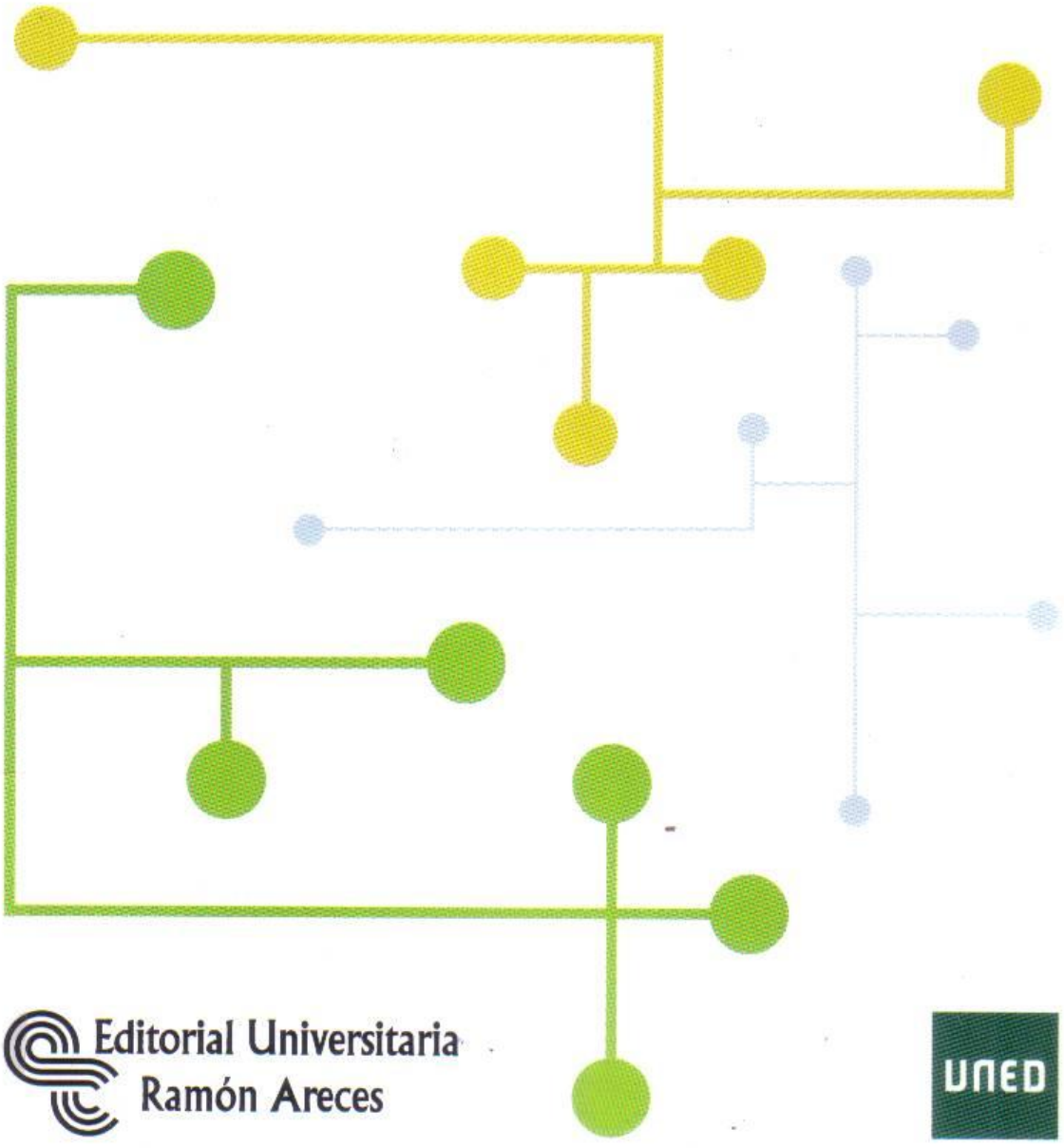
Sebastián Rubén Gómez Palomo
Eduardo Moraleda Gil

APROXIMACIÓN A LA INGENIERÍA DEL SOFTWARE



Editorial Universitaria Ramón Areces

UNED

Índice general

FUNDACIÓN CULTURAL STA. TERESA BIBLIOTECA UNED-AVILA

1. Introducción **17**
- 1.1. Introducción . . . 17
- 1.2. Objetivos . . . 18
- 1.3. ¿Qué es el software? . . . 18
  - 1.3.1. Calidad del software . . . 19
  - 1.3.2. Tipos de software . . . 21
- 1.4. ¿Cómo se fabrica el software? . . . 23
- 1.5. Mitos del software . . . 25
- 1.6. Conclusiones . . . 27
- 1.7. Ejercicios propuestos . . . 27

2. El Ciclo de Vida del Software **29**
- 2.1. Introducción . . . 29
- 2.2. Objetivos . . . 29
- 2.3. El ciclo de vida de un producto . . . 29
- 2.4. El ciclo de vida del software . . . 31
- 2.5. Fases del ciclo de vida del software . . . 32
  - 2.5.1. Análisis . . . 32
  - 2.5.2. Diseño . . . 32
  - 2.5.3. Codificación . . . 33
  - 2.5.4. Integración . . . 33
  - 2.5.5. Explotación . . . 33
  - 2.5.6. Mantenimiento . . . 33
- 2.6. Documentos que se generan en el ciclo de vida . . . 33
  - 2.6.1. Documento de requisitos del software . . . 34
  - 2.6.2. Documento de diseño del software . . . 34
  - 2.6.3. Código fuente . . . 34
  - 2.6.4. El sistema software . . . 34
  - 2.6.5. Documentos de cambios . . . 34
- 2.7. Tipos de ciclo de vida del software . . . 35
  - 2.7.1. Ciclo de vida en cascada . . . 35
  - 2.7.2. Ciclo de vida en V . . . 36
- 2.8. Prototipos . . . 38
  - 2.8.1. Prototipos rápidos . . . 39
  - 2.8.2. Prototipos evolutivos . . . 40
- 2.9. El modelo en espiral . . . 41
- 2.10. Programación extrema . . . 43

2.11. Mantenimiento del software 45
2.11.1. Evolución de las aplicaciones 46
2.11.2. Gestión de cambios 47
2.11.3. Reingeniería 47
2.12. Garantía de calidad de software 48
2.12.1. Plan de garantía de calidad 48
2.12.2. Revisiones 49
2.12.3. Pruebas 50
2.12.4. Gestión de configuración 50
2.12.5. Normas y estándares 52
2.13. Conclusiones 54
2.14. Ejercicios propuestos 54

3. Especificación de Requisitos **55**
3.1. Introducción 55
3.2. Objetivos 55
3.3. Modelado de sistemas 56
3.3.1. Concepto de modelo 57
3.3.2. Técnicas de modelado 57
3.4. Análisis de requisitos de software 62
3.4.1. Objetivos del análisis 63
3.4.2. Tareas del análisis 66
3.5. Notaciones para la especificación 69
3.5.1. Lenguaje natural 70
3.5.2. Diagramas de flujo de datos 71
3.5.3. Diagramas de transición de estados 77
3.5.4. Descripciones funcionales. Pseudocódigo 79
3.5.5. Descripción de datos 82
3.5.6. Diagramas de modelo de datos 84
3.6. Documento de especificación de requisitos 87
3.7. Ejemplos de especificaciones 95
3.7.1. Videojuego de las minas 95
3.7.2. Sistema de gestión de biblioteca 100
3.8. Conclusiones 111
3.9. Ejercicios propuestos 112

