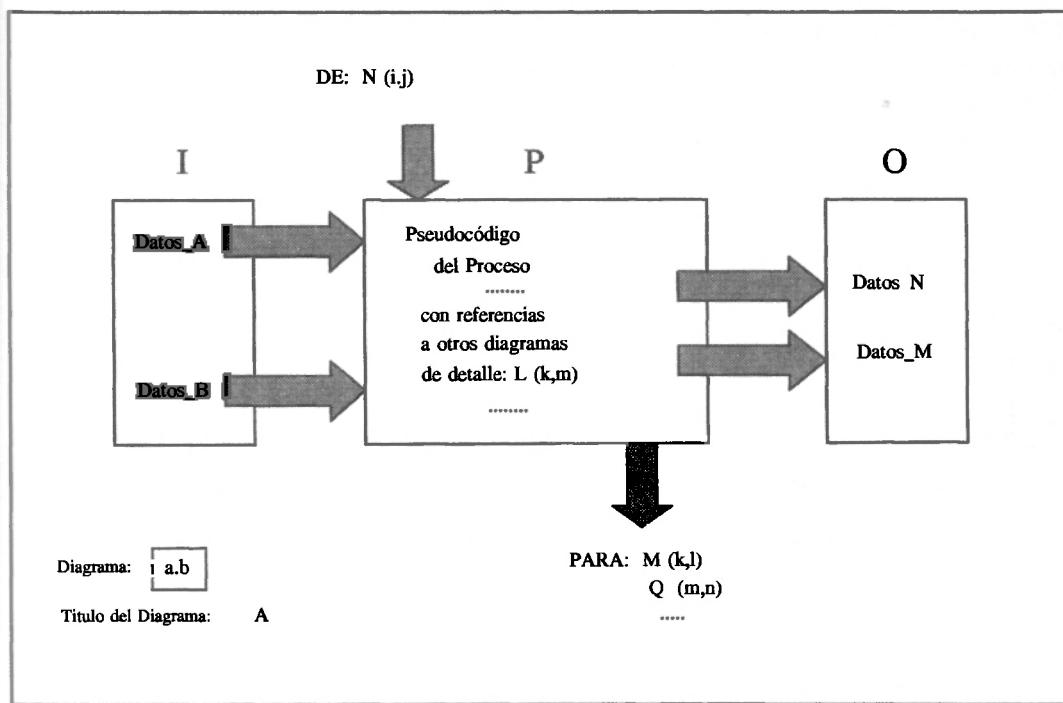


Figura 3.5 Diagrama HIPO de detalle

La figura 3.4 muestra un diagrama HIPO de contenidos que se utiliza para establecer la jerarquía entre los módulos del sistema. Como se puede observar, es asimilable a un diagrama de estructura simplificado en el que no se indican los datos que se intercambian los módulos. Cada módulo tiene un nombre (A, B, C, ...) y una referencia al correspondiente diagrama HIPO de detalle (0.0, 1.0,).

En la figura 3.5 tenemos el diagrama HIPO de detalle del módulo cuyo nombre es A y referencia (a.b). Los diagramas de detalle constan de tres zonas: Entrada (I), Proceso (P), Salida (O). En las zonas de entrada y salida se indican respectivamente los datos que entran (Datos_A y Datos_B) y salen (Datos_N y Datos_M) del módulo. En la zona central se detalla el pseudocódigo del proceso con referencia a otros diagramas de detalle de nivel inferior en la jerarquía. La lista de los diagramas referenciados se

128 Introducción a la Ingeniería de Software

listan a continuación de la partícula PARA: Por la parte superior y a continuación de la partícula DE: se indica el diagrama de detalle de nivel superior: N(i,j).

3.3.1.3 Diagramas de Jackson

Esta notación forma parte de la metodología desarrollada por Jackson [Jackson75], [Jackson83] para diseñar sistemas software a partir de las estructuras de sus datos de entrada y salida. El proceso de diseño es bastante sistemático y se lleva a cabo en tres pasos:

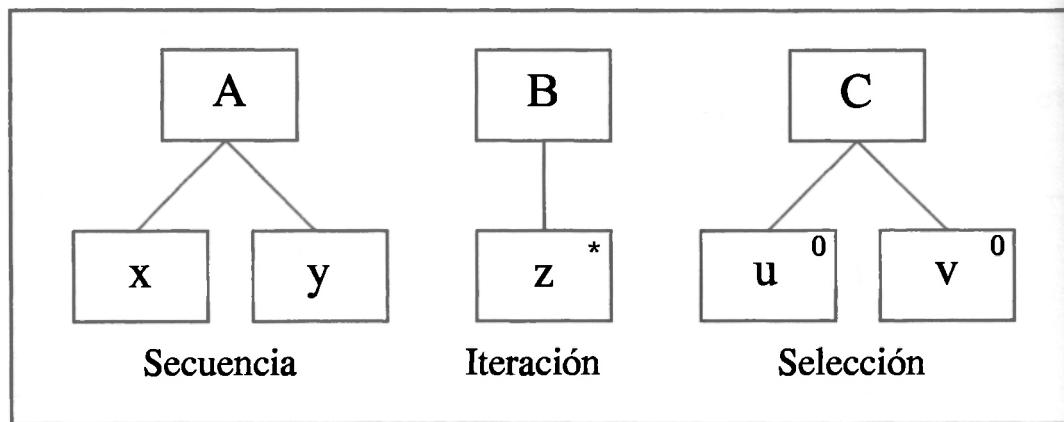


Figura 3.6 Notación de Jackson

- 1.- Especificación de las estructuras de los datos de entrada y salida
- 2.- Obtención de una estructura del programa capaz de transformar las estructuras de datos de entrada en las de salida. Este paso implica una conversión de las estructuras de datos en las correspondientes estructuras de programa que las manejan, teniendo en cuenta las equivalencias clásicas entre ambas:

TUPLA - SECUENCIA: Colección de elementos de tipos diferentes, combinados en un orden fijo.

UNIÓN - SELECCIÓN: Selección de un elemento entre varios posibles, de tipos diferentes.

FORMACIÓN - ITERACIÓN: Colección de elementos del mismo tipo.

- 3.- Expansión de la estructura del programa para lograr el diseño detallado del sistema. Para realizar este paso normalmente se utiliza pseudocódigo.

La metodología Jackson está englobada dentro de las de *Diseño Dirigido por los Datos*, que ha sido utilizado fundamentalmente para diseñar sistemas relativamente pequeños de procesamiento de datos.

La notación propuesta por Jackson se muestra en la figura 3.6 y sirve indistintamente para especificar tanto la estructura de los datos como la estructura del programa. En estos diagramas la estructura del elemento superior se detalla según queda indicado por los elementos inmediatamente inferiores. En este caso es fundamental la situación de cada elemento en el diagrama. En la figura 3.6, el elemento A es igual a la secuencia x-y leída de izquierda a derecha y nunca a la inversa. El elemento B es igual a la repetición de cero o más veces del elemento z (indicado por el símbolo "*"). El elemento C es igual a una selección entre los elementos u y v (indicado por el símbolo "O").

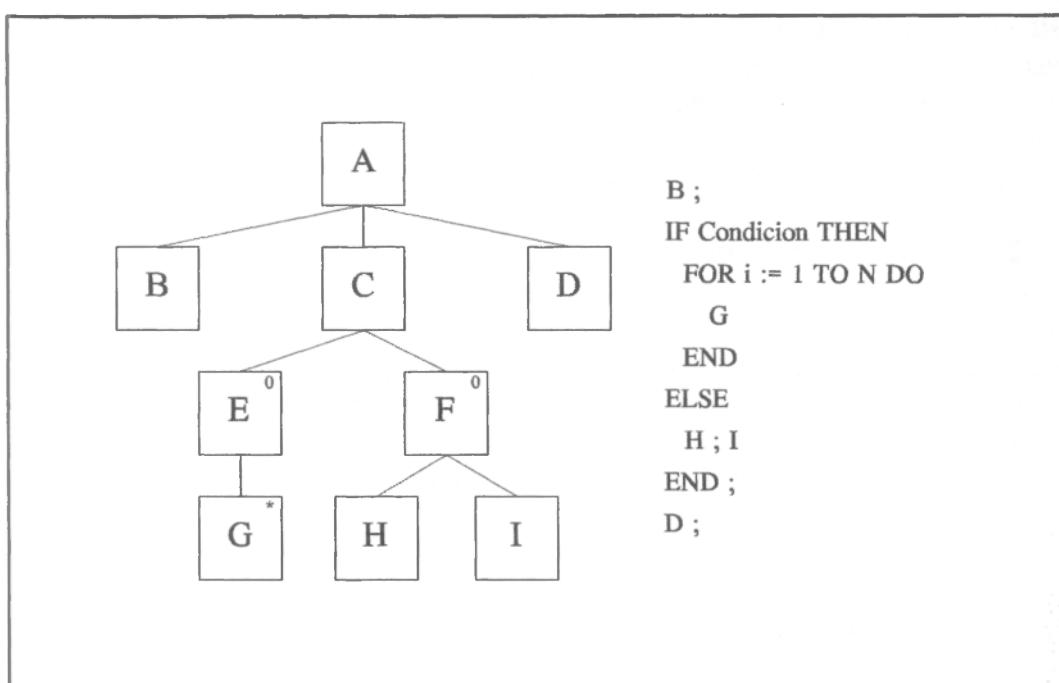


Figura 3.7 Diagrama de Jackson

En la figura 3.7 se muestra un diagrama de Jackson y la estructura de programa equivalente. Si los elementos del diagrama representan datos, la

130 Introducción a la Ingeniería de Software

estructura equivalente, expresada en notación de *diccionario de datos*, es la siguiente:

$$\begin{aligned} A &= B + C + D \\ C &= [E | F] \\ E &= \{ G \} \\ F &= H + I \end{aligned}$$

3.3.2 Notaciones estáticas

Estas notaciones sirven para describir características estáticas del sistema, tales como la organización de la información, sin tener en cuenta su evolución durante el funcionamiento del sistema. La organización de la información es uno de los aspectos en los que se hace un especial hincapié al especificar el sistema. Sin embargo, en el documento SRD de especificación sólo se realiza una propuesta de organización a grandes rasgos y de acuerdo exclusivamente con las necesidades externas del sistema. En la fase de diseño se completa la organización de la información teniendo en cuenta todos los aspectos internos: datos auxiliares, mayor eficiencia en el manejo de los datos, datos redundantes, etc. Por tanto, como resultado del diseño se tendrá una organización de la información con un nivel de detalle mucho mayor. En cualquier caso, las notaciones que se pueden emplear para describir el resultado de este trabajo son las mismas que se emplean para realizar la especificación.

3.3.2.1 Diccionario de datos

Con esta notación se detalla la estructura interna de los datos que maneja el sistema. Las características de esta notación ya han sido descritas en el apartado de descripción de datos del Tema de especificación de software. Para el diseño se partirá del diccionario de datos incluido en el documento SRD y mediante los refinamientos necesarios se ampliará y completará hasta alcanzar el nivel de detalle necesario para iniciar la codificación. En el próximo Tema se estudiarán las técnicas para realizar esta labor.

3.3.2.2 Diagramas Entidad-Relación

Esta notación permite definir el modelo de datos, las relaciones entre los datos y en general la organización de la información. Esta notación está descrita en el apartado de diagramas de modelo de datos del Tema de especificación de software. Para la fase de diseño se tomará como punto de partida el diagrama propuesto en el documento SRD, que se completará y

ampliará con las nuevas entidades y relaciones entre las mismas, que aparezcan en la fase de diseño del sistema.

3.3.3 Notaciones dinámicas

Estas notaciones permiten describir el comportamiento del sistema durante su funcionamiento. La especificación de un sistema es precisamente una descripción del comportamiento requerido desde un punto de vista externo. Al diseñar la dinámica del sistema se detallará su comportamiento externo y se añadirá la descripción de un comportamiento interno capaz de garantizar que se cumplen todos los requisitos especificados en el documento SRD. Así, las notaciones que se emplean para el diseño son las mismas utilizadas en la especificación.

3.3.3.1 Diagramas de flujo de datos

Las características de esta notación están detalladas en el Tema de especificación de software. Desde el punto de vista del diseño, los diagramas de flujo de datos serán mucho más exhaustivos que los de la especificación. Esto es debido a la necesidad de describir cómo se hacen internamente las cosas y no sólo qué cosas debe hacer el sistema.

3.3.3.2 Diagramas de transición de estados

Esta notación está descrita en el Tema de especificación de software. En el diseño del sistema pueden aparecer nuevos diagramas de estado que reflejen las transiciones entre estados internos. Sin embargo, es preferible no modificar o ampliar los diagramas recogidos en el documento SRD encargados de reflejar el funcionamiento externo del sistema.

3.3.3.3 Lenguaje de Descripción de Programas (PDL)

La notación PDL ha sido presentada en el apartado dedicado al pseudocódigo del Tema de especificación de software. Como allí se comentaba, esta notación se utiliza tanto para realizar la especificación funcional del sistema como para elaborar el diseño del mismo. La diferencia entre ambas situaciones la marca el nivel de detalle al que se desciende en la descripción funcional.

Tanto al especificar como al diseñar se utilizan las mismas estructuras básicas de la notación PDL. Sin embargo, cuando se quiere descender al nivel de detalle que se requiere en la fase de diseño, la notación PDL se

amplía con ciertas estructuras de algún lenguaje de alto nivel. Uno de los lenguajes más apropiados para estos fines es Ada. La notación denominada Ada-PDL permite declarar estructuras de datos e incluso existen compiladores para verificar el correspondiente pseudocódigo. Evidentemente, a nivel de diseño, los detalles de implementación se dejarán indicados mediante comentarios en lenguaje natural. Ada-PDL ha sido incluido como estándar en las normas IEEE. Aunque no es totalmente formal conduce a descripciones precisas, poco ambiguas, y relativamente fáciles de comprender si se conoce suficientemente el lenguaje Ada.

3.3.4 Notaciones híbridas

Estas notaciones tratan de cubrir simultáneamente aspectos estructurales, estáticos y dinámicos. En realidad algunas de las notaciones descritas anteriormente, aunque han sido clasificadas dentro de un grupo concreto, poseen también ciertas características del resto de los grupos.

Según hemos visto, la metodología de Jackson utiliza una notación estructural que describe, según los casos, la organización de la información (estática) o el comportamiento (dinámico) del sistema y está basada precisamente en aprovechar la analogía que existe entre la organización de los datos y los procedimientos o programas necesarios para su manejo. Por tanto, se podría considerar como una notación híbrida. También la metodología de diseño orientada a los datos de Warnier-Orr posee una notación que se puede considerar híbrida por las mismas razones: se parte de la organización de los datos para diseñar los procedimientos. Para un estudio en detalle de esta metodología se pueden consultar [Warnier81] y [Orr81].

Prescindiendo de las notaciones de Jackson y Warnier, el objetivo fundamental de este apartado es presentar las notaciones más modernas que se han propuesto dentro de las metodologías de diseño basado en abstracciones y de diseño orientado a objetos. Todas estas notaciones son híbridas y permiten un enfoque globalizado del diseño en el que se aglutan los aspectos estáticos (datos), dinámicos (operaciones) y de estructura del sistema.

3.3.4.1 Diagramas de abstracciones

Como ya hemos visto en este mismo Tema, el concepto de abstracción es fundamental para la estructuración de sistemas complejos. La notación que aquí se describe fue propuesta originalmente por Liskov [Liskov80] para describir la estructura de un sistema software compuesto por elementos

abstractos. En la propuesta original se contemplaban dos tipos de abstracciones: las funciones y los tipos abstractos de datos. A ellos se han añadido con símbolo propio [Cerrada00] los datos encapsulados.

Dada la afinidad existente entre los *tipos abstractos de datos* y las *clases de objetos*, al tiempo que se presenta la notación para describir abstracciones, se aprovecha para establecer una cierta analogía entre la programación basada en abstracciones y la programación orientada a objetos.

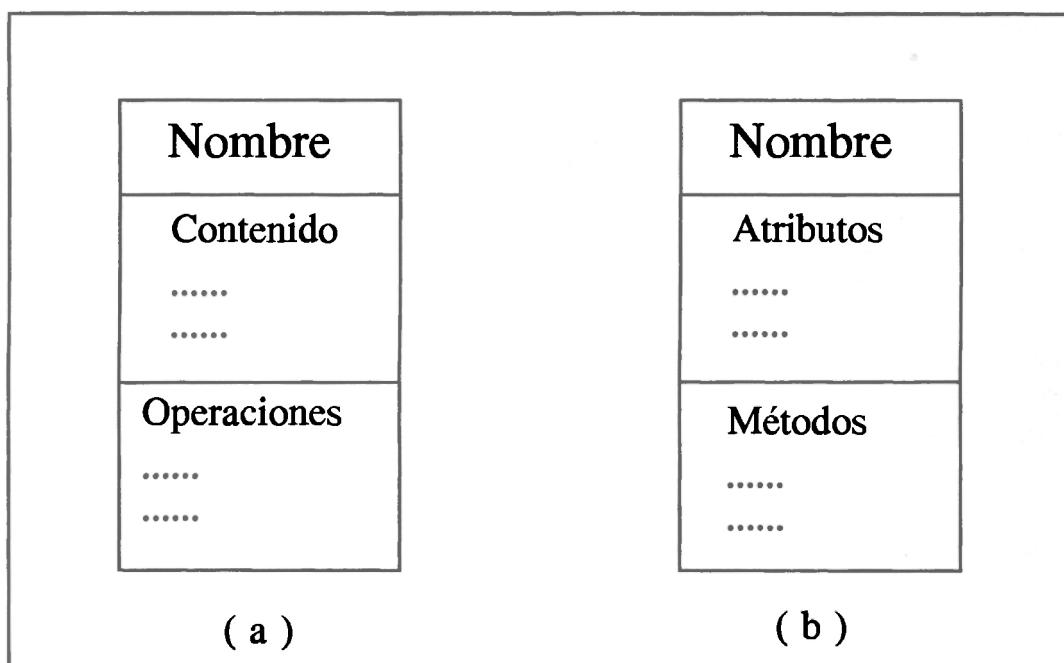


Figura 3.8 Estructura de una Abstracción (a) y de un Objeto (b)

Según se muestra en la figura 3.8 (a), en una abstracción se distinguen tres partes o elementos claramente diferenciados:

NOMBRE: Es el identificador de la abstracción

CONTENIDO: Es el elemento estático de la abstracción y en él se define la organización de los datos que constituyen la abstracción. Si se utilizase el lenguaje Modula-2, este elemento lo formarían las definiciones de constantes, tipos de datos y variables declarados en el módulo de definición.

OPERACIONES: Es el elemento dinámico de la abstracción y en él se agrupan todas las operaciones definidas para manejar el CONTENIDO de la abstracción. Si se utilizase el lenguaje Modula-2, este elemento estaría

134 Introducción a la Ingeniería de Software

formado por las definiciones de funciones y/o procedimientos declaradas en el módulo de definición.

Inicialmente la única forma de abstracción disponible en los lenguajes, y por tanto con la que únicamente se podía abordar un diseño, era la definición de subprogramas: funciones (expresiones parametrizadas) o procedimientos (acciones parametrizadas). Un subprograma constituye una operación abstracta que denominaremos *abstracción funcional*. Esta forma de abstracción no tiene la parte de CONTENIDO y sólo está constituida por una única OPERACIÓN.

Al analizar y diseñar un sistema se identifican datos de diferentes tipos con los que hay que operar. En un diseño bien organizado se puede agrupar en una misma entidad la estructura del tipo de datos con las correspondientes operaciones necesarias para su manejo. Esta forma de abstracción se denomina *tipo abstracto de datos* y tiene una parte de CONTENIDO y también sus correspondientes OPERACIONES.

Los *tipos abstractos de datos* permiten crear nuevos tipos de datos, además de los predefinidos en el lenguaje de implementación. Todos los datos concretos que se manejen en un programa deben aparecer como constantes o variables de algún tipo, sea predefinido o definido como tipo abstracto. Para operar con un dato en particular se invocará alguna de las operaciones definidas sobre su tipo, indicando a qué dato en particular queremos aplicarla. Por ejemplo, si tenemos el tipo abstracto Tarjeta y se declara:

```
VAR tarjeta4B, tarjetaVISA : Tarjeta;
```

podremos indicar con cuál de ellas queremos realizar una operación haciéndola aparecer como argumento de la misma, por ejemplo:

```
LeerTarjeta( tarjeta4B );
```

.....

```
ComprobarTarjeta( tarjetaVISA );
```

Cuando sólo se necesita una variable de un determinado tipo abstracto, su declaración se puede encapsular dentro de la misma abstracción. Así, todas las operaciones de la abstracción se referirán siempre a esa variable sin necesidad de indicarlo de manera explícita. Esta forma de abstracción se denomina *Dato encapsulado*, tiene CONTENIDO y OPERACIONES, al igual que un tipo abstracto, pero no permite declarar otras variables de su mismo tipo.

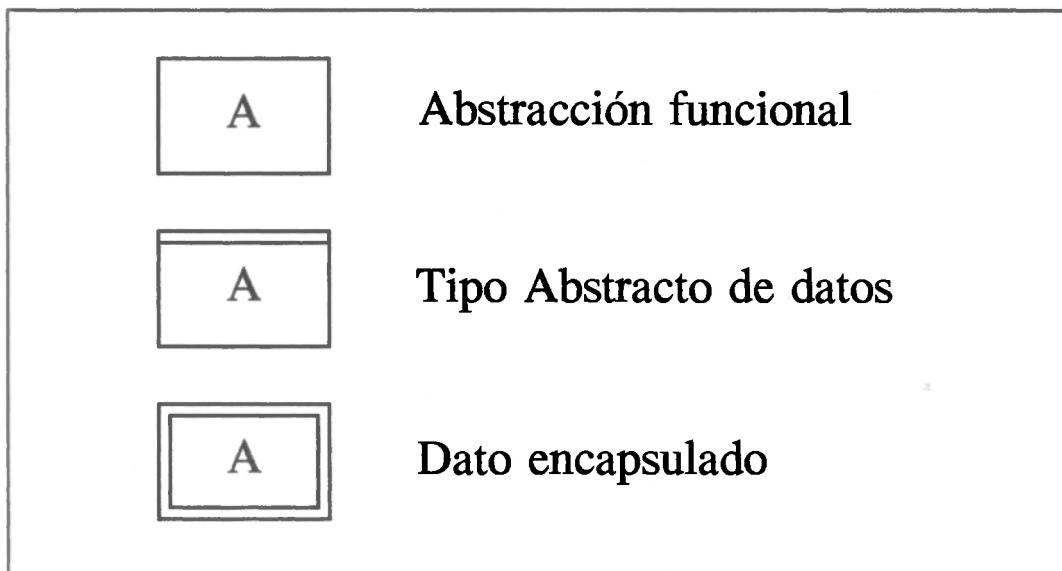


Figura 3.9 Abstracciones: notación básica

En el ejemplo anterior, si sólo se maneja una Tarjeta, se podría declarar:

```
MODULE Tarjeta;
  EXPORT LeerTarjeta, ComprobarTarjeta;
  VAR «datos de la tarjeta» ...  

  ....  

END Tarjeta;
```

e invocar las operaciones simplemente como:

```
LeerTarjeta;  

....  

ComprobarTarjeta;
```

En la figura 3.9 se muestra la notación gráfica para indicar la forma de abstracción que se está empleando.

En la figura 3.10 tenemos el diagrama de estructura de un sistema. En este caso los módulos son abstracciones en sus distintas formas posibles y la relación entre abstracciones es jerárquica e implica que la abstracción superior utiliza a la inferior. Así, la abstracción funcional A utiliza el dato encapsulado D y el tipo abstracto B. Al dato encapsulado D también lo utilizan B y la abstracción funcional C. A su vez, la abstracción funcional C también es utilizada por B.

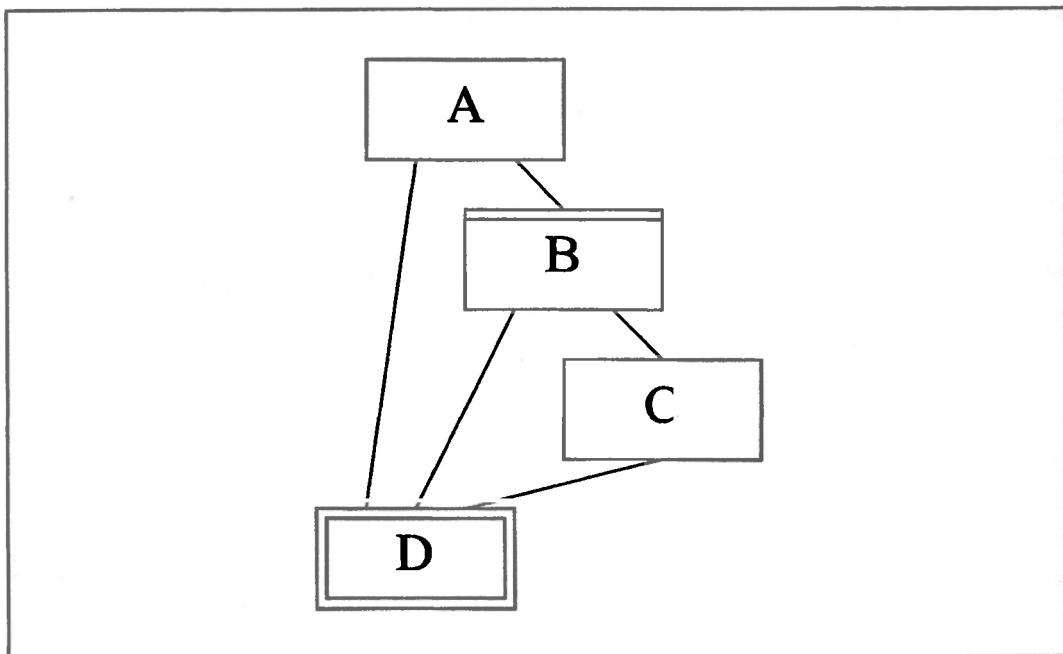


Figura 3.10 Diagrama de estructura basada en abstracciones

3.3.4.2 Diagramas de objetos

La metodología de diseño basado en abstracciones y la metodología orientada a los objetos se han desarrollado de forma paralela pero en ámbitos muy distintos. Las abstracciones se pueden considerar una propuesta de los expertos en programación, mientras que los objetos son una propuesta de los expertos en inteligencia artificial. Aunque las similitudes entre ambos conceptos son muy grandes, su desarrollo en distintos ámbitos ha propiciado la utilización de una terminología distinta para indicar lo mismo. Esto da lugar a cierta confusión.

Como se indica en la figura 3.8 (b), la estructura de un objeto es exactamente igual que la estructura de una abstracción. En cuanto a la terminología empleada se puede establecer la equivalencia que se recoge en el cuadro 3.1.

Las diferencias fundamentales entre ambos conceptos son las siguientes:

- 1.- No existe nada equivalente a los datos encapsulados ni a las abstracciones funcionales cuando se utilizan objetos en forma estricta (algunos lenguajes, como SmallTalk, suplen esta carencia permitiendo asociar atributos y métodos a las clases).

2.- Sólo entre objetos se contempla una relación de herencia.

Teniendo en cuenta la escasa diferencia entre ambos planteamientos, en el resto de este Tema utilizaremos la terminología de objetos, debido a la mayor aceptación que ha alcanzado.

ABSTRACCIONES	OBJETOS
Tipo Abstracto de Datos	----- Clase de Objeto
Abstracción Funcional	----- NO HAY EQUIVALENCIA
Dato encapsulado	----- NO HAY EQUIVALENCIA
Dato Abstracto (Variable o Constante)	----- Objeto (Ejemplar de la Clase)
Contenido	----- Atributos
Operaciones	----- Métodos
Llamada a una operación	----- Mensaje al Objeto

Cuadro 3.1 Equivalencias de terminologías

Si consideramos un objeto formado exclusivamente por sus atributos, tendremos una estructura de datos. Esta estructura, como cualquier otra, puede formar parte de un diagrama de modelo de datos E-R (Entidad-Relación) como una entidad más. Inversamente, es admisible considerar que todas las entidades en un modelo de datos son objetos (o más exactamente, clases de objetos). Como entidades, entre objetos se puede destacar cualquier tipo de relación que el diseñador considere necesaria. Además, debido a las propiedades particulares de los objetos, se pueden establecer entre ellos dos tipos de relaciones especiales:

A.- CLASIFICACIÓN, ESPECIALIZACIÓN O HERENCIA: (No contemplada entre abstracciones)

Esta relación entre objetos permite diseñar un sistema aplicando el concepto de herencia. Como ya se indicó al hablar de herencia, los

objetos "hijos" pueden adaptar las operaciones heredadas a sus necesidades concretas. Así, la herencia también se puede denominar especialización o clasificación. Para ratificar esta afirmación es completamente válido el ejemplo de las FIGURAS que hemos clasificado en ABIERTAS, CERRADAS, Las operaciones para rotar, desplazar, etc. se heredan y especializan según la estructura concreta de la figura "hija".

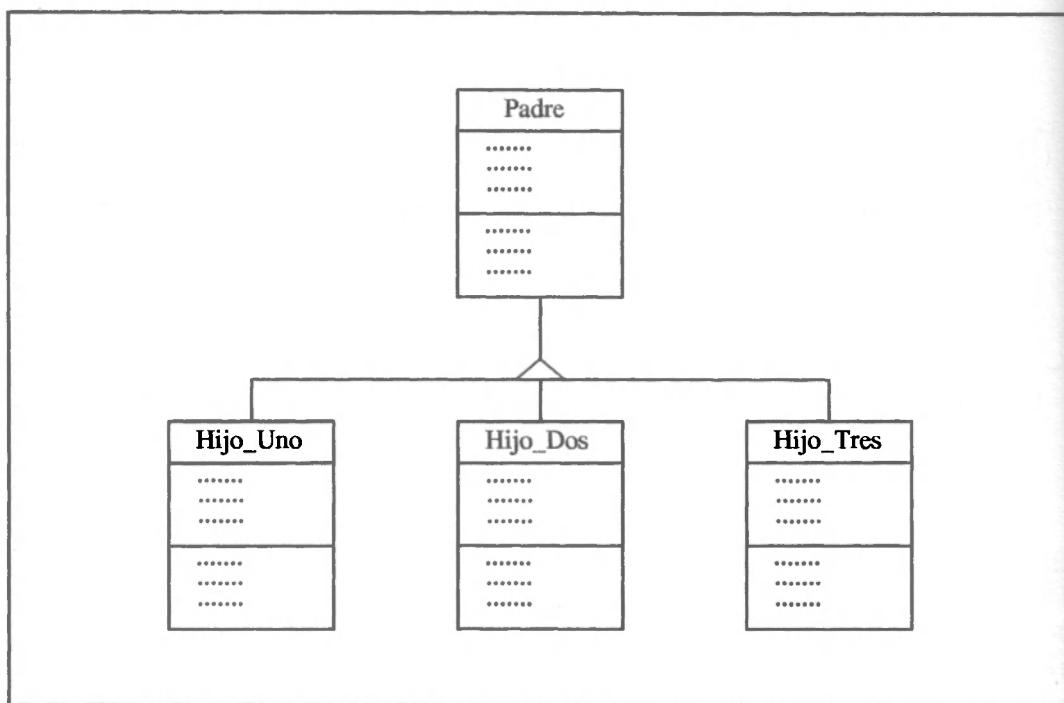


Figura 3.11 Clasificación, especialización o herencia entre objetos

En la figura 3.11 se muestra un ejemplo de herencia entre objetos. La notación empleada es la sugerida por la metodología OMT (Object Modelling Technique) descrita en [Rumbaugh91]. Con el triángulo se indica que los objetos inferiores (Hijo_Uno, Hijo_Dos e Hijo_Tres) heredan los atributos y las operaciones del objeto superior (Padre). De esta misma metodología también forma parte la notación empleada en la figura 3.8 para describir una abstracción u objeto. Hay que tener en cuenta que existe un gran número de notaciones para representar objetos y sus relaciones particulares, por lo que para trabajar con unos diagramas concretos es conveniente consultar el correspondiente manual de la herramienta de soporte o un texto específico sobre la metodología empleada.

Con la relación de herencia no es necesario indicar la cardinalidad entre las clases de objetos, que está implícita en el diagrama, ya que en él aparece expresamente cada relación de herencia entre clases.

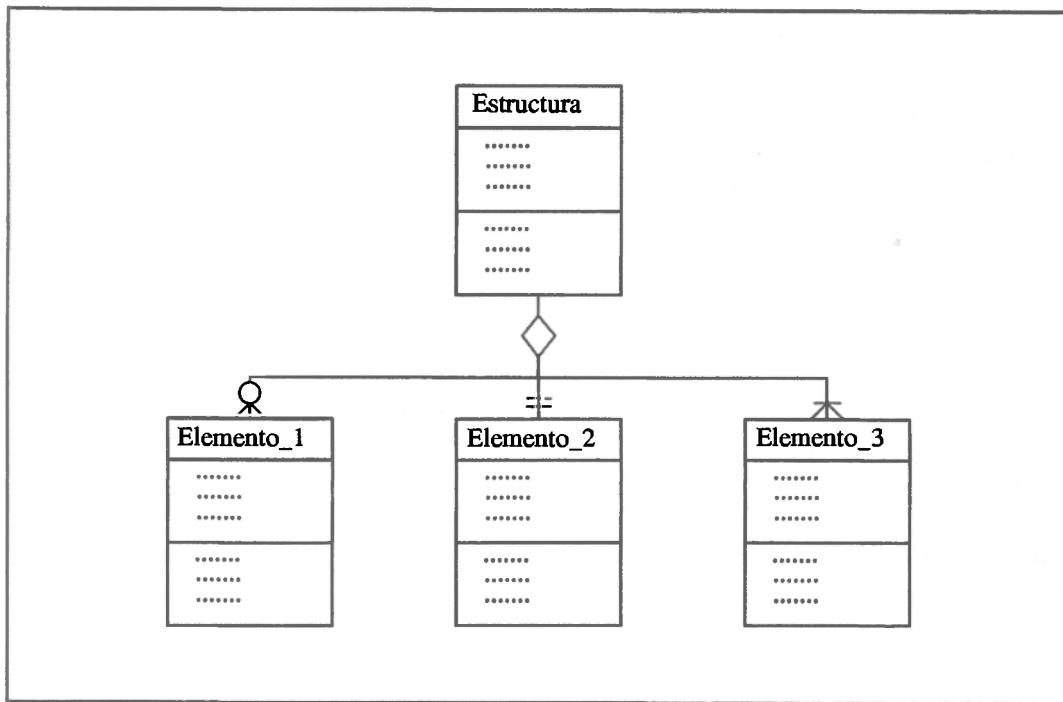


Figura 3.12 Composición de objetos

B.- COMPOSICIÓN (También válida entre abstracciones)

La relación de composición permite describir un objeto mediante los elementos que lo forman. En la figura 3.12 se muestra la relación de composición con la notación OMT. El rombo indica que el objeto **Estructura** está compuesto de **Elemento_1**, **Elemento_2** y **Elemento_3**.

En la relación de composición sólo hay que indicar la cardinalidad en un sentido. Cada objeto hijo sólo puede formar parte de un objeto padre. Sólo falta indicar qué número mínimo y máximo de objetos de cada clase hija se pueden utilizar para componer un objeto de la clase padre. Como se muestra en la figura 3.12, la cardinalidad se indica junto a cada uno de los objetos componentes. Así, para formar **Estructura** son necesarios cero o varios **Elemento_1**, uno y sólo un **Elemento_2** y al menos un **Elemento_3**.

3.4 Documentos de diseño

El resultado principal de la labor realizada en la etapa de diseño se recoge en un documento que se utilizará como elemento de partida para las sucesivas etapas del proyecto, y que denominaremos documento de diseño de software o Software Design Document (SDD). Cuando la complejidad del sistema haga que el documento SDD resulte muy voluminoso y difícil de manejar es habitual utilizar dos o más documentos para describir de forma jerarquizada la estructura global del sistema.

Existen numerosas propuestas para la organización de los documentos y de hecho probablemente cada empresa utilice uno distinto. Organismos internacionales como IEEE [IEE87], el Departamento de Defensa norteamericano [DoD88] o la Agencia Espacial Europea [ESA91] han propuesto sus propias normas de documentación, bien como simples recomendaciones o como normas de obligado cumplimiento cuando se realiza un trabajo para dichos organismos.

Las normas de la ESA establecen el empleo de un *Documento de Diseño Arquitectónico* o "Architectural Design Document" (ADD) para describir el sistema en su conjunto y otro *Documento de Diseño Detallado* o "Detailed Design Document" (DDD) para describir por separado cada uno de los componentes del sistema. Los índices de estos documentos están recogidos en los cuadros 3.2 y 3.3 respectivamente. Los contenidos de cada apartado son los siguientes:

3.4.1 Documento ADD

1. INTRODUCCIÓN

Esta sección debe dar una visión general de todo el documento ADD. Los contenidos de sus apartados (Objetivo, Ámbito, Definiciones, siglas y abreviaturas, y Referencias) serán similares a los descritos en el Tema anterior para el documento SRD pero referidos al sistema tal como se ha diseñado. Siempre que se considere interesante se puede y se debe hacer referencia a los correspondientes apartados del documento SRD.

2. PANORÁMICA DEL SISTEMA

Esta sección debe dar una visión general de los requisitos funcionales (y de otro tipo) del sistema que ha de ser diseñado, haciendo referencia al documento SRD.

- 1. INTRODUCCIÓN**
 - 1.1 Objetivo
 - 1.2 Ámbito
 - 1.3 Definiciones, siglas y abreviaturas
 - 1.4 Referencias
- 2. PANORÁMICA DEL SISTEMA**
- 3. CONTEXTO DEL SISTEMA**
 - 3.n Definición de interfaz externa
- 4. DISEÑO DEL SISTEMA**
 - 4.1 Metodología de diseño de alto nivel
 - 4.2 Descomposición del sistema
- 5. DISEÑO DE LOS COMPONENTES**
 - 5.n Identificador del componente
 - 5.n.1 Tipo
 - 5.n.2 Objetivo
 - 5.n.3 Función
 - 5.n.4 Subordinados
 - 5.n.5 Dependencias
 - 5.n.6 Interfases
 - 5.n.7 Recursos
 - 5.n.8 Referencias
 - 5.n.9 Proceso
 - 5.n.10 Datos
- 6. VIABILIDAD Y RECURSOS ESTIMADOS**
- 7. MATRIZ REQUISITOS/COMPONENTES**

Cuadro 3.2. Índice del documento ADD

3. CONTEXTO DEL SISTEMA

En esta sección se indicará si este sistema posee conexiones con otros y si debe funcionar de una forma integrada con ellos. En cada uno de sus

142 Introducción a la Ingeniería de Software

apartados se definirá la correspondiente interfase que se debe utilizar con cada uno de los otros sistemas. Si el sistema no necesita intercambiar información con ningún otro, se indicará "No existe interfaz" o "No aplicable".

4. DISEÑO DEL SISTEMA

Esta sección describe el nivel superior del diseño, en que se considera el sistema en su conjunto y se hace una primera estructuración en componentes.

4.1 Metodología de diseño de alto nivel

Se describe brevemente o se hace referencia a la metodología a seguir en el proceso de diseño de la arquitectura del sistema.

4.2 Descomposición del sistema

Se describe el primer nivel de descomposición del sistema en sus componentes principales. Se enumeran los componentes y las relaciones estructurales entre ellos.

5. DISEÑO DE LOS COMPONENTES

Las siguientes subsecciones se repiten para cada uno de los componentes mencionados en el apartado 4.2

5.n Identificador del componente

Nombre del componente. Dos componentes no podrán tener nunca el mismo nombre. Al elegir el nombre se tratará de que refleje su naturaleza. Esto simplifica la búsqueda e identificación de los componentes.

5.n.1 Tipo

Se describe la clase de componente. En algunos casos es suficiente con indicar el tipo de componente: subprograma, módulo, procedimiento, proceso, datos, etc. También es posible definir aquí nuevos tipos basados en otros más elementales. En cualquier caso, dentro del mismo documento se debe establecer una lista coherente de los tipos usados.

5.n.2 Objetivo

Se debe describir la necesidad de que exista el componente. Para ello se puede hacer referencia a un requisito concreto que se trata de cubrir. Si el requisito no forma parte del documento SRD se tendrá que detallar en este momento.

5.n.3 Función

Se describe qué hace el componente. Esto se puede detallar mediante la transformación entrada/salida que realiza o si el componente es un dato se describirá qué información guarda.

5.n.4 Subordinados

Se enumeran todos los componentes usados por éste.

5.n.5 Dependencias

Se enumeran los componentes que usan a éste. Junto a cada dependencia se podrá indicar su naturaleza: invocación de operación, datos compartidos, inicialización, creación, etc.

5.n.6 Interfases

Se describe cómo otros componentes interactúan con éste. Se tienen que establecer las distintas formas de interacción y las reglas para cada una de ellas: paso de parámetros, zona común de memoria, mensajes, etc. También se indicarán las restricciones tales como rangos, errores, etc.

5.n.7 Recursos

Se describen los elementos usados por este componente que son externos a este diseño: impresoras, particiones de disco, organización de la memoria, librerías matemáticas, servicios del sistema operativo, tamaño de "buffers", posibilidades de interbloqueos, etc.

144 Introducción a la Ingeniería de Software

5.n.8 Referencias

Se presentan todas las referencias utilizadas.

5.n.9 Proceso

Se describen los algoritmos o reglas que utiliza el componente para realizar su función como un refinamiento de la subsección 5.n.3.

5.n.10 Datos

Se describen los datos internos del componente incluyendo el método de representación, valores iniciales, formato, valores válidos, etc. Esta descripción se puede realizar mediante un diccionario de datos. Además, se indicará el significado de cada elemento.

6. VIABILIDAD Y RECURSOS ESTIMADOS

Se analiza la viabilidad de la realización del sistema y se concretan los recursos que se necesitan para llevarlo a cabo.

7. MATRIZ REQUISITOS/COMPONENTES

Finalmente, en esta sección se muestra una matriz poniendo en las filas todos los requisitos y en las columnas todos los componentes. Para cada requisito se marcará el componente o componentes encargados de que se cumpla.

3.4.2 Documento DDD

Este documento irá creciendo con el desarrollo del proyecto. En proyectos grandes puede ser conveniente organizarlo en varios volúmenes.

Como se puede ver en el cuadro 3.3, el formato de este documento es bastante similar al ADD. La diferencia entre ambos es el nivel de detalle al que se desciende. En este documento existen un mayor número de componentes y para cada uno de ellos se baja incluso hasta al nivel de codificación. Los listados fuente se recogen dentro del documento como un apéndice.

Parte 1. DESCRIPCIÓN GENERAL

1. INTRODUCCIÓN

- 1.1 Objetivo
- 1.2 Ámbito
- 1.3 Definiciones, siglas y abreviaturas
- 1.4 Referencias
- 1.5 Panorámica

2. NORMAS, CONVENIOS Y PROCEDIMIENTOS

- 2.1 Normas de diseño de bajo nivel
- 2.2 Normas y convenios de documentación
- 2.3 Convenios de nombres
(ficheros, programas, módulos, etc.)
- 2.4 Normas de programación
- 2.5 Herramientas de desarrollo de software

Parte 2. ESPECIFICACIONES DE DISEÑO DETALLADO

n. Identificador del componente

- n.1 Identificador
- n.2 Tipo
- n.3 Objetivo
- n.4 Función
- n.5 Subordinados
- n.6 Dependencias
- n.7 Interfases
- n.8 Recursos
- n.9 Referencias
- n.10 Proceso
- n.11 Datos

APÉNDICE A. LISTADOS FUENTE

APÉNDICE B. MATRIZ REQUISITOS/COMPONENTES

Cuadro 3.3. Índice del documento DDD

Dentro de la Parte 1 es interesante destacar la sección 2 dedicada a recoger todas las normas, convenios y procedimientos de trabajo que se deben

146 Introducción a la Ingeniería de Software

aplicar durante el desarrollo del sistema. La importancia de esta sección es muy grande y de ella depende que el trabajo realizado por un equipo amplio de personas tenga una estructura coherente y homogénea. La realización de esta sección debe ser la primera actividad del diseño detallado antes de iniciar el diseño propiamente dicho.

Si se utilizan las mismas normas en diversos proyectos, se puede sustituir esta sección por referencias a los documentos que contienen la información correspondiente.

Tema 4

Técnicas Generales de Diseño de Software

El diseño de software es una actividad que requiere tener cierta experiencia previa, que es difícil de adquirir sólo a través del estudio de las técnicas de diseño. Sin embargo, un conocimiento detallado de dichas técnicas es esencial para poder abordar la tarea de diseño con perspectivas de éxito.

En este Tema se hace un repaso a las técnicas más importantes de diseño que se emplean habitualmente. Todas ellas tratan de facilitar la realización del diseño. Basadas en una o varias de estas técnicas se han desarrollado metodologías completas de diseño y en algunos casos también las correspondientes herramientas CASE para diseño asistido por computador. El estudio de cualquiera de estas metodologías queda fuera del alcance de este libro. Además, ceñirse a una metodología concreta limita bastante la capacidad del diseñador a la hora de intentar enfoques alternativos de diseño. Por otro lado, existen libros enteros dedicados exclusivamente a la presentación y estudio de cada una de las metodologías y sus correspondientes herramientas.

Con carácter general, todas las técnicas tratan de conseguir como objetivos inmediatos del diseño los siguientes:

- a.- La descomposición modular del sistema. Se trata de aplicar el concepto de modularidad y obtener una división del sistema en partes o módulos.
- b.- La decisión sobre los aspectos de implementación o representación de datos que son importantes a nivel global. Se trata de que el diseñador decida sobre los algoritmos y/o las estructuras de datos fundamentales que se deben utilizar para la realización del sistema.

Este Tema comienza estableciendo ciertos requisitos y características que debe poseer la descomposición modular de un sistema. A continuación se estudian las técnicas de diseño propiamente dichas, agrupadas en

- Diseño funcional descendente
- Diseño orientado a objetos

148 Introducción a la Ingeniería de Software

- Diseño de datos

Dentro del segundo grupo se dedica un apartado específico al diseño basado en abstracciones como paso previo al diseño orientado a objetos. Los objetos incorporan respecto a las abstracciones el mecanismo de herencia y la posibilidad de establecer relaciones de especialización entre objetos.

Finalmente se incluyen los documentos de diseño de dos ejemplos totalmente desarrollados.

4.1 Descomposición Modular

Todas las técnicas de diseño están de acuerdo en la necesidad de realizar una descomposición modular del sistema como actividad fundamental del diseño y para lograrlo es necesario concretar los siguientes aspectos:

- Identificar los módulos
- Describir cada módulo
- Describir las relaciones entre módulos

La diferencia fundamental entre las distintas técnicas de diseño es precisamente lo qué se entiende por módulo en cada una de ellas y, como consecuencia, los criterios que se deben emplear para su identificación, la manera en que se describen los módulos y las posibles relaciones que se pueden establecer entre ellos.

Con un carácter lo más general posible, se puede decir que un módulo es un fragmento de un sistema software que se puede elaborar con relativa independencia de los demás. La idea básica es precisamente que la elaboración de los diferentes módulos se pueda encargar a personas distintas y que todas ellas trabajen en paralelo.

Con la definición anterior son innumerables los tipos posibles de módulos. Algunos de ellos pueden ser los siguientes:

CÓDIGO FUENTE: Contiene texto fuente escrito en el lenguaje de programación elegido. Es el tipo de módulo considerado como tal con mayor frecuencia y en el que se hace mayor hincapié en las diferentes técnicas de diseño. Su formato y organización dependen mucho de la técnica y lenguaje empleados.

■ **TABLA DE DATOS:** Se utiliza para tabular ciertos datos de inicialización, experimentales o simplemente de difícil obtención. Por ejemplo, una aplicación de control de procesos en que haya un depósito de forma irregular puede tener tabulados los volúmenes de líquido en el depósito en función del nivel. Esta tabla puede ser un módulo específico para cada forma de depósito y cuando se cambia el depósito sólo es necesario cambiar este módulo.

■ **CONFIGURACIÓN:** Un sistema se puede concebir para trabajar en entornos diversos según las necesidades de cada cliente. En estos casos, interesa agrupar en un mismo módulo toda aquella información que permite configurar el entorno concreto de trabajo. Esto es lo que sucede con los sistemas operativos que configuran el número y tipo de periféricos que el cliente adquiere. Otro ejemplo son los módulos encargados de guardar todos los mensajes de una aplicación en los distintos idiomas. Un cambio de idioma sólo requiere el cambio de este módulo.

■ **OTROS:** En general, un módulo puede servir para agrupar ciertos elementos del sistema relacionados entre sí y que se puedan tratar de forma separada del resto. Por ejemplo, para utilizar el "make" de UNIX es necesario indicar en un módulo o fichero el número y orden en que se deben combinar los otros módulos para generar el programa del sistema. También se elaboran por separado los ficheros de ayuda en línea, los manuales de usuario, etc.

■ El formato concreto de cada tipo de módulo depende de la técnica, metodología o herramienta utilizados. Por ejemplo, dependiendo del lenguaje de programación, un módulo de CÓDIGO FUENTE puede ser una única subrutina en un fichero fuente FORTRAN o un tipo abstracto de datos dentro de un MODULE de Modula-2.

Un mismo sistema se puede descomponer en módulos de muchas formas diferentes y probablemente cada diseñador argumentará razones evidentes para considerar que su propuesta es la mejor. Casi siempre el objetivo fundamental de cualquier diseño es conseguir un sistema mantenible y sólo en casos excepcionales se sacrificará este objetivo para lograr una mayor velocidad de proceso o un menor tamaño de código. Evidentemente, resulta muy difícil establecer un patrón de medida capaz de indicar de una manera precisa y cuantitativa lo buena o mala que es una determinada descomposición para facilitar su mantenimiento. Sin embargo, la experiencia acumulada permite establecer que una descomposición modular debe poseer ciertas cualidades mínimas para que se pueda considerar

150 Introducción a la Ingeniería de Software

suficientemente válida. Estas cualidades se pueden concretar en las que se estudian en los siguientes apartados.

4.1.1 Independencia funcional

En la matriz REQUISITOS/COMPONENTES del final de los documentos ADD y DDD es necesario indicar qué componente (módulo) se encargará de realizar cada uno de los requisitos (funciones) indicados en el documento SRD. Como primer paso de la descomposición se puede establecer que cada función se podrá realizar en un módulo distinto. Desde luego, si el análisis está bien hecho y las funciones son independientes, estos módulos tendrán independencia funcional entre ellos. Posteriormente y en sucesivos pasos de refinamiento del diseño se agruparán funciones afines en un mismo módulo o se subdividirán ciertos módulos en otros varios más sencillos. Para llevar a cabo esta tarea es necesario tener muy en cuenta los conceptos de abstracción, ocultación, genericidad, etc. explicados en el Tema anterior.

Para que un módulo posea independencia funcional debe realizar una función concreta o un conjunto de funciones afines, sin apenas ninguna relación con el resto de los módulos del sistema. Aunque resulta absurdo, el mayor grado de independencia se consigue cuando no existe ninguna relación entre los distintos módulos. Evidentemente al descomponer un sistema en sus partes o módulos es necesario que existan ciertas relaciones entre ellos. En todo caso se trata de reducir las relaciones entre módulos al mínimo. Una mayor independencia redundante en una mayor facilidad de mantenimiento o sustitución de un módulo por otro y aumenta la posibilidad de reutilización del módulo.

Para medir de una forma relativa la independencia funcional que hay entre varios módulos se utilizan fundamentalmente dos criterios: acoplamiento y cohesión. Estos criterios fueron sugeridos por Stevens, Constantine y Myers [Stevens74] dentro de la metodología de diseño estructurado, pero son igualmente válidos para realizar la descomposición funcional de un sistema utilizando cualquier otro tipo de técnica o metodología.