4. Fundamentos del Diseño de Software **113**
4.1. Introducción 113
4.2. Objetivos 113
4.3. ¿Qué es el diseño? 114
4.4. Conceptos de base 117
4.4.1. Abstracción 118
4.4.2. Modularidad 121
4.5. Refinamiento 122
4.5.1. Estructuras de datos 123
4.5.2. Ocultación 124
4.5.3. Genericidad 125
4.5.4. Herencia 127

4.5.5. Polimorfismo 129
4.5.6. Concurrencia 131
4.6. Notaciones para el diseño 132
4.6.1. Notaciones estructurales 133
4.6.2. Notaciones estáticas 140
4.7. Documentos de diseño 150
4.7.1. Documento ADD 150
4.7.2. Documento DDD 154
4.8. Conclusiones 155
4.9. Ejercicios propuestos 156

5. Técnicas Generales de Diseño de Software **157**
5.1. Introducción 157
5.2. Objetivos 157
5.3. Descomposición Modular 158
5.3.1. Independencia funcional 159
5.3.2. Adaptabilidad 167
5.4. Técnicas de diseño funcional descendente 169
5.4.1. Desarrollo por refinamiento progresivo 169
5.4.2. Diseño estructurado 174
5.4.3. Ejemplo: Sistema de gestión de biblioteca 177
5.5. Técnicas de diseño basado en abstracciones 178
5.5.1. Descomposición modular basada en abstracciones 179
5.5.2. Método de Abbott 180
5.5.3. Ejemplo: Videojuego de las minas 183
5.6. Técnicas de diseño orientadas a objetos 185
5.6.1. Diseño orientado a objetos 186
5.6.2. Ejemplo: Estación meteorológica 188
5.7. Técnicas de diseño de datos 196
5.8. Diseño de bases de datos relacionales 196
5.8.1. Formas normales 197
5.8.2. Diseño de las entidades 199
5.8.3. Tratamiento de las relaciones de asociación 199
5.8.4. Tratamiento de las relaciones de composición 201
5.8.5. Tratamiento de la herencia 201
5.8.6. Diseño de Índices 201
5.8.7. Ejemplo: Diseño de datos para la gestión de biblioteca 201
5.9. Diseño de bases de datos de objetos 204
5.10. Diseño de software con patrones 205
5.11. Ejemplos de diseños 210
5.11.1. Ejemplo: Videojuego de las minas 210
5.11.2. Ejemplo: Sistema de gestión de biblioteca 222
5.12. Conclusiones 231
5.13. Ejercicios propuestos 232

6. UML, Lenguaje Unificado de Modelado **233**
6.1. Introducción 233
6.2. Objetivos 233

6.3. ¿Qué es UML? . . . 234
6.4. Orígenes de UML . . . 235
6.5. Objetivos de UML . . . 236
6.6. Estructura de UML . . . 237
6.7. Diagramas UML . . . 244
6.7.1. Diagramas de casos de uso . . . 244
6.7.2. Diagrama de clases . . . 247
6.7.3. Diagramas de secuencia . . . 250
6.7.4. Diagramas de colaboración . . . 251
6.7.5. Diagrama de estado . . . 253
6.7.6. Diagrama de actividad . . . 255
6.7.7. Diagramas de componentes . . . 256
6.7.8. Diagrama de despliegue . . . 257
6.8. Conclusiones . . . 258
6.9. Ejercicios propuestos . . . 258

7. La Codificación del Software **261**
7.1. Introducción . . . 261
7.2. Objetivos . . . 261
7.3. Los lenguajes de programación . . . 262
7.3.1. Primera generación de lenguajes . . . 262
7.3.2. Segunda generación . . . 263
7.3.3. Tercera generación . . . 265
7.3.4. Cuarta generación . . . 268
7.4. Criterios de selección del lenguaje . . . 270
7.5. Aspectos metodológicos . . . 272
7.5.1. Normas y estilo de codificación . . . 272
7.5.2. Manejo de errores . . . 274
7.5.3. Aspectos de eficiencia . . . 277
7.5.4. Transportabilidad de software . . . 278
7.6. Conclusiones . . . 280
7.7. Ejercicios propuestos . . . 281

8. Pruebas de Software **283**
8.1. Introducción . . . 283
8.2. Objetivos . . . 284
8.3. Tipos de pruebas . . . 284
8.4. Pruebas de unidades . . . 285
8.4.1. Pruebas de caja negra . . . 286
8.4.2. Pruebas de caja transparente . . . 291
8.4.3. Estimación de errores no detectados . . . 296
8.5. Pruebas de unidades en programación orientada a objetos . . . 297
8.6. Estrategias de Integración . . . 298
8.6.1. Integración Big Bang . . . 298
8.6.2. Integración descendente . . . 298
8.6.3. Integración ascendente . . . 300
8.6.4. Estrategias de integración en programación orientada a objetos . . . 301
8.7. Pruebas de validación . . . 301

8.8. Pruebas del sistema 302
8.9. Conclusiones 303
8.10. Ejercicios propuestos 304

Capítulo 1

Introducción

1.1. Introducción

El software es una parte principal del entorno humano. La cantidad de aparatos de todo tipo que rodean a las personas, que se usan a diario y sin los que cada vez la vida sería más difícil de imaginar, que están controlados por un programa, por un software que rige su comportamiento es enorme.

Teléfonos, electrodomésticos, coches o cualquier vehículo que circule por una carretera, aviones, barcos, trenes, el aire acondicionado de la casa u oficina, los sistemas de control de edificios, aeropuertos o estaciones, las televisiones, los sistemas de gestión de las empresas, los robots de las fábricas de ensamblaje, la lista es interminable. Todos estos sistemas disponen de uno o más computadores, que constituyen el "hardware" del sistema, y de los programas que gobiernan su funcionamiento, que componen el "software" de los mismos.

El software está presente no sólo en los sistemas informáticos que realizan tareas de tratamiento de información, sino en un sinfín de sistemas de la más diversa complejidad. Son miles, millones de líneas de código que diariamente se programan para conseguir que todos estos sistemas funcionen como se desea. Esta tarea de construir el software la realizan los programadores, los cuales tienen que a su vez dar mantenimiento durante, en la mayoría de los casos, largo tiempo.

La ingeniería es, según la Real Academia de la Lengua, el conjunto de conocimientos y técnicas que permiten aplicar el saber científico a la utilización de la materia y de las fuentes de energía. Se pueden encontrar un sinfín de definiciones de ingeniería, en las que el denominador común es la aplicación práctica del conocimiento para la elaboración de cualquier tipo de producto o servicio.

En este libro se introduce el conjunto de técnicas y procedimientos que se han ido desarrollando a lo largo de las últimas décadas, para poder elaborar de una forma ordenada y eficiente tantas y tantas líneas de código que componen el software. Todas estas técnicas y procedimientos componen la ingeniería de software, una todavía joven rama de la ciencia.