4.1.1.1 Acoplamiento

El grado de acoplamiento entre módulos es una medida de la interrelación que existe entre ellos: tipo de conexión y complejidad de la interfase. Como escala para medir de una forma cualitativa el grado de acoplamiento entre módulos se utiliza la siguiente escala:

FUERTE:	↑ Acoplamiento por Contenido Acoplamiento Común Acoplamiento Externo
MODERADO:	↓ Acoplamiento de Control Acoplamiento por Etiqueta
DÉBIL:	↓ Acoplamiento de Datos Sin Acoplamiento Directo

Para manejar esta escala hay que saber el significado de cada tipo de acoplamiento y cuándo se produce. Con ella se trata de conocer lo que se debe hacer para conseguir un acoplamiento débil y lo que se debe evitar para que no exista un acoplamiento fuerte entre módulos. No se trata de situar con precisión el resultado de la descomposición final de un sistema dentro de esta escala, dado que esto no resulta de especial interés y además resulta difícil delimitar las fronteras entre los tipos consecutivos. Por el contrario, el objetivo es que durante el diseño se tenga en cuenta la escala para buscar descomposiciones con acoplamiento débil o, a lo sumo, moderado.

En la figura 4.1 se muestran gráficamente situaciones que dan lugar a la existencia de un acoplamiento fuerte entre módulos.

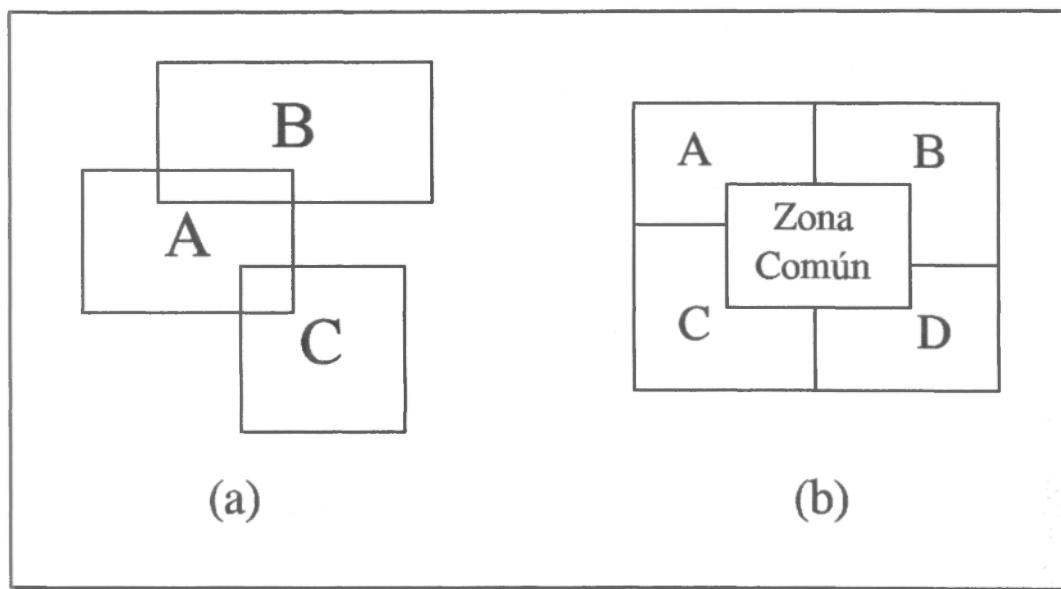


Figura 4.1 Acoplamiento fuerte: (a) por contenido; (b) común o externo

El acoplamiento *por Contenido* se produce cuando desde un módulo se pueden cambiar los datos locales e incluso el código de otro módulo. Como se muestra en la figura 4.1 (a), en realidad no existe una separación real

152 Introducción a la Ingeniería de Software

entre módulos y hay solapes entre ellos. Este tipo de acoplamiento sólo se puede lograr utilizando un lenguaje emsamblador o de muy bajo nivel y puede y debe ser evitado siempre. Resulta prácticamente imposible no sólo mantener sino también entender y depurar un sistema con acoplamiento por contenido.

Para el acoplamiento *Común* se emplea una zona común de datos a la que tienen acceso varios o todos los módulos del sistema (Ver figura 4.1 (b)). En la práctica, esto significa que cada módulo puede estructurar y manejar la zona común con total libertad sin tener en cuenta para nada al resto de módulos. El empleo de este acoplamiento exige que todos los módulos estén de acuerdo en la estructura de la zona común y cualquier cambio adoptado por uno de ellos afecta al resto que deberían ser modificados según la nueva estructura. Este tipo de acoplamiento se produce cuando se utiliza el COMMON de FORTRAN y responde a una forma de trabajo en la que la memoria disponible era escasa. La depuración y el mantenimiento de un sistema con esta descomposición resulta muy difícil y siempre que se pueda se debe evitar.

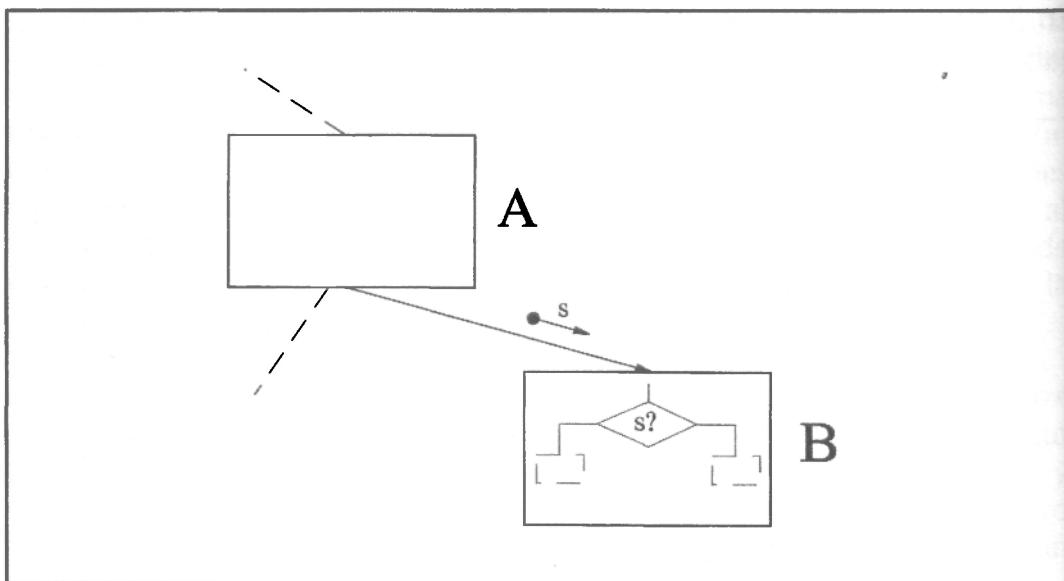


Figura 4.2 Acoplamiento moderado

En el acoplamiento *Externo* la zona común de la figura 4.1 (b) está constituida por algún dispositivo externo (disco, sensor, canal de comunicaciones, etc.) al que están ligados todos los módulos. En este caso la estructura de la zona común la impone el formato de los datos que maneja el dispositivo y cualquier modificación exige el cambio de todos los módulos. Aunque el acoplamiento externo es inevitable, siempre se deben

buscar descomposiciones en que los dispositivos externos afecten al mínimo número posible de módulos.

La figura 4.2 muestra la situación típica de un acoplamiento moderado o acoplamiento *de Control*.

En este caso, una señal (s) o dato de control que se pasa desde un módulo (A) a otro (B) es lo que determina la línea de ejecución que se debe seguir dentro de este último (B). Con este tipo de acoplamiento dentro de un mismo módulo se pueden tratar distintos casos o situaciones. Por ejemplo, es razonable que un módulo pueda generar o no una señal acústica al alcanzar un nivel de alarma y en general que el acoplamiento sirva para solicitar o anular servicios adicionales a un determinado módulo. Sin embargo, cuando se trata de utilizar este tipo de acoplamiento para aprovechar ciertas partes del módulo con fines completamente dispares se suelen crear módulos difíciles de entender, depurar, modificar y mantener. Como veremos en el próximo apartado, esto da lugar a un módulo con muy poca cohesión en que sus elementos tienen muy poca relación entre ellos.

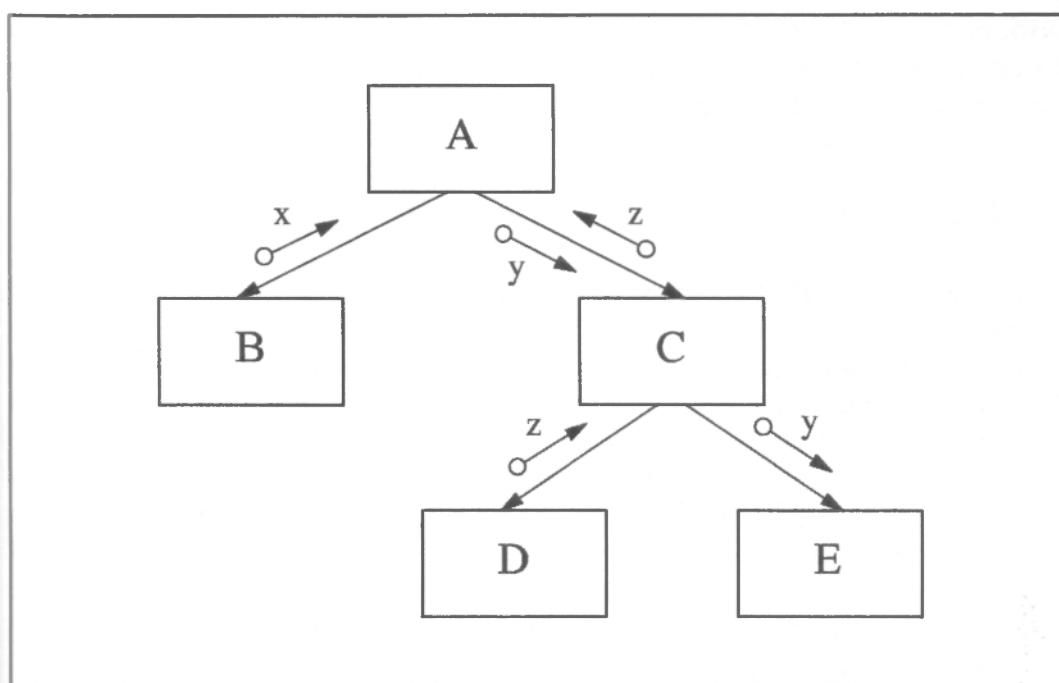


Figura 4.3 Acoplamiento débil

En la figura 4.3 tenemos una descomposición modular con acoplamiento débil.

154 Introducción a la Ingeniería de Software

Aquí, el acoplamiento se produce sólo a través del intercambio de aquellos datos que un módulo necesita de otro. Si el intercambio se realiza estrictamente con los únicos datos que se necesitan tenemos un acoplamiento *de Datos*. Este es el mejor tipo posible de acoplamiento.

Cuando en el intercambio se suministra una referencia que facilita el acceso no sólo a los datos estrictamente necesarios sino también a la estructura completa de la que forman parte (p.ej., vector, pila, árbol, grafo, etc.) tenemos un acoplamiento *por Etiqueta*.

Ambos tipos de acoplamiento débil son los más deseables en una descomposición modular. Con ellos, cualquier modificación en un módulo afecta muy poco o nada al resto. Sin embargo, cuando se utiliza un acoplamiento fuerte es necesario ser muy meticuloso y consultar siempre a todo el equipo de diseño e implementación antes de introducir ningún cambio en las zonas comunes. El equipo en su conjunto debe analizar todas las consecuencias del cambio propuesto. Además, cada cambio deberá quedar registrado junto con las razones que lo hacían necesario.

Evidentemente el acoplamiento más débil es el que no existe. Este caso es el que se produce entre los módulos E y B de la figura 4.3 entre los que no existe ningún tipo de acoplamiento directo.

4.1.1.2 Cohesión

El criterio de cohesión es complementario al de acoplamiento. Además de buscar un acoplamiento débil entre módulos es necesario lograr que el contenido de cada módulo tenga coherencia. Cuando se descompone un sistema se debe buscar un objetivo específico para cada módulo. A continuación, se tratará de agrupar en el mismo módulo todos aquellos elementos afines o que estén relacionados con el objetivo fijado para dicho módulo. Por otro lado, se debe tener en cuenta que el número de módulos no debe crecer desmesuradamente para no aumentar la complejidad del sistema. Esto último hace que ciertos elementos "sueltos" se incorporen a un determinado modulo aunque no tengan mucha afinidad con él y que como resultado se disminuya la cohesión del módulo.

Como escala para medir de forma cualitativa la cohesión de un módulo se utiliza la siguiente:

	↑	Cohesión abstraccional
ALTA:		Cohesión funcional
		Cohesión secuencial
MEDIA:		Cohesión de comunicación
		Cohesión temporal
BAJA:		Cohesión lógica
		Cohesión coincidental

La cohesión *coincidental* es la peor posible y se produce cuando cualquier relación entre los elementos del módulo es una "pura coincidencia", esto es, no guardan absolutamente ninguna relación entre ellos. El módulo se convierte en un "cajón de sastre" del que es muy difícil establecer cual es su objetivo concreto. La existencia de este tipo de módulos indica que no ha sido realizada ninguna labor de diseño y que simplemente se ha efectuado un troceado del sistema agrupando líneas de código de cien en cien.

La cohesión *lógica* se produce cuando se agrupan en un mismo módulo elementos que realizan funciones similares desde un punto de vista de usuario. Este es el caso de un módulo o biblioteca de funciones matemáticas en el que coseno, raíz cuadrada, logaritmo, etc. se agrupan debido a su carácter de herramientas matemáticas que el usuario prefiere mantener unidas. Esta misma cohesión es la que existe en los módulos de entrada/salida o cuando se diseña un módulo para el manejo de todos los mensajes de error que se producen en el sistema.

Una cohesión *temporal* es el resultado de agrupar en un mismo módulo aquellos elementos que se ejecutarán en un mismo momento. Esta es la situación que se produce en la fase de inicialización o finalización del sistema en que necesariamente se deben "arrancar" o "parar" dispositivos completamente heterogéneos: teclado, pantalla, ratón, impresora, etc.

La cohesión *BAJA* debe evitarse prácticamente siempre. De hecho, tan sólo se podría justificar una cohesión lógica o temporal y solamente en los casos dados como ejemplo u otros semejantes.

Por cohesión *de comunicación* se entiende aquella que se produce cuando todos los elementos del módulo operan con el mismo conjunto de datos de entrada o producen el mismo conjunto de datos de salida. Esto se da, por ejemplo cuando un módulo realiza operaciones diversas, pero siempre con la misma tabla de datos, y también cuando un módulo de exportación de información empaqueta distintas clases de datos pero generando siempre el mismo fichero de salida.

156 Introducción a la Ingeniería de Software

La cohesión *secuencial* es la que se produce cuando todos los elementos del módulo trabajan de forma secuencial. Esto es, la salida de un elemento del módulo es la entrada del siguiente de una manera sucesiva. Por ejemplo, un módulo para calcular el valor medio y también a continuación la desviación típica de un conjunto de valores tiene una cohesión secuencial.

Con una cohesión **MEDIA** se puede reducir el número de módulos. Sin embargo, esto no se debe realizar a costa de aumentar el grado de acoplamiento entre módulos. Por ejemplo, no se deben unir dos módulos con acoplamiento **DÉBIL** para obtener uno único con acoplamiento **MODERADO**, en el que se requiere pasar una señal para indicar con cual de los dos antiguos submódulos se quiere trabajar.

En un módulo con cohesión *funcional* cada elemento está encargado de la realización de una función concreta y específica. Con las técnicas de diseño funcional descendente este nivel de cohesión es el máximo que se puede lograr dentro de un módulo. Por tanto, con estas técnicas el objetivo será que todos los módulos tengan una cohesión funcional.

Con la técnica de diseño basado en abstracciones, un módulo con cohesión funcional sería una abstracción funcional. La cohesión *abstraccional* se logra cuando se diseña un módulo como tipo abstracto de datos con la técnica basada en abstracciones o como una clase de objetos con la técnica orientada a objetos. En ambos casos, se asocian un cierto contenido (*atributos*) u organización de datos con las correspondientes operaciones (*métodos*) encargados de su manejo.

Independientemente de la técnica empleada, una cohesión **ALTA** debe ser el objetivo que se debe perseguir en cualquier descomposición modular. Con ello, se facilitará en gran medida el mantenimiento y la posible reutilización de los módulos así diseñados.

Para conocer, y si es necesario, mejorar la cohesión de un módulo se sugiere realizar una descripción de su comportamiento [Stevens74]. A partir de la descripción, se puede establecer el grado de cohesión de acuerdo con los siguientes criterios:

- A.- Si la descripción es una frase compuesta que contiene comas o más de un verbo es muy probable que se esté incluyendo más de una función y la cohesión será **MEDIA** de tipo secuencial o de comunicación

- B.- Si la descripción contiene palabras relacionadas con el tiempo, tales como: "primero", "después", "entonces", "cuando", "a continuación", "al comenzar", etc. la cohesión será de tipo temporal o secuencial.
- C.- Si la frase de la descripción no se refiere a algo específico a continuación del verbo, es muy probable que tengamos una cohesión lógica. Por ejemplo, "escribir todos los mensajes de error" o "calcular todas las funciones trigonométricas".
- D.- Cuando se utilizan palabras tales como: "inicializar", "preparar", "configurar", etc., la cohesión será probablemente de tipo temporal.

Estos criterios serán siempre orientativos debido tanto a las deficiencias intrínsecas propias de cualquier descripción que se realiza en lenguaje natural como también a la dificultad de delimitar las fronteras entre los distintos niveles de cohesión. Así, con una descripción detallada: "medir temperatura, humedad y presión" se puede pensar en una cohesión secuencial y con una más concisa: "medir sensores" en una cohesión funcional. Está claro que la palabra "sensores" resulta más imprecisa que las tres a las que sustituye. Por otro lado, el diseñador es el encargado de matizar la necesidad de un tratamiento específico para cada tipo de sensor o si todos los sensores se deben tratar como un único tipo abstracto de datos. Esto último daría lugar a una cohesión abstraccional.

En resumen, la descomposición modular con una mayor independencia funcional se logra con un acoplamiento DÉBIL entre sus módulos y una cohesión ALTA dentro de cada uno de ellos.

4.1.2 Comprendibilidad

La dinámica del proceso de diseño e implementación de un sistema hace que los cambios sean más frecuentes de lo que sería deseable. Posteriormente, los cambios continúan durante la fase de mantenimiento hasta que se sustituye el sistema por otro nuevo. A menudo los cambios deben ser realizados por personas que no participaron ni en el diseño ni en la implementación. Para facilitar e incluso posibilitar estos cambios es necesario que cada módulo sea comprensible de forma aislada. No tiene sentido que a veces el esfuerzo de comprensión de un módulo sea mayor que el necesario para su realización.

El primer factor que facilita la comprensión de un módulo es su independencia funcional. Ya hemos visto que con una cohesión ALTA y un

158 Introducción a la Ingeniería de Software

acoplamiento DÉBIL el módulo tiene menor dependencia del resto del sistema y por tanto será más fácil entender su funcionamiento de manera aislada. Sin embargo, esto no es suficiente y es necesario además cuidar especialmente los siguientes factores:

- 1.- IDENTIFICACIÓN: Una elección adecuada del nombre del módulo y de los nombres de cada uno de sus elementos facilita mucho su comprensión. Los nombres deben reflejar de manera sencilla el objetivo de la entidad que representan. Un buen ejemplo de la dificultad de compresión que representa la utilización de una identificación inadecuada son los libros o manuales técnicos "traduztados" con un total desconocimiento del léxico utilizado habitualmente.
- 2.- DOCUMENTACIÓN: La labor de documentación de cada módulo y del sistema en general debe servir para facilitar la compresión, aclarando todos aquellos aspectos de diseño o implementación que por su naturaleza no puedan quedar reflejados de otra manera. Hay que tener en cuenta que si la documentación es muy prolífa y reiterativa se corre el riesgo de que no se lea. En cada diseño se deben establecer normas y convenios de documentación que eviten estos problemas. Estas normas formarán parte del Documento de Diseño Detallado (DDD)
- 3.- SIMPLICIDAD: Las soluciones sencillas son siempre las mejores. Un algoritmo complicado es muy difícil de entender, depurar y modificar en el caso de que sea necesario. Un esfuerzo fundamental del diseñador debe estar dedicado a simplificar al máximo las soluciones propuestas. Sólo en casos muy excepcionales (tiempo o memoria escasos), se puede sacrificar la simplicidad de un algoritmo por otro más sofisticado y difícil de comprender.

4.1.3 Adaptabilidad

Al diseñar un sistema se pretende resolver un problema concreto y se trata de obtener prácticamente un "traje a la medida". Por tanto, la descomposición modular estará muy mediatisada por el objetivo concreto del diseño. Esto dificulta bastante la adaptabilidad del diseño a otras necesidades y la posible reutilización de algunos de sus módulos. Sin embargo, es inevitable que un sistema en mayor o menor medida se adapte a los nuevos requisitos que va imponiendo el "cliente".

Independencia funcional y comprensibilidad son dos cualidades esenciales que debe tener cualquier diseño para posibilitar su adaptabilidad. No se puede abordar la tarea de adaptación de un diseño con un acoplamiento FUERTE entre sus módulos, con módulos de BAJA cohesión, mal documentados y difíciles de comprender. Con estas premisas, resulta más aconsejable y ventajoso realizar un diseño completamente nuevo que elimine todas las deficiencias del que se pretende adaptar.

Desgraciadamente las adaptaciones suelen ser múltiples y variadas a lo largo de la vida del sistema. Así, es necesario cuidar otros factores adicionales para facilitar la adaptabilidad y de ellos destacaremos los siguientes:

- 1.- PREVISIÓN: Resulta muy complicado prever que evolución futura tendrá un determinado sistema. Sólo la experiencia previa nos puede indicar que partes de sistemas semejantes han sido más susceptibles a cambios o adaptaciones. Por ejemplo: listados, presentaciones en pantalla, etc. En el diseño, estas partes se deberán agrupar en módulos con un acoplamiento lo más débil posible con el resto de los módulos del sistema. Así, las adaptaciones se podrán realizar con correcciones que sólo afectarán a los módulos previstos. Si la adaptación exige una modificación de la descomposición modular resultará tremadamente más complicada y costosa de realizar.
- 2.- ACCESIBILIDAD: Antes de abordar la nueva adaptación de un sistema es necesario conocerlo con la suficiente profundidad. Previamente a cualquier adaptación es imprescindible estudiar la estructura global del sistema y todos aquellos detalles a los que afecte de manera fundamental la adaptación. Este trabajo sólo es posible si resulta sencilla la accesibilidad a todos los documentos de especificación, diseño e implementación. Esto requiere una organización minuciosa que en la mayoría de los casos sólo es posible si se emplea una herramienta CASE. En este sentido hay que señalar las posibilidades que ofrecen los entornos de diseño y lenguajes de programación basados en el empleo de objetos. Todos estos entornos mantienen una biblioteca de objetos de fácil accesibilidad y gracias al mecanismo de herencia resulta relativamente sencilla su adaptabilidad. Esta disponibilidad permite realizar prototipos de nuevos sistemas como adaptación de otros ya existentes en un plazo muy breve de tiempo.
- 3.- CONSISTENCIA: Cuando se realizan adaptaciones sucesivas es vital mantener la consistencia entre todos los documentos de

especificación, diseño e implementación para cada nueva adaptación. La tentación de modificar exclusivamente los programas fuentes sin actualizar ningún otro documento da lugar a un caos en el que es imposible saber a qué adaptación pertenece cada nueva versión de módulo y en consecuencia nunca se podrán abordar posteriores adaptaciones. Existen herramientas para el "control de versiones y configuración" que permiten mantener automáticamente la consistencia de cada adaptación. En los entornos o lenguajes orientados a objetos no existen este tipo de herramientas y es obligatorio imponer una disciplina férrea dentro de la biblioteca de objetos para mantener la consistencia de las distintas adaptaciones. Periódicamente se debe revisar la biblioteca, eliminar objetos duplicados, reestructurar la organización de objetos, etc.

4.2 Técnicas de diseño funcional descendente

En este grupo de técnicas se incluyen todas aquellas en que la descomposición del sistema se hace desde un punto de vista funcional, es decir, se atiende fundamentalmente a la función o funciones que ha de realizar el sistema, que se van expresando poco a poco mediante funciones más sencillas, las cuales se encomiendan a módulos separados.

De esta manera los módulos corresponden precisamente a funciones concretas que se han de realizar en el sistema. Desde el punto de vista de la codificación, cada módulo corresponde esencialmente a un subprograma. Por esta razón estas técnicas de diseño conducen a estructuras modulares que pueden implementarse bien casi con cualquier lenguaje de programación, incluso aquellos como FORTRAN o COBOL que carecen de facilidades para la implementación de tipos abstractos de datos.

4.2.1 Desarrollo por refinamiento progresivo

Esta técnica corresponde a la aplicación en la fase de diseño de la metodología de programación conocida como *programación estructurada* [Dijkstra68], [Dahl72], y que condujo a la construcción de programas mediante *refinamientos sucesivos* [Wirth71].

La programación estructurada recomienda emplear en la construcción de programas sólo estructuras de control claras y sencillas, con un único punto inicial y un único punto final de ejecución. En particular son la *secuencia*, la *selección* entre alternativas, y la *iteración*.

La construcción de programas basada en el concepto de refinamiento, ya expuesto en el Tema anterior, consiste en plantear inicialmente el programa como una operación global, única, e irla descomponiendo poco a poco en función de otras operaciones más sencillas. Cada paso de descomposición consistirá en refinar o detallar la operación considerada en ese momento según una estructura de las mencionadas anteriormente, cuyos elementos serán a su vez operaciones. Por ejemplo, consideremos un programa para obtener las raíces de una ecuación de segundo grado. Los primeros refinamientos podrían ser:

```

Obtener raíces -->
  Leer coeficientes
  Resolver ecuación -->
    Calcular discriminante
    Calcular raíces -->
      SI el discriminante es negativo ENTONCES
        Calcular raíces complejas
      SI-NO
        Calcular raíces reales
      FIN-SI
    Imprimir raíces
  
```

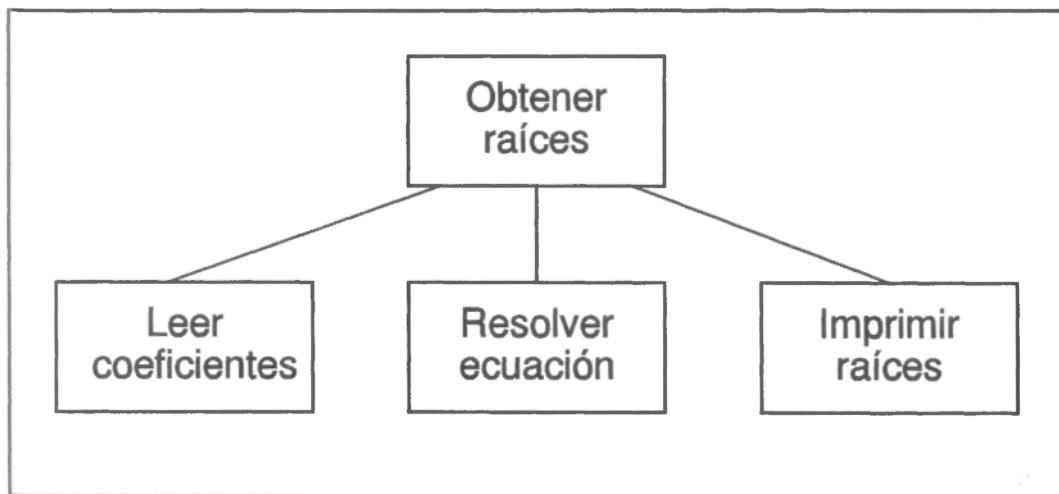


Figura 4.4 Diseño por refinamiento

La aplicación de esta técnica a la fase de diseño consiste en realizar sólo los primeros niveles de refinamiento, asignando a módulos separados las operaciones parciales que se van identificando. En el ejemplo anterior, la descomposición modular del programa resultante del primer refinamiento podría ser la que se indica en la

162 Introducción a la Ingeniería de Software

figura 4.4. A nivel de codificación, estos módulos separados se invocarían, tal como se ha dicho, como subprogramas. El paso de la estructura de refinamientos a estructura modular es inmediata, ya que los refinamientos se basan precisamente en la aplicación de estructuras de control en el programa.

4.2.2 Programación estructurada de Jackson

La técnica de Programación Estructurada de Jackson (en inglés JSP: *Jackson Structured Programming*) aparece descrita en [Jackson75]. Esta técnica sigue estrictamente las ideas de la programación estructurada en cuanto a las estructuras recomendadas (secuencia, selección, iteración) y el método de refinamientos sucesivos para construir la estructura del programa en forma descendente. Se utilizan los diagramas específicos descritos en el Tema anterior para representar tanto la estructura del programa como la de los datos con los que opera.

No hay que confundir esta metodología de diseño de programas con la denominada JSD: *Jackson System Development*, del mismo autor, descrita en [Jackson83], y que presenta algunas analogías con las técnicas de desarrollo orientadas a objetos.

La aportación principal de la programación estructurada de Jackson frente a la metodología general de programación estructurada está en las recomendaciones para ir construyendo la estructura del programa, que debe hacerse similar, en lo posible, a las estructuras de los datos de entrada y salida. Esta idea es muy apropiada para las aplicaciones denominadas antiguamente de "proceso de datos", que solían realizarse en modo "batch" y que en la actualidad han sido englobadas dentro de los denominados "sistemas de información", que operan con frecuencia en forma interactiva.

La técnica original de JSP se basa en los siguientes pasos:

- 1) Analizar el entorno del problema y describir las estructuras de datos a procesar.
- 2) Construir la estructura del programa basada en las estructuras de datos.
- 3) Definir las tareas a realizar en términos de las operaciones elementales disponibles, y situarlas en los módulos apropiados de la estructura del programa.

Como técnica de diseño los pasos significativos son los dos primeros, mientras que el tercero corresponde más bien a la fase de codificación.

Ilustraremos esta técnica mediante un ejemplo, consistente en un programa para "justificar" un texto, es decir, reagrupar las palabras en líneas e intercalar los espacios apropiados para que las líneas del texto se ajusten a los márgenes establecidos. En este ejemplo se pretende, además, respetar la forma de separar los párrafos en el texto original. Si consideramos el texto inicial:

Texto de entrada

Este párrafo y los dos siguientes son un ejemplo del texto que se pretende ajustar. Este es el primer párrafo.
Y este es el segundo párrafo, que está separado del anterior sólo por el sangrado.
Este tercer párrafo no tiene sangrado, y la separación viene dada por una línea en blanco.

se trataría de obtener como resultado algo similar a:

Texto de salida

Este párrafo y los dos siguientes son un ejemplo del texto que se pretende ajustar. Este es el primer párrafo.
Y este es el segundo párrafo, que está separado del anterior sólo por el sangrado.
Este tercer párrafo no tiene sangrado, y la separación viene dada por una línea en blanco.

En el PASO 1 del diseño identificamos las estructuras de los datos de entrada y salida. En el texto de entrada los espacios en blanco y los saltos de línea sólo son significativos en la separación entre párrafos. Podríamos describir la estructura como una iteración de elementos, bien palabras o separadores.

texto de entrada = { [separador de párrafo | palabra] }

El diagrama equivalente aparece a la izquierda de la figura 4.5.

En el caso del texto de salida, la escritura ha de hacerse al completar las líneas y justificarlas, en caso de que no sean la última del párrafo. Además se usan líneas en blanco como separación. La estructura de salida podría ser una iteración de líneas, cada una del tipo apropiado.

164 Introducción a la Ingeniería de Software

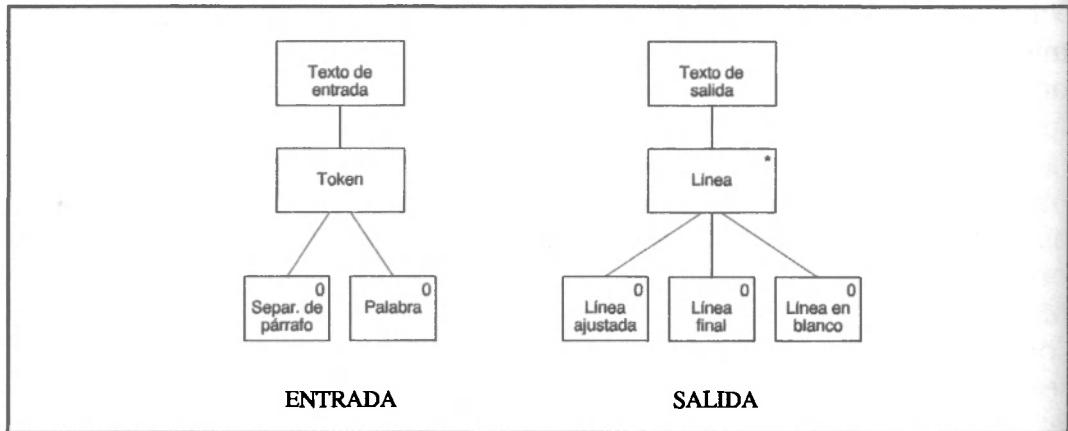


Figura 4.5 Ejemplo de estructuras de entrada y salida

texto de salida = { [línea ajustada | línea final | línea en blanco] }

El diagrama de estructura aparece a la derecha de la figura 4.5.

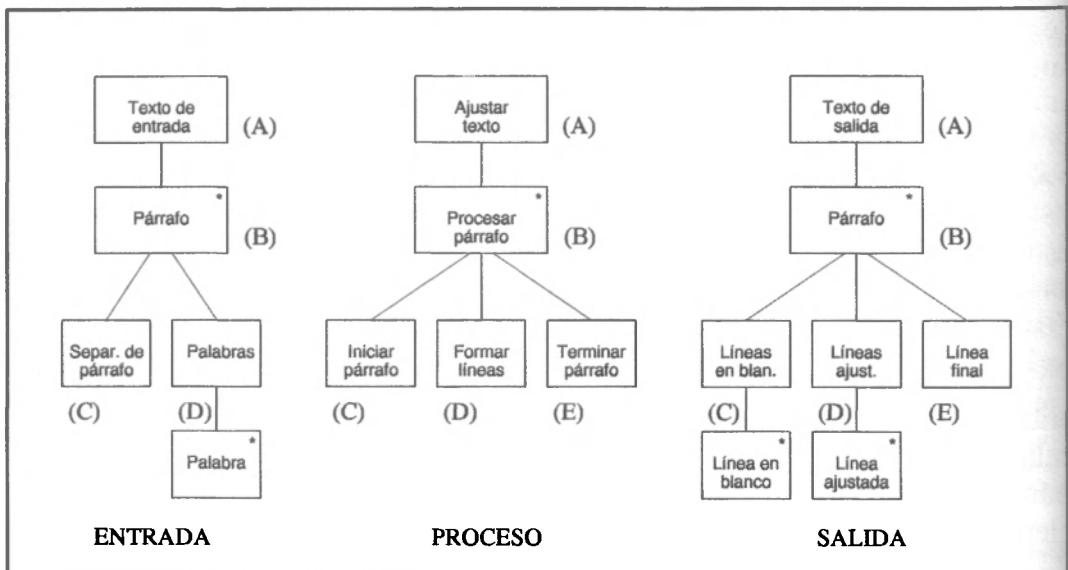


Figura 4.6 Ejemplo de diseño usando JSP

El problema ahora es encontrar, en el PASO 2, una estructura del programa que se ajuste al mismo tiempo a las dos estructuras de datos: la de entrada y la de salida.

En este caso lo que se puede hacer es buscar una unidad superior de información que englobe a las unidades de datos de la entrada y de

la salida, y usarla como unidad de tratamiento en el programa. Esta unidad superior sería el párrafo, que podría adaptarse a ambas en la forma:

```

texto = { párrafo }
párrafo de entrada = separador de párrafo + { palabra }
párrafo de salida = { línea en blanco }
+ { línea ajustada } + línea final

```

Con esta perspectiva podemos redefinir las estructuras de los datos de entrada y de salida, tal como se indica en la figura 4.6. Ahora ya se puede derivar con relativa facilidad la estructura del programa, mediante un sencillo análisis de las operaciones a realizar con cada elemento de datos. La estructura resultante podría ser la representada en el centro de la figura 4.6, donde se han señalado con las letras (A), (B), (C), (D) y (E) los elementos correspondientes de las distintas estructuras. La operación de inicio de párrafo procesa el separador de párrafo en la entrada y genera las líneas en blanco a la salida. La operación de formar líneas lee las palabras del texto de entrada y las agrupa en líneas, justificando e imprimiendo las líneas que se llenen. La operación de terminar párrafo imprime la línea final, sin ajustar.

4.2.3 Diseño estructurado

Esta técnica de diseño [Yourdon79] [Page-Jones80] es el complemento del llamado análisis estructurado. Ambas técnicas coinciden en el empleo de los diagramas de flujo de datos (DFD), descrita en el Tema anterior, como medio fundamental de representación del modelo funcional del sistema. Para el diseño se emplea la notación de diagramas de estructura, también descrita en el Tema anterior. La tarea de diseño consiste en pasar de los DFD a los diagramas de estructura.

La dificultad a superar al hacer el diseño reside en que no basta con asignar módulos a procesos o grupos de procesos relacionados entre sí, sino que hay que establecer una jerarquía o estructura de control entre los diferentes módulos, que no está implícita en el modelo funcional descrito mediante los diagramas de flujo de datos.

Por ejemplo, considerando un caso muy sencillo con sólo dos operaciones o procesos (Figura 4.7-A), cada uno de los cuales materializamos como un módulo separado, tenemos tres posibilidades de organización modular según cómo establezcamos la jerarquía de control entre ellos.

166 Introducción a la Ingeniería de Software

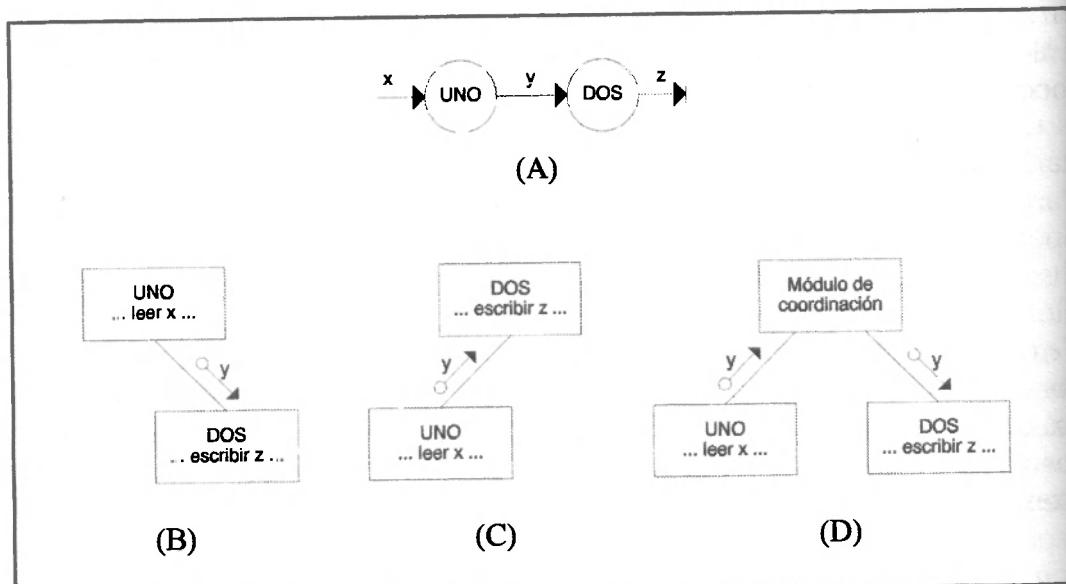


Figura 4.7 Ejemplo de estructuras alternativas

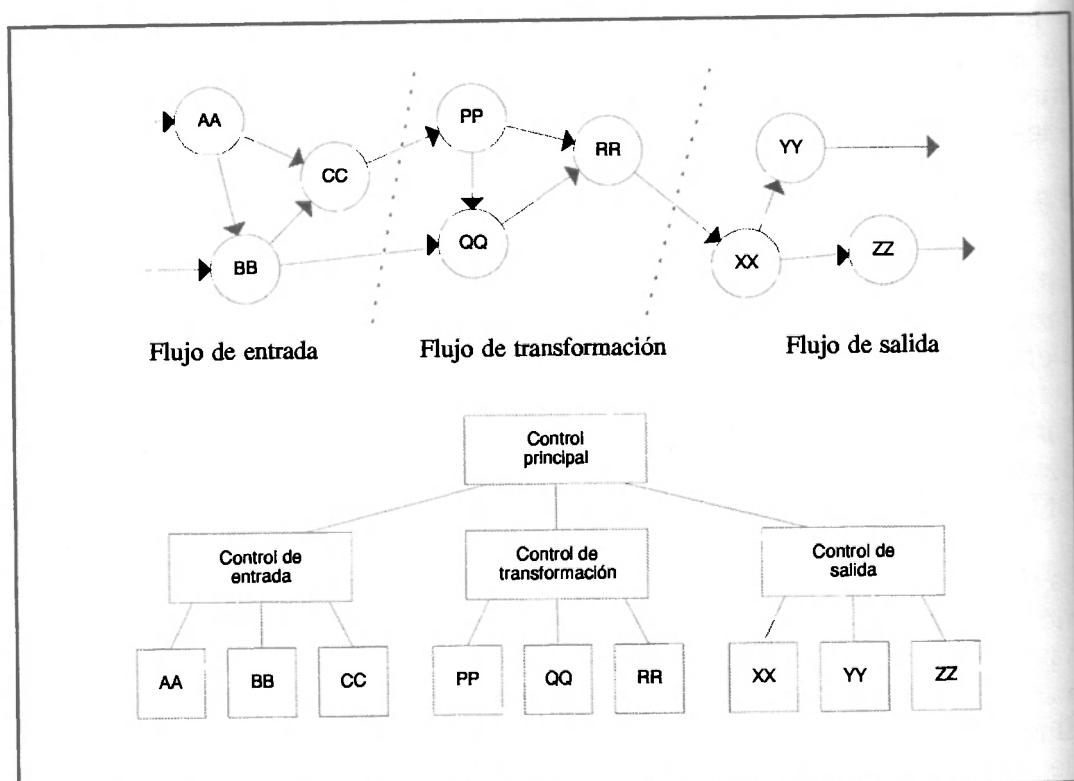


Figura 4.8 Diseño basado en el flujo de transformación

En la figura 4.7-B el módulo de la primera operación lee los datos e invoca a la segunda con los valores intermedios. En el caso de la figura 4.7-C es el módulo de la segunda operación quien tiene el control sobre la primera. En el último caso, de la figura 4.7-D, se ha introducido un módulo adicional, llamado módulo de *coordinación*, para llevar el control de los otros dos. Para establecer una jerarquía de control razonable entre las diferentes operaciones descritas en los diagramas de flujo de datos, la técnica de diseño estructurado recomienda hacer ciertos análisis del flujo de datos global. En concreto se recomienda realizar los análisis denominados de *flujo de transformación* y de *flujo de transacción*. Estos análisis pueden realizarse con mayor facilidad si se modifica parcialmente la estructura jerárquica de los diagramas de flujo de datos, construyendo un único diagrama con todos los procesos contenidos en los primeros niveles de descomposición, y prescindiendo de los almacenes de información.

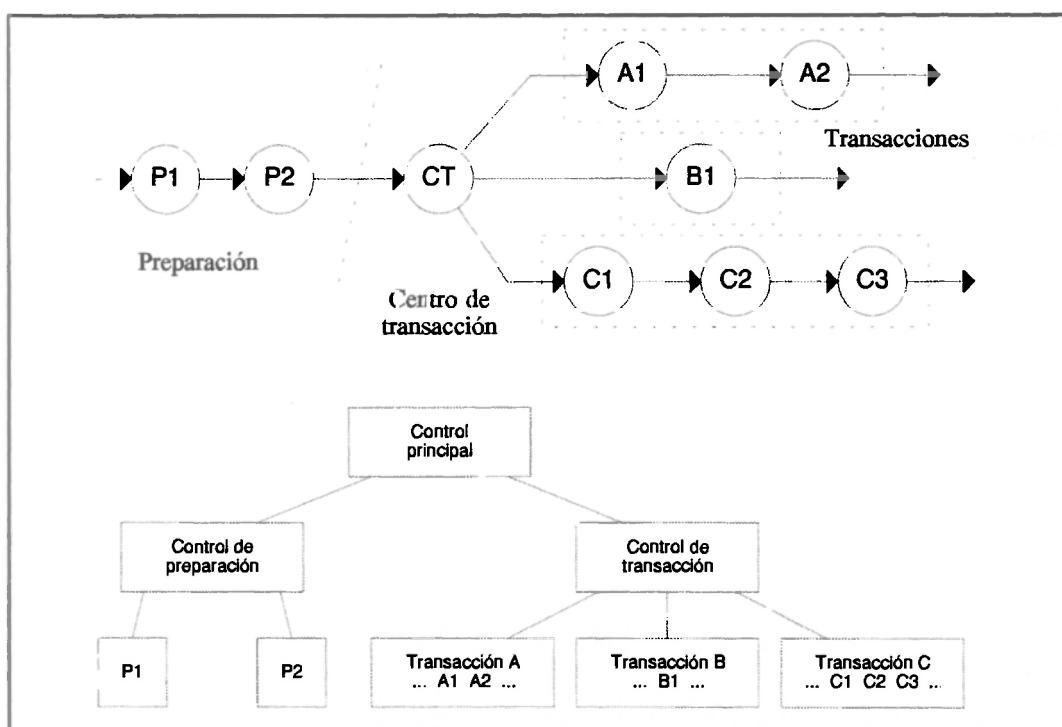


Figura 4.9 Diseño basado en el flujo de transacción

El análisis de flujo de transformación consiste en identificar un flujo global de información desde los elementos de entrada al sistema hasta los de salida, tal como se indica en la parte superior de la figura 4.8. Los procesos se deslindan en tres regiones, denominadas de flujo de entrada, de transformación, y de salida. Para obtener la estructura modular del programa se asignan módulos para las operaciones del diagrama (o grupos

168 Introducción a la Ingeniería de Software

de ellas) y se añaden módulos de coordinación que realizan el control de acuerdo con la distribución del flujo de transformación, tal como se indica en la parte inferior de la misma figura. Si se considera conveniente, a cada una de las partes se le pueden aplicar a su vez las técnicas de análisis de flujo, para refinar aún más la estructura del sistema.

El análisis del flujo de transacción es aplicable cuando el flujo de datos se puede descomponer en varias líneas separadas, cada una de las cuales corresponde a una función global o transacción distinta, de manera que sólo una de estas líneas, en general, se activa para cada entrada de datos de tipo diferente. En la parte superior de la figura 4.9 se representa un diagrama de flujo con estas características. El análisis consiste en identificar el llamado centro de transacción, a partir del cual se ramifican las líneas de flujo, y las regiones correspondientes a cada una de esas líneas o transacciones. La estructura modular del sistema puede derivarse de la distribución del flujo identificada en este análisis, tal como se indica en la parte inferior de la misma figura. Si se considera apropiado, a cada una de las transacciones se le puede aplicar por separado el análisis de flujo de transformación y/o de transacción, para refinar aún más la estructura del sistema.

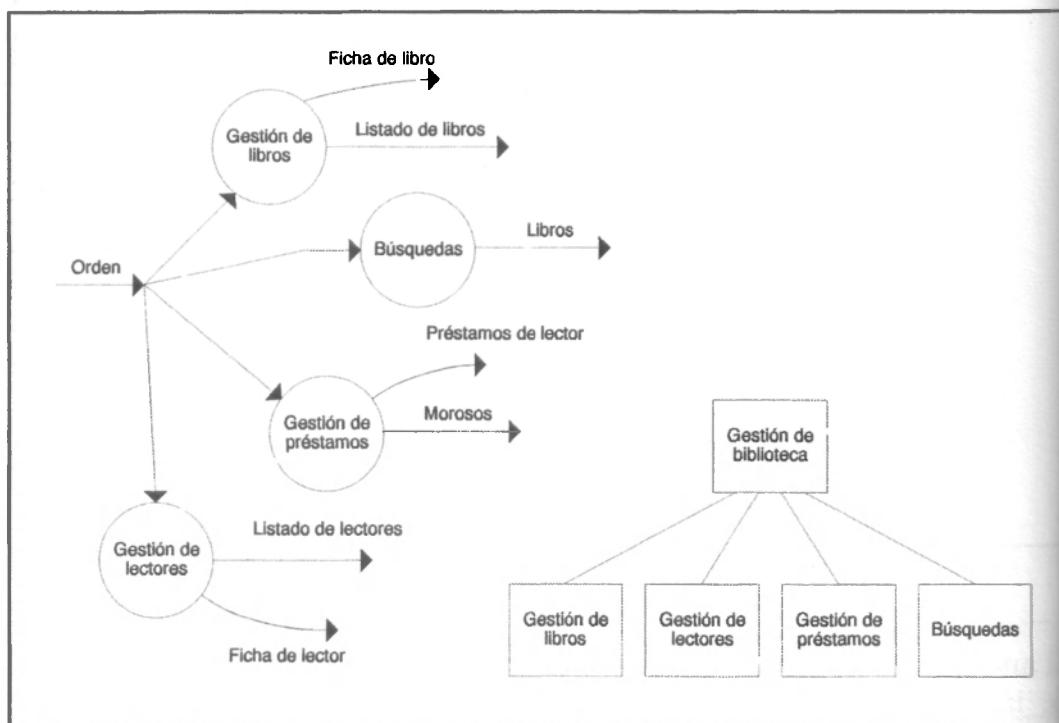


Figura 4.10 Gestión de biblioteca. Diseño inicial

4.2.4 Ejemplo: Sistema de gestión de biblioteca

Se describe aquí como ejemplo un posible diseño del caso práctico especificado en el Tema 2 con el nombre de "Sistema de Gestión de Biblioteca", empleando la técnica de diseño estructurado, por ser la más ajustada a la forma de especificación mediante diagramas de flujo de datos usada en el SRD de este ejemplo.

El primer paso será analizar los tipos de flujo de datos que circulan por el sistema. Podemos reformular el diagrama de primer nivel DFD.0 prescindiendo, para simplificar, de los almacenes de información, en la forma que se indica en la figura 4.10. En este diagrama se aprecia claramente una estructura de transacciones, en la que el centro de transacción no se corresponde con un proceso concreto, sino simplemente con la ramificación del flujo de órdenes de entrada. De aquí podemos derivar un primer nivel de estructura del sistema tal como el que aparece en dicha figura.

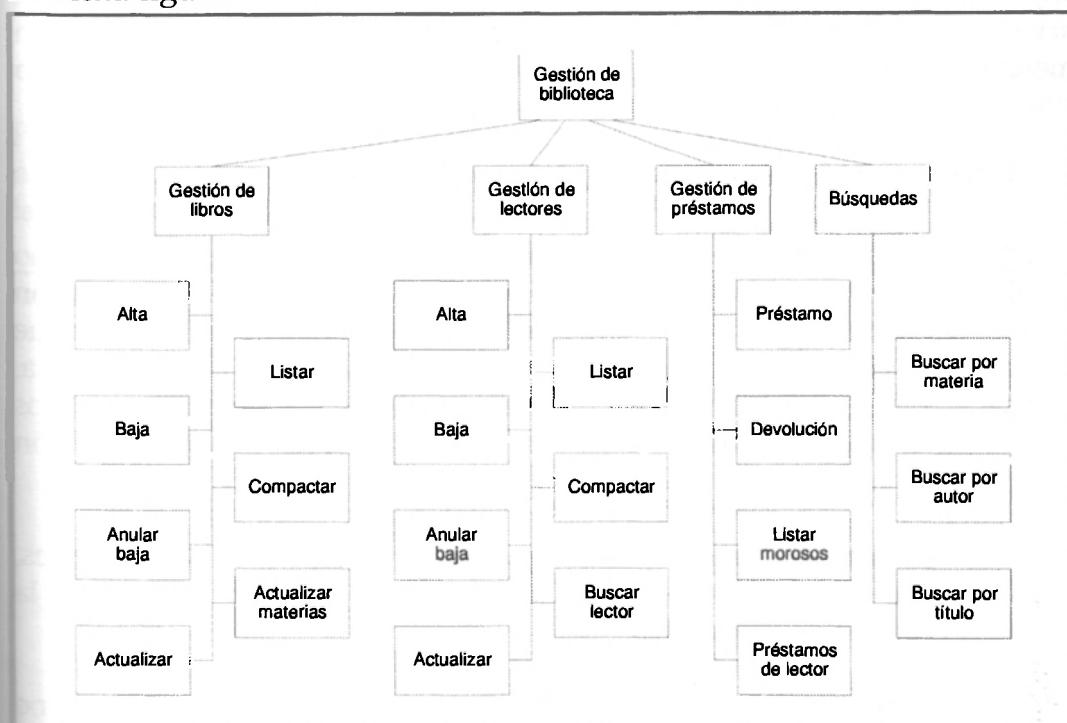


Figura 4.11 Diseño final del sistema de gestión de biblioteca

El proceso puede repetirse con cada uno de los subsistemas (gestión de libros, de lectores, etc...), para refinar aún más la estructura del sistema. El análisis de cada subsistema nos vuelve a mostrar un esquema de flujo de

170 Introducción a la Ingeniería de Software

transacción, ya que cada uno de ellos es una agrupación de funciones más sencillas, que se han asociado en un mismo proceso o diagrama por la afinidad de los datos sobre los que operan. Es inmediato, por tanto, llegar a la estructura final de la figura 4.11, en que aparece un segundo nivel de descomposición del sistema.