1.2. Objetivos

En este capítulo se da una visión inicial de lo que es el software y cómo se produce. Igualmente se profundiza en el concepto de ingeniería del software. Se pretende que el lector adquiera una idea clara de los siguientes conceptos:

- Definición de software y requisitos de calidad exigibles.
- No todo el software es igual, según su aplicación y modo de desarrollarlo puede ser muy diverso. Se da una visión de los distintos tipos de software.
- Definición del concepto de Ingeniería del software y comprensión de su origen.
- Existen algunas ideas preestablecidas acerca del software que se analizan.

1.3. ¿Qué es el software?

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

El software puede ser en sí mismo un producto que se venda, por ejemplo un procesador de textos o un programa de tratamiento de imágenes, o tan sólo una parte, en la mayoría de los casos esencial, de un producto más complejo, por ejemplo el programa que gobierna la inyección de gasoil en un motor diésel, o puede ser el medio para dar un servicio, por ejemplo el programa que permite realizar una transferencia bancaria. Lo que es indudable es que la elaboración del software ocupa a millones de personas en todo el mundo y se puede considerar una actividad económica en sí misma.

1.3.1. Calidad del software

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, sus dimensiones, etcétera. Desgraciadamente esto 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 MacCall 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 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 el mantenimiento correctivo.
- FLEXIBILIDAD. Es la facilidad para modificar el producto software. Se aplica propiamente al mantenimiento adaptativo y al perfectivo.

- 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.

Comprobar la calidad de un software es una tarea compleja. Las pruebas o ensayos consisten en hacer un producto software o una parte de él en condiciones determinadas, y comprobar si los resultados son 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 de 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.

En la figura 1.1 podemos observar de forma simplificada la evolución de la tasa de fallos de un software en el tiempo. Inicialmente esta tasa es muy alta. Según se van corrigiendo los errores se reduce rápidamente. Sin embargo, a lo largo de la vida de un software es frecuente realizar mejoras funcionales, dando lugar a distintas versiones. Cada vez que introducimos cambios en las nuevas versiones, el número de errores del software se dispara, haciendo de nuevo necesario la corrección de los mismos.

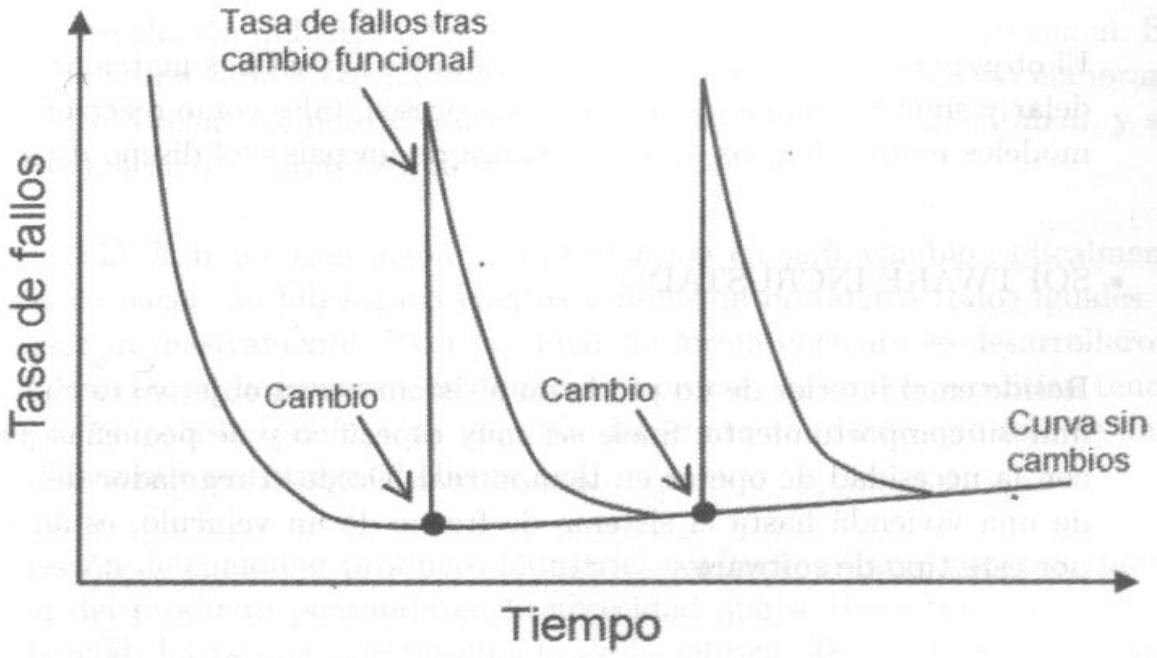


Figura 1.1: Evolución de fallos en un Sistema Software

1.3.2. Tipos de software

Clasificar el software no es una tarea fácil debido a la gran variedad de aplicaciones y métodos de desarrollo que existe. Una de las clasificaciones más completas se encuentra en [Pressman10], donde se agrupa el software en siete grandes categorías:

- SOFTWARE DE SISTEMAS

  Lo forman todos aquellos programas necesarios para dar soporte a otros programas, como los sistemas operativos, los compiladores o los programas de gestión de redes. Su principal característica es su alto grado de interacción con el hardware, ya que en muchos casos deben gestionar de forma eficiente el acceso al hardware por parte de otros programas o usuarios.

- SOFTWARE DE APLICACIÓN

  Son aplicaciones desarrolladas para resolver problemas específicos de los negocios. En esta categoría incluiríamos el software de gestión de los bancos o de las grandes empresas en general, como los ERP (Enterprise Resource Planning).

- SOFTWARE DE INGENERÍA Y CIENCIAS

El objetivo es la programación de elaborados algoritmos matemáticos para modelar y simular complejos sistemas o procesos, tales como reacciones nucleares, modelos metereológicos, la red eléctrica de un país o el diseño de un avión.

- SOFTWARE INCRUSTADO

Reside en el interior de un producto o sistema, y su objetivo es controlarlo, definir su comportamiento. Suele ser muy específico y de pequeñas dimensiones, con la necesidad de operar en tiempo real. Desde el regulador de temperatura de una vivienda hasta el sistema de frenos de un vehículo, están gobernados por este tipo de software.

- SOFTWARE DE LINEA DE PRODUCTO

Su objetivo es dar una determinada funcionalidad al consumidor. En esta categoría encontramos procesadores de texto, hojas de cálculo o las aplicaciones de contabilidad para pequeñas empresas.

- APLICACIONES WEB ("WEBAPPS")

En los últimos años se ha extendido su utilización con la generalización de los aparatos móviles con acceso a redes. Inicialmente simplemente se componían de archivos de hipertexto para la presentación de información, sin embargo hoy día tienen capacidad de cómputo y está integradas con aplicaciones y bases de datos corporativas. A través de ellas se puede operar una cuenta bancaria, realizar todo tipo de compras, utilizar juegos muy elaborados ó conocer el tiempo en cualquier parte del mundo. La comodidad, rapidez y vistosidad son determinantes a la hora de que tengan éxito.

- SOFTWARE DE INTELIGENCIA ARTIFICIAL.

Incluye aplicaciones de robótica, visión artificial, redes neuronales o sobre la teoría de juegos. Utilizan algoritmos no numéricos para la resolución de los problemas, como por ejemplo árboles lógicos de búsqueda.