4.3 Técnicas de diseño basado en abstracciones

Estas técnicas de diseño surgen cuando se identifican con precisión los conceptos de abstracción de datos y de ocultación descritos en el Tema anterior. La idea general es que los módulos se correspondan o bien con funciones o bien con tipos abstractos de datos. Las estructuras modulares resultantes pueden implementarse bien con lenguajes de programación que tengan facilidades para implementar abstracciones de datos, tales como Modula-2 o Ada, y en menor medida con lenguajes como Pascal o C. Por supuesto, pueden también implementarse con lenguajes de programación orientados a objetos, que poseen aún más facilidades. En todo caso resultan menos apropiados los lenguajes como FORTRAN o COBOL que sólo cuentan con módulos de tipo subprograma.

4.3.1 Descomposición modular basada en abstracciones

Esta técnica aparece descrita en [Parnas72] y [Liskov80]. Considerada como técnica de programación, consiste en ampliar el lenguaje existente con nuevas operaciones y tipos de datos, definidos por el usuario, de forma que se simplifique la escritura de los niveles superiores del programa. Considerada como técnica de diseño, consiste en dedicar módulos separados a la realización de cada tipo abstracto de datos y cada función importante.

Para la representación de la estructura modular basada en abstracciones, usaremos la notación introducida en el Tema anterior, basada en la propuesta inicial de [Liskov80].

Esta técnica de diseño puede aplicarse tanto en forma descendente como ascendente. En forma descendente puede considerarse como una ampliación de la técnica de refinamiento progresivo, en que al realizar un refinamiento se plantea como alternativa, además de su descomposición, el que la operación a refinar se defina separadamente como abstracción funcional, o bien como operación sobre un tipo abstracto de datos. En forma ascendente se trata de ir ampliando las primitivas existentes en el lenguaje de programación y las librerías asociadas con nuevas operaciones y tipos de

mayor nivel, más adecuados para el campo de la aplicación que se está diseñando. En muchos casos puede aplicarse simultáneamente de ambas maneras.

Volviendo al ejemplo de la ecuación de 2º grado, usado en la sección 4.2.1, podemos identificar los tipos abstractos correspondientes a un número complejo, y a una ecuación de 2º grado. Sabiendo que la fórmula que da las raíces es:

$$Ax^2 + Bx + C = 0 \quad \Rightarrow \quad x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

podemos definir sobre dichos tipos abstractos las siguientes operaciones necesarias para la aplicación:

Ecuación de 2º grado:
Leer ecuación
Escribir ecuación
Obtener raíces

Número complejo:
Escribir
Sumar, Restar, etc...
Raíz cuadrada

Para obtener las raíces se usarán las operaciones definidas sobre los números complejos. La estructura modular del programa podría ser la representada en la figura 4.12.

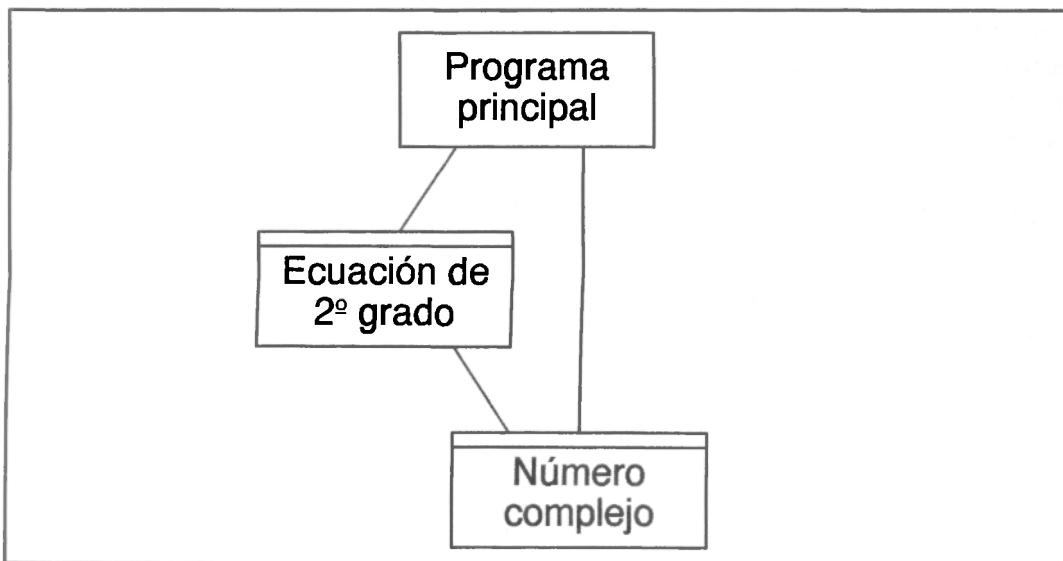


Figura 4.12 Ejemplo de diseño basado en abstracciones

4.3.2 Método de Abbott

En la descripción de la descomposición modular basada en abstracciones del apartado anterior no se ofrece ninguna guía, salvo el sentido común, para ir reconociendo aquellos elementos del modelo del sistema que son buenos candidatos para ser considerados como abstracciones, sean funciones o tipos abstractos de datos. En [Abbott83] se sugiere una forma metódica de conseguirlo a partir de las descripciones o especificaciones del sistema hechas en lenguaje natural. De hecho esta técnica puede aplicarse también, y con mayor precisión, a las descripciones más formales, que emplean notaciones precisas y no sólo lenguaje natural.

La idea es identificar en el texto de la descripción aquellas palabras o términos que puedan corresponder a elementos significativos del diseño: tipos de datos, atributos y operaciones, fundamentalmente. Los tipos de datos aparecerán como sustantivos genéricos, los atributos como sustantivos, en general, y las operaciones como verbos o bien como nombres de acciones. Algunos adjetivos pueden también sugerir valores de los atributos.

Se puede comenzar subrayando en el texto todos los términos de alguno de los tipos indicados, que sean significativos para la aplicación, y establecer dos listas: una de nombres y otra de verbos u operaciones. El siguiente paso es reorganizar dichas listas extrayendo los posibles tipos de datos, y asociándoles sus atributos y operaciones. Al reorganizar las listas hay que depurarlas eliminando los términos irrelevantes o sinónimos, y completarla con los elementos implícitos en la descripción, pero que no se mencionaban expresamente.

Podemos ilustrar esta técnica aplicándola al ejemplo del programa de ajuste de un texto, ya utilizado en este Tema. Comenzaremos con una especificación informal del mismo, en la que subrayamos los elementos significativos:

AJUSTE DEL MARGEN DERECHO DE UN TEXTO - Especificación informal

Se trata de desarrollar un programa que permita obtener textos impresos con el margen derecho bien ajustado, al mismo tiempo que se recomponen las líneas para que contengan tantas palabras como quepan en ellas.

El programa operará a partir de un texto de entrada sin ajustar y sin limitaciones de margen, el cual deberá ser impreso a la salida en forma ajustada.

El ajuste se hará de manera que se respete la separación en párrafos del texto de entrada. Dicha separación vendrá marcada por líneas en blanco y/o sangrado.

Esto quiere decir que la primera línea de un párrafo debe seguir a una línea en blanco o bien empezar por algún espacio en blanco. Las líneas segundas y siguientes del párrafo empezarán en la primera columna, y no irán precedidas de ninguna línea en blanco.

La forma de separar párrafos en el texto inicial debe respetarse en la salida. Esto quiere decir que las líneas en blanco entre párrafos deben reproducirse en la salida, así como el sangrado si lo hay.

Todas las líneas de salida, excepto las que sean final de párrafo, deberán ser ajustadas al margen derecho intercalando espacios en blanco entre las palabras. La posición del margen derecho será un parámetro global del programa.

En este marcado ya se ha omitido destacar ciertos elementos de información, tales como "... primera línea de un párrafo ... segunda y siguientes ... primera columna ...", que se refieren a las líneas del texto de entrada, cuya organización, como ya se dijo en un apartado anterior, no es significativa para el resultado a generar, salvo en lo referente a la separación entre párrafos.

A partir de este marcado elaboramos una doble lista, con los elementos correspondientes a *datos* y a *operaciones*. En estas listas se han marcado con el signo "=" los términos considerados sinónimos. También se han distinguido, comentándolos, los términos homónimos pero con diferente significado.

DATOS	OPERACIONES
textos impresos	
= salida	ajustado
margen derecho	= ajustar
= margen	= ajuste
palabras	= intercalando (espacios)
texto de entrada	recomponen
= texto inicial	quepan
separación en párrafos	ser impreso
= forma de separar párrafos	
líneas en blanco	
sangrado	
= espacio en blanco	
(comienzo de línea)	
párrafo	
líneas de salida	
final de párrafo	
espacios en blanco	
(entre palabras de salida)	

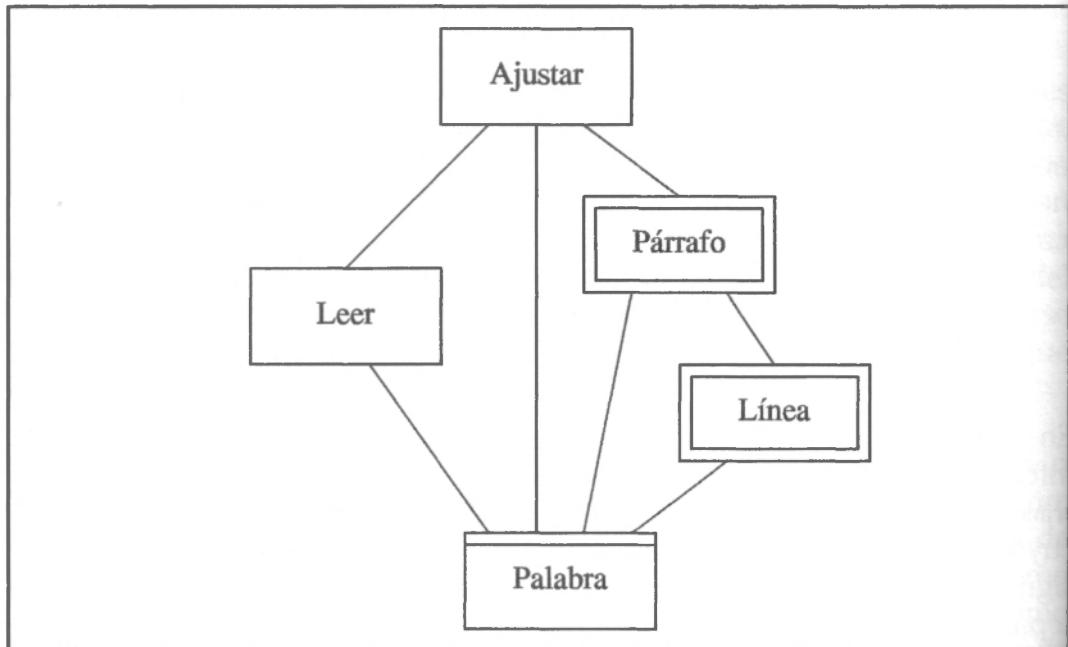


Figura 4.13 Programa de ajuste: diseño mediante abstracciones

A partir de estas listas iniciales hay que elaborar la descripción de abstracciones, indicando para cada tipo abstracto de datos cuáles son sus atributos y sus operaciones. En esa lista se pueden añadir los elementos necesarios no incluidos en las listas iniciales. En este caso se podría tener:

DATO: Palabra

Atributos:

caracteres

Operaciones:

imprimir

DATO: Separador de párrafo

Atributos:

líneas en blanco
sangrado

Operaciones:

DATO: Párrafo (de salida)

Atributos:

separador de párrafo
líneas de salida

Operaciones:

iniciar párrafo
poner palabra (= recomponer)
terminar párrafo

DATO: Línea (de salida)

Atributos:

sangrado
palabras

Operaciones:

iniciar línea
¿cabe palabra?
poner palabra
imprimir sin ajustar
imprimir ajustada

Para obtener el diseño podemos asignar un módulo a cada abstracción de datos o grupo de abstracciones relacionadas entre sí. El módulo puede corresponder a un dato encapsulado si sólo se maneja un dato de ese tipo en todo el programa. Además hay que tratar de expresar cada operación en función de las otras, para ver las relaciones de uso entre módulos y detectar posibles omisiones. También hay que añadir abstracciones funcionales para el módulo principal y otras operaciones omitidas, en su caso. En este ejemplo se detecta la omisión de una función para leer las palabras del texto de entrada y reconocer la separación entre párrafos. El diseño resultante podría ser el indicado en la figura 4.13. El tipo "separador de párrafo" no se ha materializado en un módulo separado, ya que no hay operaciones definidas sobre él y su contenido de información se reduce a un par de números enteros (líneas de separación y sangrado).

4.3.3 Ejemplo: Videojuego de las minas

Este ejemplo ha sido descrito en el Tema 2, donde se encuentra incluso el documento de especificación de requisitos, completo. Para realizar el diseño basado en abstracciones debemos empezar por recopilar toda la información disponible. Puede considerarse que la información es bastante precisa, e incluye ya la identificación de varias funciones y tipos de datos importantes.

Un primer repaso del documento de especificación de requisitos, al que podemos aplicar la técnica de Abbot para identificar elementos significativos, nos conduciría a una primera lista de tipos de datos y operaciones asociadas, por ejemplo:

DATO: Tablero	DATO: Casilla
Atributos:	Atributos:
tamaño	hay mina
datos de casillas	nº de minas vecinas
nº de minas	está destapada
nº de casillas destapadas	está marcada
Operaciones:	Operaciones:
iniciar tablero	destapar
mover cursor	poner/quitar marca
destapar	
poner/quitar marca	
	OPERACIÓN: Minas (juego)

176 Introducción a la Ingeniería de Software

DATO: Tabla de resultados

Atributos: (por nivel y entrada)

texto

tiempo

Operaciones:

anotar resultado

presentar en pantalla

OPERACIÓN: Ayuda

En esta lista se han incluido algunos tipos abstractos de datos, así como un par de abstracciones funcionales, correspondientes al programa principal (el juego de las minas) y a la invocación de la ayuda en pantalla. El siguiente nivel de refinamiento exige idear de manera más precisa la forma de realizar ciertas operaciones. Por ejemplo, se tiene el problema de cómo actualizar el cronómetro en pantalla mientras se está desarrollando el juego, y en particular mientras se está esperando que el jugador pulse alguna tecla. Otros problemas son cómo conservar los mejores resultados de una sesión para otra, y cómo actualizar un elemento en pantalla sin alterar la presentación de otros.

Los últimos problemas son relativamente fáciles de resolver. Se puede almacenar la tabla de resultados en un fichero en disco, entre sesión y sesión. La pantalla puede actualizarse por zonas, disponiendo de una función para situar el cursor de texto antes de escribir. Estas soluciones nos llevan a diseñar nuevas abstracciones de nivel más bajo que las anteriores, y que permitan realizar las operaciones indicadas, pero evitando tener que invocar directamente funciones del sistema operativo. El problema de actuar por tiempo y no sólo por la acción del jugador es algo más complejo. Se podría pensar en usar el mecanismo de interrupciones, pero resulta más sencillo realizar en el programa principal un esquema de monitor cíclico, disponiendo de una operación de lectura del teclado que no implique espera: si no se pulsa tecla se devuelve tecla nula al leer. El esquema sería:

REPETIR

 Leer tecla

 SI tecla no nula ENTONCES

 responder a la tecla

 FIN-SI

 Leer tiempo

 SI ha pasado el plazo ENTONCES

 actuar por tiempo

 FIN-SI

 HASTA fin del juego

Esto nos lleva a diseñar un módulo de cronómetro del juego, de alto nivel, que se apoya en otro de bajo nivel para leer el reloj del computador, así como otro módulo de bajo nivel para leer del teclado sin espera. El cronómetro sería un dato encapsulado, definido de la siguiente manera:

DATO: Cronómetro

Atributos:

tiempo transcurrido, en segundos

Operaciones:

iniciar

actualizar, si ha pasado un segundo

parar

El resultado final del diseño se recoge en el documento que aparece como ejemplo de documento completo de diseño arquitectónico, más adelante en este mismo Tema.

4.4 Técnicas de diseño orientadas a objetos

El diseño orientado a objetos es esencialmente igual al diseño basado en abstracciones, que de hecho se describe en muchos casos como "basado en objetos". Los objetos sólo añaden algunas características adicionales, tales como la herencia y el polimorfismo.

Se han descrito una gran variedad de metodologías particulares en este campo [Coad90], [Booch94], [Rumbaugh91], etc. Todas ellas tienen muchos elementos en común, y se distinguen sobre todo en la apariencia de los diagramas empleados para describir el diseño. Esta situación es análoga a la que ocurrió en su momento con las metodologías de análisis y diseño estructurado, basadas en los diagramas de flujo de datos.

La idea global de las técnicas de diseño orientadas a objetos es que en la descomposición modular del sistema cada módulo contenga la descripción de una clase de objetos o de varias clases relacionadas entre sí. Además del diagrama modular que describe la estructura del sistema, estas metodologías se apoyan en diagramas ampliados del modelo de datos, en que se ponen de manifiesto distintas relaciones entre los datos, independientes de las relaciones de uso, que son las que aparecen en el diagrama de estructura.

4.4.1 Diseño orientado a objetos

Puesto que la mayoría de las metodologías orientadas a objetos son bastante similares, nos limitaremos aquí a describir una técnica general para derivar el diseño a partir de las especificaciones del sistema. Realmente el diseño orientado a objetos se solapa en parte con el análisis del sistema si en él se emplean ya objetos para describir el modelo del sistema a desarrollar.

La técnica general de diseño se basa en los siguientes pasos:

- 1) Estudiar y comprender el problema a resolver. Reformular las especificaciones, si se considera conveniente, para hacerlas suficientemente precisas.
- 2) Desarrollar en líneas generales una posible solución. Describirla suficientemente, al menos de una manera informal.
- 3) Formalizar dicha estrategia en términos de clases y objetos y sus relaciones. Esto puede hacerse mediante las siguientes etapas:
 - a.- Identificar los objetos (o clases), sus atributos y componentes.
 - b.- Identificar las operaciones sobre los objetos y asociarlas a la clase u objeto adecuado.
 - c.- Aplicar herencia donde sea conveniente.
 - d.- Describir cada operación en función de las otras, y subsanar posibles omisiones.
 - e.- Establecer la estructura modular del sistema, asignando clases y objetos a módulos.

Si tras estos pasos el diseño del sistema no está suficientemente refinado, se vuelven a repetir, aplicándolos a las clases y objetos poco detallados, hasta conseguir un diseño aceptable, listo para ser codificado. A continuación se detalla algo más cada uno de los elementos de esta técnica general.

1. ESTUDIAR Y COMPRENDER EL PROBLEMA. Debe haberse realizado ya en la fase de análisis de requisitos. Sin embargo puede ser necesario repetirlo en parte, bien porque la especificación no sea suficientemente precisa, o simplemente porque el diseño va a ser realizado por personas diferentes de las que confeccionaron la especificación.

2. DESARROLLAR UNA POSIBLE SOLUCIÓN. Es posible que esto se haya hecho también durante la fase de análisis, aunque es más probable que se tenga que hacer o completar en la fase de diseño. Convendrá considerar varias alternativas y elegir la que se considere más apropiada. La solución elegida debe expresarse con suficiente detalle como para que en su descripción aparezcan mencionados casi los elementos que formarán parte del diseño. De todas maneras puede ser suficiente una descripción informal.
- 3.a IDENTIFICAR LAS CLASES Y OBJETOS. Puede hacerse siguiendo la técnica de Abbott mencionada anteriormente. En este primer paso se atiende sobre todo a identificar las clases de objetos y sus atributos. Si un objeto contiene otros objetos, no se suelen tratar como atributos, sino que se establece una relación de composición entre el objeto compuesto y los objetos componentes. Tras este primer paso puede ya confeccionarse un diagrama inicial del modelo de objetos, con las relaciones de composición, así como otras relaciones generales que se vayan identificando.
- 3.b IDENTIFICAR LAS OPERACIONES SOBRE LOS OBJETOS. También puede hacerse siguiendo la técnica de Abbott. Además de identificar las operaciones hay que decidir a qué objeto o clase se asocia. Esto puede ser un problema de diseño no trivial. Por ejemplo, la operación de escribir en pantalla la representación como caracteres de una fecha puede asociarse al objeto fecha, o bien al objeto pantalla. Cada decisión tiene sus ventajas e inconvenientes. En algunos casos puede fraccionarse la operación en varias más sencillas. Por ejemplo, se puede definir una operación de conversión de fecha a texto, sobre el objeto fecha, y otra operación de escritura del texto sobre el objeto pantalla. Las operaciones pueden reflejarse sobre el diagrama de modelo de objetos.
- 3.c APlicar HERENCIA. Una vez identificados los objetos y sus operaciones asociadas, hay que detectar analogías entre ellos, si las hay, y establecer las relaciones de herencia apropiadas, reformulando las descripciones de los objetos si es necesario. Estas relaciones de herencia se incluirán en el diagrama de modelo de objetos que se va desarrollando.
- 3.d DESCRIBIR LAS OPERACIONES. Esta es una manera de verificar que el diseño es consistente. Cada operación se describe de manera informal o en pseudocódigo, haciendo referencia únicamente a operaciones o clases de datos definidos en este mismo diseño, o bien predefinidos

180 Introducción a la Ingeniería de Software

en el lenguaje de programación a usar y otros elementos de software ya disponible. En caso necesario habrá que añadir nuevas operaciones a las ya identificadas, o incluso nuevas clases de objetos, y se actualizará el modelo de objetos.

3.e ESTABLECER LA ESTRUCTURA MODULAR. Para ello hay que asignar clases, objetos y operaciones a módulos separados. En principio se intentará que cada módulo corresponda a una clase de objetos (o a un objeto en particular). Si el módulo es demasiado complicado, ciertas operaciones pueden establecerse como módulos separados. También es posible agrupar en un solo módulo varios objetos o clases muy relacionados entre sí, para mejorar las características de acoplamiento y cohesión. Como resultado de esta etapa se obtendrá el diagrama de estructura del sistema.

Como ya se ha indicado, hay que analizar si el diseño modular resultante es apropiado para pasar a la fase de codificación. Si algún módulo es todavía demasiado complejo, o no está definido con suficiente precisión, se repetirán los pasos anteriores de diseño para ese módulo, con objeto de refinarlo.

4.4.2 Ejemplo: Estación meteorológica

Para la realización de este ejercicio de diseño se parte de una especificación o descripción informal del sistema a desarrollar:

EJEMPLO: ESTACIÓN METEOROLÓGICA

Un sistema de recogida de datos meteorológicos está formado por una serie de estaciones meteorológicas automáticas que recogen datos ambientales, realizan algún tratamiento local de dichos datos, y envían periódicamente la información recogida y elaborada a un computador de zona para su tratamiento.

Se trata de diseñar el software que ha de controlar el funcionamiento de una de estas estaciones automáticas.

Los datos medidos son:

- Temperatura
- Velocidad y dirección del viento
- Presión atmosférica
- Precipitación (lluvia)

Para ello se dispone de dispositivos de medida que aceptan las siguientes órdenes básicas:

Termómetro:

- lectura de la temperatura en ese instante

Anemómetro:

- lectura de la velocidad de viento en ese instante

Veleta:

- lectura de la dirección de viento en ese instante

Barómetro:

- lectura de la presión atmosférica en ese instante

Pluviómetro:

- iniciar medición (vaciar el depósito)
- lectura de la lluvia caída desde el inicio anterior

Los datos de los medidores deberán leerse cada minuto, excepto la precipitación que se leerá cada hora.

Con las lecturas de los instrumentos deberán realizarse los siguientes cálculos para cada periodo de una hora:

- Con los datos de temperatura, presión, y velocidad de viento, se obtendrán los valores máximo, mínimo y medio.
- Para la dirección de viento, se obtendrá la media y se marcarán para enviar todas las lecturas que se desvien en más de 15°.

La estación meteorológica dispone de otros dos dispositivos para la medida del tiempo y comunicación con el computador de zona. Las operaciones básicas son:

Reloj:

- lectura de la hora en ese instante
- poner el reloj en hora

Modem:

- recibir un mensaje
- transmitir un mensaje

La comunicación con el computador de zona se realiza por línea compartida. El computador de zona explorará cíclicamente (mediante *polling*) las estaciones meteorológicas para ver si tienen algo que comunicar. Las estaciones responderán con uno o varios mensajes, el último de los cuales indicará fin de transmisión.

Las estaciones comunicarán datos cuando hayan completado el tratamiento de los valores de una hora. El mensaje de exploración desde el computador central incluirá la hora del reloj maestro. La estación pondrá su reloj en hora si la desviación del reloj local excede de 20 segundos.

El arranque de la estación meteorológica se hará automáticamente al conectarse a la corriente, o mediante un pulsador manual. En el arranque, se esperará a una exploración del computador de zona, se pondrá el reloj en hora, se notificará que se produce dicho arranque, y se inicializará la recogida de datos.

También habrá un pulsador manual de parada que detendrá la operación de la estación.

El diseño orientado a objetos puede realizarse siguiendo los pasos indicados anteriormente:

182 Introducción a la Ingeniería de Software

1. *Estudiar y comprender el problema.* Requiere leer con atención las especificaciones y consultar todos los puntos dudosos que se encuentren. En particular habrá que conocer el formato de los mensajes a enviar y recibir.

2. *Desarrollar una posible solución.* En este caso se puede optar por mantener en memoria los datos necesarios para ir componiendo la información que habrá que enviar cada hora. Esta información será:

- Para la temperatura, presión, velocidad y dirección de viento, la suma de las medidas y el número de ellas, para obtener la media al final.
- Para la temperatura, presión y velocidad de viento, el máximo y el mínimo hasta cada momento, que serán finalmente los de toda la hora.
- Para la dirección de viento, cada una de las muestras, para poder seleccionar al final de la hora las que se desvien más del límite.

Los valores finales se calcularán al cabo de la hora, y se pondrán a cero los registros. Los valores finales se mantendrán almacenados en espera de transmitirlos a la estación central, al mismo tiempo que se va realizando la recogida de datos de la hora siguiente. Al completar una nueva hora, los nuevos datos totales se almacenan reemplazando a los de la hora anterior, aunque éstos no hayan podido ser trasmitidos a la estación central.

3.a *Identificar las clases y objetos.* Según la técnica de Abbott, marcando los términos clave como se ha hecho en la descripción informal inicial, se pueden confeccionar las siguientes listas de elementos significativos:

<u>DATOS</u>	<u>OPERACIONES</u>
datos meteorológicos	lectura
= datos ambientales	= leer
= temperatura	iniciar medición
= velocidad del viento	obtener máximo
= dirección del viento	obtener mínimo
= presión atmosférica	obtener media

= precipitación	poner el reloj en hora
= lecturas que se desvían	recibir
dispositivo de medida	= explorar
mensaje	transmitir
fin de transmisión	= responder
hora	= comunicar
estación	= notificar
pulsador manual	arranque detener

A partir de la lista de nombres de datos hay que identificar los candidatos a clases y objetos. Este paso no es trivial, y requiere cierta experiencia o habilidad para realizarlo. En este ejemplo se puede llegar a la siguiente lista de clases y atributos:

Dispositivo de medida	
lectura	
máximo	
mínimo	
media	
lecturas que se desvían	
Mensaje	
hora	
datos meteorológicos	
fin de transmisión	
Reloj	
hora	
Estación	

3.b *Identificar las operaciones sobre los objetos.* A partir de la lista de operaciones se van asignando los diferentes elementos a las clases reconocidas. En este caso, podemos llegar a la siguiente distribución:

Dispositivo de medida	
leer	
iniciar medición	
obtener máximo	
obtener mínimo	
obtener media	
Mensaje	
enviar	
recibir	

184 Introducción a la Ingeniería de Software

Reloj	
	leer
	poner en hora
Estación	
	arrancar
	detener

3.c *Aplicar herencia.* En este caso se puede observar que no todas las operaciones sobre dispositivos de medida son aplicables a cada uno de los dispositivos. Podemos identificar medidores especializados, con operaciones particulares. Así llegamos al siguiente esquema superclase/subclases:

Medidor	
	Medidor con máximo, mínimo y media
	Medidor con media y lecturas que se desvian
	Medidor con puesta a cero inicial

La primera subclase corresponde a las mediciones de temperatura, presión y velocidad de viento, mientras que la segunda corresponde a la dirección del viento, y la tercera a la precipitación. Ahora hay que redefinir los atributos y operaciones sobre estas clases. Las nuevas operaciones podrían ser:

Medidor	
	leer (lectura simple)
Medidor con máximo, mínimo y media	
	leer (acumulando y actualizando máximo y mínimo)
	obtener media
	poner a cero (el acumulador)
Medidor con media y lecturas que se desvian	
	leer (acumulando y registrando lecturas)
	obtener media
	obtener medidas desviadas
	poner a cero (el acumulador)
Medidor con puesta a cero inicial	
	iniciar medición

Todos los medidores especializados heredan la operación de lectura simple, y la mayoría de ellos la redefinen.

3.d *Describir las operaciones.* Ahora hay que comprobar que cada operación es realizable, bien directamente o en función de las otras. Por ejemplo, la clase "Medidor" y la subclase "Medidor con máximo, mínimo y media" se podrían definir, en pseudocódigo, en la forma siguiente:

```

CLASE Medidor
  ATRIBUTOS
    lectura: TipoMedida
  OPERACION Leer
    tomar una nueva 'lectura'
  FIN-CLASE

CLASE MedidorConMáximo ES-UN Medidor
  ATRIBUTOS
    máximo, minino: TipoMedida
    acumulado: TipoMedida
    numMuestras: ENTERO
  OPERACION PonerACero
    poner a cero 'acumulado' y 'numMuestras'
  OPERACION ObtenerMedia( media )
    devuelve 'media' = 'acumulado' / 'numMuestras'
  OPERACION Leer
    SUPER.Leer
    acumular la nueva 'lectura' en 'acumulado'
    incrementar 'numMuestras'
    si la nueva 'lectura' es mayor que el 'máximo', anotarla
      como nuevo 'máximo'
    si la nueva 'lectura' es menor que el 'mínimo', anotarla
      como nuevo 'mínimo'
  FIN-CLASE

```

Las operaciones que dan problemas son las de "Arrancar" y "Detener" la estación. Con un poco de reflexión se puede comprender que en realidad la estación sólo tiene una operación que realizar, que es el trabajo normal, y que podemos redefinir como "Operar". El flujo de control de esta operación puede adoptar la forma clásica de monitor cíclico, lo que conduce al siguiente pseudocódigo:

```

CLASE Estación
OPERACION Operar
    inicializar la estación
REPETIR
    SI hay mensaje ENTONCES
        recibir mensaje
        poner en hora el reloj, si hace falta
        componer mensaje con datos de la hora anterior
        enviar el mensaje
    FIN-SI
    leer el reloj
    SI ha pasado un minuto ENTONCES
        leer medidas de temperatura, viento y presión
    FIN-SI
    SI ha pasado una hora ENTONCES
        leer pluviómetro
        iniciar lectura del pluviómetro
        calcular los datos de la hora
    FIN-SI
HASTA tecla de parada pulsada
FIN-CLASE

```

Esta descripción nos lleva a descubrir que necesitamos dos operaciones adicionales sobre la clase "Mensaje", correspondientes a detectar si ha llegado un nuevo mensaje, y a componer el mensaje para enviarlo.

Con el resultado de todos estos pasos del diseño se puede ya confeccionar un diagrama de objetos del software de la estación meteorológica, tal como el que se representa en la figura 4.14. Un objeto "Estación" se compone de un gestor de "Mensajes" (modem), un "Reloj", y una colección de medidores de las clases indicadas. Los medidores son especializaciones de la clase genérica "Medidor".

3.e *Establecer la estructura modular.* Ahora se agrupan los elementos del diseño en módulos, y se construye el diagrama de estructura de la aplicación, con las relaciones de uso entre módulos. La forma más sencilla de hacerlo es asignar un módulo a cada clase de objetos. Con ello se llega a la arquitectura representada en la figura 4.15. Las relaciones de herencia de la figura 4.14 se traducen ahora a relaciones de uso. Una subclase utiliza el código de su superclase.

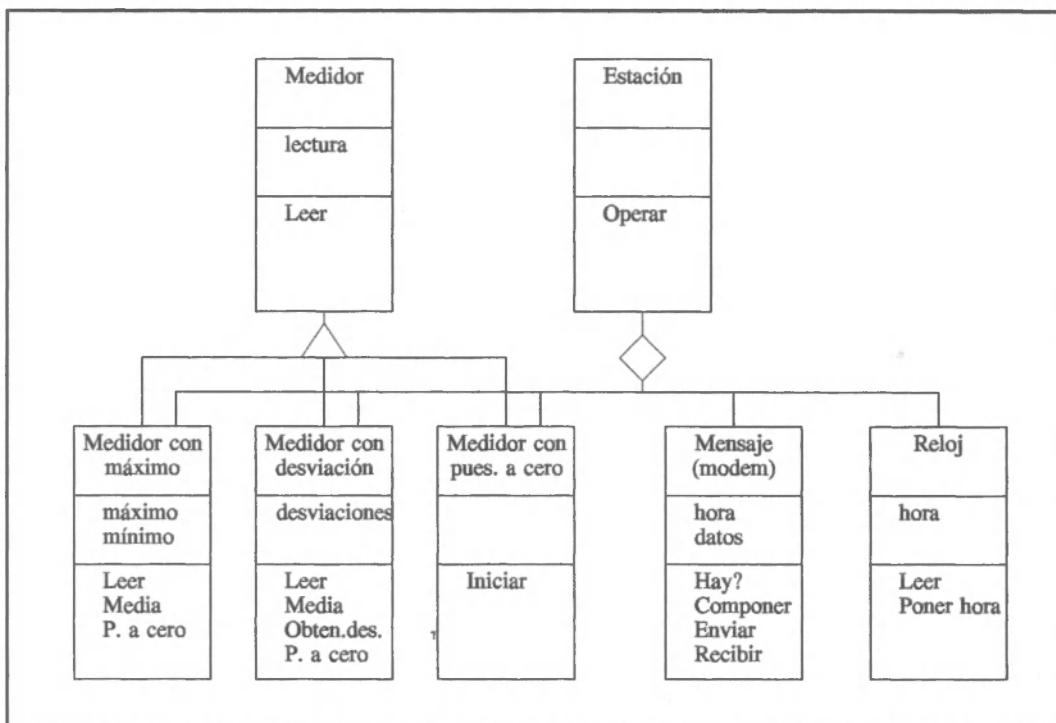


Figura 4.14 Modelo de objetos de la estación meteorológica

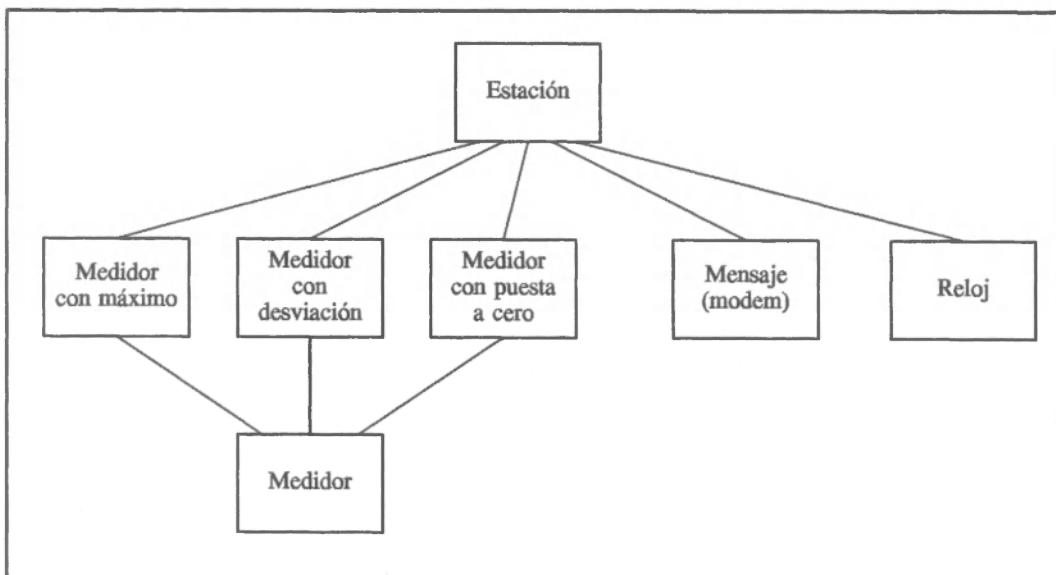


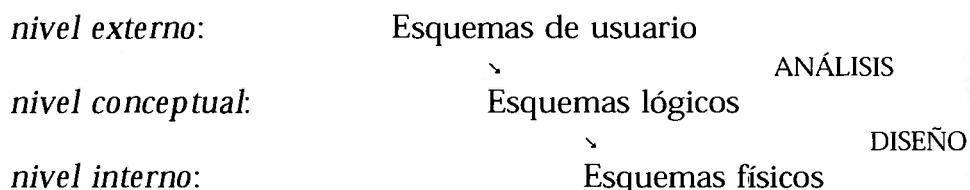
Figura 4.15 Arquitectura del software de la estación meteorológica

4.5 Técnicas de diseño de datos

La mayoría de las aplicaciones informáticas requieren almacenar información en forma permanente. La manera típica de hacerlo es apoyando esa aplicación en una base de datos subyacente. El diseño de la estructura de las bases de datos es hoy día una disciplina independiente [Date90], [DeMiguel93], al igual que otras que sirven de apoyo a la ingeniería del software.

En esta sección y las siguientes se presentan algunas ideas fundamentales sobre cómo organizar la base de datos en que se almacena la información permanente de una aplicación informática. Muchas de estas ideas pueden aplicarse también a la organización de las estructuras de datos en memoria, durante la operación de la aplicación.

La organización de la base de datos puede realizarse desde varios puntos de vista. Una forma clásica de enfocarla es en tres niveles: externo, conceptual e interno. En cada nivel se establecen esquemas de organización de los datos desde un punto de vista concreto. El paso de un nivel a otro es el siguiente:



El *nivel externo* corresponde a la visión de usuario. La organización de los datos se realiza siguiendo esquemas significativos en el campo de la aplicación, por ejemplo, en forma de ficha de cliente o de empleado.

El *nivel conceptual* establece una organización lógica de los datos, con independencia del sentido físico que tengan en el campo de aplicación. Esta organización se resume en un diagrama de modelo de datos, bien del tipo Entidad-Relación (descrito en el Tema 2) o como diagrama de Modelo de Objetos (descrito en el Tema 3).

El *nivel físico* organiza los datos según los esquemas admisibles en el sistema de gestión de bases de datos y/o lenguaje de programación elegido para el desarrollo. Si se utiliza una base de datos relacional los esquemas físicos serán esquemas de tablas.

El paso del nivel externo al nivel conceptual se puede realizar durante la etapa de análisis de requisitos. El paso del nivel conceptual al nivel interno se realizará habitualmente durante la fase de diseño.

4.6 Diseño de bases de datos relacionales

Partiendo del modelo Entidad-Relación o bien del Modelo de Objetos, es posible dar reglas prácticas, relativamente sencillas, para obtener los esquemas de las tablas de una base de datos relacional que reflejen la visión lógica de los datos, y que sean aceptablemente eficientes. En [Rumbaugh91] se sistematizan bien esas recomendaciones, que nos guían sobre la forma de traducir a esquemas de tablas los atributos de las entidades, y las relaciones entre ellas, incluyendo las de composición y herencia entre objetos.

En el modelo relacional, los aspectos de eficiencia se contemplan desde dos puntos de vista. Por una parte se establecen las llamadas *formas normales*, que tienden a evitar redundancias en los datos almacenados. Por otra parte se estudia el empleo de *índices* para mejorar la velocidad de acceso a los datos.

4.6.1 Formas normales

Las formas normales de Codd definen criterios para establecer esquemas de tablas que sean claros y no redundantes. Estos criterios se numeran correlativamente, de menor a mayor nivel de restricción, dando lugar a las formas normales 1^a, 2^a, 3^a, etc. Una tabla que cumpla con una cierta forma normal cumple también con las anteriores.

Para ilustrar las definiciones de estas formas normales, consideraremos como ejemplo una tabla en que se recoge información sobre la organización de las clases presenciales en un determinado centro educativo. En cada curso hay varios grupos de clase, y en cada grupo se imparten varias asignaturas. Cada asignatura es impartida por uno o varios profesores, cada uno de los cuales se encarga de esa asignatura en uno o varios grupos. Cada asignatura tiene, además, un profesor coordinador, que puede ser, o no, uno de los profesores que la imparten. Cada profesor tiene asignado un despacho. En la figura 4.16 se recoge toda esta información en una sola tabla.

190 Introducción a la Ingeniería de Software

Grupo	Asignatura	Profesor	Despacho	Coordinador
11, 12	Cálculo	F. Díaz	D-23	R. Pérez
13	Cálculo	A. García	D-27	R. Pérez
11	Álgebra	C. Morales	D-34	F. Arranz
12	Álgebra	M. Campos	D-21	F. Arranz
13	Álgebra	F. Arranz	D-32	F. Arranz

Figura 4.16 Tabla sin ninguna normalización

Se dice que una tabla se encuentra en *1^a forma normal* si la información asociada a cada una de las columnas es un valor único, y no una colección de valores en número variable. La tabla de la figura 4.16 no está en 1^a forma normal, porque hay alguna casilla en que se ha anotado más de un valor (varios grupos con el mismo profesor). Podemos forzar que la tabla esté en 1^a forma normal desdoblando la fila de esa casilla en tantas filas como valores hay en la misma casilla, tal como se indica en la figura 4.17.

Grupo	Asignatura	Profesor	Despacho	Coordinador
11	Cálculo	F. Díaz	D-23	R. Pérez
12	Cálculo	F. Díaz	D-23	R. Pérez
13	Cálculo	A. García	D-27	R. Pérez
11	Álgebra	C. Morales	D-34	F. Arranz
12	Álgebra	M. Campos	D-21	F. Arranz
13	Álgebra	F. Arranz	D-32	F. Arranz

Figura 4.17 Tabla en 1^a forma normal, pero no en 2^a

Se dice que una tabla está en *2^a forma normal* si está en 1^a forma normal y además hay una clave primaria (una columna o combinación de varias) que distingue cada fila, y cada casilla que no sea de la clave primaria depende de toda la clave primaria. En la tabla de la figura 4.17 la clave primaria es la combinación de las dos primeras columnas. La tabla no está en 2^a forma normal porque el dato del coordinador de la asignatura no depende de toda la clave primaria, sino sólo de parte de ella (depende de la asignatura, pero no del grupo). Se puede conseguir la segunda forma normal suprimiendo este dato de la tabla principal, y usando una tabla auxiliar para relacionar la asignatura con el coordinador, tal como se hace en las tablas de la figura 4.18.

Grupo	Asignatura	Profesor	Despacho	Asignatura	Coordin.
11	Cálculo	F. Díaz	D-23	Cálculo	R. Pérez
12	Cálculo	F. Díaz	D-23	Álgebra	F. Arranz
13	Cálculo	A. García	D-27		
11	Algebra	C. Morales	D-34		
12	Algebra	M. Campos	D-21		
13	Algebra	F. Arranz	D-32		

Figura 4.18 Tabla en 2^a forma normal, pero no en 3^a

Se dice que una tabla está en 3^a forma normal si satisface el criterio de la 2^a forma normal y además el valor de cada columna que no es clave primaria depende directamente de la clave primaria, es decir, no hay dependencias entre columnas que no son clave primaria. La tabla de la figura 4.18 no está en 3^a forma normal porque el despacho del profesor depende del profesor. Puede conseguirse la tercera forma normal eliminando de la tabla cada columna dependiente de otra no clave primaria, y usando una tabla auxiliar para registrar dicha dependencia. Esto se ha hecho en las tablas de la figura 4.19.

Grupo	Asignat.	Profesor	Asignat.	Coordin.	Profesor	Desp
11	Cálculo	F. Díaz	Cálculo	R. Pérez	F. Díaz	D-23
12	Cálculo	F. Díaz	Álgebra	F. Arranz	A. García	D-27
13	Cálculo	A. García			C. Morales	D-34
11	Algebra	C. Morales			M. Campos	D-21
12	Algebra	M. Campos			F. Arranz	D-32
13	Algebra	F. Arranz				

Figura 4.19 Tabla en 3^a forma normal

Existen criterios aún más restrictivos que definen las formas normales 4^a y siguientes, que se aplican poco en la práctica.

4.6.2 Diseño de las entidades

En el modelo relacional cada entidad del modelo E-R se traduce en una tabla por cada clase de entidad, con una fila por cada elemento de esa clase y una columna por cada atributo de esa entidad.

Si una entidad está relacionada con otras, y se quiere tener una referencia rápida entre las entidades relacionadas, se puede incluir además una columna conteniendo un código o número de referencia que identifique cada elemento de datos, es decir, cada fila de la tabla. En el modelo de objetos, esto corresponde a almacenar explícitamente en la tabla el identificador del objeto.

El número o código de referencia, si se usa, sirve perfectamente como clave primaria.

4.6.3 Tratamiento de las relaciones de asociación

En el modelo de objetos se distinguen dos tipos especiales de relación, que son las de composición (o agregación) y las de herencia (o especialización). Las demás relaciones se denominan genéricamente relaciones de asociación. En el modelo E-R todas las relaciones se consideran relaciones de asociación. En las explicaciones siguientes se atiende sólo a relaciones binarias, entre parejas de entidades.

La manera de almacenar en tablas la información de las relaciones de asociación depende de la cardinalidad de la relación. La técnica general es traducir la relación a una tabla conteniendo referencias a las tablas de las entidades relacionadas, así como los atributos de la relación, si los hay. Esto es válido para relaciones con cualquier cardinalidad, incluyendo N-N. La referencia a las entidades relacionadas se hará mediante la clave primaria de cada una. La figura 4.20 (a) recoge esta idea.

Si la cardinalidad es 1-N, es decir, la cardinalidad de un lado de la relación está limitada a 1, es posible incluir los datos de la relación en la misma tabla de una de las entidades relacionadas, tal como se indica en la figura 4.20 (b).

Si la relación es 1-1, es decir, la cardinalidad está limitada a 1 en cada lado, se pueden fundir las tablas de las dos entidades en una sola, tal como se hace en la figura 4.20 (c).

El reducir el número de tablas simplifica en cierto modo los esquemas físicos de la base de datos. Por ejemplo, algunas de las columnas con códigos de referencia pueden ya no ser necesarias, tal como se indica con líneas de puntos en la figura. Esta simplificación se hace a costa de reducir, en algunos casos, el grado de normalización.

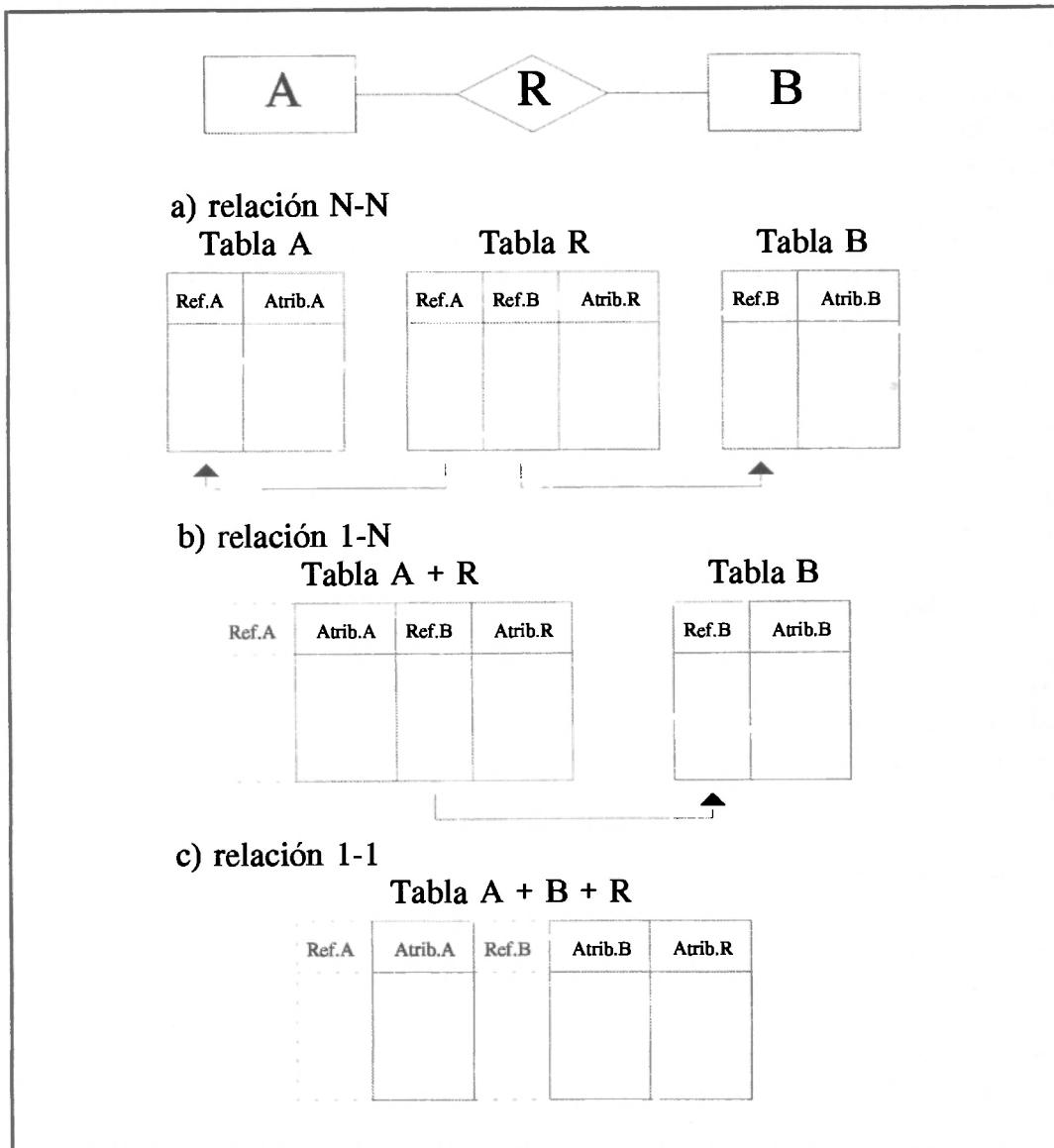


Figura 4.20 Tablas para relaciones de asociación

4.6.4 Tratamiento de las relaciones de composición

Las relaciones de composición o agregación se tratan de la misma manera que las relaciones de asociación. En las relaciones de composición la cardinalidad del lado del objeto compuesto es casi siempre 1, por lo que serán de aplicación las simplificaciones indicadas en la sección anterior.

4.6.5 Tratamiento de la herencia

Cuando una clase de objetos (entidad genérica) tiene varias subclases (entidades específicas) se pueden adoptar tres formas de almacenar en tablas la información de las entidades, tal como se recoge en la figura 4.21. En el caso (a) se usa una tabla para la superclase, con los atributos comunes, heredados por las subclases, más una tabla por cada subclase, con sus atributos específicos. En el caso (b) se han repetido los atributos comunes en las tablas de cada subclase, por lo que desaparece la tabla de la superclase. En el caso (c) se prescinde de las tablas separadas para cada subclase, y se amplía la tabla de la superclase con todos los atributos de cada una de las subclases, en forma de campos o columnas con valores opcionales, según cuál sea la subclase del objeto almacenado en cada fila de la tabla.

4.6.6 Diseño de índices

Los índices permiten acceder rápidamente a un dato concreto, reduciendo así el tiempo de acceso. Pero esto es a costa de aumentar el espacio necesario de almacenamiento y, lo que es más importante, aumentar el tiempo necesario para almacenar cada nuevo dato y más aún para modificar el valor de un atributo indexado (la modificación equivale a suprimir el elemento del índice y luego reinsertarlo con el nuevo valor).

En general, si hay que acceder a datos a través de sus relaciones con otros, será conveniente mantener índices sobre las claves primarias y columnas de referencia de las entidades relacionadas. Aparte de esto, no es fácil dar criterios generales sencillos para decidir cuándo conviene o no establecer índices adicionales sobre campos no clave.

4.6.7 Ejemplo: Diseño de datos para la gestión de biblioteca

La visión externa de los datos está constituida por los elementos "Ficha de libro" y "Ficha de lector", con los siguientes contenidos:

Ficha de libro:

- Autor(es)
- Título
- Editorial, etc.
- Materia(s)
- Lector (si está prestado)
- Fecha del préstamo (si está prestado)

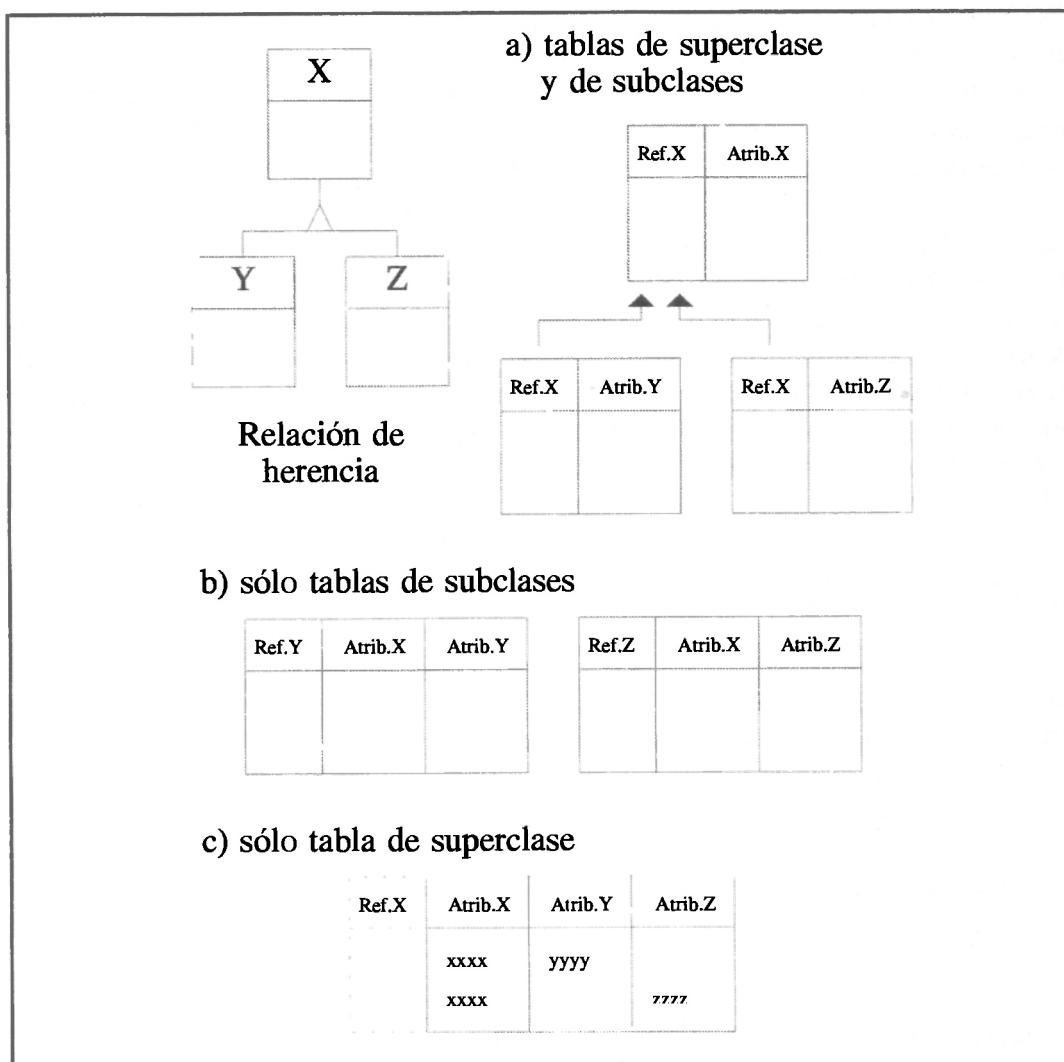


Figura 4.21 Tablas para relaciones de herencia

Ficha de lector:

Nombre
Apellidos
Domicilio
Teléfono

En el documento de especificación de requisitos correspondiente a este ejemplo, que aparece en el Tema 2, se ha planteado como modelo conceptual correspondiente a estos datos los del diagrama E-R que allí aparece, con las entidades LIBRO, LECTOR y MATERIA, y las relaciones

196 Introducción a la Ingeniería de Software

PRESTADO-A y TRATA-DE. La razón para desglosar la lista de materias de las fichas de libro es para permitir que dicha lista de materias ("thesaurus", en la terminología de los bibliotecarios) sea permanente, y todas las materias estén reseñadas aunque no haya ningún libro que trate de alguna de ellas en un momento dado.

La traducción directa de estas entidades y relaciones a tablas relacionales tiene el inconveniente de la falta de normalización en lo referente a los autores de los libros. En la visión inicial se habla del "autor" de un libro, cuando en realidad pueden ser varios. En este caso no se cumpliría la 1^a forma normal, ya que en el mismo campo tenemos una colección de datos en número variable. Para conseguir la 1^a forma normal se puede promocionar el autor como entidad independiente, y establecer la relación AUTOR-DE entre autores y libros. Esto nos lleva a rehacer el modelo conceptual según el diagrama E-R de la figura 4.22.

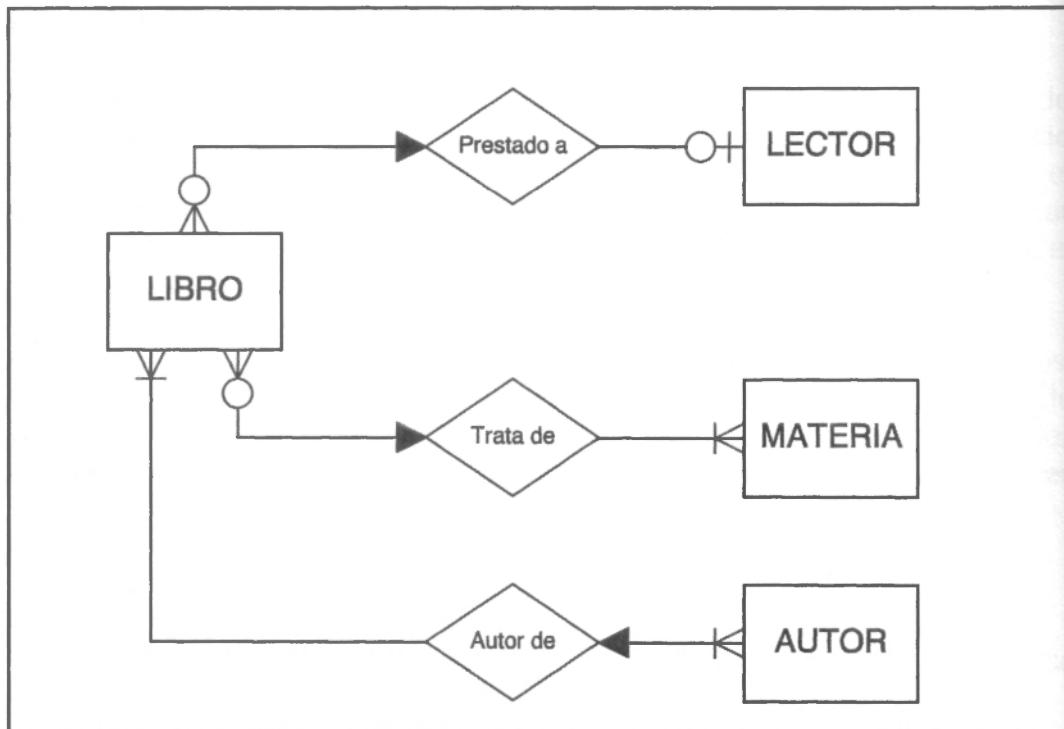


Figura 4.22 Modelo de datos revisado

Ahora se puede hacer de forma adecuada el diseño de los esquemas físicos de la base de datos relacional. Las entidades LIBRO, LECTOR, AUTOR y MATERIA se traducen a una tabla cada una, incluyendo una columna de Nº de referencia para acceso mediante las relaciones.

<u>LIBRO</u>	<u>LECTOR</u>	<u>AUTOR</u>	<u>MATERIA</u>
NºRef	NºRef	NºRef	NºRef
Título	Nombre	Nombre	Nombre
Editorial ...	Apellidos		
	Domicilio		
	Teléfono		

Las relaciones TRATA-DE y AUTOR-DE son de cardinalidad N-N, y se traducen en una tabla adicional para cada una.

<u>TRATA-DE</u>	<u>AUTOR-DE</u>
NºRef.Libro	NºRef.Libro
NºRef.Materia	NºRef.Autor

La relación PRESTADO-A es de cardinalidad 1-N. Se puede aplicar la simplificación indicada en la sección 4.6.3, incluyendo esta información en la tabla de libros.

<u>LIBRO</u>
NºRef
Título
Editorial ...
NºRef.Lector
FechaPréstamo

Con esto se tiene de nuevo un esquema similar al de la visión externa de la ficha de libro, en la que aparecen anotados el lector y la fecha del préstamo, en su caso. Estos campos de la tabla serán opcionales, y estarán vacíos si el libro no está prestado.

4.7 Diseño de bases de datos de objetos

A diferencia de las bases de datos que siguen el modelo relacional, en las bases de datos orientadas a objetos no hay todavía un conjunto estable de estructuras de información fundamentales, que se consideren primitivas de este modelo de bases de datos. En las bases de datos relacionales sólo se trabaja con la estructura tabla. En las bases de datos de objetos hay una mayor variedad de estructuras disponibles, pero distintas en cada caso.

198 Introducción a la Ingeniería de Software

En general pueden adoptarse dos enfoques en el diseño físico con estas bases de datos. El primer enfoque resulta apropiado cuando la base de datos de objetos permite usar una gran variedad de estructuras. En este caso el diseño se puede hacer como para las estructuras de datos en memoria. El sistema de gestión de base de datos aporta como complemento la persistencia de los datos.

El segundo enfoque se aplicaría cuando no existe esa gran variedad de estructuras de datos, y la base de datos de objetos resulta análoga a una base de datos relacional, en que se establece implícitamente una tabla por cada clase de objetos. En este caso se seguirían las recomendaciones ya dadas para el diseño sobre el modelo relacional. El sistema de gestión de base de datos aporta la existencia implícita de identificadores de objetos, que hacen innecesarias las columnas explícitas de códigos o números de referencia.

4.8 Ejemplos de diseños

En este apartado se recogen los documentos completos de diseño de los dos sistemas que se han ido desarrollando a lo largo del libro.

4.8.1 Videojuego de las minas

El documento ADD para el videojuego de las minas es el siguiente:

DOCUMENTO DE DISEÑO DEL SOFTWARE (ADD)

Proyecto: JUEGO DE LAS MINAS (Versión Simple)
Autor(es): J.A. Cerrada y R. Gómez
Fecha: Febrero 1999
Documento: MINAS-ADD-99

CONTENIDO

- 1. INTRODUCCIÓN**
 - 1.1 Objetivo
 - 1.2 Ámbito

- 1.3 Definiciones, siglas y abreviaturas
- 1.4 Referencias

2. PANORÁMICA DEL SISTEMA

3. CONTEXTO DEL SISTEMA

4. DISEÑO DEL SISTEMA

- 4.1 Metodología de diseño de alto nivel
- 4.2 Descomposición del sistema

5. DESCRIPCIÓN DE COMPONENTES

6. VIABILIDAD Y RECURSOS ESTIMADOS

7. MATRIZ REQUISITOS/COMPONENTES

1. INTRODUCCIÓN

1.1 Objetivo

Se trata de realizar un videojuego denominado "Juego de las Minas" cuyas reglas y características generales se detallan en el documento de requisitos del software: MINAS-SRD-99. En líneas generales, en este juego se trata de descubrir dentro del tablero la situación de las minas ocultas y situadas aleatoriamente sin que ninguna de ellas explote y en el menor tiempo posible.

1.2 Ámbito

En este desarrollo se abordará una versión simple que utilizará una interfase hombre-máquina para pantalla alfanumérica y teclado. En una fase posterior, se desarrollará una versión más elaborada que utilizará pantalla gráfica y ratón. El desarrollo de esta versión simple y la posterior deberá diferir solamente en los módulos específicos dedicados a la interfase hombre-máquina.

1.3 Definiciones, siglas y abreviaturas

Tablero: Elemento gráfico que se muestra en la pantalla del computador con forma de cuadrícula y en el que se desarrolla el juego.

Casilla: Cada uno de los elementos de los que está formada la cuadrícula del tablero.

Mina: Elemento oculto en una casilla del tablero y que si se destapa provoca la finalización del juego.

1.4 Referencias

MINAS-SRD-99: DOCUMENTO DE REQUISITOS DEL SOFTWARE del VIDEOJUEGO DE LAS MINAS.

2. PANORÁMICA DEL SISTEMA

Se recoge aquí un resumen de la descripción del sistema ya incluida en el documento de especificación de requisitos MINAS-SRD-99.

200 Introducción a la Ingeniería de Software

2.1 Objetivo y funciones

El objetivo es realizar una versión simplificada del juego de las minas. En este juego se trata de descubrir dentro del tablero la situación de las minas sin que ninguna de ellas explote y en el menor tiempo posible.

Las funciones básicas serán:

- Selección del nivel de dificultad del juego (bajo, medio, alto)
- Desarrollo de la partida según las reglas del juego
- Elaboración y mantenimiento de la tabla de mejores resultados
- Ayudas al jugador

2.2 Descripción funcional

En el juego se emplea un tablero cuadrado de N casillas de lado, semejante al que se muestra en la figura M.1.

		2	2	1	2	2	
5	!!	1	..	1	2		
!!	2	1	..	1	!!		
4	1	..	■	1	3		
3	1	1	1		
4	!!	2	1		
!!	4	1	1	1			

Figura M.1 Tablero del juego de las minas

En este tablero están ocultas un número determinado de minas que pueden estar situadas en cualquier casilla. Inicialmente se muestra el tablero con todas las casillas tapadas. En el ejemplo de la figura M.1, las casillas tapadas están en blanco. El jugador tiene que ir destapando las casillas para descubrir la situación de las minas. Cuando se destapa una casilla que tiene una mina, esta

mina explota y se acaba el juego infructuosamente. El objetivo del juego es encontrar la situación de todas las minas sin que explote ninguna de ellas.

Para facilitar la búsqueda de todas las minas, el jugador podrá marcar una casilla cuando tenga la certeza de que en ella existe una mina. En la figura M.1 la marca se indica con el símbolo "!!". Esta marca impide que pueda ser destapada la casilla por error. El jugador también podrá quitar la marca de una casilla. En todo momento se mostrará en pantalla el número de minas ocultas y todavía no marcadas.

El jugador podrá seleccionar el grado de dificultad del juego entre tres niveles: bajo, medio y alto. En cada nivel se empleará un tamaño distinto de tablero y el número de minas a descubrir también será distinto. La situación de las minas en el tablero se realizará de forma aleatoria.

Cada vez que el jugador destapa una casilla, se deberá indicar si tenía una mina, en cuyo caso finaliza el juego, o el número de minas que rodean la casilla que ha sido destapada. En la figura M.1 las casillas con el símbolo ".." indican que el número de minas que las rodean son cero. El número de minas que pueden rodear una casilla pueden ir desde 0 hasta 8.

El juego dispondrá de un cronómetro que actualizará en pantalla cada segundo el tiempo transcurrido desde el comienzo de la última partida en juego. Si el jugador consigue descubrir todas las minas, el programa registrará el tiempo invertido junto con el texto que deseé poner el jugador y actualizará la lista de los mejores tiempos registrados en el nivel correspondiente.

3. CONTEXTO DEL SISTEMA

No hay conexión con otros sistemas.

4. DISEÑO DEL SISTEMA

4.1 Metodología de diseño de alto nivel

Se utiliza la metodología de diseño modular basado en abstracciones.

4.2 Descomposición del sistema

La figura M.2 contiene el diagrama de estructura del sistema. En él aparecen los componentes principales y las relaciones de uso entre ellos, así como una serie de módulos de bajo nivel, que constituyen una librería de soporte.

Las componentes principales son las siguientes:

Abstracciones funcionales:

- Minas
- Ayuda

Tipos Abstractos de Datos:

- Casilla

Datos Encapsulados

- Tablero
- Crono
- Resultados

202 Introducción a la Ingeniería de Software

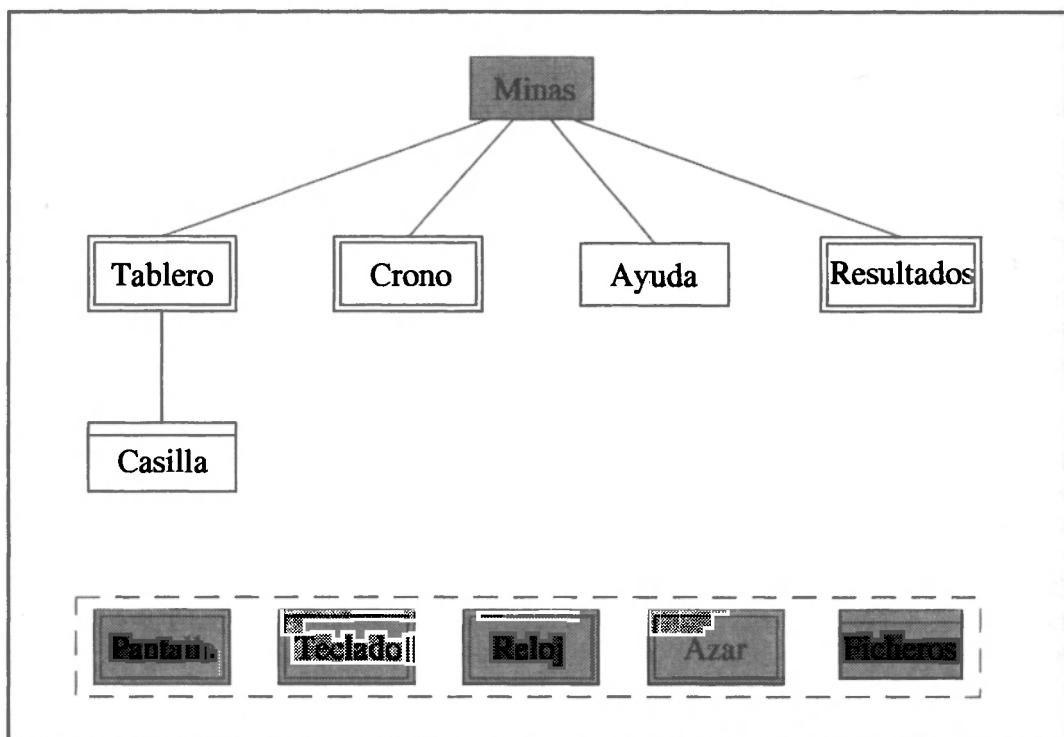


Figura M.2 Diagrama de estructura

El diagrama de estructura del sistema utiliza un módulo para cada una de estas abstracciones, según se muestra en la figura M.2. El módulo principal encargado del control del juego es Minas, que utiliza y coordina al resto de módulos. El dato encapsulado Tablero utiliza el tipo abstracto de dato Casilla. Además, en la figura se muestran varios módulos encargados de adaptar los módulos de librería de los distintos compiladores a las necesidades de este sistema. Estos módulos son:

Tipos Abstractos de Datos:

- Ficheros

Datos Encapsulados:

- Pantalla
- Teclado
- Reloj
- Azar

5. DESCRIPCIÓN DE COMPONENTES

A continuación se describen cada uno de los módulos indicados en el diagrama de estructura del sistema.

5.1 Módulo: MINAS

5.1.1 *Tipo:* Programa Principal

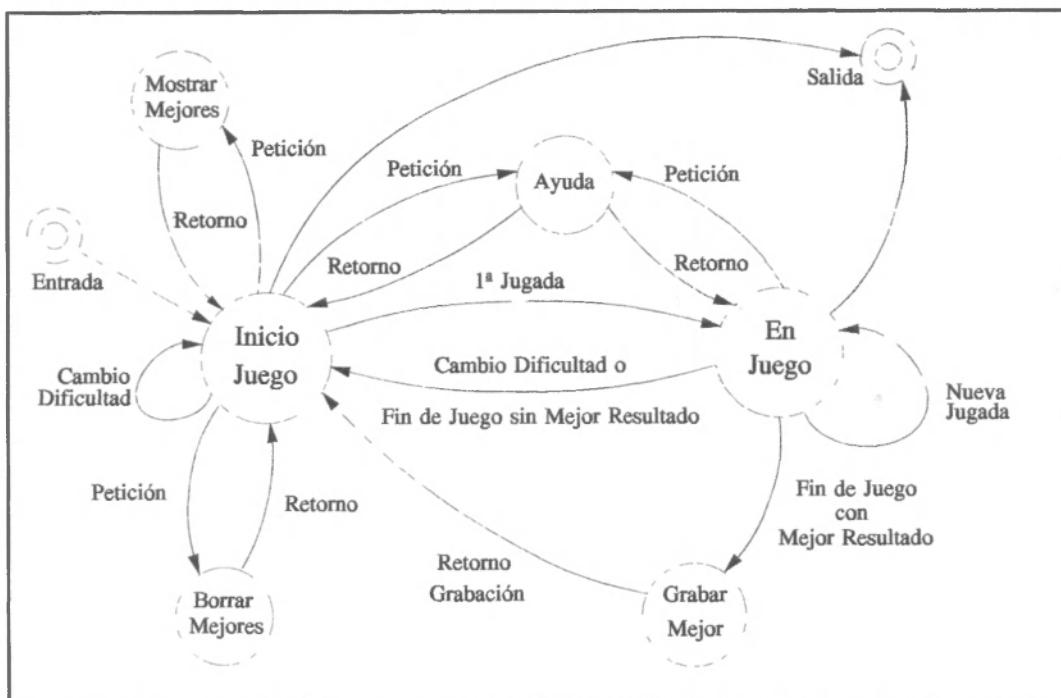


Figura M.3. Diagrama de estados del sistema

5.1.2 *Objetivo*

Efectuar el control principal del juego.

5.1.3 *Función*

Iniciar el juego y controlar la ejecución de las diferentes funciones:

- Selección del nivel de dificultad del juego (bajo, medio, alto)
- Desarrollo de la partida según las reglas del juego
- Elaboración y mantenimiento de la tabla de mejores resultados
- Ayudas al jugador

El flujo de control de este módulo equivale al del sistema completo, y se muestra en la figura M.3.

5.1.4 *Subordinados*: Tablero, Crono, Ayuda, Resultados, Pantalla, Teclado

5.1.5 *Dependencias*: Ninguna

5.1.6 *Interfases*: Ninguna

5.1.7 *Recursos*: Ninguno

5.1.8 *Referencias*: Ninguna

204 Introducción a la Ingeniería de Software

5.1.9 *Proceso:* De acuerdo con el diagrama de la figura M.3 será:

```
Iniciar juego por defecto
REPETIR
    Leer opción sin espera
    SI Inicio Juego ENTONCES
        CASO opción
            SI-ES Ayuda HACER Mostrar ayuda
            SI-ES Cambio dificultad HACER Cambiar dificultad
            SI-ES Borra resultados HACER Borrar Mejores
            SI-ES Muestra resultados HACER Mostrar Mejores
            SI-ES 1a jugada HACER
                Pasar al estado: En Juego
                Realizar jugada
            FIN-CASO
        SI NO
            Actualizar cronómetro
            CASO opción
                SI-ES Ayuda HACER Mostrar ayuda
                SI-ES Cambio dificultad HACER
                    Cambiar dificultad
                    Pasar al estado: Inicio Juego
                SI-ES Jugada HACER
                    Realizar jugada
                    SI Fin de Juego & Mejor resultado ENTONCES Grabar mejor FIN-SI
                    SI Fin de Juego ENTONCES Pasar al estado: Inicio Juego FIN-SI
            FIN-CASO
        FIN-SI
    HASTA Fin de juego
```

5.1.10 *Datos*

- Última opción
- Nivel de dificultad
- Estado del juego

5.2 Módulo: TABLERO

5.2.1 *Tipo:* Dato encapsulado

5.2.2 *Objetivo*

Este módulo realiza todas las operaciones básicas para manejar el tablero del juego.

5.2.3 *Función*

La función de este módulo es encapsular todas las operaciones posibles que se pueden realizar en el tablero en las distintas fases del juego. Las operaciones previstas son las siguientes:

- Iniciar el tablero
- Poner las minas aleatoriamente
- Mover el cursor a otra casilla

- Marcar o desmarcar la casilla apuntada por el cursor
- Destapar casillas a partir de la apuntada por el cursor
- Pintar el tablero

5.2.4 *Subordinados*: Casilla, Azar, Pantalla

5.2.5 *Dependencias*: Minas

5.2.6 *Interfases*: Para cada operación:

Operación: Iniciar tablero

Entrada: Nivel de dificultad

Salida:

Operación: Poner minas

Entrada:

Salida:

Operación: Mover cursor

Entrada: Incremento de posición X, Incremento de posición Y

Salida:

Operación: Marcar o desmarcar casilla

Entrada:

Salida:

Operación: Destapar casillas

Entrada:

Salida: SI/NO fin del juego, SI/NO objetivo alcanzado

Operación: Pintar tablero

Entrada:

Salida: Presentación en pantalla del tablero

5.2.7 *Recursos*: Ninguno

5.2.8 *Referencias*: Ninguna

5.2.9 *Proceso*: Para cada operación:

Operación: Iniciar tablero

Proceso:

Inicializar tamaño tablero y total casillas según nivel

Inicializar número de minas y marcas según nivel

Iniciar cursor

Iniciar casillas

Operación: Poner minas

Proceso:

REPETIR

Elegir casilla aleatoriamente

Poner mina en la casilla

Incrementar el nº de minas que rodean a las casillas vecinas

HASTA Total de minas

206 Introducción a la Ingeniería de Software

Operación: Mover cursor

Proceso:

- Comprobar límites del tablero
- Situar el cursor en la nueva posición
- Pintar casilla con nueva posición del cursor

Operación: Marcar o desmarcar casilla

Proceso:

- Cambiar marca de casilla
- Pintar casilla con/sin marca
- Actualizar nº de casillas marcadas

Operación: Destapar casillas

Proceso:

- Destapar y pintar casilla
- SI no hay mina & vecinas = 0 ENTONCES
 - Destapar y pintar recursivamente las casillas próximas con vecinas = 0
 - FIN-SI
- Actualizar nº de casillas tapadas
- Devolver resultado

Operación: Pintar tablero

Proceso:

- Pintar dibujo base del tablero
- REPETIR
 - Pintar casilla
- HASTA Total casillas

5.2.10 Datos

Los atributos del tablero son los siguientes:

- Número de minas
- Lado del tablero
- Número de casillas marcadas
- Número de casillas tapadas
- Posición del cursor
- Datos de las CASILLAS

5.3 Módulo: CASILLA

5.3.1 *Tipo*: Tipo abstracto de datos

5.3.2 *Objetivo*

Este módulo define el tipo abstracto de datos para una casilla del tablero.

5.3.3 *Función*

Este módulo encapsula todas las operaciones posibles que se pueden realizar con cualquier casilla del tablero y define la estructura de datos asociada a una casilla. Las operaciones previstas serán las siguientes:

- Iniciar
- Cambiar marca
- Poner mina
- Incrementar vecinas
- Destapar
- Pintar

5.3.4 *Subordinados:* Pantalla

5.3.5 *Dependencias:* Tablero

5.3.6 *Interfases:* Para cada operación:

Operación: Iniciar

Entrada:

Salida:

Operación: Cambiar marca

Entrada:

Salida: Variación de marca: Puesta, Quitada, Imposible por destapada

Operación: Poner mina

Entrada:

Salida:

Operación: Incrementar vecinas

Entrada:

Salida:

Operación: Destapar

Entrada:

Salida: SI/NO destapada, SI/NO mina, Número de vecinas

Operación: Pintar

Entrada:

Salida: Presentación en pantalla de la casilla

5.3.7 *Recursos:* Ninguno

5.3.8 *Referencias:* Ninguna

5.3.9 *Proceso:* Para cada operación:

Operación: Iniciar

Proceso:

Poner casilla sin mina

Poner casilla sin marca

Poner casilla no destapada

Poner número de vecinas a cero

Operación: Cambiar marca

Proceso:

SI no destapada ENTONCES cambiar marca FIN-SI

Devolver variación de marca

208 Introducción a la Ingeniería de Software

Operación: Poner mina

Proceso:

Poner casilla con mina

Operación: Incrementar vecinas

Proceso:

Incrementar en uno el número de vecinas

Operación: Destapar

Proceso:

SI no marcada & no destapada ENTONCES destapar FIN-SI

Devolver resultados

Operación: Pintar

Proceso:

Pintar casilla

5.3.10 *Datos*

Los atributos de una casilla son los siguientes:

- SI/NO con mina
- SI/NO destapada
- SI/NO marcada
- Número de vecinas

5.4 Módulo: CRONO

5.4.1 *Tipo:* Dato encapsulado

5.4.2 *Objetivo*

Este módulo es el encargado de proporcionar todas las operaciones necesarias para manejar el cronómetro del juego.

5.4.3 *Función*

Este módulo encapsula todas las operaciones que se pueden realizar con el cronómetro en las distintas fases del juego. Las operaciones previstas son las siguientes:

- Iniciar
- Parar
- Actualizar

5.4.4 *Subordinados:* Pantalla, Reloj

5.4.5 *Dependencias:* Minas

5.4.6 *Interfases:* Para cada operación:

Operación: Iniciar

Entrada:

Salida:

Operación: Parar

Entrada:

Salida: Valor final del cronómetro

Operación: Actualizar

Entrada:

Salida:

5.4.7 *Recursos*: Ninguno

5.4.8 *Referencias*: Ninguna

5.4.9 *Proceso*: Para cada operación:

Operación: Iniciar

Proceso:

Poner segundos a cero

Operación: Parar

Proceso:

Devolver segundos

Poner segundos a cero

Operación: Actualizar

Proceso:

Leer reloj del computador

SI Tiempo transcurrido = 1 segundo ENTONCES

 Incrementar segundos

 Actualizar segundos en pantalla

FIN-SI

5.4.10 *Datos*

Los atributos del cronómetro son los siguientes:

- Segundos

5.5 Módulo: AYUDA

5.5.1 *Tipo*: Abstracción funcional (procedimiento)

5.5.2 *Objetivo*

Este módulo es el encargado de proporcionar la información de ayuda al jugador.

5.5.3 *Función*

La función de este módulo es presentar en pantalla la información de ayuda al jugador.

5.5.4 *Subordinados*: Pantalla

5.5.5 *Dependencias*: Minas

210 Introducción a la Ingeniería de Software

5.5.6 *Interfases*

Operación: Presentar ayuda

Entrada:

Salida: Presenta en pantalla texto de ayuda

5.5.7 *Recursos*: Ninguno

5.5.8 *Referencias*: Ninguna

5.5.9 *Proceso*

Operación: Presentar ayuda

Proceso:

Presentar en pantalla el texto de ayuda

5.5.10 *Datos*

Texto a presentar, codificado en la propia función.

5.6 Módulo: RESULTADOS

5.6.1 *Tipo*: Dato encapsulado

5.6.2 *Objetivo*

Este módulo es el encargado de gestionar la tabla de los mejores resultados obtenidos por los distintos jugadores.

5.6.3 *Función*

Las operaciones sobre la tabla de resultados son las siguientes:

- Borrar resultados de un nivel de dificultad
- Comprobar si hay mejor resultado en un nivel de dificultad
- Grabar resultado en un nivel de dificultad
- Presentar mejores resultados de un nivel de dificultad

5.6.4 *Subordinados*: Pantalla, Fichero

5.6.5 *Dependencias*: Minas

5.6.6 *Interfases*: Por cada operación:

Operación: Borrar

Entrada: Nivel de dificultad

Salida:

Operación: Comprobar

Entrada: Nivel de dificultad, Segundos

Salida: SI/NO mejor resultado, posición ocupada

Operación: Grabar

Entrada: Texto a grabar, posición ocupada

Salida:

Operación: Presentar

Entrada: Nivel de dificultad

Salida: Presentación en pantalla de mejores resultados del nivel

5.6.7 Recursos:

Un fichero en disco para almacenar la tabla de resultados.

5.6.8 Referencias: Ninguna

5.6.9 Proceso: Por cada operación:

Operación: Borrar

Proceso:

Inicializar tabla de resultados

Grabar en el fichero en disco

Operación: Comprobar

Proceso:

Buscar el orden que ocupa el resultado

Devolver SI/NO es mejor y la posición que debe ocupar el resultado

Operación: Grabar

Proceso:

Desplazar los resultados peores

Copiar el texto a grabar en la posición

Grabar en el fichero en disco

Operación: Presentar

Proceso:

Presentar en pantalla la tabla de mejores resultados del nivel de dificultad

5.6.10 Datos

Tabla con los registros ordenados por nivel de dificultad y segundos invertidos en orden creciente.
Cada registro deberá tener:

- Nivel de dificultad
- Resultado en segundos invertidos
- Texto grabado por el jugador (Nombre, etc.)

5.7 Módulo: PANTALLA

5.7.1 Tipo: Dato encapsulado

212 Introducción a la Ingeniería de Software

5.7.2 *Objetivo*

Este módulo es el encargado de proporcionar todas las operaciones necesarias para el manejo de la pantalla e independizar al resto del sistema de un compilador o sistema operativo concretos.

5.7.3 *Función*

Este módulo encapsula todas las operaciones que se pueden necesitar de la pantalla. Las operaciones previstas son las siguientes:

- Limpiar
- Situar cursor
- Cambio video inverso/normal
- Salto de línea
- Escribir ristra de texto
- Escribir número entero
- Escribir carácter

5.7.4 *Subordinados:* Módulos predefinidos del compilador

5.7.5 *Dependencias:* Minas, Tablero, Casilla, Crono, Ayuda, Resultados

5.7.6 *Interfases:* Por cada operación:

Operación: Limpiar

Entrada:

Salida: Limpiar la pantalla por completo

Operación: Situar cursor

Entrada: Posición del cursor

Salida: Presentar el cursor en la posición indicada de la pantalla

Operación: Cambiar vídeo

Entrada: Normal/Inverso

Salida: Cambiar el vídeo de la posición en la que está el cursor

Operación: Salto de línea

Entrada:

Salida: Salta el cursor a la primera posición de la siguiente línea

Operación: Escribir ristra

Entrada: Ristra de texto a escribir

Salida: Presenta en pantalla el texto a partir de la posición del cursor

Operación: Escribir entero

Entrada: Entero a escribir, número de espacios de formato

Salida: Presenta en pantalla el entero en los espacios indicados a partir del cursor

Operación: Escribir carácter

Entrada: Carácter a escribir

Salida: Presenta en pantalla el carácter en la posición del cursor

5.8 Módulo: TECLADO

5.8.1 *Tipo:* Dato encapsulado

5.8.2 *Objetivo*

Este módulo es el encargado de proporcionar todas las operaciones necesarias para el manejo del teclado e independizar al resto del sistema de un compilador o sistema operativo concretos.

5.8.3 *Función*

Este módulo encapsula todas las operaciones que se pueden necesitar del teclado. Las operaciones previstas son las siguientes:

- Leer tecla con espera
- Leer tecla sin espera
- Leer ristra con espera

5.8.4 *Subordinados:* Módulos predefinidos del compilador

5.8.5 *Dependencias:* Minas

5.8.6 *Interfases:* Por cada operación:

Operación: Leer tecla con espera

Entrada:

Salida: Carácter ASCII de la tecla pulsada

Operación: Leer tecla sin espera

Entrada: Carácter ASCII testigo

Salida: Carácter ASCII testigo o de la tecla pulsada

Operación: Leer ristra con espera

Entrada:

Salida: Ristra leída

5.9 Módulo: RELOJ

5.9.1 *Tipo:* Dato encapsulado

5.9.2 *Objetivo*

Este módulo es el encargado de proporcionar todas las operaciones necesarias para el manejo del reloj e independizar al resto del sistema de un compilador o sistema operativo concretos.

5.9.3 *Función*

Este módulo encapsula la operación de lectura de reloj siguiente:

- Leer segundos

5.9.4 *Subordinados:* Módulos predefinidos del compilador

5.9.5 *Dependencias:* Crono

5.9.6 *Interfases:* Por cada operación:

Operación: Leer segundos

Entrada:

Salida: Segundos leídos en el reloj del computador

214 Introducción a la Ingeniería de Software

5.10 Módulo: AZAR

5.10.1 *Tipo:* Dato encapsulado

5.10.2 *Objetivo*

Este módulo es el encargado de proporcionar todas las operaciones necesarias para generar números aleatorios e independizar al resto del sistema de un compilador o sistema operativo concretos.

5.10.3 *Función*

Este módulo encapsula las operaciones para generar números aleatorios siguientes:

- Iniciar semilla
- Obtener un nuevo número

5.10.4 *Subordinados:* Módulos predefinidos del compilador

5.10.5 *Dependencias:* Tablero

5.10.6 *Interfases:* Por cada operación:

Operación: Iniciar semilla

Entrada:

Salida:

Operación: Obtener un nuevo número

Entrada: Rango de validez del número

Salida: Nuevo número

5.10.7 *Datos*

El atributo es el siguiente:

- Número anterior

5.11 Módulo: FICHERO

5.11.1 *Tipo:* Tipo abstracto de datos

5.11.2 *Objetivo*

Este módulo es el encargado de proporcionar todas las operaciones necesarias para el manejo de ficheros e independizar al resto del sistema de un compilador o sistema operativo concretos.

5.11.3 *Función*

Este módulo encapsula todas las operaciones que se pueden necesitar para el manejo de ficheros. Las operaciones previstas son las siguientes:

- Abrir
- Cerrar
- Leer tabla de resultados

- Escribir tabla de resultados

5.11.4 *Subordinados*: Módulos predefinidos del compilador

5.11.5 *Dependencias*: Resultados

5.11.6 *Interfases*: Por cada operación:

Operación: Abrir

Entrada: Nombre del fichero

Salida: Identificador interno del fichero

Operación: Cerrar

Entrada: Identificador interno del fichero

Salida:

Operación: Leer tabla de resultados

Entrada: Identificador interno del fichero

Salida: Tabla de resultados

Operación: Escribir tabla de resultados

Entrada: Identificador interno del fichero, tabla de resultados

Salida:

6. VIABILIDAD Y RECURSOS ESTIMADOS

El programa puede ejecutarse en una máquina tipo PC de gama baja. La configuración mínima estimada es:

- Procesador: 80486
- Memoria RAM: 512 Kb
- Pantalla: Modo texto de 25 x 80 caracteres, monocromo o color
- Disquete o disco duro: 360 Kb

7. MATRIZ REQUISITOS/COMPONENTES

Se recoge en la figura M.4. En ella se pueden observar requisitos no ligados a ningún componente. Son de dos clases. Los marcados con asterisco (*) son requisitos que no se cumplen. Los demás son requisitos no funcionales que se satisfacen con otros elementos del sistema distintos de los módulos establecidos en el diseño.

216 Introducción a la Ingeniería de Software

		MÓDULOS					
		MINAS	TABLERO	CRONO	AYUDA	RESULTADOS	CASILLA
REQUISITOS							
R.1.1	X	X	X	X	X	X	X
R.1.1.1	X	-	-	-	-	-	-
R.1.1.2	X	X	X	-	-	-	-
R.1.1.3	-	-	X	-	-	-	-
R.1.1.4	X	-	-	-	-	-	X
R.1.1.5	-	X	-	-	-	-	-
R.1.1.6	-	-	-	-	-	-	X
R.1.1.7	-	X	-	-	-	-	-
R.1.1.8	-	-	-	-	-	-	X
R.1.1.9	-	-	-	-	-	-	X
R.1.1.10	X	-	X	-	-	-	-
R.1.2	-	-	X	-	-	-	X
R.1.3	X	-	-	-	-	-	-
R.1.4	X	-	-	-	-	X	-
R.1.5	X	-	-	-	-	-	-
R.1.6	-	-	-	-	-	X	-
R.1.7	X	-	-	-	-	-	-
R.1.8	-	X	-	-	-	-	-
* R.1.9	-	-	-	-	-	-	-
R.1.10	X	-	-	-	X	-	-
R.2.1	-	-	X	-	-	-	-
R.2.2	X	-	-	-	-	-	-
R.2.3	-	X	-	-	-	-	-
R.3.1	-	-	-	-	-	-	-
R.4.1	-	X	-	-	-	-	-
R.4.2	X	-	-	-	-	-	-
R.4.3	X	-	-	-	-	-	-
R.4.4	X	-	-	-	-	-	-
R.4.5	X	-	-	-	-	-	-
R.5.1	-	-	-	-	-	-	-
* R.6.1	-	-	-	-	-	-	-
R.8.1	-	-	-	-	X	-	-
* R.9.1	-	-	-	-	-	-	-

NOTA: Los requisitos marcados con (*) no se cumplen

Figura M.4. Matriz REQUISITOS-COMponentes

4.8.2 Ejemplo: Sistema de gestión de biblioteca

En esta sección se presenta el documento de diseño de arquitectura (ADD) del sistema de gestión de biblioteca usado anteriormente como ejemplo, y cuyo documento de especificación de requisitos (SRD) se presentó en el Tema 2.

DOCUMENTO DE DISEÑO DEL SOFTWARE (ADD)

Proyecto: SISTEMA DE GESTIÓN DE BIBLIOTECA
Autor(es): M. Collado y J.F. Estivariz
Fecha: Noviembre 1999
Documento: BIBLIO-ADD-99

CONTENIDO

1. INTRODUCCIÓN

- 1.1 Objetivo
- 1.2 Ámbito
- 1.3 Definiciones, siglas y abreviaturas
- 1.4 Referencias

2. PANORÁMICA DEL SISTEMA

3. CONTEXTO DEL SISTEMA

4. DISEÑO DEL SISTEMA

- 4.1 Metodología de diseño de alto nivel
- 4.2 Descomposición del sistema

5. DESCRIPCIÓN DE COMPONENTES

6. VIABILIDAD Y RECURSOS ESTIMADOS

7. MATRIZ REQUISITOS/COMPONENTES

218 Introducción a la Ingeniería de Software

1. INTRODUCCIÓN

1.1 Objetivo

El objetivo del sistema es facilitar la gestión de una biblioteca mediana o pequeña, en lo referente a la atención directa a los usuarios. Esto incluye, fundamentalmente, el préstamo de libros, así como la consulta bibliográfica.

1.2 Ámbito

El sistema a desarrollar consistirá en un único programa que realizará todas las funciones necesarias. En particular, deberá facilitar las siguientes:

- Gestión del préstamo y devolución de libros
- Consulta bibliográfica por título, autor o materia

El sistema no ha de soportar, sin embargo, la gestión económica de la biblioteca, ni el control de adquisición de libros, ni otras funciones no relacionadas directamente con la atención a los usuarios.

El sistema facilitará la atención al público por parte de un único bibliotecario, que podrá atender a todas las funciones.

1.3 Definiciones, siglas y abreviaturas

Ninguna.

1.4 Referencias

BIBLIO-SRD-99: DOCUMENTO DE REQUISITOS DEL SOFTWARE del SISTEMA DE GESTIÓN DE BIBLIOTECA.

2. PANORÁMICA DEL SISTEMA

Se recoge aquí un resumen de la descripción del sistema ya incluída en el documento de especificación de requisitos BIBLIO-SRD-99

2.1 Objetivo y funciones

La biblioteca dispone de una colección de libros a disposición del público. Cualquier persona puede consultar los libros en la sala de lectura, sin necesidad de realizar ningún registro de esta actividad.

Los libros pueden ser también sacados en préstamo por un plazo limitado, fijado por la organización de la biblioteca, y siempre el mismo. En este caso es necesario mantener anotados los libros prestados y qué lector los tiene en su poder. Para que un lector pueda sacar un libro en préstamo debe disponer previamente de una ficha de lector con sus datos, y en particular con indicación de su teléfono, para facilitar la reclamación del libro en caso de demora en su devolución.

Un lector puede tener a la vez varios libros en préstamo, hasta un máximo fijado por la organización de la biblioteca, igual para todos los usuarios.

Los usuarios deben disponer de algunas facilidades para localizar el libro que desean, tanto si es para consultarlo en la sala como si es para sacarlo en préstamo. Se considera razonable poder localizar un libro por su autor, su título (o parte de él) o por la materia de que trata. Para la búsqueda por materias, existirá una lista de materias establecida por el bibliotecario. Cada libro podrá tratar de una o varias materias.

El objetivo del sistema es facilitar las funciones más directamente relacionadas con la atención directa a los usuarios de la biblioteca. Las funciones principales serán:

- Anotar los préstamos y devoluciones
- Indicar los préstamos que hayan sobrepasado el plazo establecido
- Búsquedas por autor, título o materia

Como complemento serán necesarias otras funciones, en concreto:

- Mantener un registro de los libros
- Mantener un registro de los usuarios (lectores)

2.2 Descripción funcional

El sistema de gestión de biblioteca será operado por una sola persona, que dispondrá de un terminal con pantalla y teclado. También existirá una impresora por la que podrán obtenerse listados y fichas de los libros y de los lectores.

La operación del sistema se hará mediante un sistema de menús para seleccionar la operación deseada, y con formularios en pantalla para la entrada y presentación de datos. Las funciones a realizar se pueden organizar en varios grupos principales, que se describen a continuación.

2.2.1 Gestión de libros

Estas funciones permiten mantener actualizado el registro (fichero) de libros existentes en la biblioteca. Las funciones de gestión de libros son las siguientes:

- Función 1.1 Alta de libro: registra un nuevo libro
- Función 1.2 Baja de libro: marca un libro como dado de baja
- Función 1.3 Anular baja de libro: suprime la marca de baja
- Función 1.4 Actualizar libro: modifica los datos del libro
- Función 1.5 Listar libros: lista todos los libros registrados
- Función 1.6 Compactar registro de libros: elimina los libros dados de baja
- Función 1.7 Actualizar lista de materias: actualiza la lista de materias consideradas

2.2.2 Gestión de lectores

Estas funciones permiten mantener actualizado el registro (fichero) de usuarios (lectores). Las funciones de gestión de lectores son las siguientes:

- Función 2.1 Alta de lector: registra un nuevo usuario (lector)
- Función 2.2 Baja de lector: marca un lector como dado de baja
- Función 2.3 Anular baja de lector: suprime la marca de baja
- Función 2.4 Actualizar lector: modifica los datos del lector
- Función 2.5 Listar lectores: lista todos los lectores registrados
- Función 2.6 Compactar registro de lectores: elimina los lectores dados de baja
- Función 2.7 Buscar lector: localiza un lector por nombre o apellido

2.2.3 Gestión de préstamos

Estas funciones permiten tener anotados los libros en préstamo, y conocer en un momento dado los que han sido retenidos más tiempo del permitido. Las funciones de gestión de préstamos son las siguientes:

- Función 3.1 Anotar préstamo: anota los datos del préstamo
- Función 3.2 Anotar devolución: elimina los datos del préstamo

220 Introducción a la Ingeniería de Software

Función 3.3 Lista de morosos: lista todos los préstamos no devueltos en el plazo fijado
Función 3.4 Préstamos de lector: lista los libros que tiene en préstamo un lector dado

2.2.4 Búsquedas

Estas funciones permiten localizar libros por autor, título o materia. Las funciones de búsqueda son las siguientes:

Función 4.1 Buscar por autor: localiza los libros de un autor dado
Función 4.2 Buscar por título: localiza los libros cuyo título contiene un texto dado
Función 4.3 Buscar por materia: localiza los libros que tratan de una materia dada

2.3 Modelo de datos

El modelo conceptual se describe en el diagrama Entidad-Relación, revisado, que aparece en la Figura B.1.

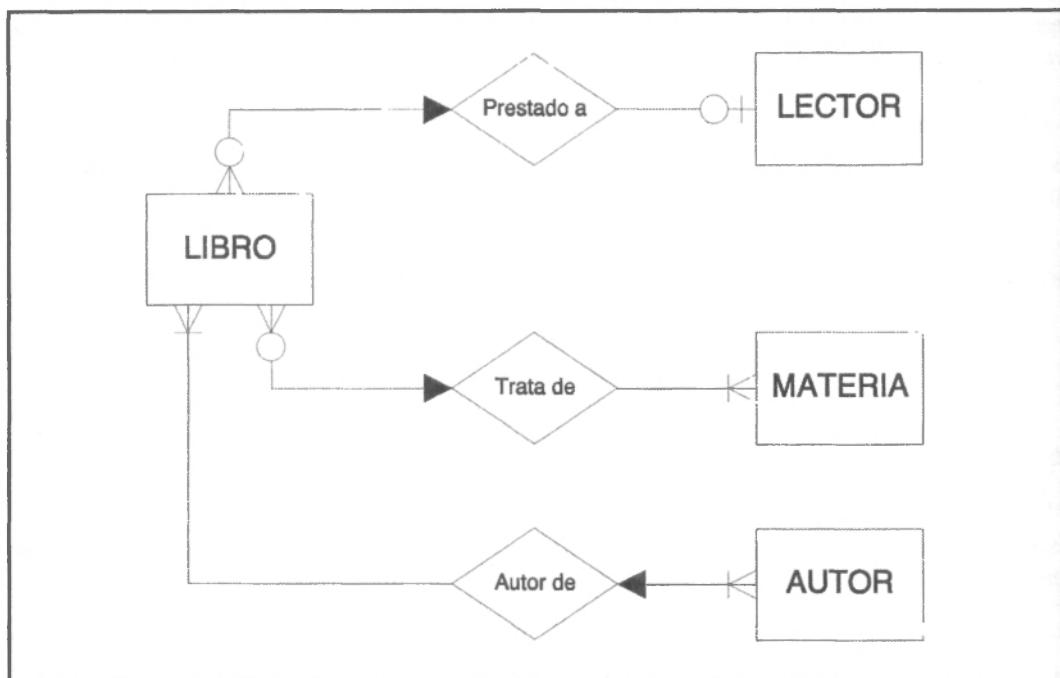


Figura B.1. Modelo de datos ENTIDAD-RELACIÓN

Este modelo amplía el descrito en el documento de requisitos BIBLIO-SRD-99. Las entidades de datos principales y las relaciones entre ellas son las siguientes:

- Entidad Libro: Contiene los datos de identificación del libro
- Entidad Autor: Contiene el nombre y apellidos del autor
- Entidad Lector: Contiene los datos personales del lector
- Entidad Materia: Contiene el término clave que identifica la materia
- Relación Autor-de: Enlaza un libro con su(s) autor(es)
- Relación Prestado-a: Enlaza un libro con el lector que lo ha sacado en préstamo. Contiene la fecha del préstamo
- Relación Trata-de: Enlaza un libro con la(s) materia(s) de la(s) que trata

Los esquemas físicos correspondientes a este modelo conceptual se describen en el apartado 5.1 *Módulo: BASEDATOS*.

La Ficha de Libro es una visión externa de los datos de un libro, para ser presentada en pantalla, o impresa como ficha o como líneas de detalle en un listado, y que incluye:

- Número de referencia
- Título
- Autor(es)
- Colección
- Editorial
- Materia(s)

La Ficha de Lector es una visión externa que incluye los datos de un lector almacenados en la tabla LECTORES, para ser presentada en pantalla o impresa como ficha o como líneas de detalle en un listado.

3. CONTEXTO DEL SISTEMA

No existe conexión con otros sistemas.