1.4. ¿Cómo se fabrica el software?

Antes de la revolución industrial, los productos realizaban de forma artesanal. El artesano aprendía una serie de técnicas y procedimientos que le permitían elaborarlos manualmente. Dichas técnicas y procedimientos rara vez se documentaban, y se transmitían de maestro a aprendiz a lo largo de los años.

En el siglo XIX la industrialización y la producción en serie cambio radicalmente esta forma de hacer. Se fabricaban cientos y miles de productos todos iguales y se comercializaban masivamente. Para producir de forma eficiente se desarrollaron una serie de técnicas, procedimiento, estándares y normas, los cuales permitían tener perfectamente controlado todo el ciclo de vida del producto, desde que éste es una simple idea, hasta que se entrega al cliente final.

La elaboración de cualquier producto industrial conlleva un largo proceso, desde la concepción del producto pensando en la necesidad que se desea que cubra y su mercado potencial, hasta que sale terminado de la fábrica. Todo este proceso está concebido para lograr los objetivos de calidad y coste que se consideren necesarios para poder venderlo.

De igual forma, 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, aumentó también la complejidad de las aplicaciones a programar, y se apreció 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 automaticen las labores triviales y repetitivas.

La identificación formal del problema origina una frenética actividad para la creación de metodologías concretas de desarrollo y, en general, en la concepción de la ingeniería del software como disciplina. A finales de los años 60, se acuña el termino Ingeniería del Software en un congreso de la OTAN de manera formal. Con esta denominación se designa el empleo en el desarrollo del software, de técnicas y procedimientos típicos de la ingeniería en general.

El software tiene una particularidad especial frente a cualquier producto físico que podamos imaginar: una vez diseñado, este se puede replicar con tremenda facilidad, sin necesidad de un proceso de fabricación propiamente dicho. A pesar de ello, la ingeniería del software se ha desarrollado a partir de la ingeniería industrial.

La norma de calidad [ISO1694908] es un conjunto de normas de calidad y gestión de la calidad, adoptadas por la industria del automóvil como herramienta básica para desarrollar los procesos necesarios en la producción de un vehículo de forma competitiva y satisfactoria para el cliente.

En el capítulo 2 de este libro se analizan los procesos más importantes para la producción del software y diferentes modelos para organizarlos, de forma similar a los procesos mostrados en la figura 1.2 para la industria del automóvil.

Figura 1.2: Enfoque de calidad en la industria del automóvil

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

Concretando, se tomará la definición de ingeniería del software de [SWEBOK04]. Ingeniería de software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software, y el estudio de estos enfoques, es decir, la aplicación de la ingeniería al software.

A lo largo de los años 70 aparecen las herramientas CASE (Computer Aided Software Engineering) de diseño asistido por ordenador, que se aplican ampliamente durante los 80. Las herramientas CASE tradicionales apoyaban a las actividades anteriores a la programación o codificación, para la que se seguías empleando herra-

mientas tradicionales, como los compiladores, que funcionaban de forma totalmente separada de la herramienta CASE.

En los 90 se amplía el campo de aplicación de las herramientas de ingeniería 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).

Con la llegada del siglo XXI, el desarrollo e implantación de Internet, muchos de los planteamientos de las herramientas CASE han evolucionado. Por un lado la posibilidad de distribuir el desarrollo de los proyectos en diferentes localizaciones y por otro la obligada necesidad de la utilización de esta plataforma como marco para el funcionamiento de la mayoría de las aplicaciones. Otro de los grandes retos que ha aparecido y triunfado recientemente son los "smartphones" o teléfonos inteligentes. Por el momento sólo se utilizan como plataformas de funcionamiento, pero no hay duda de que en breve formarán parte de las infraestructuras empleadas para el desarrollo del software.

A lo largo de estos períodos de tiempo fue surgiendo una importante comunidad científica en torno a la ingeniería de software. Dos organizaciones han liderado dicha comunidad, ACM e IEEE Computer Society, que han promovido activamente la puesta en práctica de esta disciplina. IEEE ha desarrollado la guía SWEBOK con el objeto de crear una acreditación para la profesión de ingeniero del software en Estados Unidos. Dicha guía da forma a los conocimientos necesarios para dominar esta disciplina y los diferenciales frente a otras relacionadas con el software, como las ciencias de la computación o la gestión de proyectos de software.

Todavía a día de hoy, la ingeniería de software se encuentra en una situación permanente de fuerte evolución, con avances continuos en las técnicas que resultan sin embargo insuficientes en cada momento. De hecho se discute si el término ingeniería del 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.

1.5. 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 inmateriali-

dad,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.

Alguna 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 manos 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 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 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.

1.6. Conclusiones

El software lo componen el conjunto de programas que gobiernan el comportamiento de cualquier sistema basado en computador. En muchos casos el software tiene entidad en sí misma, y puede ser considerado un producto propiamente dicho.

La aplicación del conocimiento y del método científico a la elaboración del software, dan lugar a la disciplina que se conoce como ingeniería del software.

El ciclo de vida del software, igual que el de cualquier otro producto que se pueda elaborar, es la evolución del mismo desde el momento que se concibe hasta que se deja de utilizar.

1.7. Ejercicios propuestos

1. Piénsese en diez sistemas con los que se interactúe usted la vida diaria que estén controlados por algún tipo de programa. Hágase un listado e intente describir la funcionalidad de cada uno de ellos.
2. Supóngase el diagrama de las tasas de fallos de un coche en el tiempo. ¿Qué diferencias puede tener respecto a un producto de software?
3. Compárase el ciclo de vida de una lavadora con el de un programa de edición de texto. Elabórese un listado con las particularidades de cada uno de ellos.
4. Repásese las siete categorías de software y dar diez ejemplos de cada una de ellas.
5. Una práctica muy extendida para reutilizar software es "cortar y pegar" fragmentos de código. Esto puede llevar a que un trozo de código se encuentre

repetido muchas veces en un programa. ¿Cómo afecta esto al mantenimiento del programa? ¿Cómo se podrían solventarse los efectos negativos del código duplicado?

6. Los principales términos acuñados en este capítulo son software e ingeniería de software. Explíquense razonadamente las diferencias entre ambos términos.

7. Arguméntese la corrección o falsedad de las siguientes afirmaciones:

   - "Los requerimientos de software cambian continuamente, pero el cambio se asimila con facilidad debido a que el software es flexible".
   - "Una vez que se escriba el programa y se hace que funcione, el trabajo ha terminado"
   - "La ingeniería de software hará que se genere documentación voluminosa e innecesaria, e invariablemente retrasará el trabajo".

Capítulo 2

El Ciclo de Vida del Software

2.1. Introducción

Uno de los conceptos que se estudian en este curso es el ciclo de vida del software. En este capítulo se va a definir qué es el ciclo de vida de un producto, cómo es el ciclo de vida del software, qué fases tiene y qué posibles esquemas organizativos pueden tener esas fases, dando lugar a ciclos de vida diferentes. Se presentará además qué tipo de ciclo de vida es el más adecuado según qué proyecto o producto se esté desarrollando. La descripción pormenorizada de las diferentes fases del ciclo de vida del software serán descritas en posteriores capítulos de este libro.

2.2. Objetivos

El alumno, después de leer este capítulo, debe conocer qué es el ciclo de vida de un producto comercial y en particular del software. Así mismo debe conocer las diferentes fases que integran este ciclo de vida, la documentación resultante de cada fase y los diferentes enfoques en la organización del ciclo de desarrollo del software. Para completar el capítulo se presenta la fase de mantenimiento del software.

2.3. El ciclo de vida de un producto

El ciclo de vida es algo propio de un ser vivo; nace, crece o cambia, se reproduce o diversifica y muere o desaparece. El estudio de cualquier concepto productivo, desde una óptica del ciclo de vida, lleva implícito una evolución temporal que debe ser tenida en cuenta en todos los momentos del ciclo. Cuando un ser vivo nace, igulamente lleva implícitas cualquiera de las fases posteriores por las que va a pasar su vida.

Contemplar la producción de un bien desde este planteamiento obliga, desde el momento de su concepción, a tener en cuenta que a lo largo de su vida útil va a pasar por la secuencia de fases de su ciclo.

Cuando la producción de un bien se hacía de modo artesanal, se contemplaba como un hecho aislado que se realizaba por encargo y se entregaba cuando estaba finalizada su elaboración. Dependía de la habilidad del artesano, de su honestidad y de su capacidad de convicción, que el cliente quedara satisfecho con el producto encargado.

Desde el punto de vista de la producción industrial de cualquier bien, hay que distinguir entre las fases de ingeniería, producción y comercialización. En cada una de estas fases se encuentran distintos ciclos de vida de los productos que se "ingenian" o desarrollan, se producen y finalmente se comercializan.

El ciclo de vida en la ingeniería pasa por etapas como prediseño o diseño preliminar, diseño básico, diseño ejecutivo y diseño final. También se añaden a este ciclo de vida las acciones de diseño necesarias durante el período de explotación o funcionamiento del producto, acciones de mantenimiento y de mejora del producto.

Si se plantea cómo sería el ciclo de vida en la producción industrial de un bien, también tendría sus fases. Desde la implantación de su producción, introducción de cambios, adaptaciones o nuevos elementos en la planta de producción, al mantenimiento de la producción del bien, aseguramiento de la calidad de la producción, optimización de los recursos, eficiencia de los procesos que intervengan en la producción y la eliminación de la producción del bien con el reciclaje de los recursos existentes para producir el bien, reciclado de la mano de obra y eliminación final de la producción. Todas estas actividades serían realizas por la dirección de la planta de producción o serían encargadas a una empresa de ingeniería que lo organizase.

Si se analiza el ciclo de vida de un producto en el mercado, se encontraría la evolución que experimentan las ventas de ese producto mientras se encuentra disponible en el mercado. Las fases por las que pasa este ciclo suelen ser: introducción en el mercado, crecimiento, madurez y declive. Dependiendo del tipo de producto del que se trate, estas fases pueden tener duraciones, actividades y costes distintos. Estas son actividades propias del marketing y se realizan por la sección de marketing de la empresa o son subcontratadas a empresas especializadas en estos campos.

Teniendo en cuenta las características particulares del producto "software" ya explicadas en el capítulo 1, a continuación se verá cómo es el ciclo de vida del software.

2.4. El ciclo de vida del software

Según lo que se ha explicado en el epígrafe anterior, deberían contemplarse cualquiera de los posibles ciclos de vida mencionados. El software es un producto comercial, que debe ser generado de acuerdo con las pautas utilizadas en el sector de la producción industrial y que debe ser comercializado con prácticas adecuadas del marketing.

Así, cuando el software se está diseñando, se debe tratar como un producto de ingeniería Cuando se está produciendo, como cualquier producto producido por la industria. Finalmente, cuando se está comercializando, deben seguirse las pautas habituales del mercado. Evidentemente deben considerarse las particularidades del producto en cualquiera de las citadas fases, pero no se debe olvidar lo genérico del mismo.

El ciclo de vida del software que se va a presentar en sus diferentes modalidades, estará centrado en la parte de ingeniería, no en vano, ya que éste es un curso para ingenieros. Se dejará para otras asignaturas u otras profesiones los otros ciclos de vida. Se estudiará el ciclo de vida de la ingeniería de software. El ciclo de vida del software abarca el proceso de desarrollo y el mantenimiento necesario durante su explotación.

Las fases en el proceso de desarrollo del software suelen ser las siguientes:

- Análisis
- Diseño
- Codificación
- Integración
- Mantenimiento

Cada una de estas fases lleva consigo una serie de tareas que deben realizarse, también conocidas como actividades. Estas tareas generan, como resultado, documentos donde se presenta el trabajo realizado. Estas diferentes fases deben poder completarse por grupos de trabajo independientes, quienes trabajarán de manera secuencial o simultáneamente. El producto del trabajo de un grupo será utilizado por el grupo de trabajo siguiente.

Imaginemos un arquitecto que, después de escuchar los deseos o requisitos de un cliente y analizar sus preferencias, le plantea un posible diseño de la casa que le

ha encargado. El trabajo del arquitecto consiste en recoger toda la información del cliente y procesarla para comprobar que las peticiones del cliente son posibles de realizar dentro del presupuesto establecido. Después elabora unos planos y una memoria de ejecución del proyecto. Los planos y el proyecto son el resultado de esta fase.

Estos documentos serán utilizados posteriormente por el cliente que los apruebe, por la administración que los supervise y la empresa constructora que materialice el proyecto. Algo extremadamente importante es que estos mismos documentos servirán para que todas las partes implicadas puedan **validar** sus compromisos; el arquitecto con el propietario, el constructor con el arquitecto y con el propietario, el propietario con la administración. El modo de hacerlo será validar que la realidad construida coincide con lo descrito en los planos.

Esto que se acaba de describir es un conjunto de actividades de ingeniería o arquitectura encaminadas a la elaboración de un proyecto. Cuando lo que se quiere construir es software en lugar de una casa, una máquina o una explotación agrícola, las fases de desarrollo de la elaboración del proyecto son bastantes parecidas.

2.5. Fases del ciclo de vida del software

Como se ha comentado en el epígrafe anterior las fases habituales en el proceso de software son: análisis, diseño, codificación, integración y mantenimiento.

2.5.1. Análisis

En esta fase se procede a analizar las necesidades que tienen los usuarios del futuro sistema software y que deben ser satisfechas mediante el funcionamiento del mismo. El cliente que realiza el encargo expone sus necesidades, requisitos que debe cumplir el software y la empresa que va a realizarlo los recoge y analiza. De acuerdo con esto, la empresa, elabora una especificación precisa del sistema a desarrollar.

2.5.2. Diseño

Consiste en elaborar un esquema o diseño donde se contemplen los elementos necesarios para que el sistema funcione según con lo especificado en el análisis. En esta fase no sólo se debe diseñar el sistema para su funcionamiento, también debe establecerse la organización del sistema para su construcción. Un adecuado diseño permite la optimización de los recursos en la producción del mismo. El resultado de la fase de diseño suele ser un documento de carácter gráfico, donde se presentan todos los elementos componentes del sistema y la organización pormenorizada de cada uno de ellos. En la fase de diseño se elaboran los planos de lo que se va a construir.