4. DISEÑO DEL SISTEMA

4.1 Metodología de diseño de alto nivel

Se utiliza la metodología de DISEÑO ESTRUCTURADO, basada en la descomposición funcional del sistema.

4.2 Descomposición del sistema

La estructura modular del sistema aparece representada en la Figura B.2. Los módulos identificados son los siguientes:

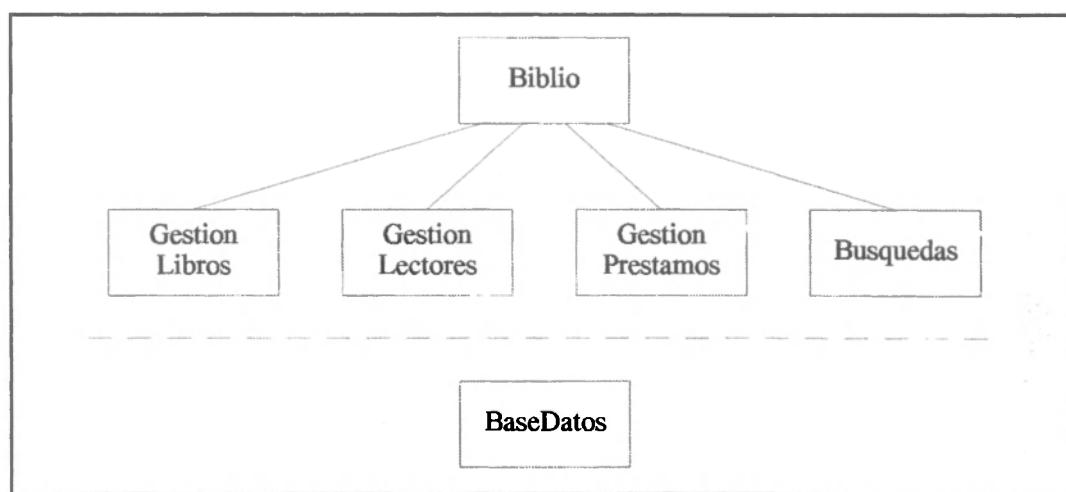


Figura B.2. Arquitectura del sistema

222 Introducción a la Ingeniería de Software

- **BIBLIO:** Es el programa principal. Realiza el diálogo con el operador, en lo referente a la selección de función.
- **GESTIONLIBROS, GESTIONLECTORES, GESTIONPRESTAMOS, BUSQUEDAS:** Módulos que realizan las funciones principales del sistema.
- **BASEDATOS:** Módulo que contiene la base de datos del sistema.

El módulo BaseDatos está realizado físicamente por el SGBD comercial que se utiliza.

5. DESCRIPCIÓN DE COMPONENTES

5.1 Módulo: BASEDATOS

5.1.1 *Tipo:* Base de datos relacional

5.1.2 *Objetivo:* Este módulo contiene la base de datos relacional que almacena la información persistente del sistema.

5.1.3 *Función*

Almacenar los datos que se indican.

5.1.4 *Subordinados:* Ninguno

5.1.5 *Dependencias:* GESTIONLIBROS, GESTIONLECTORES, GESTIONPRESTAMOS, BUSQUEDAS

5.1.6 *Interfases*

El modelo físico de datos es el siguiente:

- Tabla: LIBROS

Campo	Tipo	Long.	Indice	Descripción
NumRef	Num.	4	Sí	Número de referencia del libro
Titulo	Texto	60	No	Título del libro
Coleccion	Texto	20	No	Colección a la que pertenece, opcional
Editorial	Texto	30	No	Nombre de la editorial
Lector	Num.	4	Sí	Número de referencia del lector que lo ha sacado en préstamo
FechaPres	Fecha		Sí	Fecha en que se prestó

- Tabla: LECTORES

Campo	Tipo	Long.	Indice	Descripción
NumRef	Num.	4	Sí	Número de referencia del lector
Apellidos	Texto	40	No	Apellidos del lector
Nombre	Texto	20	No	Nombre del lector
Domicilio	Texto	40	No	Domicilio del lector
Telefono	Num.	7	No	Número de teléfono del lector

- Tabla: MATERIAS

<u>Campo</u>	<u>Tipo</u>	<u>Long.</u>	<u>Indice</u>	<u>Descripción</u>
NumRef	Num.	3	Sí	Número de referencia de la materia
Nombre	Texto	20	No	Nombre de la materia (término o palabra clave que la identifica)

- Tabla: AUTORES

<u>Campo</u>	<u>Tipo</u>	<u>Long.</u>	<u>Indice</u>	<u>Descripción</u>
NumRef	Num.	4	Sí	Número de referencia del autor
Nombre	Texto	30	No	Nombre y apellidos del autor

- Tabla: AUTOR-DE

<u>Campo</u>	<u>Tipo</u>	<u>Long.</u>	<u>Indice</u>	<u>Descripción</u>
Libro	Num.	4	Sí	Número de referencia del libro
Autor	Num.	4	No	Número de referencia del autor

- Tabla: TRATA-DE

<u>Campo</u>	<u>Tipo</u>	<u>Long.</u>	<u>Indice</u>	<u>Descripción</u>
Libro	Num.	4	Sí	Número de referencia del libro
Materia	Num.	3	No	Número de referencia de la materia

5.1.7 *Recursos:* Ninguno

5.1.8 *Referencias:* Ninguna

5.1.9 *Proceso:* Ninguno

5.1.10 *Datos (ver Interfases)*

5.2 Módulo: BIBLIO

5.2.1 *Tipo:* Abstracción funcional (programa principal)

5.2.2 *Objetivo:* Este es el programa principal de la aplicación

5.2.3 *Función*

Este módulo se encarga de realizar el diálogo con el usuario en lo referente a la selección de la función deseada en cada momento. También debe realizar las operaciones oportunas al comienzo y al final de una sesión de trabajo.

5.2.4 *Subordinados:* GESTIONLIBROS, GESTIONLECTORES, GESTIONPRESTAMOS, BUSQUEDAS

5.2.5 *Dependencias:* Ninguna

5.2.6 *Interfases:* No aplicable

5.2.7 *Recursos:* Ninguno

224 Introducción a la Ingeniería de Software

5.2.8 Referencias: Ninguna

5.2.9 Proceso

- Iniciar la sesión
- REPETIR
 - Presentar menú principal
 - Elegir opción
 - CASO opción DE
 - Gestión de libros:
 - Presentar menú de gestión de libros
 - Elegir opción
 - CASO opción DE
 - Alta de libro: Realizar alta
 - Baja de libro: Realizar baja

.....
FIN-CASO

Gestión de lectores:

.....
Gestión de préstamos:

.....
Búsquedas:

.....
FIN-CASO

HASTA fin de sesión

- Terminar la sesión

5.2.10 Datos (ver BASEDATOS)

5.3 Módulo: GESTIONLIBROS

5.3.1 Tipo: Abstracción funcional (colección de funciones)

5.3.2 Objetivo: Realizar las funciones de mantenimiento del registro de libros disponibles en la biblioteca.

5.3.3 Función: Las funciones realizadas por este módulo son las siguientes:

Función ALTA LIBRO: registra un nuevo libro

Entrada:

Salida:

Usa: MATERIAS

Actualiza: LIBROS, AUTORES, AUTOR-DE, TRATA-DE

Efecto: Compone una nueva ficha de libro asignando código automáticamente y leyendo los datos del libro por pantalla (las materias se seleccionarán de entre la lista de materias registradas). Registra la ficha en el fichero de libros, y además la imprime.

Excepciones:

Proceso:

- Asignar número de referencia al nuevo libro
- Editar la ficha del libro mediante un formulario en pantalla
 - El autor o autores se seleccionan de la lista de autores, añadiendo nuevos nombres o editando los existentes, si es necesario
 - La materia o materias se seleccionan de la lista de materias, pero no se pueden añadir o modificar las materias existentes
- Pedir confirmación
- SI se confirma ENTONCES
 - Registrar la ficha del libro en las tablas de LIBROS, AUTOR-DE y TRATA-DE
 - Imprimir la ficha del libro
- SI-NO
 - Anular la asignación de nuevo número de referencia y las modificaciones en las tablas
- FIN-SI

Función BAJALIBRO: marca un libro como dado de baja

Entrada:

Salida:

Usa: AUTORES, MATERIAS, AUTOR-DE, TRATA-DE

Actualiza: LIBROS

Efecto: Lee el código del libro por pantalla, y marca la ficha de ese libro en el fichero de libros como dada de baja.

Excepciones: Si no existe el libro con el código indicado, o ya estaba dado de baja, da un aviso.

Proceso:

- Lee el número del libro por pantalla
- Busca el libro en la tabla LIBROS
- SI el libro no existe ENTONCES
 - Da mensaje de que no existe
- SI NO
 - Presenta la ficha del libro en pantalla
 - Pide confirmación de la baja
 - SI se confirma ENTONCES
 - Marca el libro como borrado en la tabla LIBROS
- FIN-SI
- FIN-SI

Función ANULARBAJALIBRO: suprime la marca de baja

Entrada:

Salida:

Usa: AUTORES, MATERIAS, AUTOR-DE, TRATA-DE

226 Introducción a la Ingeniería de Software

Actualiza: LIBROS

Efecto: Lee el código del libro por pantalla, y anula la marca de dado de baja en la ficha de ese libro en el fichero de libros.

Excepciones: Si no existe el libro con el código indicado, o no estaba dado de baja, da un aviso.

Proceso:

- Lee el número del libro por pantalla
- Busca el libro en la tabla LIBROS
- SI el libro no existe o no está marcado como borrado ENTONCES
 - Da el mensaje de aviso apropiado

SI-NO

- Presenta la ficha del libro en pantalla
- Pide confirmación de la anulación de la baja
- SI se confirma ENTONCES
 - Anula la marca de borrado del libro en la tabla LIBROS

FIN-SI

FIN-SI

Función EDITARLIBRO: modifica los datos del libro

Entrada:

Salida:

Usa: MATERIAS

Actualiza: LIBROS, AUTORES, AUTOR-DE, TRATA-DE

Efecto: Lee el código del libro por pantalla, localiza la ficha de ese libro, y actualiza los datos de ese libro, también por pantalla. Registra la ficha actualizada en el fichero de libros, y la imprime.

Excepciones: Si no existe el libro con el código indicado, da un aviso.

Proceso:

- Lee el número de referencia del libro por pantalla
 - Busca el libro en la tabla LIBROS
 - SI el libro no existe o está marcado como borrado ENTONCES
 - Da el mensaje de aviso apropiado
- SI NO
- Editar la ficha del libro mediante un formulario en pantalla
 - El autor o autores se seleccionan de la lista de autores, añadiendo nuevos nombres o editando los existentes, si es necesario
 - La materia o materias se seleccionan de la lista de materias, pero no se pueden añadir o modificar las materias existentes
 - Pedir confirmación
 - SI se confirma ENTONCES
 - Registrar la ficha modificada del libro en las tablas de LIBROS, AUTOR-DE y TRATA-DE
 - Imprimir la ficha modificada del libro
- SI-NO
- Anular las modificaciones en las tablas
- FIN-SI
- FIN-SI

Función LISTARLIBROS: lista todos los libros registrados

Entrada:

Salida:

Usa: LIBROS, AUTORES, MATERIAS, AUTOR-DE, TRATA-DE

Actualiza:

Efecto: Imprime un listado con todos los datos de todos los libros, incluso los que estén dados de baja, indicando si están prestados.

Excepciones:

Proceso:

- PARA-CADA libro registrado HACER

- Imprimir una o varias líneas del listado, con los datos de la ficha del libro y la indicación de si está prestado o dado de baja
(el listado se hará paginando en la forma habitual)

FIN-PARA

Función COMPACTARLIBROS: elimina los libros dados de baja

Entrada:

Salida:

Usa:

Actualiza: LIBROS, AUTORES, AUTOR-DE, TRATA-DE

Efecto: Suprime del fichero de libros las fichas de libros que estén dados de baja, liberando el espacio que ocupaban.

Excepciones:

Proceso:

- PARA-CADA libro registrado HACER

- SI está marcado como borrado ENTONCES

- Borrar las líneas que correspondan de las tablas de AUTOR-DE y TRATA-DE

FIN-SI

FIN-PARA

- Compactar las tablas de LIBROS, AUTOR-DE y TRATA-DE

- PARA-CADA autor HACER

- SI no queda ningún libro de ese autor ENTONCES

- Borrar ese autor

FIN-SI

FIN-PARA

- Compactar la tabla de AUTORES

Función EDITARMATERIAS: actualiza la lista de materias

Entrada:

Salida:

Usa: TRATA-DE

Actualiza: MATERIAS

228 Introducción a la Ingeniería de Software

Efecto: Se edita por pantalla la lista de materias a considerar, pudiendo añadir, suprimir materias, o modificar sus nombres.

Excepciones: Si se intenta suprimir una materia habiendo libros registrados que tratan de ella, se pedirá confirmación especial. Si se fuerza la eliminación, se suprimirá también la referencia a esa materia de los libros que traten de ella.

Proceso:

- Editar la lista de materias en pantalla, permitiendo
 - Añadir materias
 - Modificar el nombre de una materia
 - Suprimir materia:
 - Si hay libros que tratan de esa materia, pedir confirmación
 - SI se confirma la supresión ENTONCES
 - Borrar las entradas en TRATA-DE correspondientes a esa materia

FIN-SI

5.3.4 *Subordinados:* BASEDATOS

5.3.5 *Dependencias:* BIBLIO

5.3.6 *Interfases* (ver *Función*)

5.3.7 *Recursos:* Ninguno

5.3.8 *Referencias:* Ninguna

5.3.9 *Proceso* (ver *Función*)

5.3.10 *Datos* (ver módulo BASEDATOS)

5.4 Módulo: GESTIONLECTORES

.....

5.5 Módulo: GESTIONPRESTAMOS

.....

5.6 Módulo: BUSQUEDAS

.....

6. VIABILIDAD Y RECURSOS ESTIMADOS

Esta aplicación puede ejecutarse en una máquina tipo PC de gama baja. Los recursos mínimos estimados son los siguientes:

- Procesador: 80486
- Sistema Operativo: DOS
- Memoria RAM: 2 Mb
- Pantalla: Modo texto de 25 x 80 caracteres, monocromo o color
- Disco duro:
 - 4 Mb para el SGBD
 - 1 Mb para los programas de la aplicación
 - 4 Mb para los datos de la aplicación (10.000 libros y 10.000 lectores)

7. MATRIZ REQUISITOS/COMPONENTES

Se recoge en la Figura B.3. En ella pueden observarse requisitos no ligados a ninguna componente. Son de dos clases. Los marcados con asterisco (*) son requisitos que no se cumplen. Los demás son requisitos no funcionales que se satisfacen con otros elementos del sistema distintos de los módulos establecidos en el diseño.

		MÓDULOS					
		BASEDATOS	BIBLIO	GESTIONLIBROS	GESTIONLECTORES	GESTIONPRESTAMOS	BUSQUEDAS
REQUISITOS							
R.1.1	X	-	-	-	-	-	-
R.1.2	-	X	-	-	-	-	-
F.1.1	-	-	X	-	-	-	-
F.1.2	-	-	X	-	-	-	-
F.1.3	-	-	X	-	-	-	-
F.1.4	-	-	X	-	-	-	-
F.1.5	-	-	X	-	-	-	-
F.1.6	-	-	X	-	-	-	-
F.1.7	-	-	X	-	-	-	-
F.2.1	-	-	-	X	-	-	-
...
R.2.1	X	-	-	-	-	-	-
R.2.2	-	-	-	-	-	-	-
R.2.3	-	-	-	-	-	-	-
R.2.4	-	-	-	-	-	-	-
R.4.1	-	X	-	-	-	-	-
R.4.1.1	-	X	-	-	-	-	-
R.4.2	X	-	-	-	-	-	-
* R.4.3	-	-	-	-	-	-	-
* R.4.4	-	-	-	-	-	-	-
R.4.5	-	-	-	-	-	-	-
R.4.6	-	-	-	-	-	-	-
R.4.7	-	X	X	X	X	-	-
R.4.8	-	X	-	-	-	-	-
R.5.1	-	-	-	-	-	-	-
R.6.1	X	-	-	-	-	-	-
R.7.1	-	-	-	-	-	-	-
R.8.1	-	-	-	-	-	-	-
* R.8.2	-	-	-	-	-	-	-
* R.9.1	-	-	-	-	-	-	-
R.9.2	X	-	-	-	-	-	-
R.10.1	-	-	-	-	-	-	-

NOTA: Los requisitos marcados con (*) no se cumplen

Figura B.3. Matriz REQUISITOS-COMPONENTES

Tema 5

Codificación y Pruebas

En este Tema se hace un rápido repaso a las últimas fases del ciclo de vida: codificación, pruebas de unidades, fase de integración y pruebas del sistema; con ellas se materializa el diseño y se obtiene el producto final de cualquier desarrollo. Entre todas estas fases existe una gran interrelación. Las técnicas de prueba de unidades son imprescindibles para depurar la codificación de cada módulo o unidad por separado. Las pruebas del sistema nos aseguran que la integración de todas las unidades se realiza correctamente. Cuando una prueba de unidad o de sistema no se supera correctamente es necesario modificar la codificación y ésta posteriormente tendrá que ser probada de nuevo.

5.1 Codificación del diseño

La fase de codificación constituye el núcleo central de cualquiera de los modelos de desarrollo de software: ciclo de vida, uso de prototipos, modelo en espiral, etc. Hasta la década de los 70, en el desarrollo de software prácticamente todo el trabajo se centraba en la fase de codificación. Actualmente, también sucede lo mismo al desarrollar pequeños sistemas que son realizados enteramente por una única persona en un plazo de tiempo muy breve. Simplificando, se puede decir que las etapas previas de análisis y diseño tienen como misión fundamental la de organizar y traducir los requisitos del *cliente* en unidades o módulos de programa que puedan ser codificados de forma independiente y sencilla. La importancia de la fase de codificación dentro del desarrollo de software es evidente si se tiene en cuenta que en ella se elabora el producto fundamental de todo el desarrollo: los programas fuente.

Un elemento esencial dentro de la codificación es el lenguaje de programación que se emplea. En los próximos apartados de este mismo Tema se repasan el desarrollo histórico de los lenguajes, sus prestaciones básicas y algunos criterios de selección. Existen cientos de lenguajes y cada

234 Introducción a la Ingeniería de Software

uno de ellos tiene sus propias peculiaridades lo que hace que sea más aconsejable su utilización en unos casos que en otros. Por ejemplo, COBOL está pensado para codificar sistemas de información mientras que C, Pascal o Ada son más adecuados para sistemas de control o de cálculo. El uso de un determinado lenguaje limita e impone al programador una determinada forma de trabajo, pero el resultado de la codificación no debería quedar excesivamente mediatizado por el lenguaje utilizado.

El estudio de una determinada metodología de programación queda fuera de los objetivos de este libro. Sin embargo, dada la importancia que los aspectos metodológicos tienen para lograr una codificación de buena calidad se dedicará un apartado de este Tema al repaso de aquellos aspectos que inciden de una manera decisiva en la calidad. Antes de iniciar la fase de codificación es fundamental establecer por escrito cuál será la metodología de programación que se empleará por todos miembros del equipo de trabajo. Normalmente, todas las empresas de desarrollo de software tienen su propia metodología de programación que utilizan en todos sus desarrollos. La metodología empleada tiene tanta importancia como el lenguaje de programación elegido.

En la fase de codificación pueden intervenir un gran número de programadores y durante un tiempo muy largo. Como parte de la metodología de programación y para garantizar la adecuada homogeneidad en la codificación se deben establecer las normas y estilo de codificación. Solamente una disciplina rigurosa en el empleo de las normas establecidas es capaz de asegurar una codificación de calidad. Además un estilo homogéneo hace posible un trabajo coordinado entre los programadores ya que el entendimiento entre todos ellos resulta mucho más fácil. Como ventaja adicional, el empleo de estas normas facilita de forma considerable el mantenimiento posterior y la reusabilidad del software codificado.

Cuando los resultados de las pruebas no sean satisfactorios habrá que realizar cambios en la codificación. Ambos aspectos, codificación y pruebas, están muy relacionados. Por ejemplo, es normal que para depurar un fragmento de código se incluyan en el mismo código ciertos puntos de test que nos informen del paso del programa por dichos puntos. Estos puntos de test forman parte del mismo código a depurar y también pueden ser causa de errores. La codificación se debe estructurar para facilitar su depuración y las modificaciones derivadas de las pruebas. Así, resultará más sencillo y barato localizar las causas de una disfunción y la posterior modificación del fragmento de programa afectado.

5.2 Lenguajes de programación

Los lenguajes de programación son el medio fundamental que tenemos para realizar la codificación. Aunque los lenguajes han evolucionado mucho desde los años 50, todavía suelen estar más próximos a la forma de trabajar de los computadores que a la forma de pensar del ser humano. A pesar de todo, resulta muy interesante conocer la evolución que se ha producido en los lenguajes y observar cómo han mejorado sus prestaciones a lo largo de los últimos treinta años. Con los lenguajes actuales resulta más sencillo obtener un software de calidad, fácil de mantener y con posibilidades reales de reutilización.

El conocimiento de las prestaciones de los lenguajes permite aprovechar mejor sus posibilidades y también salvar sus posibles deficiencias. Evidentemente no existe un único lenguaje ideal para todo y a veces es necesario trabajar con varios lenguajes para las distintas partes de un mismo desarrollo. Por ejemplo, un lenguaje ensamblador para los controladores de dispositivos hardware (teclado, motores, válvulas, etc.), un lenguaje estructurado (Pascal, C, Ada, etc.) para los algoritmos y un lenguaje de cuarta generación para la interfase hombre-máquina.

El desarrollo de los lenguajes ha sido posible gracias a la experiencia acumulada en la codificación de los más diversos proyectos de software. Cuando se logra un consenso global respecto a la necesidad de una nueva prestación esto provoca su incorporación al lenguaje. Este es el caso del lenguaje C++ que es el resultado de incorporar la posibilidad de definir y utilizar objetos en C. Así, en los lenguajes de programación quedan reflejados los avances metodológicos que se producen para mejorar y simplificar el desarrollo de software.

5.3 Desarrollo histórico

Aunque la historia de los computadores es relativamente reciente, son cientos y cientos los lenguajes que han sido diseñados con los más diversos objetivos. La utilidad de la mayoría de ellos ha sido meramente experimental o de investigación. Sólo un número relativamente pequeño se utiliza o ha sido utilizado en el desarrollo de software a escala industrial. El estudio detallado de tan sólo los dos o tres más representativos queda completamente fuera de los objetivos de este libro. Incluso resulta difícil realizar una clasificación coherente en la que cada lenguaje se pueda catalogar indiscutiblemente dentro de un grupo concreto y determinado.

El estudio de la evolución histórica de los lenguajes es una buena manera de adquirir una visión panorámica de los mismos. En todo caso, dentro de esta evolución se suelen distinguir cuatro generaciones de lenguajes que se solapan en el tiempo. Los lenguajes de las nuevas generaciones coexisten con los de las generaciones anteriores hasta que se imponen por completo los de mejores prestaciones.

5.3.1 Primera generación

Los primeros computadores eran máquinas de válvulas tremadamente costosas y los programas que podían ejecutar eran muy pequeños. La programación se hacía introduciendo directamente en su memoria el escaso número de instrucciones del programa en forma de códigos binarios. Para realizar esta labor era esencial conocer la estructura interna del computador, tarea ésta que estaba reservada a un número reducido de personas.

Durante la década de los 50 el número de computadores disponibles aumentó y la tarea de su programación empezó a tener entidad propia. Para simplificar esta labor tan tediosa y disminuir las posibilidades de error se crearon los lenguajes ensambladores. Un lenguaje ensamblador consiste fundamentalmente en asociar a cada instrucción del computador un nemotécnico que recuerde cuál es su función. Estos nemotécnicos constituyen un lenguaje algo más próximo al ser humano que el puro código máquina a base de unos y ceros. Los ensambladores se consideran la primera generación de lenguajes, con un nivel de abstracción muy bajo.

Actualmente, con cada nuevo computador que se diseña, el fabricante debe proporcionar de forma inmediata su correspondiente ensamblador. Por tanto, existen tantos ensambladores como computadores distintos. La programación en ensamblador resulta compleja, da lugar a errores difíciles de detectar, exige que el programador conozca bastante bien la arquitectura del computador y sobre todo se necesita adaptar la solución a las particularidades de cada computador concreto. Sólo está justificada la utilización del ensamblador en aquellos casos en los que no se puede programar con ningún lenguaje de alto nivel. Esta situación es cada día más rara y se da fundamentalmente cuando se requiere una optimización del código para aplicaciones con especificaciones muy críticas de tiempo real. En cualquier caso, sólo se programarán en ensamblador pequeños fragmentos de programa que se podrán insertar, en forma de macros o subrutinas, dentro del programa escrito en un lenguaje de alto nivel.

5.3.2 Segunda generación

El aumento de la capacidad de memoria y disco de los computadores hizo posible abordar programas más grandes y complejos. Cuando estos programas se realizaban enteramente en ensamblador resultaban muy difíciles de depurar, entender y mantener, lo que aumentaba los costes y el tiempo de desarrollo. A finales de la década de los 50 se comenzaron a desarrollar los primeros lenguajes de alto nivel. Su característica más notable era que no dependían de la estructura de un computador concreto y por primera vez se programaba en "alto nivel", de manera simbólica.

Esta segunda generación de lenguajes supone un paso trascendental en la creación de herramientas para el desarrollo de software. Con ellos, se incorporan los primeros elementos realmente abstractos. Por ejemplo, aparecen tipos de datos (numéricos o de caracteres) no soportados directamente por las instrucciones de la máquina. También se puede ignorar en parte la organización interna de la memoria del computador para pasar a trabajar con variables simbólicas de distintos tamaños y estructuras (vectores o matrices) según las necesidades de cada programa. Además se dispone de las primeras estructuras de control para la definición de bucles genéricos o la selección entre varios caminos alternativos de ejecución, etc. Todo esto hizo que estos lenguajes alcanzaran inmediatamente una amplia difusión y que se utilizaran de forma masiva. Todavía hoy se utilizan estos mismos lenguajes o alguno de sus descendientes directos de la tercera generación. Algunos de los lenguajes más representativos de esta generación son FORTRAN, COBOL, ALGOL y BASIC.

FORTRAN (FORmula TRANslator): Es un lenguaje que fue pensado para programar fundamentalmente aplicaciones científicas o técnicas en las que tiene una gran importancia el cálculo numérico. Con el paso de los años se han sucedido sucesivas versiones: FORTRAN-IV, FORTRAN-66, FORTRAN-77, FORTRAN-90 que han ido corrigiendo deficiencias anteriores y se han adaptado a las nuevas propuestas metodológicas incorporadas por otros lenguajes. Desde el punto de vista metodológico, una deficiencia importante que todavía conserva FORTRAN es el manejo casi directo de la memoria mediante la sentencia COMMON.

A pesar de su origen científico, FORTRAN se ha utilizado y se continúa utilizando en el desarrollo de una amplia gama de aplicaciones de todo tipo que incluyen la realización de sistemas de gestión de bases de datos, sistemas en tiempo real, etc.

238 Introducción a la Ingeniería de Software

COBOL: Es el lenguaje con el que están escritos casi todos los sistemas para el procesamiento de información (contabilidad, nóminas, facturación, control de almacén, etc.). Hay que tener en cuenta que este tipo de sistemas supone más del 70% de todo el software que se desarrolla. Aunque actualmente se continúa utilizando COBOL para estos desarrollos, los lenguajes de la cuarta generación están consiguiendo desplazarlo paulatinamente.

ALGOL: Se considera el precursor de muchos de los lenguajes de la tercera generación y especialmente de Pascal y todos sus descendientes, tales como Modula-2 y Ada. Se puede decir que es el primer lenguaje que da una gran importancia a la tipificación de datos. También en este caso existen diversas versiones: ALGOL-60, ALGOL-68, aunque su difusión a nivel industrial o comercial no ha sido tan importante como en el caso de FORTRAN.

BASIC: Fue pensado para la enseñanza de la programación y es de destacar su sencillez y facilidad de aprendizaje. Gracias a los escasos recursos que necesitaba un intérprete de BASIC, con la llegada de los primeros computadores de sobremesa se produjo una rápida difusión de este lenguaje y con él se desarrollaron las aplicaciones más diversas. Son innumerables las versiones que existen de este lenguaje y en algunas de ellas se incorporan herramientas de todo tipo. Por ejemplo, existen versiones para trabajar con distintos tipos de bases de datos, con entornos de ventanas (windows) o incluso para aplicaciones de tiempo real. Estas versiones han sido desarrolladas fundamentalmente para trabajar con los actuales computadores personales.

5.3.3 Tercera generación

A esta generación pertenecen gran parte de los lenguajes que se utilizan actualmente para el desarrollo de software. A finales de los 60 y principios de los 70 se consolidan las bases teóricas y prácticas de la *programación estructurada*. Con esta nueva metodología, la programación pierde definitivamente su consideración de "arte" destinado a demostrar las habilidades del programador para convertirse en una tarea productiva que se debe realizar de forma metódica y sin concesiones a la genialidad. Hay que tener en cuenta que ya entonces el número de personas que participaban en el desarrollo de un nuevo compilador, sistema operativo u otro tipo de aplicación podía ser de varias decenas. Aparece así, la necesidad de una nueva generación de lenguajes fuertemente tipados, que

faciliten la estructuración de código y datos, con redundancias entre la declaración y el uso de cada tipo de dato. Con ello se facilita la verificación en compilación de las posibles inconsistencias de un programa. Algunos de los lenguajes más importantes de esta generación son los siguientes:

PASCAL: Aparece en 1971 y se puede considerar el primer lenguaje de esta generación. Aunque fue diseñado fundamentalmente para la enseñanza de la *programación estructurada*, su sencillez y buenas prestaciones hizo que pronto se utilizara también para el desarrollo de todo tipo de aplicaciones científico-técnicas o de sistemas. Pascal permite la estructuración del código mediante procedimientos o funciones que se pueden definir de manera recursiva. La tipificación de datos es bastante rígida y no se contempla en forma apropiada la compilación separada.

MODULA-2: Es el descendiente más directo de Pascal y con su diseño se trataban de paliar las carencias más importantes de su predecesor. Ambos lenguajes han sido diseñados por Niklaus Wirth. Las deficiencias fundamentales de Pascal se deben a que no fue pensado como un lenguaje para el desarrollo de software a gran escala. En Modula-2 se incorpora la estructura de módulo y queda separada la especificación del módulo de su realización concreta. Esto facilita una compilación segura y permite la ocultación de los detalles de implementación. Con Modula-2 se facilita enormemente la aplicación inmediata de los conceptos fundamentales de diseño: modularidad, abstracción y ocultación. Además, Modula-2 incorpora ciertos mecanismos básicos de concurrencia. Al contrario de lo que sucedió con Pascal, la utilización de Modula-2 para el desarrollo de aplicaciones es todavía más bien escasa.

C: Aparece en 1975 y su desarrollo es prácticamente paralelo a Pascal. Originalmente fue diseñado para la codificación del sistema operativo UNIX. Posteriormente ha sido utilizado ampliamente para desarrollar software de sistemas y aplicaciones científico-técnicas de todo tipo. A pesar de su consideración de lenguaje de alto nivel posee ciertas características que le hacen muy flexible y capaz de optimizar el código tanto como si se utilizara un lenguaje ensamblador. Por otro lado, esto mismo hace que deba ser utilizado con bastante cuidado. Por ejemplo, la importancia de los punteros dentro del lenguaje y su íntima relación con vectores, matrices y en general con el manejo casi directo de la memoria puede tener resultados desastrosos si no se toman las debidas precauciones. También posee un conjunto de operadores muy amplio que permite

240 Introducción a la Ingeniería de Software

hacer cualquier cosa con cualquier tipo de dato sin apenas ninguna restricción. Esto puede dificultar bastante la depuración de un programa cuando se utilizan indiscriminadamente.

ADA: Es el lenguaje patrocinado por el Departamento de Defensa de los EEUU, principal consumidor de software del mundo, para imponerlo como estándar en todos sus desarrollos de sistemas con computador englobado ("embedded computer systems") y de tiempo real. Es un descendiente de Pascal mucho más potente y complejo. Aunque desde principios de los 80 se viene hablando de las ventajas de Ada como lenguaje universal, todavía no está suficientemente extendido. Esto se ha debido fundamentalmente a la dificultad de aprendizaje de un lenguaje tan complejo y a que la puesta a punto de los compiladores ha requerido más tiempo del previsto. Los *package* de Ada facilitan la codificación de un diseño modular y la aplicación de los conceptos de abstracción y ocultación. Ada permite la definición de elementos genéricos y dispone de mecanismos para la programación concurrente de tareas y la sincronización y cooperación entre ellas.

Dentro de esta misma generación y de forma paralela, se han ido desarrollando lenguajes asociados a otros paradigmas de programación o modelos abstractos de cómputo (orientación a objetos, funcional o lógico) como enfoques alternativos a la programación imperativa representada por Pascal, C, Modula-2 y Ada. Los lenguajes que soportan estos paradigmas tienen una aplicación bastante orientada hacia el campo para el cual han sido diseñados. Algunos de los más representativos son los siguientes:

SMALLTALK: Este lenguaje es el precursor de los lenguajes *orientados a objetos*. Aunque las primeras versiones son de principios de los 70, los entornos de trabajo disponibles eran deficientes y su difusión fue muy limitada. Durante la década de los 80, las estaciones de trabajo con pantallas gráficas y procesadores más potentes permiten disponer de entornos más amigables y las nuevas versiones del lenguaje alcanzan mayor difusión.

C++: Es un lenguaje que incorpora al lenguaje C los mecanismos básicos de la programación *orientada a objetos*: Ocultación, clases, herencia y polimorfismo. Con este lenguaje se trata de aprovechar la amplia difusión que tiene C y facilitar su utilización cuando se emplean técnicas de análisis y diseño orientadas a objetos.

EIFFEL: Este lenguaje de finales de los 80 supone probablemente el intento más serio de diseño de un lenguaje completamente nuevo *orientado a objetos*, tomando como estructura básica de partida un lenguaje imperativo estructurado. Eiffel permite la definición de clases genéricas, herencia múltiple y polimorfismo. Aunque su difusión todavía no es muy grande es previsible que la adquiera en un futuro próximo.

LISP: Es el precursor de los lenguajes *funcionales*. Las primeras versiones aparecieron junto a los lenguajes de 2^a generación. Aunque actualmente compiten con él un gran número de lenguajes funcionales, continúa siendo uno de los más utilizados en aplicaciones de inteligencia artificial y sistemas expertos. En LISP todos los datos se organizan en forma de listas y para su manejo se emplean exclusivamente funciones, que normalmente son recursivas. LISP está pensado especialmente para la manipulación de símbolos, demostración de teoremas y resolución de problemas teóricos.

PROLOG: Este lenguaje es el representante más importante de los lenguajes *lógicos* y se utiliza fundamentalmente para la construcción de sistemas expertos. En el desarrollo de la propuesta de "quinta generación", el lenguaje PROLOG se tomó como un punto de arranque. Los elementos fundamentales que utiliza son: hechos y reglas; ambos constituyen la base de la representación del conocimiento. A partir de ellos se pueden inferir nuevos hechos o reglas que se incorporan a la base del conocimiento. Cuando el número de hechos y reglas crece, el proceso de inferencia puede resultar tremadamente costoso y lento.

5.3.4 Cuarta generación

Los lenguajes de la cuarta generación (L4G) tratan de ofrecer al programador un mayor nivel de abstracción, prescindiendo por completo del computador. Normalmente estos lenguajes no son de propósito general y en realidad se pueden considerar herramientas específicas para la resolución de determinados problemas en los campos más diversos. Se trata de lograr que cualquiera, sin grandes conocimientos sobre programación de computadores, pueda utilizar estos lenguajes para realizar aplicaciones sencillas. Por otro lado, no es aconsejable utilizar estas herramientas para desarrollar aplicaciones complejas debido a lo ineficiente que resulta el código que generan. En todo caso, se pueden utilizar para la realización de

242 Introducción a la Ingeniería de Software

prototipos que posteriormente serán mejorados con lenguajes de la tercera generación.

Desde hace diez o quince años han venido surgiendo lenguajes o herramientas 4G. Sin embargo, y sobre todo en los últimos años, la amplia difusión de los computadores personales de altas prestaciones: potencia de cálculo, pantalla gráfica, ratón, etc; han sido determinantes para su desarrollo más amplio. Sería tremadamente prolijo hacer una relación de herramientas (la mayoría de ellas tienen marcas comerciales) e incluso resulta difícil establecer una clasificación. Una posible agrupación según su aplicación concreta sería la siguiente:

BASES DE DATOS: Estos lenguajes permiten acceder y manipular la información de la base de datos mediante un conjunto de órdenes de petición (QUERY) relativamente sencillas. Con estas órdenes se pueden establecer condiciones para el simple acceso a determinados datos, se pueden agrupar datos que verifican una determinada relación entre ellos, se pueden efectuar cálculos entre los datos u otras operaciones mucho más sofisticadas. La ventaja fundamental de estos lenguajes es que dotan de una gran versatilidad a la base de datos y permiten que pueda ser el propio usuario quien diseñe sus propios listados, informes, resúmenes, cálculos, etc., cuando los necesite.

GENERADORES DE PROGRAMAS: Con ellos se pueden construir elementos abstractos fundamentales en un cierto campo de aplicación sin descender a los detalles concretos que se necesitan en los lenguajes de la tercera generación. La generación de los correspondientes programas, en algún lenguaje de la tercera generación, se realiza de acuerdo a un conjunto de reglas que se establecen a priori y que dependen del lenguaje destino. Evidentemente, el programa generado se puede modificar o adaptar cuando la generación automática no resulta completamente satisfactoria. Con la utilización de estos lenguajes siempre se produce un ahorro considerable de tiempo de desarrollo. La mayoría de los generadores de programas sirven para realizar aplicaciones de gestión y el lenguaje destino es COBOL. Sin embargo, recientemente se han desarrollado herramientas CASE para diseño orientado a objeto que se pueden utilizar para aplicaciones de sistemas y que generan programas en C, C++ o Ada.

CÁLCULO: Existe una amplia gama de herramientas de cálculo destinadas a simplificar las más diversas tareas científicas o técnicas.

Inicialmente estas herramientas fueron simplemente aplicaciones más o menos sofisticadas que ofrecían distintos métodos o algoritmos para resolver un problema concreto en el campo de la estadística, la investigación operativa, el control, etc. Sin embargo, algunas de estas herramientas han evolucionado de tal manera que se han convertido en verdaderos L4G. Con ellos se puede programar prácticamente cualquier problema dentro de un determinado campo. Como ejemplos concretos se pueden citar los siguientes: hojas de cálculo; herramientas de cálculo matemático, herramientas de simulación y diseño para control, etc.

OTROS: En este grupo se pueden englobar todo tipo de herramientas que permitan una programación de cierta complejidad y para ello utilicen un lenguaje. Como ejemplos concretos se pueden citar las herramientas para la especificación y verificación formal de programas, lenguajes de simulación, lenguajes de prototipos, etc.

5.4 Prestaciones de los lenguajes

Es muy importante conocer las prestaciones que ofrecen los lenguajes en su conjunto y cada uno de ellos en particular. Esto facilita mucho la tarea de selección del más adecuado para una determinada aplicación. Existen libros enteros dedicados a realizar estudios comparativos entre distintos lenguajes [Pratt84], [Sebesta89], pero esto queda fuera del alcance de este libro. En este apartado se agrupan las prestaciones de los lenguajes en cuatro grandes apartados y se hace un pequeño repaso de las estructuras más importantes dentro de cada grupo.

5.4.1 Estructuras de control

En este grupo nos referiremos al conjunto de sentencias o instrucciones de los lenguajes que se encuadran dentro de la parte ejecutiva de un programa. Se incluyen aquí algunas estructuras específicas para el manejo de excepciones y para la programación concurrente, además de las clásicas para la programación estructurada.

5.4.1.1 Programación estructurada

Actualmente cualquier lenguaje imperativo dispone de sentencias que facilitan la programación estructurada y esto se puede considerar un

244 Introducción a la Ingeniería de Software

requisito mínimo para realizar cualquier desarrollo software. Para ello, siempre existen sentencias para programar:

SECUENCIA: Normalmente escribiendo una tras otra las sentencias

SELECCIÓN: Normalmente mediante una construcción *if - then - else*

ITERACIÓN: Al menos mediante la construcción *while - do*

El conjunto de sentencias de los lenguajes suele ser algo más rico y se dispone de otras sentencias para la selección:

SELECCIÓN GENERAL: mediante un *if - then - elif - then ... else*

SELECCIÓN POR CASOS: mediante una sentencia *case - of*

y para la iteración:

REPETICIÓN: mediante una sentencia *repeat - until*

BUCLE CON CONTADOR: mediante una sentencia *for - to - do*

BUCLE INDEFINIDO: mediante las sentencias *loop* y *exit*

Por otro lado, para facilitar el desarrollo por refinamientos sucesivos, todos los lenguajes permiten definir subprogramas mediante el empleo de procedimientos o funciones. Normalmente, estos subprogramas se pueden definir en forma *recursiva*. Las ventajas respecto a la sencillez y claridad que representa el empleo de la *recursividad* en la resolución de determinados problemas es indiscutible.

5.4.1.2 Manejo de excepciones

Durante la ejecución de un programa se pueden producir errores o sucesos inusuales que denominaremos genéricamente *excepciones*. Los errores pueden tener distintos orígenes: errores humanos, fallos hardware, errores software. También pueden tratarse como excepciones situaciones no erróneas pero poco frecuentes: datos de entrada vacíos, valores fuera del rango habitual, etc.

Errores humanos: El operador puede introducir datos erróneos, aunque el programa verifique la validez de cada dato por separado cuando se introduce. Resulta muy complejo comprobar todo un conjunto de datos de entrada si están relacionados entre sí. Por ejemplo, entre varios datos de entrada se puede obtener un resultado intermedio que es cero y se producirá una excepción si se utiliza como cociente en una división por cero.

Fallos hardware: Cualquier mal funcionamiento de un periférico (teclado, ratón, disco, coprocesador matemático, sensor, actuador, etc.) puede dar como resultado un dato erróneo que se introduce al programa. También se pueden producir errores debidos a la capacidad limitada de los recursos: memoria, disco, coprocesador matemático, etc.

Errores software: El programa puede tener algún defecto no detectado en la fase de pruebas.

Datos de entrada vacíos: Por ejemplo, un programa para ordenar una colección de datos, u obtener estadísticas, puede que reciba ocasionalmente una colección de datos vacía, sin ningún elemento.

Valores fuera de rango: A veces se puede invocar una operación o función sobre un operando o argumento no admisible, tal como una división por cero o la raíz cuadrada de un número negativo.

Si las situaciones anteriores no han sido previstas, el programa puede abortar. Los sistemas operativos son los encargados de detectar los fallos hardware y de evitar que los errores humanos o de software puedan llegar a afectar al resto de los usuarios del computador o al mismo sistema operativo. Por ejemplo: la violación de la partición de memoria asignada a otro programa o de las zonas de disco o memoria reservadas al propio sistema operativo. La medida correctora del sistema operativo es siempre abortar el programa para evitar males mayores. Este hecho no es admisible en un programa que debe funcionar siempre y en cualquier circunstancia. Por ejemplo, el control de una central nuclear, los frenos ABS de un coche, una centralita telefónica, un cajero automático, etc.

Para facilitar la claridad del programa es preferible, siempre que sea posible, escribir la parte principal del código atendiendo sólo a las situaciones normales. Esto exige algún mecanismo apropiado para desviar automáticamente la ejecución hacia un fragmento de código apropiado en caso de que ocurra alguna situación anormal. Esta transferencia de control se contempla en lenguajes que incorporan el *manejo de excepciones*.

Para concretar un poco más como se realiza el manejo de las excepciones, a continuación se describe muy someramente la propuesta de Ada, que sí nos permite manejar las excepciones dentro del programa. En Ada existen varias excepciones predefinidas que son las siguientes:

246 Introducción a la Ingeniería de Software

CONSTRAINT_ERROR	=>	Fuera de Rango
NUMERIC_ERROR	=>	Operaciones aritméticas
PROGRAM_ERROR	=>	Ejecución anómala de una sentencia
STORAGE_ERROR	=>	Falta de memoria
TASKING_ERROR	=>	Fallo en tareas concurrentes

Cada una de estas excepciones agrupa varias causas distintas y se dispara automáticamente cuando se detecta cualquier fallo o error del tipo correspondiente. Por ejemplo, la excepción CONSTRAINT_ERROR se puede disparar tanto por un valor entero que excede el máximo como por un índice de una matriz fuera del tamaño definido.

Ada también dispone de un controlador predefinido para las excepciones predefinidas, que se ejecuta cuando en nuestro programa no se realiza ningún tratamiento específico para alguna de ellas. El tratamiento predefinido suele ser dar un mensaje de error y abortar el programa. La programación de un control específico de las excepciones se hace al final de cada unidad de programa o módulo con la siguiente sintaxis:

```
begin
    BLOQUE_UNIDAD
exception
    when EXCEPCION_UNO => TRATAMIENTO_UNO;
    when EXCEPCION_DOS => TRATAMIENTO_DOS;
    .....
end;
```

El BLOQUE_UNIDAD es la secuencia de sentencias de la unidad de programa. EXCEPCION_XXX pueden ser cualquiera de las predefinidas u otras que pueda definir el programador. Los TRATAMIENTO_Xxx se programan como secuencias de sentencias en la forma habitual. Cuando se dispara una excepción al ejecutar el BLOQUE_UNIDAD, la ejecución continúa con el correspondiente TRATAMIENTO. En general, este tratamiento tendrá como objetivo analizar, si es posible, cuál fue la causa de la excepción, registrar la situación encontrada y en todo caso recuperar el estado del sistema para que pueda continuar su ejecución normal.

En Ada, después de la ejecución del TRATAMIENTO se abandona por completo la ejecución de la unidad. Por el contrario, en otros lenguajes (p.e. PL/1) se reanuda la ejecución inmediatamente después del punto en el que se disparó la excepción.

Cuando se produce la excepción en una unidad que no tiene programado su tratamiento, la excepción se propaga a la unidad que la llamó. Esta propagación continúa hasta alcanzar la unidad más externa y ejecutar el tratamiento predefinido si no existe ningún otro.

Ada permite la definición de nuevas causas de excepción. Este mecanismo sirve para cortar una línea de ejecución bruscamente cuando se detecta, por ejemplo, una situación de alarma que requiere un tratamiento inmediato al margen del habitual. Otra aplicación es matizar las excepciones predefinidas y adaptarlas a las necesidades de nuestro programa dado que en la mayoría de los casos las excepciones predefinidas no dan demasiada información de la causa concreta de la excepción. A cada nueva excepción se le debe dar un nombre y efectuar su declaración, que se debe hacer al comienzo de la unidad con la siguiente sintaxis:

Sobre_Presion, Tabla_Llena: exception

Para indicar desde el programa que se ha producido la excepción se utiliza la construcción:

raise Sobre_Presión o raise Tabla_Llena

Evidentemente, el disparo de la excepción estará condicionado a la comprobación de una determinada situación. Por ejemplo:

if Presión > Maxima_Presion then raise Sobre_Presion end if;

Como se indicó anteriormente, el tratamiento de estas excepciones se programa igual que las predefinidas:

```

exception
  when Sobre_Presion => ....
  when NUMERIC_ERROR => ...
  ...
end
```

Un análisis más detallado del manejo de excepciones en Ada se puede encontrar en cualquier manual o libro específico del lenguaje. En cualquier caso, siempre que se necesite manejar excepciones se deben estudiar las alternativas que ofrecen los distintos lenguajes para evitar trabajar a bajo nivel, interceptando las acciones del sistema operativo.

248 Introducción a la Ingeniería de Software

5.4.1.3 Concurrencia

En el Tema 3 se hizo una pequeña introducción al concepto de concurrencia y a algunos de los aspectos fundamentales que se deben tener en cuenta para su diseño y programación:

- Tareas que se deben ejecutar concurrentemente
- Sincronización de tareas
- Comunicación entre tareas
- Interbloqueos (en inglés "deadlock")

Para los tres primeros aspectos se han sistematizado estructuras de control específicas. Algunas de ellas ya han sido incorporadas a ciertos lenguajes para facilitar la programación concurrente. El cuarto aspecto es un problema que no se puede sistematizar globalmente y que se debe estudiar en cada caso particular.

Todos los lenguajes concurrentes permiten declarar distintas tareas y definir la forma en que se ejecutarán concurrentemente. Sin embargo, existen distintas formas de abordar este aspecto:

CORRUTINAS: No existen tareas sino corrutinas. Las corrutinas tienen una estructura semejante a subprogramas, pero entre ellas se pueden transferir el control de ejecución en cualquier momento y no sólo de una forma jerarquizada según el estricto orden de llamada/retorno. En realidad la concurrencia se limita a un avance de la ejecución de todas las corrutinas pero secuenciando ese avance por un acuerdo entre ellas. Esta propuesta es la que utiliza Modula-2.

FORK-JOIN: Una tarea puede arrancar la ejecución concurrente de otras mediante una orden **fork**. La concurrencia se finaliza con un **join** invocado por la misma tarea que ejecutó el fork y con el que ambas tareas se funden de nuevo en una única. Es posible realizar todos los **fork-join** que se deseen, implicando a cualquier número de tareas y con cualquier nivel de anidamiento. Esta propuesta es la que se emplea en el sistema operativo UNIX y en el lenguaje PL/I.

COBEGIN-COEND: Todas las tareas que se deben ejecutar concurrentemente se declaran dentro de una construcción **cobegin T₁ | T₂ | ... | T_n coend**. Al alcanzar el **cobegin** se inicia la ejecución de todas las tareas (T₁, T₂, ..., T_n). La finalización de la concurrencia exige que todas las tareas hayan acabado. Es posible anidar varios **cobegin-coend**. Esta propuesta se utiliza en el lenguaje ALGOL68.

PROCESOS: Cada tarea se declara como un proceso. Todos los procesos declarados se ejecutan concurrentemente desde el comienzo del programa y no es posible iniciar ninguno nuevo. Esta propuesta es la utilizada por Pascal Concurrente. En Ada se utiliza esta misma propuesta, pero además es posible lanzar dinámicamente un proceso en cualquier momento.

Para lograr la sincronización y la cooperación entre tareas, los lenguajes concurrentes disponen de estructuras con las que se pueden resolver ambos aspectos. Además, prácticamente cada lenguaje propone estructuras diferentes y aunque cada una tiene sus ventajas e inconvenientes, la mayoría de ellas son intercambiables, en cuanto a que permiten resolver los mismos problemas con una mayor o menor dificultad. Las estructuras con una mayor aceptación, clasificadas en dos grandes grupos, son las siguientes:

- | | |
|------------------------|---|
| VARIABLES COMPARTIDAS: | { Semáforos
Regiones críticas condicionales
Monitores |
| PASO DE MENSAJES: | { "Communicating Sequential Processes" (CSP)
Llamada a procedimientos remotos
"Rendezvous" de Ada |

El primer grupo necesita que todas las tareas se ejecuten en un mismo computador (multiprogramación) o en distintos computadores pero utilizando una memoria compartida (multiproceso) para que todas las tareas tengan acceso a las VARIABLES COMPARTIDAS que emplean para comunicarse.

Además, el segundo grupo se puede utilizar para la sincronización y cooperación entre tareas que se ejecutan en computadores que no tienen ninguna memoria común (procesos distribuidos) y que están conectados por una red de comunicaciones que les permite comunicarse mediante el PASO DE MENSAJES.

Para ilustrar los problemas de sincronización y cooperación entre tareas utilizaremos los semáforos que son la primera estructura que se propuso ya en 1968 por Dijkstra. Un semáforo es un nuevo tipo abstracto de datos del que se pueden declarar variables de una forma similar a:

```

VAR
  Sincro : semaphore := 0;
  Arbitro : semaphore := 1;

```

250 Introducción a la Ingeniería de Software

El tipo **semaphore** sólo puede tomar valores positivos entre cero y un valor máximo, que depende de la implementación. En la declaración se le puede dar un valor inicial. Por ejemplo, la variable Sincro se inicializa a cero y la variable Arbitro se inicializa a uno.

El tipo abstracto **semaphore** tiene definidas dos únicas operaciones, P y V, que realizan, en forma indivisible, lo siguiente:

```
P( Sincro ): IF Sincro = 0 THEN
    la tarea que ejecuta P( Sincro ) se suspende y se
    pone en cola
    ELSE (* Sincro > 0 *)
        Sincro := Sincro - 1;
    END
```

```
V( Sincro ): IF hay alguna tarea esperando en la cola THEN
    continúa la ejecución la primera tarea de la cola
    ELSE
        Sincro := Sincro + 1;
    END
```

Un semáforo se puede utilizar para la sincronización entre dos tareas. Por ejemplo, con la declaración del semáforo Sincro inicializado a 0, analizaremos el comportamiento de las siguientes tareas:

```
PROCESS Productor;
.....
BEGIN
    LOOP
        elaborar un dato;
        V(Sincro)
    END
END Productor;
```

```
PROCESS Consumidor;
.....
BEGIN
    LOOP
        P(Sincro);
        utilizar el dato
    END
END Consumidor;
```

Inicialmente, la tarea Consumidor esperará suspendida en P(Sincro) hasta que la tarea Productor haya elaborado el primer dato y ejecute V(Sincro) dejando Sincro con un valor igual a 1. Esto mismo sucederá con los siguientes datos si la velocidad del Productor es inferior a la del Consumidor. Por el contrario, si la tarea Productor puede elaborar varios datos por anticipado, con los correspondientes V(Sincro) se indicará a la tarea Consumidor que cuando ejecute P(Sincro) puede continuar su ejecución puesto que Sincro es mayor que cero y hay

datos ya elaborados para consumir. De esta manera se logra la sincronización entre ambas tareas.

El semáforo también se puede utilizar para programar la cooperación entre tareas. Por ejemplo, con la declaración del semáforo Arbitro inicializado a 1 se puede compartir el acceso a un recurso común: disco, impresora, etc. En este caso tendríamos los siguientes procesos o tareas:

```
PROCESS Uno;
.....
BEGIN
    LOOP
        .....
        P( Arbitro )
        usar recurso
        V( Arbitro )
        .....
    END
END Uno;

PROCESS Dos;
.....
BEGIN
    LOOP
        .....
        P( Arbitro );
        usar recurso
        V( Arbitro )
        .....
    END
END Dos;
```

La primera tarea que ejecute P(Arbitro) dejará Arbitro igual a 0. Esto hace esperar a la otra tarea al ejecutar su P(Arbitro). La espera se prolongará hasta que la primera finalice la utilización del disco o la impresora y ejecute el V(Arbitro). Dependiendo del resto de trabajo de cada tarea la alternancia en la utilización del disco o de la impresora puede ser cualquiera, pero siempre se garantiza que hay exclusión mutua entre ellas y que la cooperación es satisfactoria.

Podemos decir que los semáforos son estructuras de bajo nivel para programar la concurrencia entre tareas. Por ejemplo, si no se sigue la disciplina convenida:

- Se puede usar el recurso sin realizar el P(Arbitro)
- Se puede bloquear el recurso si no se realiza el V(Arbitro)
- Es fácil equivocarse y hacer un P(Sincro) en vez de P(Arbitro)
- No se distingue un semáforo para sincronizar de otro para cooperar

Todos estos errores son difíciles de detectar y depurar. Las propuestas posteriores: regiones críticas, monitores, CSP, etc., evitan en gran medida todos estos problemas. Como siempre lo adecuado es

252 Introducción a la Ingeniería de Software

estudiar las estructuras disponibles en el lenguaje que se pretende utilizar.

5.4.2 Estructuras de datos

En este grupo nos referiremos a las distintas formas que emplean los lenguajes para estructurar los datos que manejan.

5.4.2.1 Datos simples

Todos los lenguajes pueden manejar datos de tipo entero, pero hay que tener en cuenta su rango. Algunos lenguajes tienen dos o más rangos de enteros: normal, corto, largo, etc.

También es normal manejar datos reales en todos los lenguajes. En este caso hay que tener en cuenta su precisión, aunque a veces se dispone de reales con precisión simple y con doble precisión. En algunos lenguajes más dedicados al cálculo científico se pueden manejar directamente datos complejos.

Actualmente, prácticamente todos los lenguajes y computadores manejan los datos de tipo carácter empleando habitualmente el código ASCII. Aunque es bastante raro emplear alguna otra tabla de código, es conveniente comprobarlo. Para manejar los caracteres agrupados la mayoría de los lenguajes disponen de datos de tipo *ristra de caracteres* ("string"). Sin embargo, no existe una forma única de organización y operación con estos datos y es aconsejable estudiar la forma específica del lenguaje empleado.

En lenguajes como Pascal y sus descendientes, es posible definir datos simples por enumeración, de la siguiente forma:

```
TYPE Color = (Rojo, Amarillo, Azul, Verde, Gris);
```

Cuando un lenguaje no tiene esta posibilidad, se pueden emplear directamente los enteros y es responsabilidad del programador hacer una asociación y manipulación correctas. Con este tipo de datos se aumenta mucho la claridad de los programas y se evitan errores. Un tipo de dato enumerado del que sí disponen la mayoría de los lenguajes es el tipo lógico (verdadero, falso) o booleano.

Los datos de tipo *subrango* permiten acotar el rango de un tipo de dato ordinal para crear un nuevo tipo de dato. Por ejemplo:

```
TYPE ColoresBasicos = [ Rojo .. Azul ];
```

En los lenguajes que tienen esta posibilidad resulta más sencilla la depuración y el mantenimiento de los programas, si se dispone de detección automática en ejecución de los valores que se salen del rango.

5.4.2.2 Datos compuestos

Las posibilidades de definir datos compuestos o estructurados en los lenguajes de programación varían de unos a otros, e incluso dependen de la versión concreta de compilador o de la versión del estándar del lenguaje. Los datos compuestos siempre se definen como combinaciones de otros tipos de datos simples y compuestos ya definidos.

Prácticamente todos los lenguajes tienen la posibilidad de definir y manejar formaciones (vectores o matrices, en inglés "arrays") y solamente es necesario familiarizarse con la sintaxis y semántica de cada uno.

La definición de datos compuestos de elementos heterogéneos (esquema tupla) se realiza en la mayoría de los lenguajes mediante la estructura registro ("record"). Por ejemplo:

```
TYPE Circulo = RECORD
    CentroX, CentroY: REAL;
    Radio: INTEGER;
    Pintado: Color
END;
```

En Pascal y sus descendientes la estructura registro se usa también para definir el esquema unión como registro con campos variantes. En lenguaje C existe una construcción separada para ello.

En lenguajes como FORTRAN, en los que no existen los registros, hay que recurrir a un ARRAY o formación para agrupar varios datos en una sola estructura.

Pascal y sus descendientes tienen la posibilidad de definir datos de tipo *conjunto* tomando como referencial un tipo enumerado o subrango. Por ejemplo:

```
TYPE Mezcla = SET OF Color;
```

254 Introducción a la Ingeniería de Software

En lenguajes como C, que no disponen de esta posibilidad, se puede recurrir a las operaciones para manejo de bits, de bajo nivel, para usar un grupo de bytes como estructura conjunto.

El empleo de datos dinámicos está resuelto de formas muy diversas en los distintos lenguajes. Por ejemplo, en LISP las listas manejadas dinámicamente son los datos fundamentales. En Pascal, Modula-2, C y Ada se realiza mediante punteros. En FORTRAN no existe esta posibilidad.

5.4.2.3 Constantes

En los primeros lenguajes sólo se contemplaba el empleo de constantes literales, expresadas directamente por su valor numérico o de caracteres. En los lenguajes modernos se pueden declarar constantes simbólicas, con nombre.

En Pascal las constantes han de declararse antes que los tipos. Así, sólo es posible que las constantes puedan ser de alguno de los tipos predefinidos: entero, real, carácter o booleano.

En Modula-2 no existe un orden obligatorio para las declaraciones. Esto permite declarar también constantes de un tipo enumerado o conjunto. Por ejemplo:

```
CONST  
    Pintura = Amarillo;  
    MiColor = Mezcla {Rojo, Verde }
```

Sin embargo, en Modula-2 no se pueden declarar constantes de tipo estructurado: formación o registro. Esto sí es posible en Ada utilizando un agregado: agrupación de expresiones separadas por comas y encerradas entre paréntesis. Por ejemplo:

```
Tabla: constant array (0..1, 0..2) of REAL :=  
        ( (1.0, 2.0, 3.0), (M*N, 0.0, 0.0) );
```

Cualquiera de las expresiones puede incluir valores simbólicos que se evaluarán una única vez en ejecución. Esto permite una inicialización dinámica de las constantes y "congelar" un valor constante para el resto de la ejecución del programa después de calculado.

5.4.2.4 Comprobación de tipos

Las operaciones que se pueden realizar con los datos de un programa dependen del nivel de comprobación de tipos que corresponda al lenguaje utilizado. Según se propone en [Fairley85] al menos se pueden distinguir los cinco niveles siguientes:

- Nivel 0: Sin tipos
- Nivel 1: Tipado automático
- Nivel 2: Tipado débil
- Nivel 3: Tipado semirrígido
- Nivel 4: Tipado fuerte

NIVEL 0 (Sin Tipos): A este nivel pertenecen los lenguajes en los que no se pueden declarar nuevos tipos de datos y todos los datos que se utilizan deben pertenecer a sus tipos predefinidos. Estos lenguajes han sido pensados para realizar aplicaciones en un campo específico y resulta muy complejo utilizarlos fuera de ese campo. Según el campo de aplicación del lenguaje, los datos predefinidos son numéricos: BASIC; caracteres y numéricos: COBOL; listas: LISP; formaciones multidimensionales: APL; ristras de caracteres: Snobol; etc. En todos ellos, no es necesario que el compilador realice ninguna comprobación de tipos. Es responsabilidad exclusiva del programador distinguir qué representa cada dato.

NIVEL 1 (Tipado automático): En este caso es el compilador el encargado de decidir cuál es el tipo más adecuado para cada dato que utiliza el programador. Asimismo, es también el compilador el encargado de convertir al tipo adecuado los operandos de una expresión cuando estos son incompatibles entre sí o cuando lo son con el operador utilizado en la expresión. Por ejemplo, si se utiliza un dato lógico en una expresión aritmética se convertirá previamente a numérico con la equivalencia: 0 = verdadero y 1 = falso, o si se utiliza un dato numérico en una expresión de caracteres se convertirá a su representación como ristra de dígitos. Todo esto se realiza automáticamente y aparentemente libera al programador de cualquier comprobación. Sin embargo, si se desconocen las reglas de selección y conversión de tipos que utiliza el compilador, los resultados pueden ser una sorpresa. Un lenguaje que utiliza esta forma de comprobación de tipos es PL/1.

NIVEL 2 (Tipado débil): También en este nivel se realiza una conversión automática de tipos pero solamente entre datos que poseen ciertas

256 Introducción a la Ingeniería de Software

similitudes. Así, no es posible la conversión automática de un valor lógico a numérico o viceversa pero sí son factibles las conversiones entre enteros y reales. En expresiones con operandos de distintos tipos (enteros, reales, etc.), las conversiones siempre se hacen hacia el tipo de mayor rango o precisión (real). FORTRAN es el lenguaje típico de los que utilizan esta comprobación de tipos. El punto en el que la comprobación de tipos resulta muy débil en FORTRAN es en los subprogramas. Entre declaración y utilización de un subprograma no se realiza ninguna comprobación ni en el número ni en los tipos de los argumentos.

NIVEL 3 (Tipado semirrígido): El lenguaje más representativo de este nivel es Pascal. Todos los datos que se quieran usar deben ser declarados previamente con sus correspondientes tipos. Son tipos incompatibles los que se declaran por separado con nombres distintos aunque tengan la misma estructura. Nunca es posible realizar operaciones entre datos de tipos incompatibles. Respecto a los procedimientos y funciones, se comprueban el número de los argumentos y el tipo de cada uno de ellos para que coincida la declaración con su utilización. Sin embargo, en un tipado semirrígido existen algunas vías de escape que permiten evitar todo lo anterior. Por ejemplo, en Pascal no se comprueban los tipos de los argumentos de aquellas funciones o procedimientos que son pasados a su vez como argumentos de otros procedimientos o funciones. Pero la vía de escape más usada en este tipo de lenguajes es la compilación separada ya que no existe ninguna forma de comprobación de tipos en tiempo de montaje o de ejecución. Por tanto, puede ser distinto lo que se declara y compila en un módulo de lo que se compila y usa en otro módulo.

NIVEL 4 (Tipado fuerte): En este nivel no existe ninguna escapatoria posible y el programador está obligado a hacer explícita cualquier conversión de tipo que necesite realizar. Las comprobaciones de tipo se realizan en compilación, carga y ejecución. Para llevar a cabo esta comprobación tan exhaustiva de tipos es necesario disponer, además del compilador para el lenguaje, de un entorno único de desarrollo de programas que englobe toda la información necesaria. Los lenguajes que tienen este nivel son Ada y Modula-2.

5.4.3 Abstracciones y objetos

La importancia del concepto de abstracción ya ha sido discutida en el Tema 3. Desde el punto de vista del diseño se indicaban tres formas fundamentales de abstracción:

- Abstracciones funcionales
- Tipos abstractos
- Máquinas abstractas

Para la codificación de un diseño basado en las dos primeras formas, los lenguajes de propósito general disponen de ciertas estructuras que simplifican bastante esta labor. Sin embargo, para la programación de una máquina abstracta se requieren herramientas mucho más específicas (p.ej.: lenguaje 4G, simulador, etc.), lo que se escapa bastante del objetivo de este apartado.

5.4.3.1 Abstracciones funcionales

Las abstracciones funcionales han sido durante mucho tiempo un medio fundamental para el diseño, estructuración y codificación de los programas; por ello, todos los lenguajes disponen de alguna construcción para su definición. Según el lenguaje, la denominación puede variar: subprogramas, subrutinas, procedimientos, funciones, etc. En todos ellos se tienen que definir el nombre y la *interfaz* de la abstracción, se tiene que codificar la *operación* que realiza y finalmente se dispone de un mecanismo para la utilización de la abstracción.

Normalmente, en todos los lenguajes permanece oculta la codificación de la *operación* para quien hace uso de la misma. Sin embargo, dependiendo de las reglas de visibilidad de cada lenguaje, en la codificación se podrá o no acceder a datos externos. Pascal, Modula-2 y Ada tienen una visibilidad por bloques, lo que permite consultar y modificar cualquier dato de los bloques más externos. En FORTRAN las subrutinas se deben realizar en ficheros aparte y por tanto hay total opacidad desde dentro hacia fuera y al contrario. Conviene recordar que en FORTRAN para acceder a datos globales se dispone de la sentencia COMMON.

5.4.3.2 Tipos abstractos de datos

Para la codificación de un tipo abstracto de datos se deben agrupar en una entidad única el *contenido* o *atributos* de los datos de la abstracción y las *operaciones* definidas para el manejo del contenido. Además, debe existir un mecanismo de ocultación que impida el acceso al *contenido* por una vía distinta a las que ofrecen las *operaciones* definidas.

Pascal no dispone de ninguna estructura específica para agrupar datos con funciones/procedimientos y sus reglas de visibilidad por bloques no permiten lograr el nivel de ocultación adecuado. Empleando un fichero

258 Introducción a la Ingeniería de Software

aparte, las subrutinas de FORTRAN sí tendrían el nivel de ocultación adecuado pero no sería fácil definir los *atributos* como tipo, y sólo sería posible tener una subrutina por fichero.

Modula-2 dispone de la estructura **MODULE** con la que se pueden definir de una forma agrupada el contenido y operaciones de un tipo abstracto:

```
DEFINITION MODULE Ejemplo;
  TYPE Contenido;
  PROCEDURE OperacionUno( ... );
  PROCEDURE OperacionDos( ... );
  ....
END Ejemplo.
```

Además, en esta definición aparecen los únicos elementos visibles del tipo abstracto que se necesitan para poder hacer uso de él. Permanecerán ocultos todos los detalles de la codificación o implementación del tipo abstracto. En el caso de Modula-2, esto último se hace por separado de la siguiente forma:

```
IMPLEMENTATION MODULE Ejemplo;
  ....
  PROCEDURE OperacionUno( ... );
  ....
  END OperacionUno;
  PROCEDURE OperacionDos( ... );
  ....
  END OperacionDos;
  ....
BEGIN
  ...
END Ejemplo.
```

El lenguaje Ada dispone de la estructura *package* que es muy semejante al **MODULE** de Modula-2

5.4.3.3 Objetos

Como se indicó en el Tema 3, desde el punto de vista del diseño existe un gran paralelismo entre abstracciones y objetos. Sin embargo, desde el punto

de vista de la programación las diferencias son algo más importantes. Los conceptos de polimorfismo y herencia aparecen muy ligados a los objetos y para su codificación resulta necesario que los lenguajes de programación dispongan de unas construcciones específicas.

Modula-2 no dispone de mecanismos de herencia ni polimorfismo. Así, aunque es un lenguaje perfectamente válido para codificar un diseño basado en abstracciones, resulta muy complicado tratar de codificar un diseño orientado a objetos si entre ellos existe alguna relación de clasificación, especialización o herencia.

El lenguaje Ada dispone de estructuras para la programación de ciertos tipos de polimorfismo. Por un lado, se pueden declarar elementos genéricos (*generic*) y por otro, también es posible aplicar *sobrecarga* a los operadores y las funciones. Sin embargo, no es posible realizar polimorfismo de *anulación* o *diferido* que son los más ligados al mecanismo de herencia.

Solamente con los denominados genéticamente lenguajes orientados a objetos resulta factible la codificación de diseños orientados a objetos con herencia simple o múltiple y polimorfismo. Algunos de ellos son los siguientes:

- Smalltalk
- Objective - C
- Object Pascal
- C++
- Eiffel

Para un estudio más detallado de la programación orientada a objetos y las peculiaridades de cada uno de estos lenguajes se puede consultar [Budd94].

5.4.4 Modularidad

Según se indicaba en el Tema 3, el concepto de modularidad está ligado a la división del trabajo y al desarrollo en equipo de un proyecto software. En los primeros lenguajes de programación las herramientas para la división en partes o módulos no estaban vinculadas nunca al lenguaje y normalmente eran las herramientas generales del sistema operativo (gestores de ficheros, montadores de enlaces, cargadores, etc.), las encargadas de facilitar este trabajo. A medida que el tamaño de los proyectos aumenta se ve más clara la importancia de disponer dentro de los mismos lenguajes de mecanismos adecuados para facilitar el trabajo en equipo. La primera calidad que se exige es la *compilación separada*, que permite preparar y compilar separadamente el código de cada módulo.

260 Introducción a la Ingeniería de Software

Además, en los lenguajes modernos se introducen redundancias entre la definición y uso de cada módulo que podrán ser verificadas automáticamente por el compilador del lenguaje. Cuando es posible comprobar en tiempo de compilación que el uso de un elemento es consistente con su definición, diremos que la compilación separada es *segura*.

En FORTRAN se requiere un fichero diferente para cada una de las subrutinas o módulos en los que se divide un proyecto. Cada fichero se compila por separado del resto y se obtiene otro fichero reubicable en el que sólo se conserva con seguridad el nombre de la subrutina. Cuando se unen todos los ficheros reubicables la única comprobación posible es si están disponibles o no todas las subrutinas utilizadas. Evidentemente, esta compilación no es segura puesto que no es posible verificar si coinciden el número de argumentos de la subrutina en su definición y en su utilización, ni por supuesto el tipo de cada uno de ellos.

Pascal fue diseñado para la realización de programas monolíticos y no está contemplada totalmente en la definición estándar del lenguaje una compilación separada por partes. Hay que tener en cuenta que Pascal tenía como objetivo la enseñanza de la programación y no el desarrollo de medianos o grandes proyectos. No obstante, la amplia difusión alcanzada por Pascal ha llevado a que en sus compiladores actuales se pueda realizar una compilación separada empleando para ello las "unidades" o "módulos". Sin embargo, esta compilación tampoco es segura por las mismas razones que no lo es en FORTRAN. Precisamente esta es una vía de escape que existe en Pascal para evitar un tipado fuerte.

Para conseguir que una compilación separada sea segura se debe comprobar que la definición de unos elementos en un módulo es consistente con el uso que se hace de ellos desde otro módulo. Una forma de conseguir esto es separar de manera lógica la definición de los elementos, o interfaz del módulo, de la codificación de dichos elementos. La interfaz del módulo será lo único visible para el resto de los módulos y en la compilación de estos últimos se podrá comprobar que el uso coincide con la definición dada, permaneciendo oculto cómo ha sido o será finalmente codificado. Por otro lado, en la compilación de la codificación del módulo se comprobará también que concuerda con su definición.

Modula-2 debe su nombre a la importancia que tuvo el concepto de modularidad en su diseño. Como se mostró en una sección anterior, para la codificación de cada módulo es necesario definir su interfaz con una estructura **DEFINITION MODULE** y su codificación mediante una

estructura **IMPLEMENTATION MODULE**. La compilación es segura puesto que la definición de la interfaz se utiliza tanto para comprobar el uso por otros módulos como para comprobar la codificación del módulo de implementación. En Ada se adopta la misma solución y para ello se utilizan las construcciones *package* y *package body* para la definición y la implementación, respectivamente.

En C ANSI la parte de definición (prototipos) se puede escribir en un fichero aparte. Mediante una directiva `#include`, ese fichero se puede compilar formando parte de otro fichero, bien sea el de implementación o bien otro módulo que usa esa definición. En ambos casos, el compilador comprobará la coherencia de todo el código como si fuera un único fichero.

5.5 Criterios de selección del lenguaje

Después del repaso a las diferentes prestaciones que ofrecen los lenguajes no parece sencilla la selección de uno determinado para el desarrollo de un proyecto. El lenguaje de programación es probablemente uno de los elementos más importantes de cualquier desarrollo puesto que es la herramienta de trabajo que utilizarán mayor número de personas y que además tiene una influencia decisiva en la depuración y mantenimiento de la aplicación. A priori, en la decisión deberían prevalecer los criterios técnicos, pero existen otros factores de tipo operativo que deben ser tenidos muy en cuenta. Algunos de los criterios de selección que deberían ser analizados son los siguientes:

IMPOSICIÓN DEL CLIENTE: En algunos casos es directamente el cliente el que fija el lenguaje que se debe utilizar. Las razones para esta imposición pueden ser de diversa índole. Por ejemplo, el lenguaje Ada fue diseñado para imponerlo como estándar en todos los desarrollos de sistemas con computador empotrado o englobado (*embedded computer systems*) para el Departamento de Defensa norteamericano; se trataba con ello de disminuir los costes de desarrollo y mantenimiento que se producen cuando se utilizan cientos de lenguajes diferentes. En otras ocasiones el cliente no es tan drástico y simplemente establece una relación reducida de lenguajes que se pueden usar en sus desarrollos.

TIPO DE APLICACIÓN: Aunque con las prestaciones de cualquier lenguaje de última generación se pueden realizar aplicaciones para los más diversos campos, no parece muy realista pensar en un lenguaje universal que sirva

262 Introducción a la Ingeniería de Software

para todo. Por el contrario cada día aparecen nuevos lenguajes orientados a un campo de aplicación concreta.

Sólo en aplicaciones de tiempo real muy crítico (robótica, aeroespaciales, etc.) o para hardware muy especial, sin compiladores de lenguajes de alto nivel, estaría justificado el empleo de lenguajes ensambladores. En todo caso, la parte realizada en ensamblador debería quedar reducida exclusivamente a lo imprescindible.

Para aplicaciones de gestión lo habitual es utilizar COBOL, aunque cada día se utilizan más los lenguajes de cuarta generación. En las aplicaciones del área técnica/científica, tradicionalmente ha sido FORTRAN el lenguaje más empleado, aunque tanto Pascal como C también se utilizan con bastante frecuencia. Para aplicaciones de sistemas y tiempo real se utilizan bastante C, Modula-2 y Ada. Los lenguajes Lisp y PROLOG todavía están entre los más usados para las aplicaciones de inteligencia artificial. Finalmente, para aplicaciones con un diseño orientado a objetos se pueden utilizar C++, Eiffel o cualquier otro diseñado para este fin.

DISPONIBILIDAD Y ENTORNO: Desde luego no existen compiladores de todos los lenguajes para todos los computadores. Por tanto, como paso previo, es necesario comprobar qué compiladores existen para el computador elegido. Normalmente, el número de ellos será bastante reducido. Puede resultar interesante realizar un estudio comparativo de los compiladores disponibles en cuanto a la memoria y tiempo de ejecución del código que generan para un programa sencillo de prueba. En este sentido se debe tener en cuenta si el código generado se ejecuta directamente o se interpreta posteriormente.

Un factor muy importante a tener en cuenta en la selección, es el entorno que acompaña a cada compilador. El desarrollo será más sencillo cuanto más potentes sean las herramientas disponibles: editor, montador de enlaces, depurador, control de versiones, manejo de librerías de programas realizados en ensamblador o en otros lenguajes, etc. Por otro lado, también se deben considerar las facilidades de manejo de estas herramientas y lo "amigable" que resulta su utilización, por ejemplo, no es lo mismo un entorno de ventanas y ratón que otro basado en menús.

EXPERIENCIA PREVIA: Aunque se supone que el equipo de trabajo posee una buena metodología de trabajo y se adaptaría rápidamente a un nuevo lenguaje y entorno, siempre que sea posible es más rentable aprovechar la experiencia previa. Cuando las condiciones de trabajo no se modifican el rendimiento aumenta y se disminuyen las posibilidades de error. En la mayoría de las empresas de software están establecidos tanto el lenguaje de

programación que utilizan preferentemente como su entorno de desarrollo y en todo caso, a lo largo del tiempo, se van actualizando con nuevas versiones. Hay que tener en cuenta que la formación de todo el personal es una inversión muy importante.

REUSABILIDAD: Este aspecto es importante tanto por la posibilidad de utilizar software ya realizado como por el hecho de dejar disponibles para otros proyectos partes del software desarrollado. Así, por un lado, es muy interesante conocer exhaustivamente las librerías disponibles y por otro, resulta muy conveniente disponer de herramientas dentro del entorno del lenguaje para la organización de dichas librerías en las que se facilite la búsqueda y el almacenamiento de los módulos reutilizables. Este tipo de herramientas todavía no están muy extendidas pero es previsible que se utilicen mucho en un futuro muy próximo.

TRANSPORTABILIDAD: Los desarrollos se realizan para un computador concreto, pero a lo largo del tiempo este se queda obsoleto. Por esta y otras razones, es bastante habitual que se traten de trasladar las aplicaciones de unos computadores a otros. Para facilitar esta labor es muy interesante que el lenguaje utilizado sea transportable. La transportabilidad está ligada a que exista un estándar del lenguaje que se pueda adoptar en todos los compiladores. Pese a todo siempre existirán pequeños detalles que no resultarán completamente compatibles y que deberán ser modificados para adaptarlos al nuevo computador.

USO DE VARIOS LENGUAJES: Aunque no es aconsejable mezclar varios lenguajes en un mismo proyecto, hay ocasiones en las que las distintas partes del mismo resultan mucho más sencillas de codificar si se utilizan diferentes lenguajes. En todo caso, para tomar esta decisión se debe hacer un estudio de la compatibilidad entre los compiladores y en definitiva de los pros y los contras de utilizar uno o varios lenguajes.

5.6 Aspectos metodológicos

En este apartado se repasan ciertos aspectos metodológicos que pueden mejorar la codificación bajo determinados puntos de vista: claridad, manejo de errores, eficiencia y transportabilidad. Un estudio más profundo de todos los aspectos de una buena metodología de programación queda fuera del alcance de este libro.

5.6.1 Normas y estilo de codificación

Cuando se inicia la fase de codificación de un proyecto es fundamental fijar las normas que deberán respetar todos los programadores para que el resultado del trabajo de todo el equipo sea completamente homogéneo. Es habitual que las empresas dedicadas al desarrollo de software tengan ya establecidas un conjunto de normas que con carácter general aplican a todos sus proyectos.

Casi todos los lenguajes permiten un formato libre en la escritura de sus sentencias e ignoran los espacios en blanco redundantes y los comentarios. Así, al mismo texto de un programa se le pueden dar distintos formatos que pueden resultar muy significativos desde el punto de vista del lector humano, al tiempo que son irrelevantes desde el punto de vista del compilador. Para la puesta a punto y sobre todo para el mantenimiento de un programa es esencial que éste resulte fácil de entender por todos, sus actuales y futuros lectores, incluyendo a su propio autor. Por tanto, se deben fijar normas concretando un estilo de codificación. No se trata de proponer el *mejor* estilo posible puesto que este sería muy difícil de establecer y provocaría muchas discusiones estériles. Lo más importante es fijar un estilo concreto y que todo el equipo lo adopte y lo respete.

Para fijar un estilo de codificación se deberán concretar al menos los siguientes puntos:

- Formato y contenido de las cabeceras de cada módulo:
 - Identificación del módulo
 - Descripción del módulo
 - Autor y fecha
 - Revisiones y fechas
 -
- Formato y contenido para cada tipo de comentario:
 - Sección
 - Orden
 - Al margen
 -
- Utilización de encolumnado
 - Tabulador = N° espacios
 - Máximo indentado
 - Formato selección
 - Formato iteración

- Elección de nombres
 - Convenio para el uso de mayúsculas y minúsculas
 - Nombres de ficheros
 - Identificadores de elementos del programa

En el libro de Fundamentos de Programación [Cerrada00] se concretan estos puntos para configurar el estilo de todos los programas escritos a lo largo del texto.

Además del estilo, en las normas se deben incluir todas aquellas restricciones o recomendaciones que puedan contribuir a mejorar la claridad del código y a simplificar su mantenimiento posterior. Por ejemplo, se pueden fijar las siguientes restricciones de carácter general:

- El tamaño máximo de las subrutinas no debe superar P páginas.
- Las subrutinas tendrán un máximo de N argumentos.
- Se deben incluir en un fichero todas las ristras de caracteres que utilice el programa. Esto facilitará la personalización y los cambios.
- Evitar el anidamiento excesivo de sentencias IF.
- ... etc. ...

Con los mismos fines, también se puede limitar el empleo de algunas sentencias del lenguaje. Por ejemplo, si se utiliza el lenguaje C para la codificación, se pueden dar normas como las siguientes:

- Evitar el empleo de goto
- No usar los operadores de autoasignación ($+=$, $-=$, $*=$, $/=$, $\%=$, $<<=$, $>>=$, $\&=$, $\wedge=$, $|=$), ya que pueden resultar confusos
- ... etc. ...

5.6.2 Manejo de errores

Durante la ejecución de un programa se pueden producir fallos que tienen como origen las más diversas causas:

- Introducción de datos incorrectos o inesperados

266 Introducción a la Ingeniería de Software

- Anomalías en el hardware (p.ej.: ruido en las comunicaciones)
- Defectos en el software (p.ej.: erratas no depuradas del programa)

Algunas de estas causas no pueden ser eliminadas completamente, por lo que si se quiere evitar su efecto indeseable habrá que introducir elementos correctores en el código del programa. En lo que sigue consideraremos los errores en el funcionamiento de un programa desde el punto de vista del software, fundamentalmente, pero antes de analizar la manera de organizar el código para atenuar o evitar errores, comenzaremos por definir de manera precisa los conceptos básicos a tener en cuenta:

Defecto: Errata o "gazapo" de software. Puede permanecer oculto durante un tiempo indeterminado, si los elementos defectuosos no intervienen en la ejecución del programa. Esto depende de los datos particulares con los que se opere en cada momento. En sistemas en tiempo real, también depende del momento preciso en que se reciban estímulos externos.

Fallo: Es el hecho de que un elemento del programa no funcione correctamente, produciendo un resultado (parcial) erróneo. Si un elemento software defectuoso interviene en una ejecución particular del programa, dará lugar normalmente a un fallo.

Error: Se define como un estado inadmisible de un programa al que se llega como consecuencia de un fallo. Típicamente consiste en la salida o almacenamiento de resultados incorrectos.

Esta terminología no se emplea siempre con la precisión requerida. Por ejemplo, en la bibliografía sobre pruebas o ensayos de programas se suele usar el término "error" como sinónimo de "defecto".

Al codificar un programa se pueden adoptar distintas actitudes respecto al tratamiento de los errores (o más precisamente, de los fallos). Analizaremos brevemente cada una de ellas:

NO CONSIDERAR LOS ERRORES:

Con esta actitud, la más cómoda desde el punto de vista de la codificación, se declina toda responsabilidad si se produce algún error. Para ello, se exigirá como requisito que todos los datos que se introduzcan deberán ser correctos y que el programa no deberá tener ningún defecto. Cuando no se cumpla alguno de estos requisitos, no será posible garantizar el resultado correcto del programa. Evidentemente, esta postura no es realista y sólo

puede ser válida para sistemas con muy baja probabilidad de fallo o muy poca trascendencia de los mismos.

PREVENCIÓN DE ERRORES:

Consiste en detectar los fallos antes de que provoquen un error. La técnica general de prevención se conoce como *programación a la defensiva* (en inglés "defensive programming") y consiste en que cada programa o subprograma esté codificado de manera que desconfíe sistemáticamente de los datos o argumentos que se le introducen, y devuelva siempre:

- a.- El resultado correcto, si los datos son válidos, o bien
- b.- Una indicación precisa del fallo, si los datos no son válidos

En el caso (b), además, el código del programa debe evitar operar con los datos incorrectos de forma que el estado después de la operación sea erróneo. Si han sido considerados todos los posibles fallos, no se puede producir ningún error en el programa.

Conviene recordar que un *error* es un estado incorrecto (o resultado incorrecto) del programa. La prevención de errores evita estos resultados incorrectos, a base de no producir resultados en caso de *fallo*. De todas maneras una ventaja principal de la programación a la defensiva es evitar la propagación de errores, facilitando así el diagnóstico de los *defectos*.

RECUPERACIÓN DE ERRORES:

Cuando no se pueden detectar todos los posibles fallos, es inevitable que en el programa se produzcan errores. En este caso se puede hacer un tratamiento del error con el objetivo de restaurar el programa en un estado correcto y evitar que el error se propague. Este tratamiento exige dos actividades diferentes y complementarias:

- 1.- DETECCIÓN del error
- 2.- RECUPERACIÓN del error

Para la detección de un error hay que concretar qué situaciones se considerarán erróneas y realizar las comprobaciones adecuadas en ciertos puntos estratégicos del programa. Por su parte, en la recuperación se tienen que adoptar decisiones sobre cómo corregir el estado del programa para llevarlo a una situación consistente. Estas decisiones pueden afectar a otras partes del programa diferentes y alejadas de aquella en la que se produce la detección del error.

268 Introducción a la Ingeniería de Software

Para la recuperación de errores, existen dos esquemas generales:

- Recuperación hacia adelante
- Recuperación hacia atrás

La *recuperación hacia adelante* trata de identificar la naturaleza o el tipo de error (p.ej.: fuera de rango, sobreexpresión, etc.), para posteriormente tomar las acciones adecuadas que corrijan el estado del programa y le permitan continuar correctamente su ejecución. Este esquema se puede programar mediante el mecanismo de *excepciones* que ya ha sido descrito en un apartado anterior.

La *recuperación hacia atrás* trata de corregir el estado del programa restaurándolo a un estado correcto anterior a la aparición del error, todo ello con independencia de la naturaleza del error. Con este esquema se necesita guardar periódicamente el último estado correcto del programa. En la nueva operación se parte de ese último estado correcto para obtener otro nuevo estado. Si ese nuevo estado es también correcto, la operación se da por terminada satisfactoriamente. En caso de error, se restaura el estado anterior y se trata de realizar la misma operación por un camino o algoritmo diferente.

Este esquema se utiliza habitualmente en los sistemas basados en transacciones. Una *transacción* es una operación que puede terminar con éxito, modificando el estado del sistema ("commit"), o con fallo, en cuyo caso la transacción se aborta ("abort") y se restaura el estado inmediatamente anterior, de manera que la operación no produce ningún efecto. Los esquemas de transacciones se usan para mantener la consistencia en las bases de datos (p.ej.: en las transacciones bancarias). En estos casos, lo fundamental es asegurar que el estado del programa sea siempre correcto aunque se queden pendientes de realizar ciertas operaciones.

En general, todos los programas que realizan una previsión o recuperación de errores se denominan genéricamente *tolerantes a fallos*.

5.6.3 Aspectos de eficiencia

Actualmente, la eficiencia en la codificación no es tan importante como lo fue en los primeros computadores. La potencia de cálculo y la cantidad de memoria disponible en los computadores actuales permite asegurar que prácticamente nunca será necesario sacrificar la claridad en la codificación

por una mayor eficiencia. La eficiencia se puede analizar desde varios puntos de vista:

- Eficiencia en memoria
- Eficiencia en tiempo

El bajo costo de la memoria hace que cuando se precisa cierto ahorro resulte suficiente con el que se puede obtener de forma automática empleando un compilador que disponga de posibilidades de compresión de memoria. Por otro lado, cuando el volumen de información a manejar sea excesivamente grande para la memoria disponible, será durante la fase de diseño detallado cuando se deban estudiar las distintas alternativas y optar por el algoritmo que optimice más la utilización de la memoria.

La eficiencia en tiempo adquiere su mayor importancia en la codificación de sistemas para tiempo real con plazos muy críticos. En ocasiones una mayor eficiencia en tiempo se logra disminuyendo la eficiencia en memoria. Por ejemplo, cuando un cálculo es muy complejo y no hay suficiente tiempo para llevarlo a cabo se puede adoptar como solución tabular dicho cálculo y simplemente consultarla cada vez que se necesite.

La primera vía para conseguir un ahorro de tiempo importante es realizar en la fase de diseño detallado un estudio exhaustivo de las posibles alternativas del problema y adoptar el algoritmo más rápido. Aunque existen compiladores capaces de optimizar el código que generan para aumentar su eficiencia en tiempo, las mejoras que se pueden obtener son difíciles de prever. Existen formas bastante simples de obtener un ahorro de tiempo significativo utilizando técnicas de codificación tales como las siguientes:

- Tabular los cálculos complejos según se mencionó anteriormente.
- Expansión en línea: Si se emplean macros en lugar de subrutinas se ahorra el tiempo necesario para la transferencia de control y el paso de argumentos.
- Desplegado de bucles: En la evaluación de la condición de un bucle se emplea un tiempo que se puede ahorrar repitiendo el código de forma sucesiva. Por ejemplo, si se repite 10 veces seguidas el código interno del bucle, las evaluaciones se reducen a la décima parte.
- Simplificar las expresiones aritméticas y lógicas.
- Sacar fuera de los bucles todo lo que no sea necesario repetir.
- Utilizar estructuras de datos de acceso rápido (p.ej.: vectores en lugar de listas con encadenamiento)

270 Introducción a la Ingeniería de Software

- Evitar las operaciones en coma flotante y realizarlas preferiblemente en coma fija.
- Evitar las conversiones innecesarias de tipos de datos.
- ... etc. ...

5.6.4 Transportabilidad de software

Realizar una aplicación transportable implica un esfuerzo adicional, que en muchos casos se rentabiliza rápidamente. La transportabilidad permite usar el mismo software en distintos computadores actuales y futuros. Por tanto, la transportabilidad no sólo es rentable a corto plazo para aprovechar el mismo software en distintos computadores sino también a medio y largo plazo para facilitar el mantenimiento/adaptación de la aplicación a las nuevas arquitecturas y prestaciones de los computadores.

Como factores esenciales de la transportabilidad se pueden destacar los siguientes:

- Utilización de estándares
- Aislar las peculiaridades

Un producto software desarrollado exclusivamente sobre *elementos estándar* (lenguaje, base de datos, librerías gráficas, etc.) es teóricamente transportable sin ningún cambio, al menos entre plataformas que cuenten con el soporte apropiado de dichos estándares. La falta de estándares es uno de los problemas que dificulta la transportabilidad. En este caso, se deben utilizar lenguajes, compiladores, bases de datos y herramientas en general, que tengan una amplia difusión y que se puedan considerar estándares "de facto". De todos ellos, se procurarán evitar aquellos elementos no consolidados por completo y que puedan estar sujetos a futuros cambios o revisiones.

La mejor manera de *aislar las peculiaridades* es destinar un módulo específico para cada una de ellas. El transporte se resolverá recodificando y adaptando solamente estos módulos específicos al nuevo computador. Las peculiaridades fundamentales de los computadores suelen estar vinculadas a los elementos siguientes:

ARQUITECTURA DEL COMPUTADOR:

La arquitectura del computador determina la longitud de palabra (8, 16, 32 ó 64 bits) y de esto se derivan la representación interna de los valores enteros y reales. Cuando no se desbordan los rangos o precisiones del computador no resulta muy compleja la transportabilidad debido a que será

el compilador el encargado de tener en cuenta todos los detalles. Aunque afortunadamente este problema es cada día menos frecuente, la cosa se complica cuando se llega a desbordar la capacidad de la longitud de palabra del computador. Para resolver esto es necesario definir un nuevo tipo abstracto de dato con el rango o precisión ampliados y crear para él todas las operaciones necesarias mediante funciones. En Ada el nombre de estas funciones pueden ser directamente operadores (+, -, *, /, ...). Inevitablemente, la implementación de estos nuevos tipos abstractos será bastante ineficiente respecto a los tipos propios del computador realizados por hardware.

La tabla de códigos de caracteres utilizados es otra causa de problemas. Actualmente, prácticamente todos los computadores utilizan la tabla ASCII. Sin embargo, para facilitar la transportabilidad lo mejor es no aprovechar nunca en la codificación el orden de los caracteres en una tabla concreta.

SISTEMA OPERATIVO:

Los lenguajes incorporan de un modo u otro el acceso a servicios del sistema operativo para realizar tareas como las siguientes:

- Entrada/salida (teclado, pantalla, ratón, etc.)
- Manejo de ficheros (abrir, leer, escribir, etc.)
- Manejo de librerías (numéricas, utilidades, etc.)

En muchas aplicaciones es inevitable hacer uso de algunas o todas estas facilidades que son específicas de cada sistema operativo. Los lenguajes de alto nivel disponen de procedimientos o funciones predefinidos para la realización de las operaciones más elementales dentro de estas tareas y siempre que sea posible deben ser las únicas que se utilicen. En algunos compiladores concretos, el nivel de sofisticación de estas operaciones puede ser mayor y esto simplificará bastante la codificación. Sin embargo, su transportabilidad será bastante incierta dado que son operaciones que no se encontrarán en todos los compiladores con carácter general.

Lo habitual es que se necesiten operaciones más complejas y particularizadas que las predefinidas en los lenguajes. En este caso, se deben concretar y especificar claramente cada una de ellas. Estas nuevas operaciones se agruparán en módulos de entrada/salida, para manejo de ficheros, librerías matemáticas, etc. propios de cada aplicación. Para cada módulo y operación se definirá una interfaz única y precisa en toda la aplicación. El resto de la aplicación utilizará estos módulos de forma independiente del sistema operativo. En la implementación de estos módulos se podrán utilizar las operaciones disponibles en el lenguaje y el

sistema operativo. Para transportar la aplicación a otro sistema operativo sólo será necesario realizar una nueva implementación de estos módulos a partir del nuevo sistema operativo y sus operaciones más o menos sofisticadas.

5.7 Técnicas de prueba de unidades

A lo largo de la fase de codificación se introducen de manera inadvertida múltiples errores de todo tipo e incorrecciones respecto a las especificaciones del proyecto. Todo ello debe ser detectado y corregido antes de entregar al cliente el programa acabado. Como sucede con cualquier otro producto (mecánico, electrónico, etc.), para garantizar su calidad es necesario someter al programa a diversas pruebas destinadas a detectar los errores o verificar su funcionamiento correcto. Según la utilización final del programa, las pruebas pueden ser más o menos exhaustivas. Para un software crítico (aeronáutica, nuclear, automoción, etc.), el costo de las pruebas puede ser la partida más importante del costo de todo el desarrollo.

Para evitar el caos de una prueba global única, se deben hacer pruebas a cada unidad o módulo según se avanza en la codificación del proyecto. Esto facilitará enormemente las posteriores pruebas de integración entre módulos y las pruebas del sistema total que deben realizarse en cualquier caso. En este apartado se describen ciertas técnicas para realizar las pruebas de las unidades.

5.7.1 Objetivos de las pruebas de software

Aunque pueda resultar paradójico, el principal objetivo de las pruebas debe ser conseguir que el programa funcione incorrectamente y que se descubran sus defectos. Esto exige elaborar minuciosamente un juego de casos de prueba destinados a someter al programa a un máximo número posible de situaciones diferentes. Para elaborar los casos de prueba se debe tener en cuenta lo siguiente:

- Una buena prueba es la que encuentra errores y no los encubre
- Para detectar un error es necesario conocer cuál es el resultado correcto
- Es bueno que no participen en la prueba el codificador o diseñador
- Siempre hay errores, y si no aparecen se deben diseñar pruebas mejores
- Al corregir un error se pueden introducir otros nuevos

- Es imposible demostrar la ausencia de defectos mediante pruebas

Probar completamente cada módulo es inabordable y además no resulta rentable ni práctico. Como se muestra en la figura 5.1, con las pruebas sólo se explora una parte de todas las posibilidades del programa. Se trata de alcanzar un compromiso para que con el menor esfuerzo posible se puedan detectar el máximo número de defectos y sobre todo aquellos que puedan provocar las más graves consecuencias.

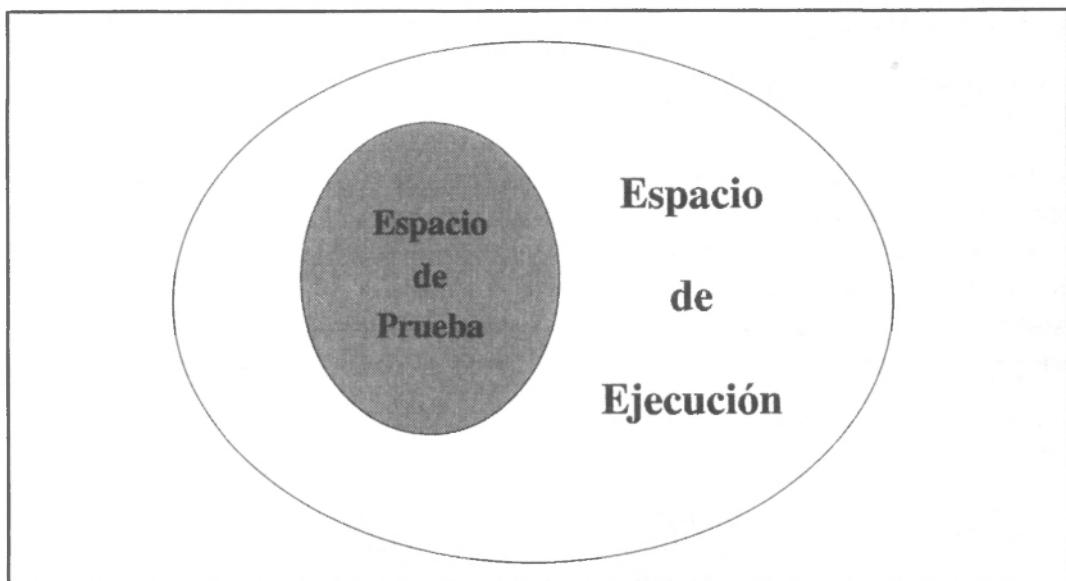


Figura 5.1. Dominio de las pruebas

Para garantizar unos resultados fiables, además del juego de casos de prueba, es esencial que todo el proceso de prueba se realice de la manera más automática posible. Esto exige crear un entorno de prueba que asegure unas condiciones predefinidas y estables para las sucesivas pasadas, que necesariamente se deben efectuar después de corregir los errores detectados en cada pasada anterior.

Las pruebas de unidades se realizan en un entorno de ejecución controlado, que puede ser diferente del entorno de ejecución del programa final en explotación. El entorno deberá proporcionar, al menos, un informe con los resultados de las pruebas y un registro de todos los errores detectados con su discrepancia respecto al valor esperado. A veces para establecer el entorno de pruebas serán suficientes las utilidades del sistema operativo preparando en un fichero los casos de prueba y recogiendo en otro fichero los resultados obtenidos que se compararán posteriormente con los esperados. Si se necesita un entorno más sofisticado que registre tiempos u

274 Introducción a la Ingeniería de Software

otros parámetros de la prueba será necesario desarrollar el correspondiente programa.

Las técnicas de prueba de unidades responden a dos estrategias fundamentales:

- Pruebas de "caja negra"
- Pruebas de "caja transparente"

A continuación se describen las técnicas principales aplicables en cada caso.

5.7.2 Pruebas de caja negra

Según muestra la figura 5.2, la estrategia de *caja negra* ("black box") ignora por completo la estructura interna del programa y se basa exclusivamente en la comprobación de la especificación entrada-salida del software. Se trata de verificar que todos los requisitos impuestos al programa, como a cualquier otro producto, se cumplen. Esta es la única estrategia que puede adoptar el cliente o cualquier persona ajena al desarrollo del programa.

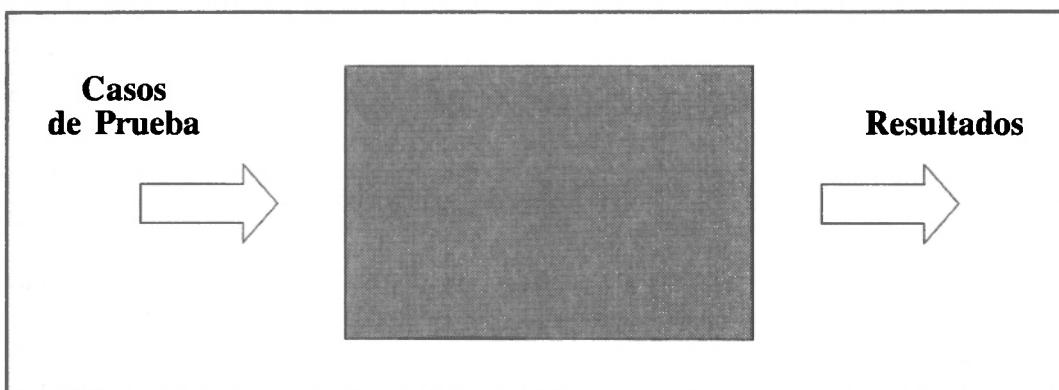


Figura 5.2. Pruebas de Caja Negra

La elaboración de unos buenos casos de prueba que permitan conocer el correcto funcionamiento de la caja negra no resulta trivial, y para ello se deben emplear todos los métodos que estén a nuestro alcance. Esta tarea requiere cierta dosis de ingenio y hay personas mejor capacitadas que otras para llevarla a cabo. Como si se tratara de un juego, el objetivo es descubrir los errores o incorrecciones del módulo "sospechoso" y para ello hay que diseñar un "interrogatorio" amplio y coherente.

Existen ciertos métodos basados fundamentalmente en la experiencia que ayudan bastante en la elaboración de casos de prueba. Todos ellos se

pueden y se deben utilizar de forma conjunta y complementaria. Algunos de los usados más ampliamente son los siguientes:

PARTICIÓN EN CLASES DE EQUIVALENCIA:

Según se muestra en la figura 5.3 se trata de dividir el espacio de ejecución del programa en varios subespacios. Cada subespacio o clase equivalente agrupa a todos aquellos datos de entrada al módulo que resultan equivalentes desde el punto de vista de la prueba de caja negra. La equivalencia podría corresponder intuitivamente a que el algoritmo de cálculo, tal como se describe externamente, siga los mismos pasos en su ejecución. Por ejemplo, si estamos probando una función que realiza la raíz cuadrada será suficiente con probar cualquier número positivo, el cero y cualquier número negativo o tal vez sea más adecuado probar con un cuadrado perfecto positivo, un valor positivo sin raíz exacta, el cero, un cuadrado perfecto negativo y un valor negativo arbitrario. De una forma tenemos tres clases equivalentes y de otra cinco.