2.5.3. Codificación

En esta fase se produce materialmente lo que va a hacer funcionar el sistema software. Se construirá, por separado, cada uno de los elementos que se han definido en la fase de diseño utilizando para ello las herramientas pertinentes: lenguajes de programación, sistemas de bases de datos, sistemas de información, etc. Así mismo se construirán los elementos necesarios para la comprobar que lo construido funciona correctamente.

2.5.4. Integración

Después de construidos todos los elementos se procede a unirlos todos con el objetivo de construir el sistema completo. En esta fase deben realizarse pruebas exhaustivas para garantizar que el conjunto funciona durante la explotación.

2.5.5. Explotación

Esta fase no forma parte del ciclo de desarrollo de un producto software aunque si influye en el resto de las fases que se están describiendo. Esta fase comprende el período de funcionamiento de la aplicación. Es el objetivo final del producto desarrollado y según su devenir marcará fases posteriores de desarrollo como la de mantenimiento.

2.5.6. Mantenimiento

Durante la fase de explotación del software es necesario realizar cambios, bien para corregir errores no detectados en las fases de desarrollo o para introducir mejoras. Cualquier sistema que se ponga en funcionamiento durante un período de tiempo recibe una casuística ampliada sobre la supuesta en su desarrollo. Ante estas nuevas situaciones de funcionamiento el sistema debe evolucionar para responder a las nuevas demandas. Esta evolución se desarrolla en la fase de mantenimiento.

El resultado de cada una de estas fases se plasma en un documento. Estos documentos permiten independizar las fases y posibilitan que grupos de personas distintas trabajen en cada fase especializándose según la fase en la que se trabaje en analista, diseñador, programador, etc.

2.6. Documentos que se generan en el ciclo de vida

Los trabajos realizados en cada una de las fases del ciclo de vida se describen formalmente en cada uno de los siguientes documentos.

2.6.1. 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 de cómo lo debe hacer. En un proyecto de arquitectura sería una lista detallada, acompañada con diagramas, de todas los elementos que quiere el propietario para su casa.

2.6.2. 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 así como la combinación de unas con otras. En un proyecto de arquitectura equivaldría a los planos que diseña el arquitecto indicando todas y cada una de las partes del edificio, los detalles constructivos particulares incluyendo directivas de cómo se deben unir todos los elementos.

2.6.3. Código fuente

Como producto de la fase de codificación. Contiene los programas fuente, en el lenguaje de programación elegido, debidamente comentados para conseguir que se entiendan con claridad. En cualquier otro proyecto está sería la fase de construcción o de fabricación de cada uno de los elementos necesarios para construir lo que se quiere.

2.6.4. El sistema software

Es lo que se denomina el ejecutable como producto de la fase de integración. Obviamnte no se trata de un tocumento escrito, sin embargo deben documentarse las pruebas realizadas para su puesta en marcha y debe describirse el procedimiento de integración.

2.6.5. Documentos de cambios

Tras cada modificación realizada en la fase de mantenimiento del software debe quedar constancia escrita de las acciones realizadas. Estos documentos suelen tener la siguiente estructura para cada modificación realizada: Información sobre el problema detectado, descripción de la solución adoptada y las modificaciones realizadas sobre el sistema para aplicar dicha solución.

2.7. Tipos de ciclo de vida del software

En esta sección se verá cómo se pueden organizar las diferentes fases que integran el ciclo de vida del software según diferentes enfoques.

2.7.1. Ciclo de vida en cascada

El ciclo de vida en cascada es la secuenciación de las distintas fases de la producción del software que se han descrito. Como elementos de unión entre cada fase aparecen los diferentes documentos que se generan en cada fase. En la figura 2.1 se puede ver la organización de un ciclo de vida en en cascada.

Cada fase se separa claramente de la siguiente lo que permite la independización su realización. Los elementos de unión entre las fases son los documentos generados en las mismas. El modelo en cascada obliga a terminar cada fase antes de comenzar con la siguiente. Cada fase fundamenta su trabajo en los resultados de la anterior. Para detectar errores lo antes posible y evitar que se propaguen a las fases posteriores se establecen procesos de revisión al completar cada fase, antes de pasar a la siguiente.

Esta revisión se realiza sobre la documentación generada en cada fase de manera formal siguiendo una lista de comprobaciones establecida de antemano. Si se detectan errores en una fase será necesario corregirlos en esa fase y todos los puntos del ciclo de vida anteriores.

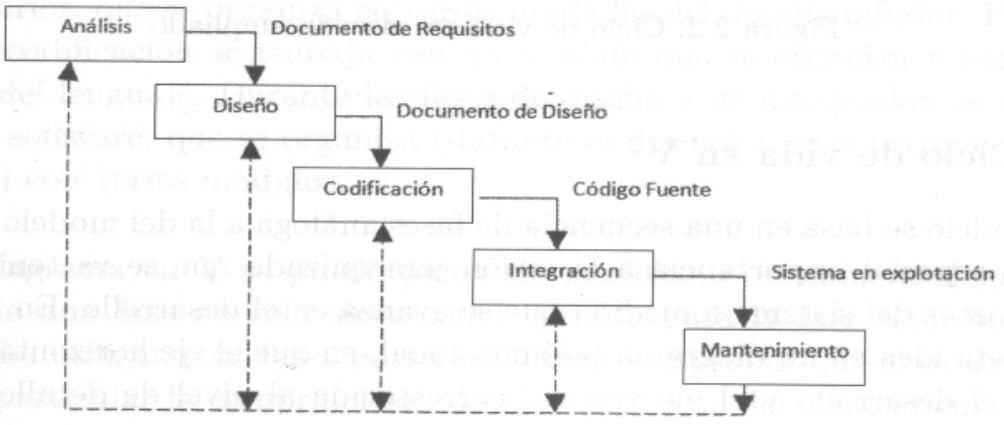


Figura 2.1: Ciclo de vida en cascada

El modelo de ciclo de vida en cascada se puede ampliar y pormenorizar hasta el nivel que se desee dependiendo de la complejidad del sistema que se esté desarrollando. En la figura 2.2 se puede ver un ciclo de vida ampliado donde se contemplan fases adicionales como la Ingeniería de sistemas o la arquitectura del sistema.