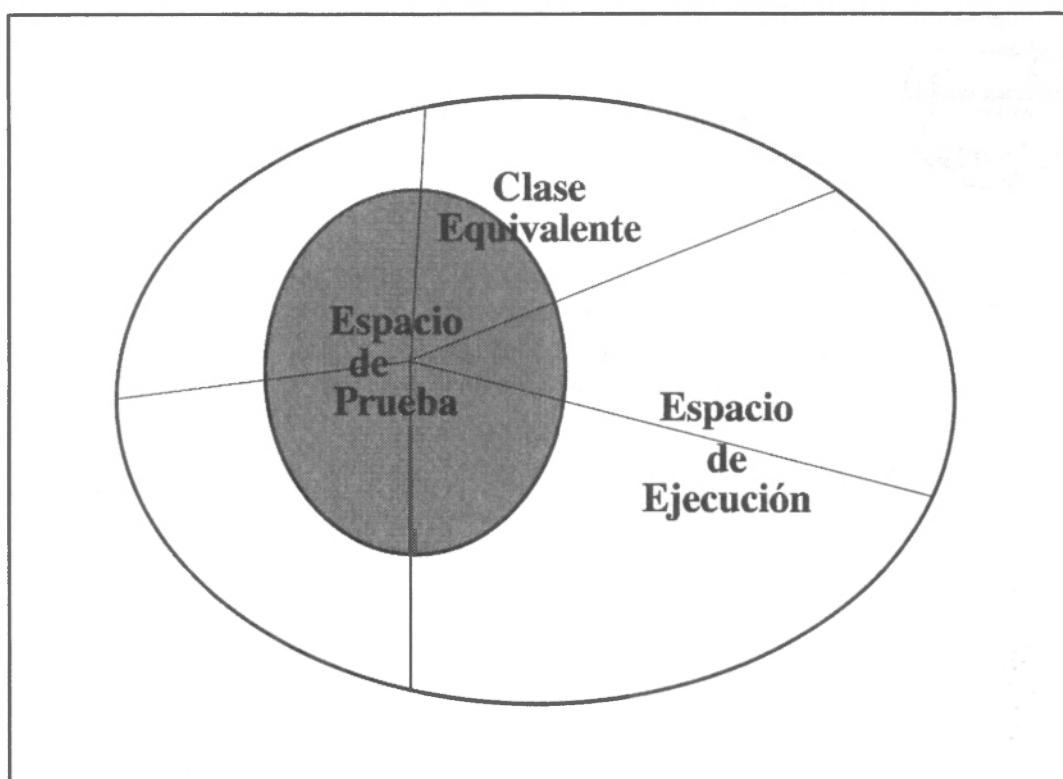


Figura 5.3. Clases de equivalencia

Los pasos que se deben seguir con este método son los siguientes:

1. Determinar las clases equivalentes apropiadas

276 Introducción a la Ingeniería de Software

2. Establecer pruebas para cada clase de equivalencia: Se deben proponer casos o datos de pruebas válidos e inválidos para cada una de las clases definidas en el paso anterior.

Con este método, si las clases se eligen adecuadamente, se reduce bastante el número de casos que se necesitan para descubrir un defecto. Según cómo se caractericen las clases de equivalencia se pueden emplear las siguientes directrices:

A. Rango de valores:

Ejemplo:	0 <= edad < 120 años
Casos válidos:	1 dentro del rango 33 años
Casos inválidos:	1 mayor y 1 menor -5 y 132 años

B. Valor específico:

Ejemplo:	Clave = OCULTA
Casos válidos:	1 igual OCULTA
Casos inválidos:	1 distinto Otra578

C. Conjunto de valores:

Ejemplo:	Operaciones = compra, venta, cambio
Casos válidos:	1 por elemento compra, venta, cambio
Casos inválidos:	1 fuera del conjunto regalo

Hay que tener en cuenta que un caso de prueba válido para una clase puede ser también un caso de prueba inválido para otra y asimismo puede ocurrir que un caso de prueba bien elegido sea válido para varias clases. Así, para la elaboración del juego de casos de pruebas se pueden refinar los pasos indicados anteriormente:

1. Definir las clases equivalentes
2. Definir una prueba que cubra tantos casos válidos como sea posible de cualquier clase
3. Marcar las clases cubiertas y repetir el paso anterior hasta cubrir todos los casos válidos de todas las clases
4. Definir una prueba específica para cada caso inválido

ANÁLISIS DE VALORES LÍMITE:

Muchos programas se construyen codificando primero un tratamiento general, y retocando luego el código para cubrir casos especiales. Por esta y otras razones es bastante normal que los errores tengan cierta tendencia

a aparecer precisamente al operar en las fronteras o valores límite de los datos normales.

El método de *análisis de valores límite* (en inglés "boundary analysis"), como se muestra en la figura 5.4, hace un especial hincapié en las zonas del espacio de ejecución que están próximas al borde. Este método es complementario del anterior y también en éste se deben proponer casos válidos y casos inválidos. Por ejemplo, para el rango de edades indicado en el apartado anterior, los casos de prueba serían:

Límite inferior:	-1	0	1
Límite superior:	119	120	121

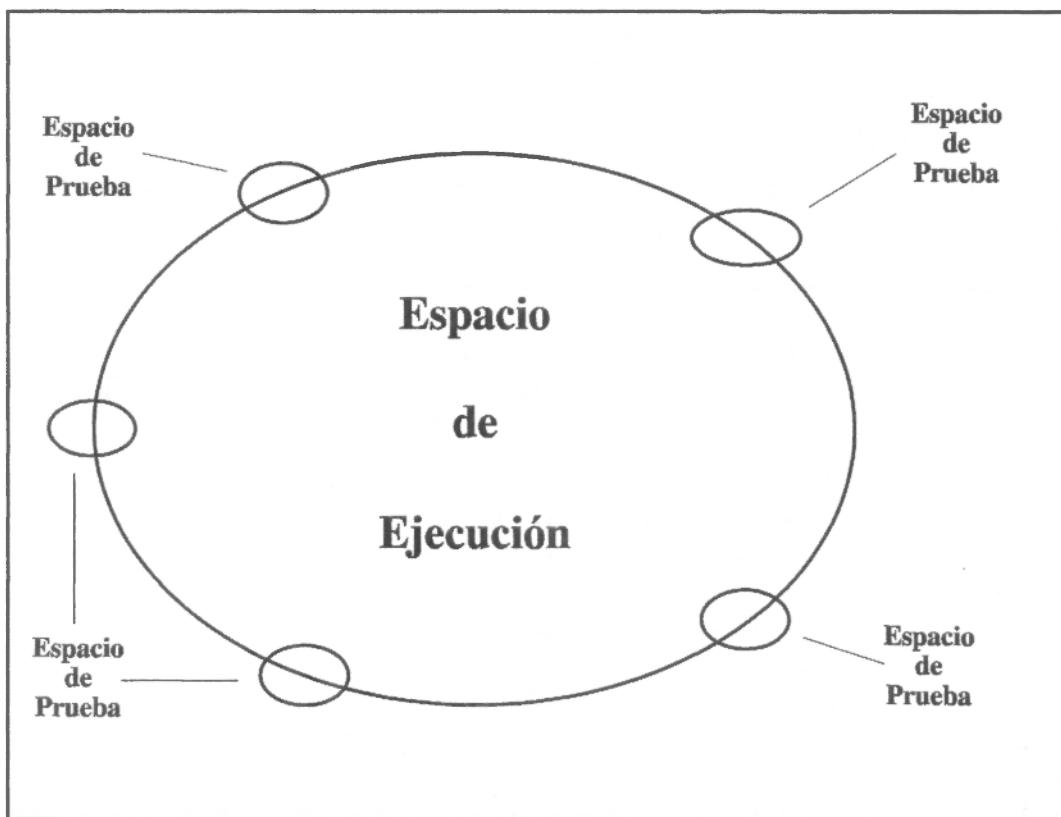


Figura 5.4. Pruebas de valores límite

Los errores en los límites pueden tener consecuencias más graves de lo normal debido a que pueden provocar la necesidad de unos recursos extra que no estaban previstos. Por ejemplo, si un sistema de control de una central nuclear debe poder atender 20 alarmas simultáneamente, en las pruebas se podrán provocar 21, 22 e incluso 30 alarmas y comprobar que

278 Introducción a la Ingeniería de Software

se detecta esta situación y se avisa al operador sin abortar el programa y producir la parada del sistema por violación de memoria.

En este caso, las directrices a seguir en la elaboración de casos de prueba serían las siguientes:

1. Entradas: Probar con los mismos valores límite y justo fuera de límites. Por ejemplo, si la precisión en los cálculos = 5 cifras, probar 5 y 6
2. Salidas: Probar con los mismos valores límite y justo fuera de límites. Por ejemplo: Para listados con Nº líneas/página = 70, probar 70 y 71
3. Memoria: Probar con tamaños nulos, límite y superior al límite de todas las estructuras de información. Por ejemplo, para pilas, colas, tablas, etc., probar los casos vacío, lleno y sobrelleno (un elemento más de lleno)
4. Recursos: Probar con ningún recurso, límite de recursos y superior al límite. Por ejemplo, si máximo Nº de terminales = 30, probar 0, 30 y 31
5. Otros: Pensar en otras situaciones límite y establecer las pruebas

Este método también se puede utilizar con la estrategia de caja transparente en la medida que se conozcan las estructuras internas del programa.

COMPARACIÓN DE VERSIONES:

Cuando una unidad o módulo es especialmente crítico se puede hacer un desarrollo *multi-versión* (en inglés "N-version") encargando la codificación de diferentes versiones a distintos programadores. Todos ellos utilizarán las mismas especificaciones de partida y deberían obtener módulos completamente intercambiables. Sin embargo, esto no suele ocurrir debido a variaciones en los algoritmos empleados, diferencias de matiz en la interpretación de la especificación, diferentes precisiones, etc.

Por otro lado, se elaborará un juego de casos de prueba mediante los métodos habituales. Hay que tener en cuenta que no siempre se conocen de forma completamente exacta los resultados esperados y que en algunos casos, como sucede en los sistemas de tiempo real, resulta casi imposible conocer a priori cuáles serán esos resultados.

A continuación, se someterán todas las versiones al mismo juego de casos de prueba de una forma completamente automática. Los resultados de las distintas versiones se comparan entre ellos y con los esperados. Cualquier

discrepancia entre las distintas versiones se debe analizar hasta discernir si una versión es errónea y sus causas. Cuando todas las versiones produzcan los mismos resultados y éstos coincidan con los deseados se puede suponer que todas son equivalentes y correctas, y se puede utilizar cualquiera de ellas.

La redundancia que se introduce con la codificación de varias versiones aumenta las garantías de que el módulo funciona correctamente y que cumple las especificaciones. Sin embargo, no es un criterio infalible puesto que un error en la especificación se trasladará a todas las versiones.

EMPLEO DE LA INTUICIÓN:

Como ya ha sido comentado, la elaboración de pruebas requiere ingenio. En este sentido, es muy importante dedicar cierto tiempo a preparar pruebas que planteen situaciones especiales y que puedan provocar algún error. Para ello, las personas ajenas al desarrollo del módulo suelen aportar un punto de vista mucho más distante y fresco que las que participan en él. En esta forma de trabajo es fundamental la intuición aunque ésta suele ir muy ligada a la experiencia previa en otras situaciones similares.

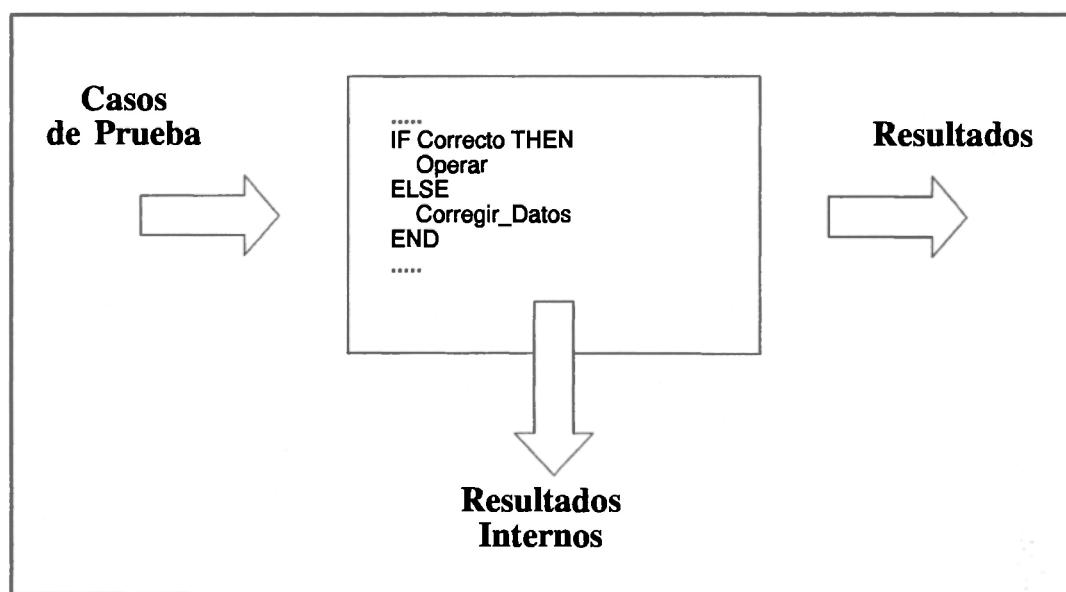


Figura 5.5. Pruebas de caja transparente

5.7.3 Pruebas de caja transparente

Según se muestra en la figura 5.5, en la estrategia de prueba de *caja transparente* ("white box", "clear box", "glass box") se conoce y se tiene en

280 Introducción a la Ingeniería de Software

cuenta la estructura interna del módulo. En la elaboración de los casos de prueba se trata de conseguir que el programa transite por todos los posibles caminos de ejecución y ponga en juego todos los elementos del código. Por tanto, los casos de prueba deben conseguir que:

- Todas las decisiones se ejecuten en uno y otro sentido
- Todos los bucles se ejecuten en los supuestos más diversos posibles
- Todas las estructuras de datos se modifiquen y consulten alguna vez

En principio puede parecer que las únicas pruebas realmente necesarias son las que sirven para demostrar el funcionamiento según las especificaciones y esto se puede lograr con una estrategia de caja negra. Sin embargo, un programa es una entidad de una complejidad muy superior a la del más sofisticado artilugio mecánico, hidráulico, electrónico, etc., para los que sí suele ser suficiente con someterlos a pruebas de caja negra. Como ejemplo se puede decir que un sencillo bucle que se pueda ejecutar entre 0 y 10 veces y que tenga en su interior sólo 5 caminos diferentes se puede ejecutar de casi 10 millones de formas distintas.

Es cierto que en la mayoría de los programas sólo se llegan a ejecutar un número bastante reducido de todos los caminos posibles. Pero también es cierto que no resulta fácil prever a priori qué caminos son los que se ejecutarán y cuáles no. Cuando se elaboran las pruebas de caja negra pueden quedar perfectamente inexplorados caminos que en un funcionamiento habitual no serán muy frecuentados pero que sí son decisivos en situaciones concretas y que si no han sido probados convenientemente pueden producir un error en el peor momento.

Las pruebas de caja negra y de caja transparente deben ser complementarias y nunca excluyentes. De hecho es conveniente aplicar el método de análisis de valores límite para elaborar pruebas de caja transparente teniendo en cuenta la estructura del programa. Está claro que las pruebas de caja transparente deben ser propuestas por alguien que haya participado o conozca la codificación en detalle. En este caso, los métodos más ampliamente utilizados son los siguientes:

CUBRIMIENTO LÓGICO:

No parece muy razonable proponer casos de prueba para conseguir que un programa se ejecute de todos los billones o trillones de formas posibles, pero al menos debemos conseguir cierto *cubrimiento lógico* de todas las secciones de código y no dejar ninguna sin ejecutar alguna vez.

Dado un fragmento de código como el que muestra el diagrama de flujo de la figura 5.6, denominaremos *camino básico* a cualquier recorrido que siguiendo las flechas de las líneas de flujo nos permita ir desde el punto inicial (0) al punto final (9) del diagrama. Se utiliza aquí un diagrama de flujo por su carácter gráfico, pero los razonamientos se pueden trasladar de forma inmediata a un fragmento de código escrito en cualquier lenguaje de programación. Cada rombo del diagrama debe representar un predicado lógico simple, esto es, no puede ser una expresión lógica con algún operador OR, AND, etc. Hay que tener en cuenta que cualquier expresión lógica siempre se puede representar utilizando un rombo por cada predicado lógico simple.

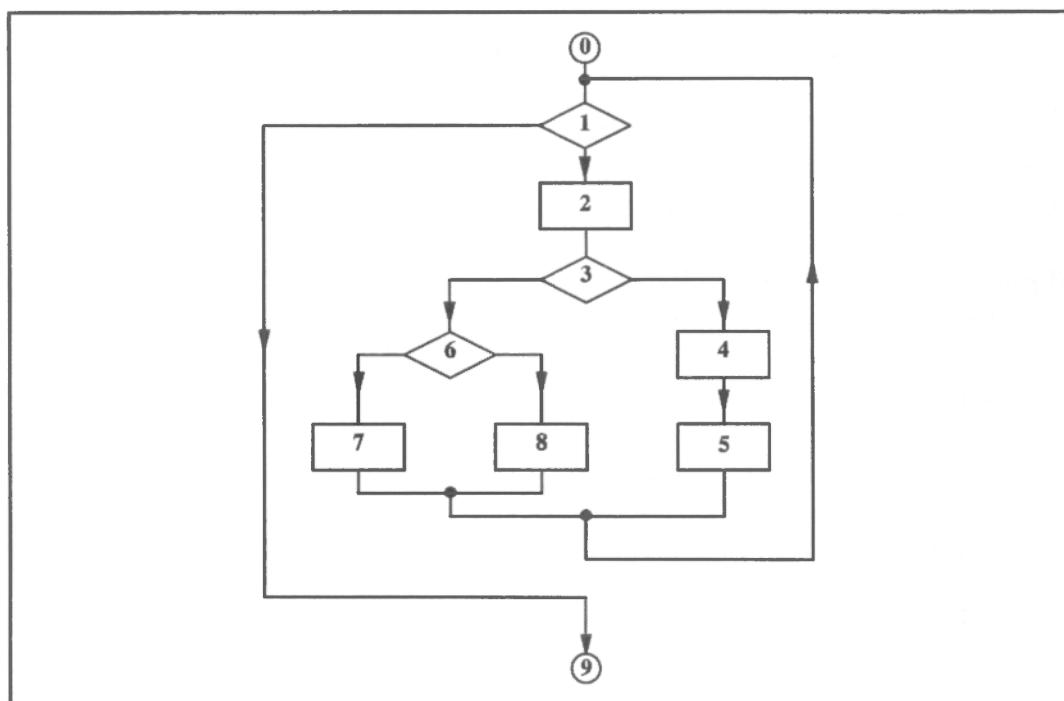


Figura 5.6. Diagrama de flujo con 3 predicados lógicos simples

Primeramente, se trata de determinar un conjunto de caminos básicos que recorran todas las líneas de flujo del diagrama alguna vez. Como máximo, el número de caminos básicos necesarios vendrá determinado por el número de predicados lógicos simples o rombos que tenga el diagrama de flujo de acuerdo con la siguiente fórmula:

$$\text{Nº máximo de caminos} = \text{Nº predicados} + 1$$

282 Introducción a la Ingeniería de Software

En el caso de la figura 5.6 tenemos:

$$\text{Nº máximo de caminos} = 3 + 1 = 4$$

y para este ejemplo serían suficientes los siguientes cuatro caminos básicos:

- Camino 1: 0 - 1 - 9
- Camino 2: 0 - 1 - 2 - 3 - 4 - 5 - 1 - 9
- Camino 3: 0 - 1 - 2 - 3 - 6 - 8 - 1 - 9
- Camino 4: 0 - 1 - 2 - 3 - 6 - 7 - 1 - 9

Existen otros caminos, tales como:

$$\text{Camino 5: } 0 - 1 - 2 - 3 - 4 - 5 - 1 - 2 - 3 - 6 - 7 - 1 - 9$$

pero ninguno de ellos utiliza alguna línea de flujo nueva no utilizada previamente por los cuatro primeros. Lo que sí es posible es sustituir un camino por otro siempre que con el conjunto se recorran todas las líneas.

El método del cubrimiento lógico consiste en elaborar casos de prueba para que el programa recorra un determinado conjunto de caminos siguiendo ciertas pautas. Para ello, se pueden establecer distintos niveles de cubrimiento:

NIVEL I: Se trata de elaborar casos de pruebas para que se ejecuten al menos una vez todos los caminos básicos, cada uno de ellos por separado.

NIVEL II: Se trata de elaborar casos de prueba para que se ejecuten todas las combinaciones de caminos básicos por parejas. En el ejemplo de la figura 5.7 serían necesarios 7 caminos para probar las combinaciones por parejas. Estos caminos podrían ser:

- Camino 1: 0 - 1 - 11
- Camino 2: 0 - 1 - 2 - 3 - 5 - 6 - 7 - 1 - 11
- Camino 3: 0 - 1 - 2 - 3 - 5 - 6 - 8 - 1 - 11
- Camino 4: 0 - 1 - 2 - 3 - 5 - 9 - 10 - 1 - 11
- Camino 5: 0 - 1 - 2 - 4 - 5 - 6 - 7 - 1 - 11
- Camino 6: 0 - 1 - 2 - 4 - 5 - 6 - 8 - 1 - 11
- Camino 7: 0 - 1 - 2 - 4 - 5 - 9 - 10 - 1 - 11

NIVEL III: Se trata de elaborar casos de prueba para que se ejecuten un número significativo de las combinaciones posibles de caminos.

Como se ha comentado, cubrir todas las combinaciones posibles resulta inabordable.

Como mínimo las pruebas de cada módulo deben garantizar el nivel I. Según los distintos autores, con el cubrimiento lógico se pueden quedar sin detectar el 50 % de los errores y es necesario utilizar otros métodos. En concreto, nunca se podrá detectar la falta de un fragmento de código.

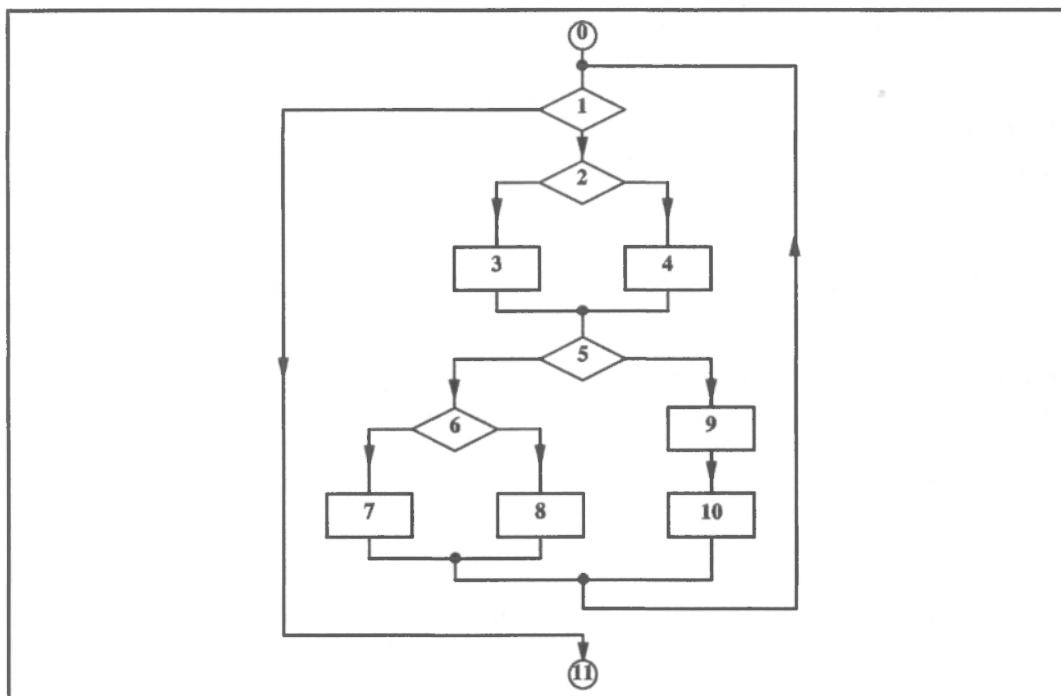


Figura 5.7. Diagrama de flujo con 4 predicados lógicos simples

PRUEBAS DE BUCLES:

Los bucles constituyen un elemento esencial en cualquier programa y se debe prestar especial atención a la elaboración de pruebas para ellos. Con este fin, distinguiremos entre las siguientes situaciones:

Bucle con número no acotado de repeticiones. Se elaborarán pruebas para:

- Ejecutar el bucle 0 veces
- Ejecutar el bucle 1 vez
- Ejecutar el bucle 2 veces
- Ejecutar el bucle un número moderado de veces
- Ejecutar el bucle un número elevado de veces

284 Introducción a la Ingeniería de Software

Bucle con número máximo (M) de repeticiones. Se elaborarán pruebas para:

- Ejecutar el bucle 0 veces
- Ejecutar el bucle 1 vez
- Ejecutar el bucle 2 veces
- Ejecutar el bucle un número intermedio de veces
- Ejecutar el bucle M-1 veces
- Ejecutar el bucle M veces
- Ejecutar el bucle M+1 veces

Bucles anidados. El número de pruebas crece de forma geométrica con el nivel de anidamiento; para reducir este número se utilizará la siguiente técnica:

1. Ejecutar todos los bucles externos en su número mínimo de veces para probar el bucle más interno con el algoritmo de bucle que corresponda.
2. Para el siguiente nivel de anidamiento, ejecutar los bucles externos en su número mínimo de veces y los bucles internos un número típico de veces, para probar el bucle del nivel con el algoritmo de bucle que corresponda.
3. Repetir el paso 2 hasta completar todos los niveles.

Bucles concatenados. Si son independientes se probarán cada uno por separado con alguno de los criterios anteriores. Si están relacionados (p.ej.: el índice final de uno es el inicial del siguiente), se empleará un enfoque similar al indicado para los bucles anidados.

EMPLEO DE LA INTUICIÓN:

También con la estrategia de caja transparente merece la pena dedicar un cierto tiempo a elaborar pruebas que sólo por intuición podemos estimar que plantearán situaciones especiales. Es obvio que para ello es necesario conocer muy en detalle la estructura del módulo y tener alguna experiencia previa.

5.7.4 Estimación de errores no detectados

Aunque sea constatable que no aparecen nuevos errores y haya sido considerable el esfuerzo empleado con las pruebas más diversas y sofisticadas, resulta imposible demostrar que un módulo no tiene defectos. Este hecho resulta poco tranquilizador, por lo que conviene obtener alguna

estimación estadística de las erratas que pueden permanecer todavía sin ser detectadas.

Desde luego, el juego de pruebas sólo ejercita el módulo en una parte de sus posibilidades y siempre pueden quedar situaciones sin explorar. Para tener una estimación del número de defectos que quedan sin detectar se puede utilizar la siguiente estrategia:

1. Anotar el número de errores que se producen inicialmente con el juego de casos de prueba: E_I (p.ej.: $E_I = 56$).
2. Corregir el módulo hasta que no tenga ningún error con el mismo juego de casos de prueba.
3. Introducir aleatoriamente en el módulo un número razonable de errores: E_A (p.ej.: $E_A = 100$) en los puntos más diversos. Esta labor deberá ser realizada por alguien que no conozca el juego de casos de prueba.
4. Someter el módulo con los nuevos errores al juego de casos de prueba y hacer de nuevo el recuento del número de errores que se detectan: E_D (p.ej.: $E_D = 95$).
5. Para el juego de casos de prueba considerado, suponiendo que se mantiene la misma proporción estadística, el porcentaje de errores sin detectar será el mismo para los errores iniciales que para los errores deliberados. Por tanto, el número estimado de errores sin detectar será:

$$E_E = (E_A - E_D) * (E_I / E_D) = (100 - 95) * (56/95) \approx 3 \text{ errores}$$

Dependiendo de lo crítico que resulte el software y del resultado obtenido con esta estrategia, se debe estudiar la conveniencia de elaborar nuevos casos de prueba.

5.8 Estrategias de integración

Las unidades o módulos de un producto software se han de integrar para conformar el sistema completo. Desgraciadamente, en esta fase de integración también aparecen nuevos errores debidos a las más diversas causas:

- Desacuerdos en la interfaz
- Interacción indebida entre módulos

286 Introducción a la Ingeniería de Software

- Imprecisiones acumuladas
- ... etc.

Durante la fase de integración se debe proceder en forma sistemática, siguiendo una estrategia bien definida, para facilitar la depuración de los errores que vayan surgiendo. Entre las estrategias básicas de integración tenemos:

- Integración "Big Bang"
- Integración descendente (en inglés "Top-Down")
- Integración ascendente (en inglés "Bottom-Up")

Estas estrategias pueden emplearse independientemente o en forma combinada.

5.8.1 Integración *Big Bang*

Esta estrategia consiste en realizar la integración de todas las unidades en un único paso. Como es fácil suponer la cantidad de errores que aparecen de golpe puede hacer casi imposible la identificación de los defectos que los causan, provocando un auténtico caos. Sólo para sistemas muy pequeños se puede justificar la utilización de esta estrategia. La ventaja fundamental es que se evita la realización del software de "andamiaje" que se requiere con las otras dos estrategias progresivas.

5.8.2 Integración descendente

Como se muestra en la figura 5.8, en esta estrategia de integración se parte inicialmente del módulo principal (Módulo P) que se prueba con módulos de "andamiaje" o sustitutos (en inglés "stubs") de los otros módulos usados directamente (Sust. A, ...). Los módulos sustitutos se van reemplazando, uno por uno, por los verdaderos y se realizan las pruebas de integración correspondientes. La forma en que se produce la sustitución e integración de los módulos verdaderos puede ser estrictamente por niveles, por ramas o una mezcla, según vayan estando disponibles.

La codificación de los sustitutos es un trabajo adicional que conviene simplificar al máximo y para el que se pueden adoptar distintas soluciones:

- No hacer nada y que sólo sirva para comprobar la interfaz
- Imprimir un mensaje de seguimiento con información de la llamada
- Suministrar un resultado fijo
- Suministrar un resultado aleatorio

- Suministrar un resultado tabulado u obtenido con un algoritmo simplificado

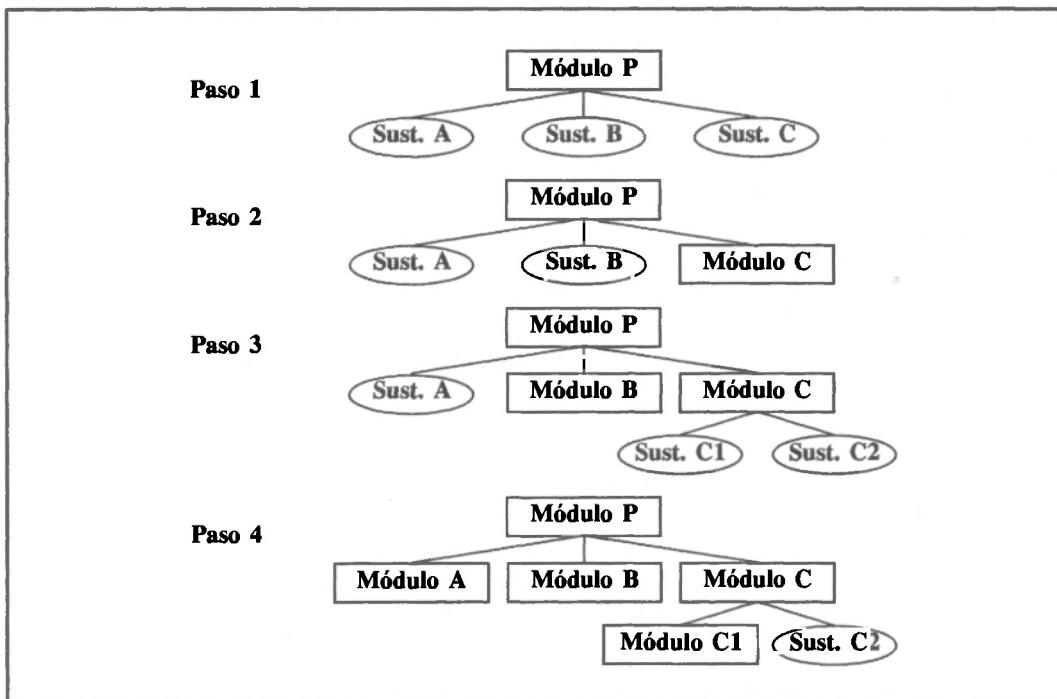


Figura 5.8. Integración descendente

La ventaja fundamental de la integración descendente es que se ven desde el principio las posibilidades de la aplicación. Esto permite mostrar muy pronto al cliente un prototipo sencillo y discutir sobre él posibles mejoras o modificaciones.

Los inconvenientes más importantes son:

1. La integración estrictamente descendente limita en cierta forma el trabajo en paralelo.
2. Al conducir la integración de los nuevos módulos desde otros módulos ya integrados y definitivos se tienen bastantes limitaciones para hacer pruebas especiales o dirigidas a un objetivo específico. Para lograr esto es necesario desarrollar nuevos módulos o hacer una integración híbrida ascendente-descendente.

5.8.3 Integración ascendente

En la estrategia de integración ascendente se empieza por codificar por separado y en paralelo todos los módulos de nivel más bajo. Para probarlos

288 Introducción a la Ingeniería de Software

se escriben módulos gestores o conductores (en inglés "drivers") que los hacen funcionar independientemente o en combinaciones sencillas. Por ejemplo, según se muestra en la figura 5.9, el Gestor A es el encargado de realizar las pruebas de integración entre los módulos A1 y A2 antes de que se tenga disponible el módulo definitivo A. Los Gestores se van sustituyendo uno a uno por los módulos de mayor nivel según se van codificando, al tiempo que se van desarrollando nuevos gestores si hace falta. En este caso, el orden de sustitución puede ser cualquiera salvo para los últimos pasos en que se necesita que todos los módulos inferiores estén disponibles.

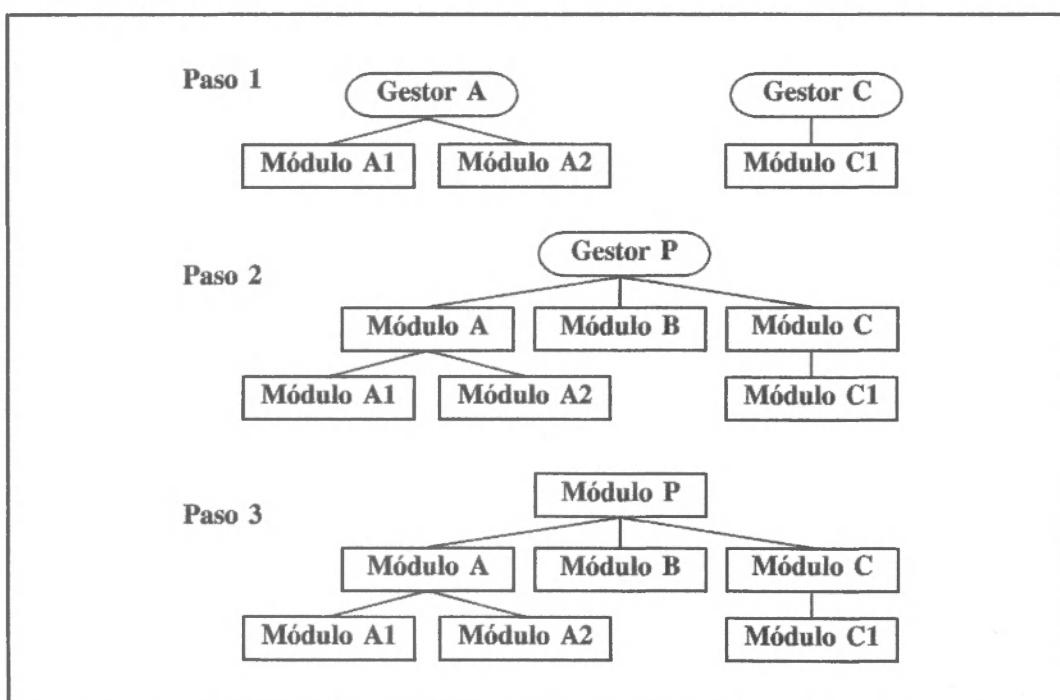


Figura 5.9. Integración Ascendente

Aunque en algunos casos se puede prescindir de los Gestores, su interés radica fundamentalmente en su capacidad para probar de forma explícita situaciones especiales o infrecuentes.

Las ventajas de la integración ascendente coinciden con los inconvenientes de la integración descendente:

- Facilita el trabajo en paralelo
- Facilita el ensayo de situaciones especiales

Por el contrario, el inconveniente más importante es que resulta difícil ensayar el funcionamiento global del producto software hasta el final de su integración.

La mejor solución es utilizar una integración ascendente con los módulos de nivel más bajo y una integración descendente con los de nivel más alto. Esto se denomina *integración sandwich*.

5.9 Pruebas de sistema

Finalizada la integración, es el momento de probar el sistema completo para ver si verdaderamente cumple las especificaciones en un entorno real de trabajo. En todas estas pruebas se suele aplicar una estrategia de caja negra al sistema en su conjunto.

5.9.1 Objetivos de las pruebas

Según el objetivo perseguido, tendremos diferentes clases de pruebas.

PRUEBAS DE RECUPERACIÓN:

Sirven para comprobar la capacidad del sistema para recuperarse ante fallos. Con estas pruebas, además de provocar el fallo, se debe comprobar si el sistema lo detecta, si lo corrige cuando así está especificado y si el tiempo de recuperación es menor del también especificado.

PRUEBAS DE SEGURIDAD:

Sirven para comprobar los mecanismos de protección contra un acceso o manipulación no autorizada. Las pruebas deberán intentar violar los mecanismos previstos como si de un intruso se tratara.

PRUEBAS DE RESISTENCIA:

Sirven para comprobar el comportamiento del sistema ante situaciones excepcionales. Las pruebas deberán forzar el sistema por encima de su capacidad y prestaciones normales para verificar cuál es su comportamiento y cómo se va degradando en estos casos.

290 Introducción a la Ingeniería de Software

PRUEBAS DE SENSIBILIDAD:

Sirven para comprobar el tratamiento que da el sistema a ciertas singularidades relacionadas casi siempre con los algoritmos matemáticos utilizados. Las pruebas deben buscar combinaciones de datos que puedan provocar alguna operación incorrecta o poco precisa en un entorno del problema especialmente sensible.

PRUEBAS DE RENDIMIENTO:

Sirven para comprobar las prestaciones del sistema que son críticas en tiempo. Estas pruebas son fundamentales para los sistemas de tiempo real y/o sistemas con computador englobado. Para medir los tiempos se suelen necesitar equipos de instrumentación externos (emuladores, analizadores lógicos, osciloscopios, etc.). Por ejemplo, se puede forzar al sistema provocando múltiples interrupciones simultáneamente para medir el tiempo máximo que emplea en tratarlas.

5.9.2 Pruebas alfa y beta

Para comprobar que un producto software es realmente útil para sus usuarios es conveniente que estos últimos intervengan también en las pruebas finales del sistema. De esta manera se pueden poner de manifiesto nuevas deficiencias no caracterizadas hasta entonces. Por ejemplo:

- Mensajes ininteligibles para el usuario
- Presentación inadecuada de resultados
- Deficiencias en el manual de usuario
- Procedimientos de operación inusuales
- ... etc. ...

Para la realización de estas pruebas se pueden necesitar semanas o meses porque el usuario necesita ir aprendiendo el manejo del nuevo sistema. Es probable que los problemas más graves aparezcan ya al comienzo y por ello es aconsejable que alguien del equipo de desarrollo acompañe al usuario durante la primera toma de contacto. Se denominan *pruebas alfa* a unas primeras pruebas que se realizan en un entorno controlado donde el usuario tiene el apoyo de alguna persona del equipo de desarrollo y a su vez esta misma persona puede seguir muy de cerca la evolución de las pruebas.

Posteriormente, en las *pruebas beta*, uno o varios usuarios trabajan con el sistema en su entorno normal, sin apoyo de nadie, y anotando cualquier

problema que se presente. En algunos sistemas pueden quedar registradas automáticamente las últimas operaciones que han dado lugar al problema. Sin embargo, es muy importante que sea el usuario el encargado de transmitir al equipo de desarrollo cuál ha sido el procedimiento de operación que le llevó al error. Esta información resulta vital para abordar la corrección.

Tema 6

Automatización del Proceso de Desarrollo

Este Tema se dedica a analizar los elementos de soporte informático del proceso de desarrollo, conocidos genéricamente como técnicas CASE. Dada la variedad de elementos existentes, se realiza un esfuerzo por clasificar de alguna manera las herramientas disponibles, para presentarlas de una manera coherente.

6.1 Entornos de desarrollo software

La palabra *entorno* (en inglés "environment") se viene utilizando en informática desde hace algunos años para designar el contexto en el cual se realiza una determinada actividad, y en particular para designar la combinación de instrumentos disponibles para ello. Además, para aplicar con propiedad el nombre de "entorno" a una determinada colección de instrumentos se requiere que dicha colección constituya un conjunto coherente, de forma que todos los elementos se combinen unos con otros de manera apropiada y que, en lo posible, aparezcan más como una única herramienta global que como una colección de herramientas independientes.

Un *entorno de desarrollo de software* (en inglés SEE: Software Engineering Environment) consiste, por tanto, en el conjunto de los elementos disponibles para realizar dicho desarrollo, y especialmente los instrumentos informáticos que facilitan esta labor. Las técnicas de soporte informático del desarrollo de software se designan habitualmente con las siglas inglesas CASE (Computer Aided Software Engineering).

La reciente evolución de las técnicas de ingeniería de software ha hecho aparecer una multitud de herramientas de soporte de diferentes actividades del desarrollo, que al tiempo que facilitan la producción de software introducen a veces cierta confusión debida a la proliferación de enfoques diferentes que se dan al concebir las distintas herramientas disponibles hoy día.

Las siguientes secciones tratan de dar una visión coordinada de las actuales técnicas CASE, desde las más elementales y clásicas hasta las más evolucionadas, que apenas empiezan a utilizarse.

6.2 Panorámica de las técnicas CASE

Tomando el término CASE en su aspecto más general, se aborda aquí una presentación de conjunto de los diferentes elementos informáticos de apoyo a la labor de desarrollo de software.

6.2.1 Soporte de las fases de desarrollo

Históricamente el soporte informático del desarrollo de software ha ido progresando desde las fases centrales del ciclo de vida hacia los extremos. Las primeras herramientas disponibles se han centrado exclusivamente en la fase de codificación (o programación). La evolución del soporte informático de programación ha ido ligado a la evolución de los lenguajes. De hecho, la palabra "entorno" se ha aplicado inicialmente en informática a la actividad de programación antes que a la de desarrollo global de software.

El *entorno de programación* clásico (que aún no recibía este nombre) se organizaba en torno a un compilador, al que se añadían un editor de textos, para la preparación del código fuente, y un montador de enlaces, para permitir la preparación de módulos por separado y su posterior combinación en un programa ejecutable único.

Un enfoque alternativo del entorno de programación clásico lo constituyen los intérpretes de lenguajes interactivos, que combinan en una sola herramienta las funciones de edición y ejecución del programa.

La aparición de metodologías concretas de análisis y diseño dió lugar a la progresiva aparición de herramientas para soporte de las mismas. Estas herramientas han recibido en muchos casos la denominación de *herramientas CASE*, dando lugar así a una cierta confusión al limitar la aplicación del término CASE a sólo estas fases del ciclo de vida. La metodología más extendida, al menos en cuanto a la variedad de productos comerciales que la soportan, es la de análisis y diseño estructurado, en sus distintas variantes.

Para la fase de pruebas e integración se puede disponer de herramientas de ensayo, que permiten ejecutar automáticamente programas de prueba, y

comparar los resultados obtenidos con los esperados, decidiendo así si la prueba se ha realizado con éxito.

Para la fase de mantenimiento se dispone de soporte de gestión de configuración, que habitualmente se aplica también a las fases anteriores de codificación, pruebas e integración. Este soporte incluye la gestión de versiones y el control de cambios.

El futuro de las técnicas CASE está en el soporte completo de todo el ciclo de vida. Por el momento esto no se consigue con un producto único, aunque sí se han establecido esquemas generales para integrar diversos elementos en un entorno común. Estos entornos integrados se han designado inicialmente con las siglas inglesas IPSE (Integrated Project Support Environment). Actualmente se prefiere hablar de ICASE (Integrated CASE) o de ISEE (Integrated Software Engineering Environments).

6.2.2 Formas de organización

Un entorno de desarrollo de software, con independencia de que cubra un campo más o menos amplio del ciclo de vida, suele ser una combinación de elementos que automatizan una variedad mayor o menor de funciones. El entorno en su conjunto puede estar organizado de diferentes maneras.

Una manera de construir un entorno de desarrollo es combinar varias herramientas distintas, cada una de las cuales realiza una función diferenciada. La combinación de herramientas exige, en general, que el producto resultante de unas pueda usarse como elemento de entrada de otras. De esta manera la actividad de desarrollo puede organizarse como cadenas de trabajos.

Un ejemplo típico es el entorno de programación clásico, con la cadena editor-compilador-montador que permite pasar del texto fuente al programa ejecutable. Este núcleo básico puede ampliarse con otras herramientas tales como las que se describen más adelante en este Tema.

Otra forma de organizar el entorno es disponer de un almacén común de información, denominado *repositorio*. En este almacén se guarda la información en un formato común a todas las herramientas, de manera que cualquiera de ellas puede aplicarse a los datos disponibles en cada momento. Las herramientas CASE de análisis y diseño están organizadas habitualmente de esta manera. Por ejemplo, con la metodología de análisis estructurado, el repositorio contiene la lista completa de todos los datos usados en la aplicación en desarrollo, y esta lista se usa tanto para rotular

296 Introducción a la Ingeniería de Software

las flechas durante la edición de los diagramas de flujo de datos, como para comprobar que las especificaciones de proceso contienen referencias a los datos de entrada y salida, o para imprimir el diccionario de datos completo.

Finalmente podemos hablar de entornos que están organizados como una única herramienta global, capaz de realizar todas las funciones necesarias. Este enfoque resulta muy ventajoso de cara a la comodidad de utilización, pero también encierra el peligro de resultar demasiado cerrado, y no facilitar su combinación con otros elementos de soporte del desarrollo.

6.2.3 Objetivo de un entorno de desarrollo

La diversidad de herramientas y entornos disponibles hoy día, y la consiguiente dificultad para clasificar todo este conjunto de elementos, se debe en buena parte a la diversidad de objetivos parciales que tratan de satisfacerse al diseñar cada herramienta. Según cuál sea el objetivo buscado, las herramientas o entornos de trabajo adoptarán una u otra forma, resultando a veces elementos muy diferentes entre sí.

Uno de los objetivos parciales, y de hecho uno de los más antiguos, es dar soporte a la programación en un lenguaje concreto. En este caso tenemos entornos ligados a un lenguaje en particular. Los intérpretes interactivos son un ejemplo clásico. En algunos casos el entorno como tal es tan importante como el lenguaje. Esto ha ocurrido con el lenguaje SmallTalk, para el que se desarrolló el primer entorno de trabajo basado en una interfaz gráfica con ventanas y menús. También en el caso del lenguaje Ada se ha tratado de crear un entorno estándar, y no sólo un lenguaje de programación.

Otro de los objetivos puede ser dar soporte a una metodología de desarrollo concreta, independiente en la mayoría de los casos del lenguaje de programación utilizado. Así se han construido, como ya se ha indicado, las llamadas herramientas CASE que soportan las metodologías particulares de análisis y diseño de software.

También encontramos entornos de trabajo centrados en la planificación y control del proceso de desarrollo de un proyecto. Estos entornos incluyen herramientas para la organización y control de tareas, ayudas para la planificación de reuniones, medida de la productividad, etc.

Otro objetivo, algo especial, es ayudar al desarrollo de entornos de desarrollo. Tenemos en este caso los llamados *meta-entornos*, que permiten construir entornos o herramientas para soportar una metodología o lenguaje determinado, a partir de su descripción formal, de igual manera que hay

herramientas para construir casi automáticamente un compilador a partir de la descripción formal del lenguaje.

Los ejemplos anteriores no agotan la lista de objetivos posibles, pero son ya un exponente de la variedad de enfoques que pueden darse a la hora de crear una nueva herramienta o entorno de desarrollo.

6.3 Una clasificación pragmática

Un primer intento de clasificar la variedad de entornos de desarrollo lo encontramos en [Dart87]. Esta clasificación puede considerarse como realizada desde un punto de vista pragmático, poco formal, definiendo grupos de entornos similares por analogías de diferentes clases, unas veces de organización y otras de objetivos. Esa clasificación cubre la mayoría de los productos existentes en el momento, pero no otros de gran difusión actual, como son las herramientas de 4^a generación. Supliendo esta omisión, tenemos las siguientes agrupaciones:

- Entornos asociados a un lenguaje
- Entornos orientados a estructura
- Entornos basados en herramientas
- Entornos asociados a una metodología
- Entornos de 4^a generación

A continuación se describen cada uno de estos grupos.

6.3.1 Entornos asociados a un lenguaje

Un primer paso hacia la idea de un entorno de programación integrado lo constituyen los intérpretes de los lenguajes de programación interactivos, tales como los intérpretes de LISP o BASIC. Por ejemplo, un intérprete de BASIC permite editar el programa (habitualmente en memoria), almacenarlo en un fichero externo, recuperar programas preparados de antemano y, por supuesto, ejecutar el programa en edición en cualquier momento, de manera inmediata. Los intérpretes tradicionales de BASIC operaban a base de introducir órdenes en forma de líneas de texto. Si una línea de texto comienza por un número, se entiende que es una orden de edición, y la línea se almacena como línea del programa, o reemplaza a otra ya existente con ese número. Si la orden no comienza con un número se entiende que es una orden de ejecución inmediata. En particular hay una orden para ejecutar el programa. El intérprete suele incluir algunas facilidades de depuración, tales como ejecutar el programa paso a paso, o examinar y

298 Introducción a la Ingeniería de Software

modificar los valores de las variables en mitad de la ejecución del programa.

Más potentes resultan los entornos evolucionados para el lenguaje LISP, como por ejemplo Interlisp. Este entorno posee amplias facilidades para desarrollo incremental, incluyendo potentes órdenes de edición, corrección automática de errores de mecanografiado, manejo de la historia de órdenes, con posibilidades de anular o repetir anteriores órdenes de edición, etc. Los entornos modernos se apoyan en sistemas de ventanas para la interacción con el usuario. Así se puede disponer de una ventana para la edición en modo pantalla del programa fuente, otra ventana para realizar en ella la ejecución, y otra ventana para examinar el contenido de los valores simbólicos. La ejecución paso a paso va marcando en la ventana de edición cada una de las expresiones que se van ejecutando.

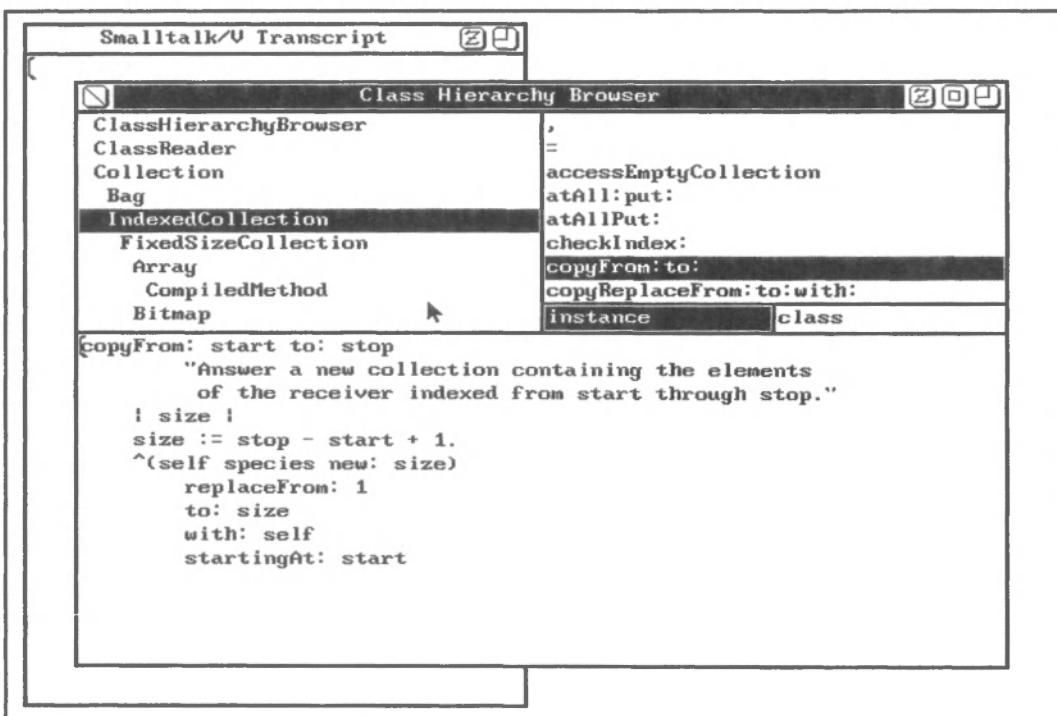


Figura 6.1 Apariencia de un editor de clases en SmallTalk

Un caso especial digno de mención aparte son los entornos asociados al lenguaje SmallTalk. Este lenguaje, además de ser el pionero de los lenguajes de programación orientados a objetos, lleva asociado los primeros entornos gráficos interactivos basados en ventanas, menús, iconos y dispositivo apuntador (ratón o similar). Un entorno SmallTalk se diferencia de los entornos asociados a otros lenguajes en que el programa que se está

preparando no constituye un elemento separable del entorno de desarrollo, sino que se materializa como una versión modificada del propio entorno.

En otras palabras, podemos decir que para desarrollar un programa en SmallTalk lo que se hace es añadir nuevos elementos a los ya existentes en el entorno de desarrollo, de manera que se disponga finalmente de una función equivalente al programa deseado. El entorno de desarrollo en sí está escrito en su mayor parte en el propio lenguaje SmallTalk, de manera que los programas en desarrollo pueden apoyarse en la amplia colección de definiciones de clases y objetos que constituyen el entorno de programación. Estas clases soportan de manera inmediata el manejo de diferentes estructuras de datos, de las ventanas gráficas, y la edición de textos.

Un elemento central del entorno de programación SmallTalk es el editor de clases. En la figura 6.1 se muestra el aspecto de un entorno particular de SmallTalk (SmallTalk/V, de DigiTalk) cuando se ha activado este elemento. El editor de clases opera mediante una ventana con varios paneles. En los paneles superiores aparece la jerarquía de clases definidas en el entorno. Al seleccionar una de ellas se presentan los nombres de los métodos definidos en la clase seleccionada. Seleccionando a su vez uno de esos métodos se presenta en el panel inferior el texto fuente de su definición, que puede ser editada y recompilada. Usando el panel de edición y la función de compilación asociada a él se pueden definir también nuevos métodos asociados a una clase, así como definir nuevas clases.

Otro caso particular es el de los entornos asociados al lenguaje Ada. Este lenguaje ha surgido de un proceso de normalización promovido por el Departamento de Defensa de Estados Unidos. En este proceso se ha definido no sólo el lenguaje en sí, sino la arquitectura y funciones fundamentales de los entornos de programación para este lenguaje, tal como se refleja en la figura 6.2. El nombre en código asignado a la definición estándar del entorno Ada es Stoneman [Buxton84].

El entorno se apoya en un núcleo base (KAPSE: *Kernel Ada Program Support Environment*) que debe estar disponible en cualquier sistema que permita desarrollos en Ada. En la actualidad está constituido por una librería de funciones denominada CAIS (*Common APSE Interface Set*). Sobre esta base se debe disponer de al menos una serie de herramientas básicas: editor, compilador, montador, etc., que constituyen el entorno mínimo (MAPSE: *Minimal Ada Program Support Environment*). En diferentes instalaciones puede haber herramientas adicionales. El entorno completo, incluyendo todas las herramientas disponibles en cada caso, se denomina, en general, APSE (*Ada Programming Support Environment*).

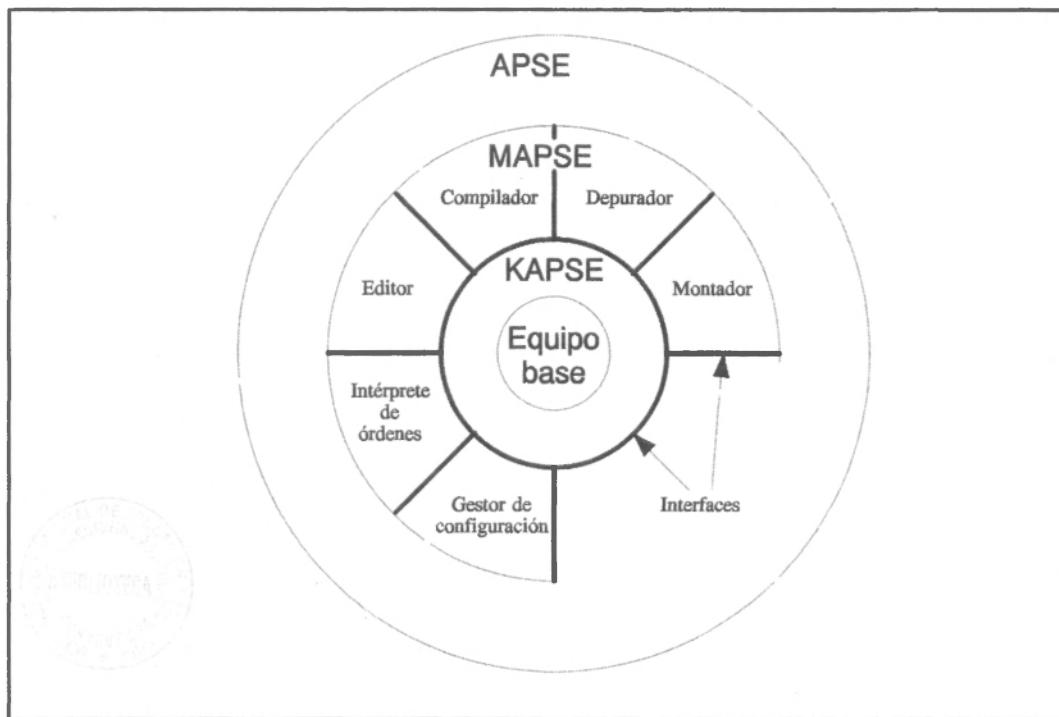


Figura 6.2 Esquema de un entorno Ada

6.3.2 Entornos orientados a estructura

En estos entornos se almacena la información correspondiente al programa en forma estructurada, y no simplemente como texto. Para el caso del programa fuente, esto quiere decir que no se almacena el texto fuente, sino su equivalente como árbol de sintaxis abstracta (AST: *Abstract Syntax Tree*). La edición del programa se consigue mediante un editor de estructura, que permite construir o modificar un programa operando sobre los elementos de su estructura: sentencias, declaraciones, expresiones, etc. En lugar de insertar, borrar o reemplazar caracteres o líneas del texto, el editor permite insertar, borrar o reemplazar expresiones, sentencias, declaraciones de variables, bloques de programa, etc.

El ejemplo clásico de entorno orientado a estructura lo constituye el "Cornell Program Synthesizer" [Teitelbaum91]. Este entorno permite desarrollar programas en lenguaje PL/I operando sobre la estructura codificada del programa. Las operaciones sobre esa estructura incluyen tanto la edición como la ejecución del programa, y permiten operar con programas a medio construir, mientras no se intente trabajar con un elemento aún no definido. El entorno se basa en *plantillas* (*templates*) que describen las estructuras

básicas y la manera de presentarlas en pantalla en forma de texto. Por ejemplo, la plantilla correspondiente a una sentencia IF es

```
"IF (" condición
") \nTHEN" sentencia-1
"ELSE" sentencia-2
"\r"
```

Los elementos en cursiva son huecos (*placeholders*) en la plantilla, que habrán de ser rellenados con elementos concretos. Los elementos literales, entre doble apóstrofo ("") incluyen códigos de control del formato de presentación. Puesto que la plantilla incluye la forma en que se presenta el elemento en pantalla, el estilo de encolumnado del programa es totalmente automático.

Posteriormente este proyecto se amplió, desarrollando una herramienta para generar automáticamente un entorno de programación, similar al de PL/I, para cualquier otro lenguaje de programación, partiendo de una descripción formal de la sintaxis y semántica del lenguaje en forma de gramática de atributos. Esta herramienta se ha denominado "Synthesizer Generator" [Reps84].

Otro proyecto en que se han desarrollado herramientas para la construcción de entornos de programación orientados a estructura es el proyecto Gandalf [Habermann86], cuyo elemento central es un generador de editores denominado ALOEGEN.

6.3.3 Entornos basados en herramientas

Los entornos de esta clase consisten en una colección de herramientas (en inglés "toolkit" o "toolbox") relativamente independientes. Para que el conjunto se pueda considerar con propiedad un entorno de desarrollo es necesario que las distintas herramientas sean compatibles entre sí. Además debe existir algún medio de hacerlas funcionar en forma combinada, para facilitar su manejo.

Un ejemplo clásico es el denominado "entorno de programación UNIX", que incluye una variada colección de programas de utilidad para el desarrollo de programas en lenguaje C, tales como las siguientes:

cc	compilador/montador
vi	editor de textos

302 Introducción a la Ingeniería de Software

lint	verifica la consistencia de programas
ar	gestor de librerías
cdb	depurador simbólico
prof	presenta el perfil de ejecución
cxref	generador de referencias cruzadas
cb	"prettyprinter"
make	automatiza la compilación y montaje

A esta colección se pueden añadir otras herramientas para tratar ficheros de texto, en general, y preparar documentación. Entre ellas:

troff, nroff	procesadores de texto
SCCS	sistema de control de versiones
awk	programa de manipulación de texto
grep	busca patrones en ficheros de texto
what	busca texto de identificación
... etc. ...	

La colección de herramientas disponible en UNIX es amplísima, superando el centenar de utilidades. La compatibilidad viene dada por los formatos de ficheros tratados: texto, fundamentalmente, así como código objeto y código ejecutable. Para combinar unas herramientas con otras se dispone de las facilidades propias de este sistema operativo. Muchas herramientas pueden operar como filtros y ejecutarse en cadena conectando la salida de un programa como entrada del siguiente.

En realidad la forma más flexible de combinar herramientas es usar las posibilidades de los "shell" o intérpretes de órdenes. Los lenguajes de órdenes de UNIX disponen de construcciones similares a las del lenguaje C y pueden usar variables. Usando un lenguaje de órdenes se puede programar la realización automática de operaciones complejas que formen parte del proceso de desarrollo de software, facilitando la aplicación de normas o reglas particulares de programación o desarrollo.

Otros sistemas operativos incluyen colecciones de herramientas similares a algunas de las mencionadas. Por ejemplo, sobre VMS de DEC existe el conjunto VaxSet (o el más moderno DECSet) que comprende editor orientado a lenguaje, gestor de versiones, analizador de código fuente, etc.

Los entornos basados en herramientas suelen presentar como ventaja el ser bastante abiertos, permitiendo la incorporación de nuevas herramientas de diferentes fabricantes o desarrolladas por los propios usuarios. Como inconveniente se les puede achacar la falta de una interfaz de usuario

interactiva y uniforme, y ciertas limitaciones en la representación de información compleja si se fuerza el uso de ficheros de texto como medio fundamental de intercambio de información.

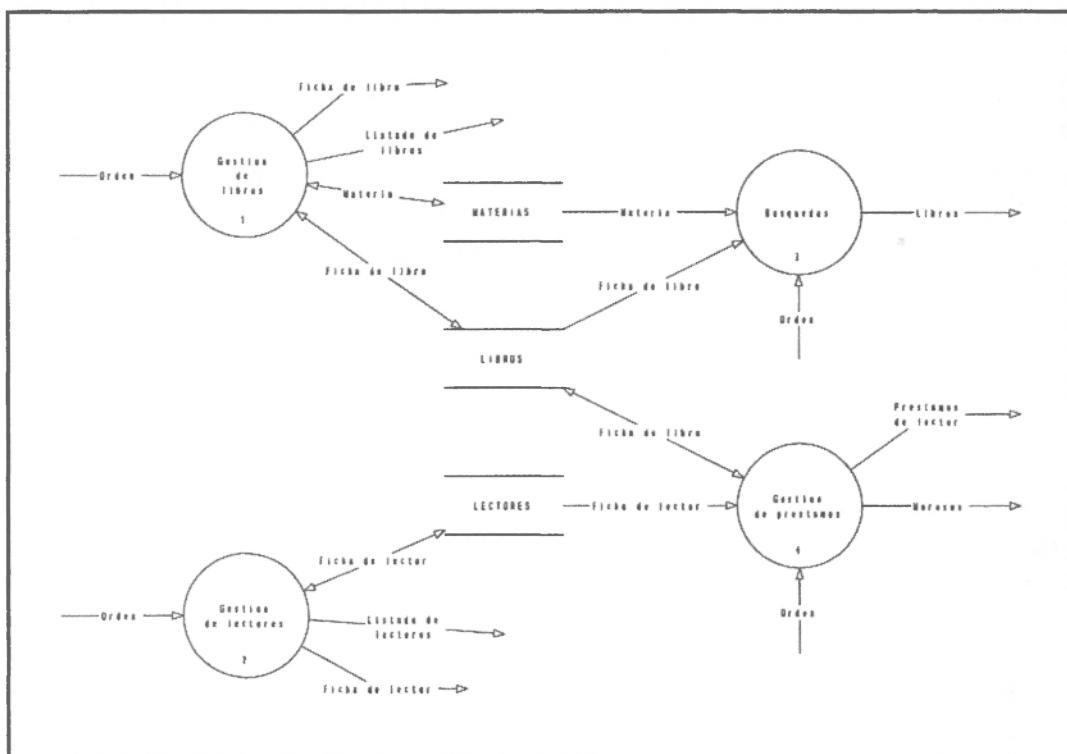


Figura 6.3 Ejemplo de DFD realizado con una herramienta CASE

6.3.4 Entornos asociados a metodología

El establecimiento de metodologías de análisis y diseño en los primeros años de existencia de la ingeniería de software como disciplina consolidada, dio lugar a continuación a la aparición de numerosas herramientas de soporte de dichas metodologías. Como ya se ha indicado, muchas de estas herramientas se han denominado comercialmente "herramientas CASE", aunque realmente sólo soporten algunas fases del ciclo de vida de desarrollo. La principal ventaja de estas herramientas es que suelen constituir entornos de trabajo bien integrados y con una interfaz de usuario uniforme.

La integración de los distintos elementos del entorno se suele conseguir mediante el empleo de un almacén único o *repositorio CASE* para almacenar todos los elementos de información contemplados en la

304 Introducción a la Ingeniería de Software

metodología soportada. Como ejemplo podemos considerar las metodologías de análisis y diseño estructurado, basadas en el empleo de diagramas de flujo de datos. El repositorio contiene la información correspondiente a toda la familia de diagramas de flujo de datos, así como las descripciones de cada dato y de cada proceso. El entorno de análisis y diseño permite editar separadamente cada diagrama de flujo y cada descripción de proceso (miniespecificación), así como el diccionario de datos. Algunos elementos de información, almacenados sólo una vez en el repositorio, aparecen en varios diagramas o descripciones. La herramienta CASE facilita también el mantenimiento o al menos detección de la consistencia entre las diferentes descripciones o diagramas del producto software en desarrollo.

```
Project: C:\YSELECT\SYSTEM\BIBLI094\  
Title: Gestión de biblioteca  
Date: 9-Jan-95 Time: 16:57  
  
This report generates a data dictionary listing, in alphabetical order.  
The report lists the name, type and BNF attribute for each item.  
Note type abbreviations are: D = Discrete Flow  
                           CD = Continous Flow  
                           C = Control Flow  
                           DS = Data Store  
                           CS = Control Store  
  
The format of the report is.  
  
NAME          TYPE BNF  
  
Ficha de lector      D No BNF  
Ficha de libro       D No BNF  
Fichas              D [ Ficha de libro | Ficha de lector ]  
LECTORES            DS { Ficha de lector }  
LIBROS               DS { Ficha de libro }  
Libros              D No BNF  
Listado de lectores  D No BNF  
Listado de libros    D No BNF  
Listados             D [ Listado de libros | Listado de lectores | Morosos ]  
Materia              D No BNF  
MATERIAS             DS { Materia }  
Morosos              D No BNF  
Orden                D No BNF  
Prestamos de lector  D No BNF  
Respuesta            D [ Libros | Prestamos de lector ]
```

Figura 6.4 Ejemplo de diccionario de datos de la figura anterior

En la figura 6.3 aparece como ejemplo un diagrama de flujo de datos construido con una de estas herramientas (SELECT/Yourdon). En la figura 6.4 se muestra la versión inicial del diccionario de datos, correspondiente

al mismo ejemplo, generado por la herramienta CASE tras editar manualmente la estructura (en notación BNF) de algunos datos.

Algunas herramientas CASE permiten también generar automáticamente partes del código de la aplicación en desarrollo. Por ejemplo, a partir de las descripciones del diccionario de datos se puede generar directamente la DATA DIVISION de un programa en COBOL.

6.3.5 Entornos de 4^a generación

Los entornos de desarrollo basados en herramientas de 4^a generación son relativamente recientes, aunque hoy día su uso está bastante extendido. La mayoría de estos entornos se apoyan en un sistema de gestión de base de datos dotado de un lenguaje de consulta, al que se añaden algunas herramientas complementarias.

Estos entornos suelen estar orientados al usuario final. El desarrollo de un sistema de información consiste en definir los esquemas de la base de datos subyacente y programar las operaciones o transacciones principales. Las operaciones de manipulación pueden programarse directamente con el lenguaje de consulta de la base de datos, o ayudarse con herramientas específicas para el diseño de formularios de visualización y edición de datos en pantalla, generación de listados o informes, generación de sistemas de menús para el diálogo con el usuario, etc.

Muchas de estas herramientas de ayuda pueden considerarse como entornos de programación visuales, en que se edita en pantalla el esquema de una consulta, de un formulario, de un listado, etc. A partir de este esquema se genera automáticamente un programa equivalente en el lenguaje básico de acceso a la base de datos.

6.4 Una clasificación por niveles

La clasificación de la sección anterior sigue unos criterios pragmáticos, tratando de agrupar los productos CASE según afinidades. Podemos encontrar clasificaciones más modernas en [Fernström92] y [Fuggetta93]. Estas clasificaciones se basan principalmente en las funciones realizadas por los distintos productos CASE, y su situación en el entorno global de desarrollo.

306 Introducción a la Ingeniería de Software

Un criterio base de estas clasificaciones es el nivel de funcionalidad del producto CASE, correspondiente a la actividad de desarrollo que automatiza o soporta. Los niveles que se distinguen son:

Producto CASE	Actividad soportada
Servicio	Operación
Herramienta	Tarea
Banco de trabajo	Papel o perfil profesional
Entorno de desarrollo	Proceso de desarrollo de software

El *nivel de servicio* corresponde a un producto que realiza una función u operación elemental, atómica, que una vez invocada no ha de interrumpirse. Desde el punto de vista del proceso de desarrollo, es una operación que ha sido automatizada completamente; por ejemplo, la compilación de un programa fuente.

El *nivel de herramienta* corresponde a un producto software que permite invocar diferentes servicios u operaciones correspondientes a una misma actividad individualizada dentro del proceso de desarrollo. Un ejemplo típico es el de un editor (de texto, diagrama o documento).

El nivel de *banco de trabajo* o *equipo de herramientas* (en inglés: "Workbench", "Toolkit" o "Toolbox") corresponde a un producto CASE que automatiza o soporta un perfil concreto de actividad profesional dentro del proceso de desarrollo. Por ejemplo, la actividad del analista, o del diseñador, o del "bibliotecario", etc. Un banco de trabajo suele englobar varias herramientas, a menudo fuertemente integradas, con una interfaz de usuario uniforme. De esta manera podemos disponer de un banco o equipo de análisis, de diseño, etc. En el caso particular de la actividad de codificación, el banco de trabajo corresponde a lo que se denomina *entorno de programación*.

El nivel de *entorno de desarrollo* corresponde a un producto CASE que soporte el proceso completo de desarrollo de software. Como ya se ha indicado, se suele designar con las siglas IPSE o ICASE.

Los dos primeros niveles mencionados se describen a veces como uno solo, en cuyo caso se denomina herramienta, por ejemplo, tanto a un editor como a un compilador.

6.5 Herramientas de software

Aquí se describen productos correspondientes a los dos primeros niveles de la clasificación anterior. En primer lugar se describen las herramientas tradicionales, y luego las más modernas. Algunas de ellas han sido mencionadas ya al hablar de entornos basados en herramientas.

6.5.1 Herramientas clásicas

El repertorio de herramientas que podemos considerar tradicionales incluye las siguientes:

EDITOR DE TEXTO: Permite editar ficheros de texto. En su forma tradicional considera el texto formado por líneas y caracteres. A veces incluye funciones orientadas a palabras, pero en general no atiende para nada a la naturaleza del texto en edición, es decir, no tiene en cuenta si el texto es un programa fuente, o en general, si sigue una reglas de construcción determinadas.

COMPILADOR: Traduce un fichero de texto fuente a otro de código objeto, normalmente reubicable. El código objeto no suele ser ejecutable directamente.

MONTADOR DE ENLACES (en inglés "linker"): Permite construir un programa ejecutable combinando varios ficheros objeto. Habitualmente maneja librerías de funciones en código objeto, realizando la inclusión automática de aquellas que son efectivamente usadas en el programa.

GESTOR DE LIBRERÍA: Permite combinar varios ficheros objeto en una librería única. La librería suele incluir un índice que permite localizar rápidamente dentro de ella el fragmento de código objeto que contenga un nombre simbólico o referencia externa determinada.

HERRAMIENTA "MAKE": Sirve para automatizar la actualización de un conjunto de ficheros a partir de otros. Típicamente sirve para automatizar la compilación y montaje de programas a partir de los ficheros fuente. Opera a partir de un fichero de dependencias en el que se indica de qué ficheros depende uno dado, y qué órdenes hay que ejecutar para regenerarlo en caso necesario. Estas órdenes son invocadas selectivamente dependiendo de si la fecha de actualización del fichero final es anterior o posterior a las de los ficheros de partida.

308 Introducción a la Ingeniería de Software

INTÉRPRETE INTERACTIVO: Constituye en sí mismo casi un entorno de programación completo (si lo es, habrá que clasificarlo con el nivel de banco de trabajo y no de herramienta). Un intérprete interactivo engloba funciones equivalentes a las de edición, compilación, montaje y ejecución de un programa.

COMPILADOR/INTÉRPRETE: Es un procesador de un lenguaje interpretado en forma no interactiva. Incluye un compilador que traduce el programa fuente a código intermedio, y un intérprete de ejecución de dicho código intermedio que contiene la librería de todas las funciones básicas de soporte del lenguaje. No incluye funciones de edición del programa.

DEPURADOR ABSOLUTO: Permite ejecutar un programa en forma controlada, instrucción por instrucción, o hasta un punto concreto. También permite examinar y modificar el contenido de la memoria y los registros del procesador. Como sólo hace referencia a direcciones de memoria o instrucciones de máquina, puede ser utilizado con programas escritos en cualquier lenguaje y preparados con cualquier compilador. Su inconveniente es que resulta muy incómodo de usar.

DEPURADOR SIMBÓLICO: Realiza una función análoga al anterior pero trabajando con referencia al código fuente, lo cual resulta mucho más cómodo. La ejecución por pasos opera sobre las sentencias del lenguaje, y el acceso a la memoria se hace con referencia a las variables del programa. El depurador simbólico necesita una tabla de información generada por el compilador, que ha de estar preparado para ello.

6.5.2 Herramientas evolucionadas

En este apartado se describen otras herramientas que han alcanzado ya una amplia difusión. Algunas de ellas pueden casi considerarse clásicas, mientras que otras se emplean habitualmente sólo en entornos de desarrollo avanzados.

EDITORES ORIENTADOS A LENGUAJE: Normalmente son editores de estructura. El programa fuente no se almacena internamente como texto, sino que se codifica su estructura sintáctica/semántica. Ya se ha hablado de esta clase de herramientas al mencionar los entornos orientados a estructura. Aunque presentan claras ventajas respecto a un editor de textos convencional, todavía no se emplean con profusión, ya que

por debajo de un cierto nivel de estructura (típicamente el de expresión aritmética) resulta más cómoda la edición del código fuente como texto.

HERRAMIENTA "MAKE" AUTOMÁTICA: Las herramientas clásicas de tipo "make" necesitan una lista explícita de dependencias entre ficheros y acciones de actualización. Hay casos en que las tareas de compilación y montaje pueden automatizarse totalmente detectando las dependencias entre módulos mediante el análisis del código fuente. De esta manera se puede combinar una herramienta de detección de dependencias, que genera el fichero de dependencias, con una herramienta "make" clásica. También se puede incorporar la función "make" en el propio compilador, de manera que la compilación de un módulo provoca automáticamente la recompilación, en caso necesario, de aquellos de los que depende.

MANEJADOR DE VERSIONES: Este tipo de herramienta permite almacenar de manera organizada y eficiente una serie de versiones de un mismo elemento de software. Todas las herramientas de esta clase pueden manejar ficheros de texto, y algunas permiten también almacenar versiones de ficheros binarios. Ya se ha mencionado como ejemplo las utilidades de SCCS para UNIX. Las versiones suelen designarse mediante una numeración correlativa, a varios niveles; por ejemplo 1.1, 1.2, 2.1, etc. Algunas herramientas de esta clase usan un fichero separado para las versiones de cada elemento de software. Otras operan con librerías que agrupan las distintas versiones de todos los elementos de software correspondientes a un mismo sistema o subsistema en desarrollo. Es frecuente que estas herramientas se utilicen automáticamente desde las utilidades de tipo "make" al recompilar una aplicación en desarrollo.

PROCESADORES/ANALIZADORES DE CÓDIGO FUENTE: En este grupo se pueden incluir diferentes herramientas que procesan el texto fuente para obtener mediciones (tamaño, complejidad, etc.), generar tablas de referencias, encolumnar según un estilo normalizado ("prettyprinters"), comprobar la consistencia de módulos entre sí, etc. Muchas de estas utilidades pueden ser consideradas como complemento de los compiladores tradicionales, y las funciones que realizan podrían estar incorporadas directamente a dichos compiladores.

PROCESADORES DE DOCUMENTOS: Estas herramientas no son específicas de las labores de desarrollo de software, pero son un soporte

310 Introducción a la Ingeniería de Software

fundamental para ellas. Los procesadores de documentos, llamados también procesadores de texto ("word processors"), facilitan la edición de documentación. Existen dos enfoques fundamentales. En uno de ellos se usa un editor convencional para preparar un fichero con el texto del documento, sobre el que se realiza un marcado para señalar las partes del documento y/o el estilo de presentación. La herramienta de proceso de documento genera el texto impreso en forma apropiada a partir de este fichero con el texto marcado. Las utilidades "troff" y "nroff" en UNIX son un ejemplo clásico de este enfoque, que también lo siguen TeX y sus variantes.

El otro enfoque corresponde a los editores de documentos que presentan en pantalla una representación más o menos exacta del resultado impreso previsto, y permiten editar el documento directamente sobre dicha presentación. La misma herramienta realiza las funciones de edición y de composición e impresión del documento.

HERRAMIENTAS DE CONTROL DE PRUEBAS: Ayudan a la realización de pruebas unitarias o de integración. Las hay de varias clases. Hay herramientas para comprobar el grado de cubrimiento o ejecución de código durante las pruebas. Otras herramientas invocan la ejecución de programas de prueba en forma automática, comprobando que los resultados coinciden con los esperados. También hay herramientas para facilitar las pruebas automáticas de programas interactivos, simulando la entrada de órdenes o datos por teclado o ratón, en forma predefinida.

HERRAMIENTAS DE CONTROL DE CAMBIOS: Ayudan a la realización del desarrollo en forma incremental, y al mantenimiento de aplicaciones. Una herramienta de este tipo mantiene una configuración consistente del sistema en desarrollo, denominada *línea base*, y sólo permite incorporar cambios en forma controlada, garantizando que las modificaciones mantienen el sistema en estado operativo. Estas herramientas pueden apoyarse en herramientas tipo "make" para realizar las actualizaciones, en herramientas de control de pruebas para garantizar el correcto funcionamiento del sistema, y en herramientas de manejo de versiones para almacenar la evolución de la línea base.

PROCESADORES DE FICHEROS DE TEXTO: En este grupo de herramientas se pueden incluir un gran número de programas de utilidad disponibles en sistemas UNIX, muchos de los cuales han sido transportados o

adaptados a otros sistemas operativos, constituyendo un verdadero "cajón de sastre" del que se puede echar mano ocasionalmente para realizar pequeñas (o grandes) tareas en forma cómoda. Algunas de estas herramientas se han mencionado ya al hablar de entornos basados en herramientas. Destacaremos aquí solamente la utilidad "awk" de UNIX, que es un compilador/intérprete de un lenguaje de manipulación de ficheros de texto que facilita las operaciones de tratamiento de ficheros de datos. La forma básica de operación es realizar el tratamiento línea por línea, con acciones particulares para cada clase de línea de texto, identificada por un patrón determinado.

Esta panorámica no agota la variedad de herramientas de apoyo al desarrollo de software existentes hoy día, pero es suficiente para ilustrar las posibilidades de soporte informático de las actividades de desarrollo.

6.5.3 Herramientas de 4^a generación

Las herramientas mencionadas en los apartados anteriores suelen ser usadas por los ingenieros de software y programadores para desarrollar nuevos productos software. Las técnicas de 4^a generación buscan la aproximación al usuario final, permitiendo que éste realice sus propios desarrollos. Algunas de estas técnicas se han mencionado ya en este Tema. Citaremos ahora las herramientas más conocidas en este campo.

HOJA DE CÁLCULO: Es una herramienta para facilitar la realización automática de cálculos basados en una plantilla o formulario, habitualmente de tipo matricial o tabular. Se puede especificar que determinadas casillas de la tabla se rellenen automáticamente con datos calculados a partir de las otras. Los datos o las fórmulas se editan en forma interactiva. Algunas herramientas modernas de este tipo permiten también la presentación gráfica de los valores almacenados en la hoja de cálculo.

PROCESADORES DE DOCUMENTOS: Estas herramientas se han mencionado ya en el apartado anterior. Las herramientas de 4^a generación adoptan siempre el segundo de los enfoques mencionados allí, presentando en pantalla el documento a medida que se va editando.

GESTORES DE BASES DE DATOS: Permiten definir los esquemas de las bases de datos, y realizar operaciones generales de consulta y manipulación, tales como la inserción de nuevos datos, modificación de los

312 Introducción a la Ingeniería de Software

existentes, búsquedas, etc. Suelen estar basados en el modelo relacional.

LENGUAJES DE 4^a GENERACIÓN: Permiten programar operaciones de tratamiento de datos. Normalmente están asociados a un gestor de base de datos, en cuyo caso engloban un lenguaje de consulta y actualización (*query language*), tal como SQL.

GENERADORES DE PROGRAMAS: También suelen estar asociados a un sistema de gestión de base de datos. Son herramientas para la preparación interactiva de esquemas de informes, formularios en pantalla, menús, etc. A partir de los esquemas editados en forma interactiva, generan programas en el lenguaje de 4^a generación asociado.

Realmente hay otras muchas herramientas o programas de utilidad que pueden incluirse en la categoría de software de 4^a generación, pero aquí nos limitamos a considerar el software de desarrollo.

6.6 Entornos integrados

Los siguientes niveles (banco o equipo de trabajo, entorno de desarrollo) se pueden analizar atendiendo no sólo a las funciones que realizan, o las herramientas que incluyen, sino también a la manera en que estas herramientas se integran en un conjunto coherente. La integración [Thomas92] puede presentar diferentes aspectos:

- Integración de datos
- Integración del control
- Integración de presentación
- Integración del proceso

A continuación se estudia cada uno de estos aspectos.

6.6.1 Integración de datos

La *integración de datos* significa que la información almacenada en el entorno es gestionada de manera uniforme, con independencia de las transformaciones concretas que se hagan con cada elemento de información. Esta forma de integración debe conseguir:

- Interoperabilidad entre herramientas

- No redundancia de datos, evitando duplicaciones
- Consistencia de datos, evitando incoherencias
- Paso de datos de una herramienta a otra

La integración de datos puede conseguirse de distintas maneras [Chen92]:

TRANSFERENCIA DIRECTA de datos de una herramienta a otra. Es una manera muy eficiente de compartir datos, pero poco flexible. Resulta complicada cuando hay que integrar muchas herramientas diferentes.

TRANSFERENCIA MEDIANTE FICHEROS. Ésta es la manera más sencilla de conseguir el paso de información de una herramienta a otra. En la actualidad existe incluso un formato normalizado para intercambio de datos (CDIF: CASE Data Interchange Format) propuesto por la EIA (Electronic Industries Association) norteamericana.

TRANSFERENCIA BASADA EN COMUNICACIÓN. Es una alternativa a la transferencia mediante ficheros, y puede ser usada en sistemas distribuidos y en sistemas abiertos.

REPOSITORIO COMÚN. Esta forma de integración es la que se utiliza con preferencia en entornos modernos, que buscan un grado de integración elevado. Se estudia con detalle más adelante.

6.6.2 Integración del control

La *integración del control* consiste en permitir la combinación flexible de funciones para cumplir con las particularidades del proceso y actividades que hay que informatizar. El mayor grado de integración se consigue cuando desde una herramienta se pueden invocar funciones suministradas por otra herramienta. Para que una herramienta suministre algún servicio a otra será necesario compartir información entre ellas; por tanto, la integración del control exige como paso previo la integración de los datos.

La integración del control se puede conseguir mediante un sistema de gestión de procesos y mensajes. Mediante un sistema de mensajes se pueden invocar funciones entre herramientas a diferentes niveles: herramienta-herramienta, herramienta-servicio, o servicio-servicio.

Un sistema de gestión de procesos permite invocar como acción única una combinación de acciones realizadas por distintas herramientas. Ya se ha mencionado como ejemplo el empleo del lenguaje de órdenes asociado a un

"shell" de un sistema operativo para programar secuencias de acciones que automaticen determinadas actividades del proceso de desarrollo.

6.6.3 Integración de la presentación

La *integración de la presentación* trata de realizar la interacción con el usuario de manera uniforme, con independencia de la función o herramienta en uso en un momento dado. De esta manera se reduce el esfuerzo de memoria y atención necesario para utilizar el entorno. Para ello se deben conseguir los objetivos propios de un sistema amigable (en inglés *user-friendly*):

- Limitar el número de formas de interacción diferentes usadas por el conjunto de las herramientas y servicios.
- Usar formas de interacción y presentación adecuadas al modelo mental que el usuario tiene del entorno.
- Satisfacer los tiempos de respuesta esperados y dar indicación del avance de la operación en caso de tratamientos de larga duración.
- Mantener información útil a disposición del usuario.

La integración de la presentación se puede basar en el empleo de sistemas de interfaz de usuario normalizados, tales como Motif u Open Look. Sin embargo el uso de uno de estos sistemas no garantiza en sí mismo un nivel de integración suficiente, ya que dicha interfaz sólo implica una forma de presentación uniforme, pero no su adecuación al modelo mental del usuario, que debe conseguirse mediante un diseño adecuado del diálogo, implícito en el flujo de control de cada herramienta o servicio.

6.6.4 Integración del proceso

La *integración del proceso* consiste en que las herramientas se combinan de tal manera que apoyan o fuerzan de manera efectiva el uso de una metodología de desarrollo definida. Este modo de integración exige, en general, la existencia de una buena integración de control y datos.

El proceso de desarrollo puede definirse en base a los siguientes elementos:

- Un *paso* del desarrollo es una unidad de trabajo concreta que produce un resultado; por ejemplo, la revisión del documento de diseño.
- Un suceso o *evento* de desarrollo es una condición que ocurre durante la ejecución de un paso y que puede desencadenar la ejecución de una acción asociada; por ejemplo, la compilación sin errores de un módulo puede desencadenar la ejecución de los programas de prueba de esa unidad.
- Una *restricción* del desarrollo es una limitación que debe cumplirse; por ejemplo, que no debe haber dos personas editando el mismo fichero al mismo tiempo, o que el autor de un módulo no debe encargarse de las pruebas del mismo.

Un buen grado de integración del proceso exige que los pasos, eventos y restricciones que definen de manera natural la metodología de desarrollo a utilizar, sean representables y tratables dentro del entorno.

6.6.5 El repositorio CASE

El repositorio CASE es un almacén común en el que se guarda toda la información necesaria para la operación de un grupo de herramientas o de un entorno de desarrollo. En su forma más sencilla, el repositorio facilita las funciones de almacenamiento y recuperación de datos, normalmente en forma concurrente multiusuario, y el mantenimiento de relaciones entre los datos. El repositorio puede suministrar, además, funciones de gestión de versiones, de seguridad, mediante control de acceso, y de gestión de transacciones.

Para proporcionar las funciones de almacenamiento y recuperación de datos se requiere, según [Chen92]:

- Un servicio de *metamodelo*, que permita definir las estructuras de datos que han de almacenarse en el repositorio.
- Un servicio de *consulta y actualización* (query), que permita acceder y manipular la información contenida en el repositorio.
- Un servicio de *vistas*, que permita definir subconjuntos de datos y operaciones que constituyan el subentorno de trabajo de ciertas

316 Introducción a la Ingeniería de Software

actividades, y entre los que haya que mantener relaciones concretas de consistencia.

- Un servicio de *intercambio de datos*, que facilite la importación y exportación de información mediante ficheros externos o canales de comunicación.

La información que interesa almacenar en un repositorio CASE cubre un espectro muy amplio. Su contenido puede incluir:

- Código fuente
- Código objeto
- Código ejecutable
- Órdenes de compilación y montaje
- Dependencias entre módulos
- Estructura modular
- Programas de prueba
- Datos de prueba
- Resultados de pruebas
- Especificaciones de módulos
- Información de cambios
- Métricas de calidad
- Documentos de diseño
- Documentos de requisitos
- Informes de revisiones
- Definición de la metodología de desarrollo
- Información de planificación del proyecto
- Información de gestión del proyecto
- Información de la empresa
- ... etc. ...

Integrar toda esta información en forma homogénea en un repositorio único no resulta fácil, por supuesto. Ésta es una de las dificultades a las que hay que enfrentarse a la hora de construir entornos de desarrollo capaces de soportar todo el ciclo de vida del software.

6.7 Bancos o equipos de trabajo

Como ya se ha indicado, un banco de trabajo debe integrar las herramientas necesarias para dar soporte a un determinado perfil profesional o actividad general de desarrollo. Para que el banco de trabajo merezca este nombre, debe conseguir:

- Integración de la presentación, con una interfaz de usuario homogénea y consistente
- Integración del control, con la posibilidad de invocar cómodamente una herramienta o cadena de herramientas
- Integración de datos, preferiblemente mediante un repositorio común

Según la clase de actividad soportada, tendremos distintos bancos o equipos de trabajo, entre ellos los siguientes:

EQUIPO DE ANÁLISIS Y DISEÑO, denominado a veces herramienta CASE, o CASE superior ("upper" CASE). Debe ser capaz de soportar completamente una determinada metodología. Corresponde a lo que en el apartado 6.3.4 se ha denominado "entorno asociado a metodología". Muchos de estos entornos cubren las dos fases (análisis y diseño), mientras que otros sólo cubren una de ellas. El repositorio almacena en forma estructurada todos los elementos de información definidos en la metodología soportada, con referencias entre ellos, evitando redundancias y garantizando la consistencia.

Por ejemplo, para soportar la metodología de análisis y diseño estructurado se incluyen editores de diagramas de flujo de datos, diccionario de datos, especificaciones de proceso, diagramas entidad-relación, diagramas de estructura modular, diagramas de transición de estados, etc. La modificación de un elemento desde un editor puede implicar su modificación a nivel global, con lo que el cambio se apreciará en todos los diagramas o descripciones en que aparezca dicho elemento.

ENTORNO DE PROGRAMACIÓN, es el banco de trabajo para la actividad de codificación, y puede extenderse también al diseño detallado y a las pruebas de unidades. Con mucha frecuencia aparece como entorno asociado a un lenguaje determinado, tal como se describió en las secciones 6.3.1 y 6.3.2. También puede construirse como colección de herramientas independientes, como se indicaba en la sección 6.3.3, pero en este caso no suele alcanzar un nivel de integración elevado.

Un entorno de programación integrado debe incluir las funciones de edición orientada al lenguaje, compilación y montaje automáticos, depuración, evaluación de métricas, etc.

EQUIPO DE VERIFICACIÓN Y VALIDACIÓN, capaz de facilitar las tareas de inspección y pruebas de módulos y sistemas. En muchos casos suele

318 Introducción a la Ingeniería de Software

estar ligado al entorno de programación, ya que con frecuencia se necesita operar sobre el código fuente. Puede incluir funciones de:

- Análisis estático, con evaluación de métricas de calidad y generación de matrices o grafos de llamadas entre funciones y módulos.
- Generación de tablas de referencias cruzadas.
- Análisis dinámico, con evaluación del cubrimiento de código y perfil de ejecución.
- Gestión de pruebas, automatizando la realización de ensayos y comprobando los resultados. Se necesita para realizar cómodamente pruebas de regresión.

EQUIPO DE CONSTRUCCIÓN DE INTERFAZ DE USUARIO. Permite definir cómodamente el esquema de diálogo con el usuario, así como los elementos, habitualmente gráficos, de interacción. La definición de la interfaz de usuario suele hacerse en forma interactiva, manipulando directamente la imagen de pantalla correspondiente a cada elemento de interacción (menú, ícono, botón, etc.).

En la mayoría de los casos el desarrollo de la interfaz de usuario equivale al desarrollo del programa principal de la aplicación. Con frecuencia se incluye un generador de código que produce el texto fuente de dicho programa principal en un lenguaje de programación de uso general que debe ser compatible, a nivel de llamadas a funciones, con el lenguaje de programación usado para desarrollar el resto de la aplicación.

EQUIPO DE GESTIÓN DE CONFIGURACIÓN, que permita almacenar diferentes versiones de los elementos del proyecto, definir distintas configuraciones y controlar los cambios sucesivos. Es fundamental que se combine bien con el entorno de programación si se quiere realizar desarrollo incremental, modificando progresivamente la configuración de módulos del sistema en construcción. También es indispensable para gestionar cómodamente las tareas de mantenimiento.

Un enfoque tradicional de estos bancos de trabajo consiste en usar un repositorio central para las versiones estables, y materializar las configuraciones como copias temporales, en un directorio separado, de versiones estables combinadas con versiones en desarrollo. Un enfoque más evolucionado consiste en interceptar el acceso al sistema

de ficheros básico, definiendo las configuraciones como sistemas virtuales de ficheros, en los que un mismo nombre de fichero hace referencia a una u otra versión dependiendo de la configuración en que se trabaje.

EQUIPO DE INGENIERÍA INVERSA. Debe facilitar la extracción de información de diseño y los elementos abstractos a partir de un código o sistema software ya desarrollado. Puede incluir herramientas para generar tablas de referencias cruzadas, diagramas de flujo de ejecución, recolectar las cabeceras de funciones y procedimientos, reagrupar y reordenar el código fuente, etc. Dada la variedad de situaciones y estilos que pueden aparecer en el código de antiguas aplicaciones, no es posible sistematizar esta actividad, y sólo puede contarse con ayudas parciales.

EQUIPO DE GESTIÓN DE PROYECTOS, que facilita la confección de planes de trabajo, con asignación de tiempos y recursos a las diferentes tareas, y el seguimiento de su realización. Algunas veces se incluyen también herramientas para automatizar la agenda de trabajo personal de los miembros del equipo, los calendarios de reuniones, intercambio de mensajes, etc.

6.8 Entornos orientados al proceso

Un entorno de desarrollo orientado al proceso debe ser capaz de soportar todas las actividades del ciclo de vida de desarrollo, y además hacerlo de acuerdo con una metodología concreta, es decir, siguiendo un modelo de desarrollo definido. Al hablar aquí de metodología nos referimos a metodología global de desarrollo de software, representada a nivel general por un modelo de ciclo de vida. Por supuesto, esta metodología de desarrollo puede detallarse tanto como se desee, especificando además metodologías concretas de análisis, diseño, programación, etc. Como ya se ha indicado, un entorno global de estas características se suele designar con las denominaciones IPSE, ICASE, o simplemente ISEE (integrated Software Engineering Environment).

La característica principal que distingue un entorno de esta clase de un banco de trabajo amplio es el soporte explícito de un modelo global de desarrollo. El entorno debe poseer, por tanto, las características de integración del proceso, tal como se describían en la sección 6.6.4, además de las de integración de datos, control y presentación.

320 Introducción a la Ingeniería de Software

Para conseguir este nivel de integración es necesario contar con un modelo formal del proceso de desarrollo. A diferencia de las metodologías parciales de análisis y diseño, este modelo de desarrollo suele construirse más o menos a medida de cada empresa productora de software. Por esta razón no existen, en general, productos comerciales disponibles en forma inmediata. Un ISEE de uso general deberá permitir, entonces:

- construir la definición formal del modelo del proceso de desarrollo, y
- asegurar la aplicación práctica del modelo definido.

Como ya se ha indicado, no hay, en general, entornos ISEE disponibles directamente como productos comerciales. Lo que sí existen son esquemas generales de arquitectura de entornos orientados al proceso, que en algunos casos han dado lugar al desarrollo de colecciones de herramientas que facilitan, al menos en parte, las funciones deseadas. Mencionaremos a continuación algunas de estas propuestas.

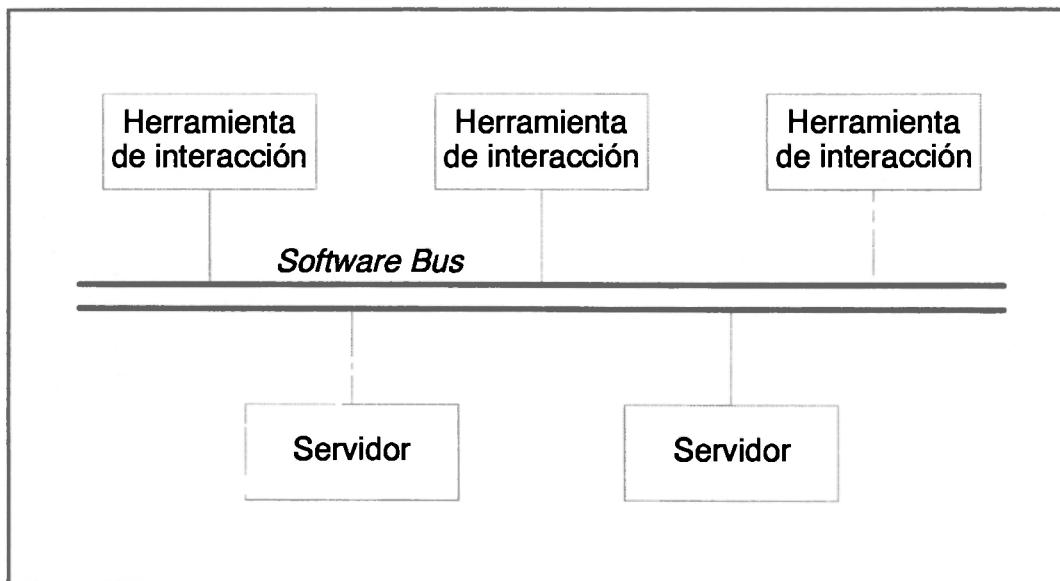


Figura 6.5 Arquitectura del entorno ESF

PCTE (PORTABLE COMMON TOOL ENVIRONMENT) es una arquitectura de entorno integrado basada en un repositorio común. El elemento principal de esta propuesta es la definición de la interfaz de acceso a dicho repositorio. El repositorio puede almacenar objetos de diferentes clases, tales como documentos, diagramas o especificaciones de distintos elementos definidos en el modelo de desarrollo. Sobre este repositorio pueden operar herramientas que

automaticen las actividades previstas en el modelo del proceso. Existen implementaciones de repositorios que cumplen con la especificación PCTE, y también algunas colecciones de herramientas, como las correspondientes al proyecto PACT.

ESF (EUREKA SOFTWARE FACTORY) define otro modelo de arquitectura [Fernström92], cuyo elemento central de integración es el denominado "software bus", que constituye una interfaz normalizada para interconexión de herramientas, tal como se indica en la figura 6.5. En ESF se distinguen dos clases de herramientas: *servidores* y *herramientas de interacción*. Los servidores pueden realizar las funciones de repositorio, tanto centralizado como distribuido, y suministrar servicios o funciones automatizadas. Las herramientas de interacción permiten la comunicación con los usuarios, que pueden acceder a los repositorios y a los servicios a través de ellas.

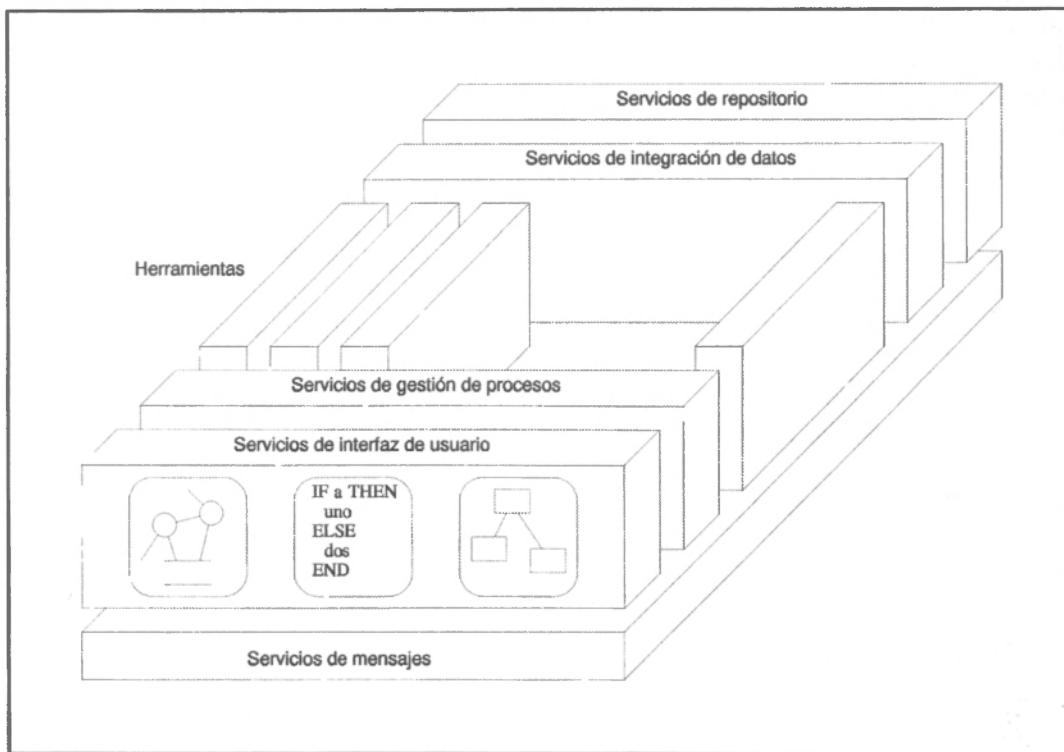


Figura 6.6 Modelo de referencia NIST/ECMA

El MODELO NIST/ECMA, propuesto por estos organismos (National Institute of Standards and Technology, y European Computer Manufacturers Association). En este modelo [Chen92], representado en la figura 6.6, se contempla una infraestructura fija, compuesta por

322 Introducción a la Ingeniería de Software

elementos que proporcionan integración de datos, basada en un repositorio común, integración de presentación, mediante un soporte global de interfaz de usuario, e integración del control, basada en la gestión de procesos y mensajes. El entorno puede particularizarse para un modelo de desarrollo determinado instalando sobre estos elementos fijos una colección particular de herramientas.

En ausencia de productos CASE listos para usar y capaces de integrarse en una arquitectura uniforme, se puede adoptar un enfoque pragmático, y combinar de la mejor manera posible productos heterogéneos para construir un entorno global. Un ejemplo concreto es el entorno ESSDE, recomendado para proyectos de la ESA (European Space Agency). Este entorno está construido alrededor de dos productos comerciales concretos: Concerto (de SEMA Group) y Rational Ada. El primero es una herramienta CASE de diseño que soporta la metodología HOOD. El segundo es un entorno de programación en lenguaje ADA. A estos elementos se han añadido herramientas adicionales para gestión de configuración, preparación de documentación, etc.