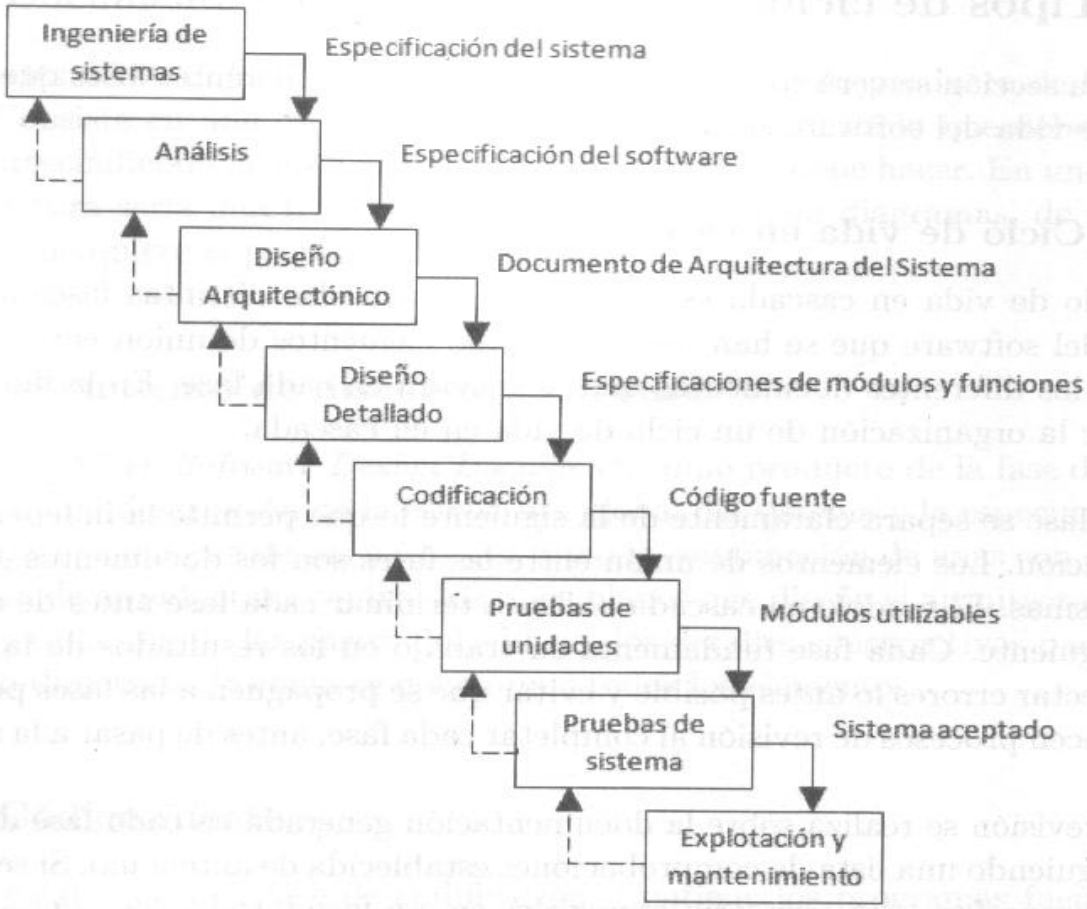


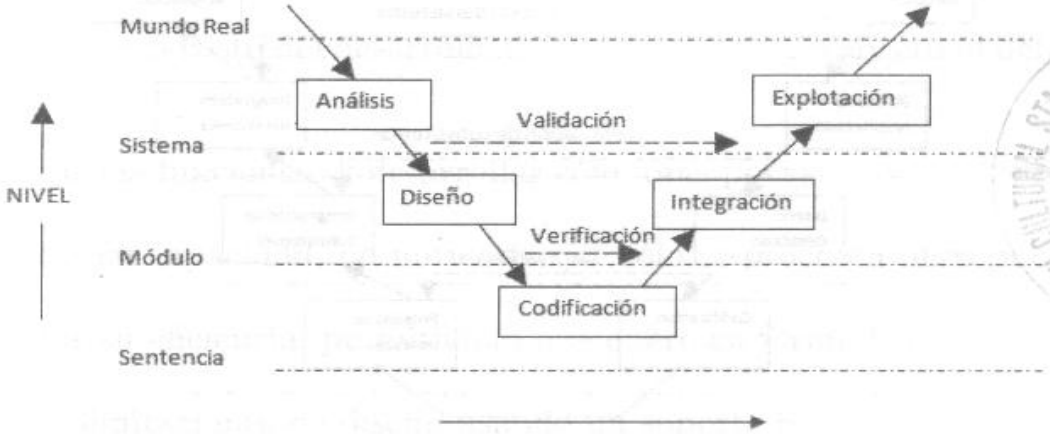
Figura 2.2: Ciclo de vida en cascada ampliado

2.7.2. Ciclo de vida 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 2.3 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 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 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.



BIBLIOTECA
CULTURAL STA. TERESA
UNED-AVILA

Figura 2.3: Ciclo de vida en V

El diagrama de la figura 2.3 corresponde a un modelo en V simplificado. 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 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 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 se denomina ***verificación***, y en el diagrama simplificado de la figura 2.3 aparece a un 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 puede establecer modelos de ciclo en V más elaborados, con un mayor número de fases. En la figura 2.4 se representa una variante ampliada de este modelo.

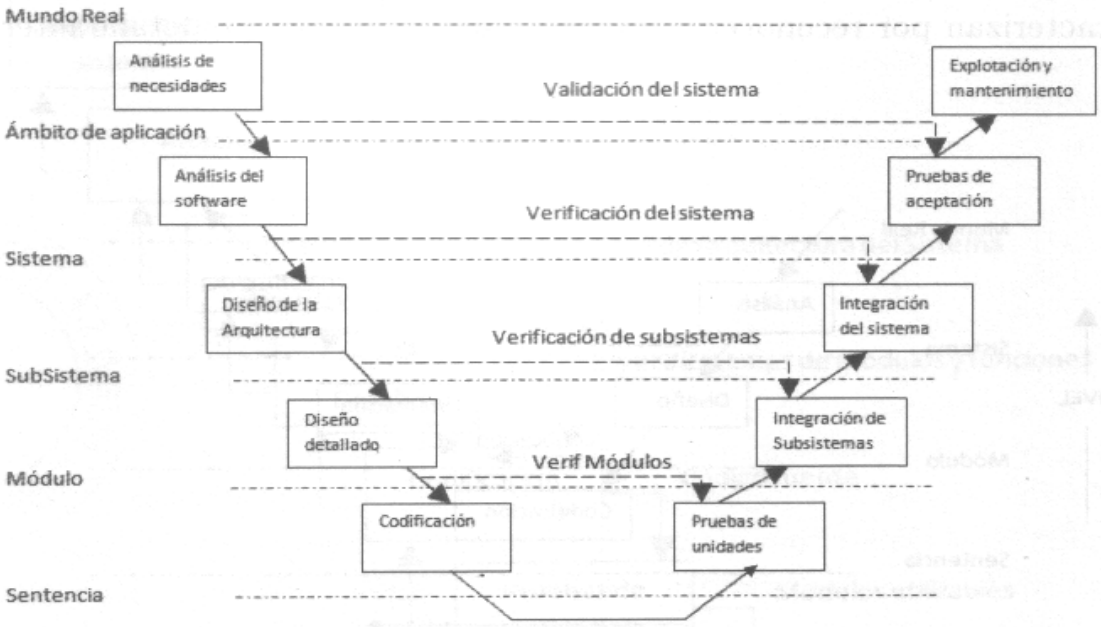


Figura 2.4: Ciclo de vida en V, ampliado

2.8. Prototipos

Qué es un prototipo 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.

2.8.1. 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.

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 2.5 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.

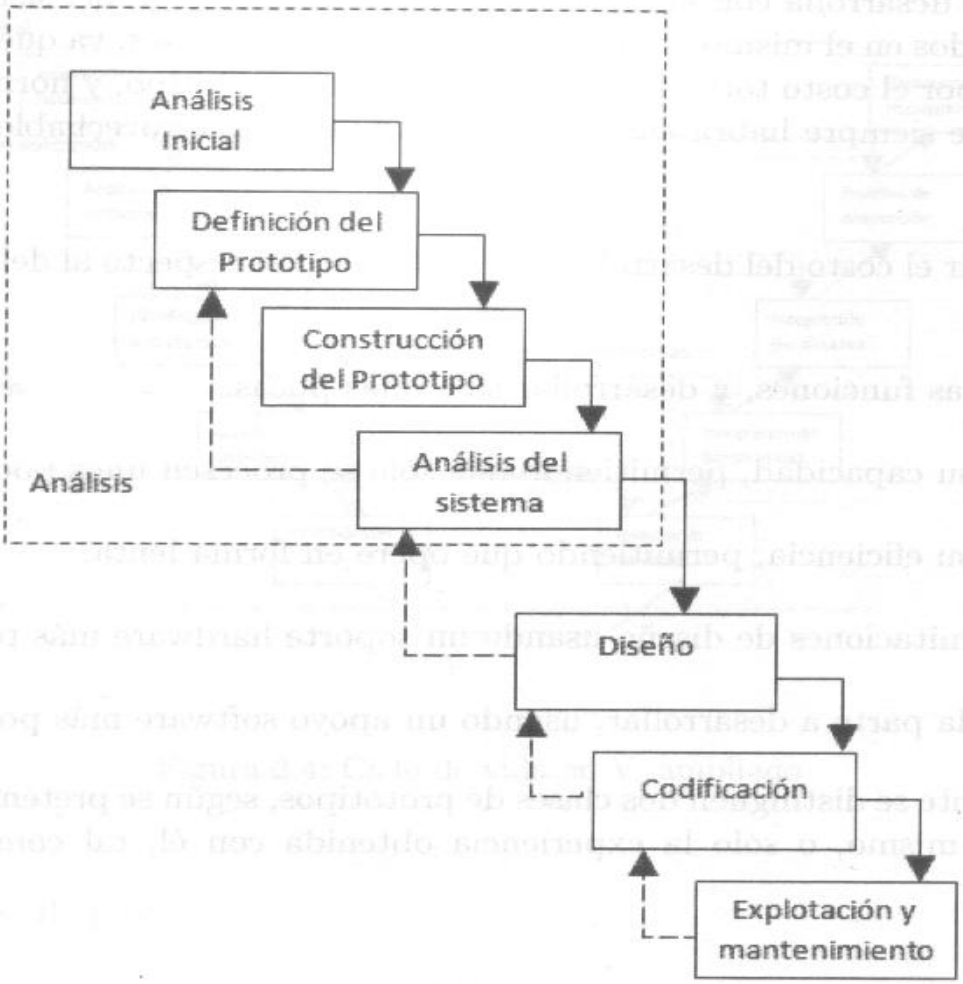


Figura 2.5: Análisis con prototipo rápido

2.8.2. 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 ir convirtiendolo en el sistema final mediante adiciones sucesivas.

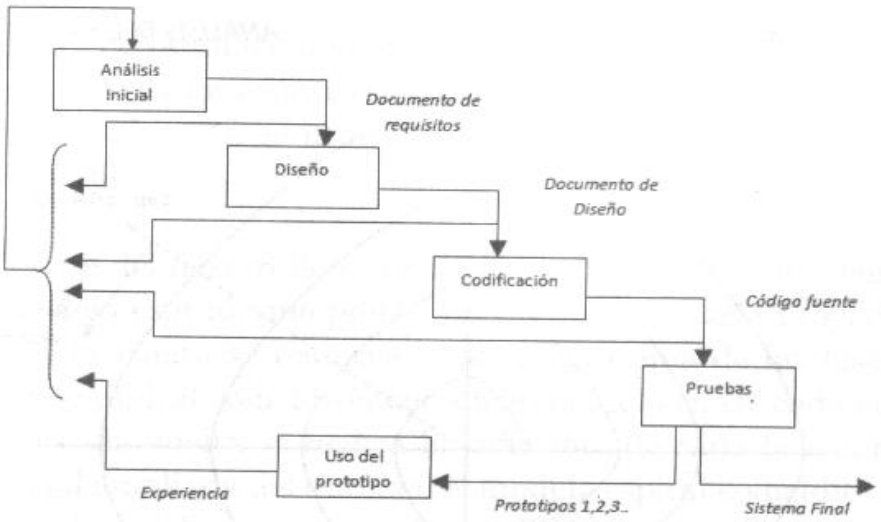


Figura 2.6: Modelo del ciclo de vida evolutivo

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

2.9. 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 2.7

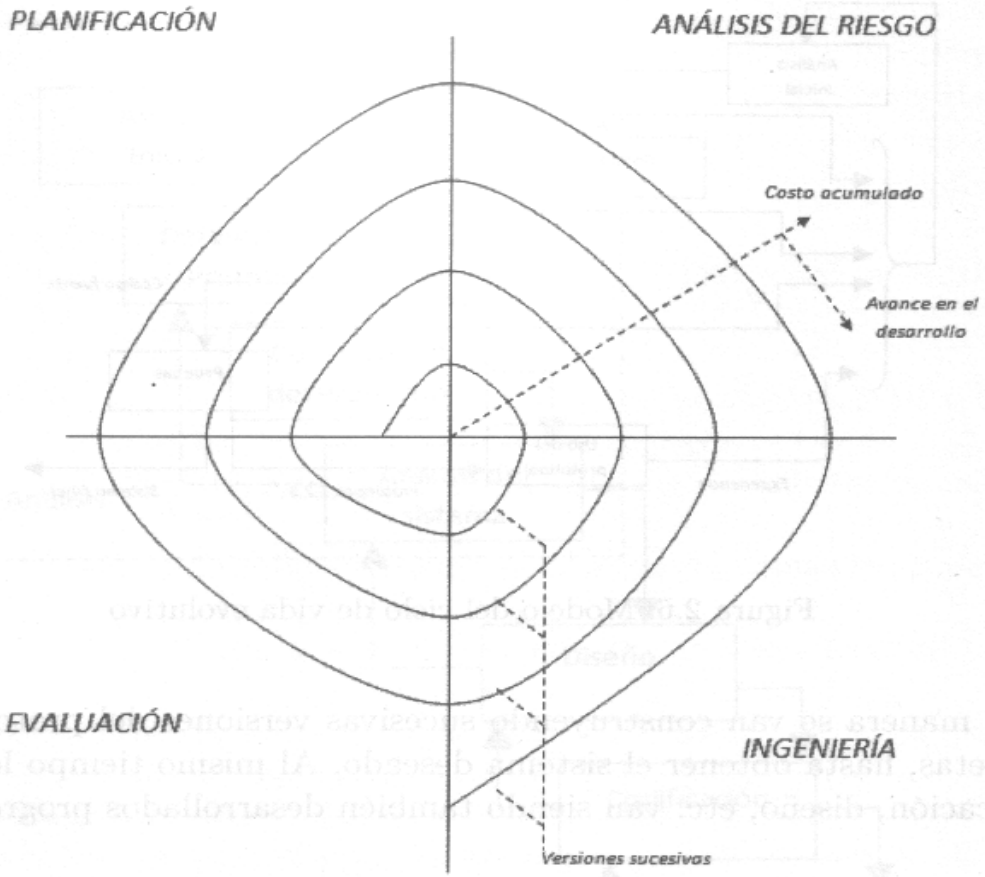


Figura 2.7: Modelo en espiral

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.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.

2.10. Programación extrema

Uno de los riesgos mayores, a los que se enfrentan las empresas de software, es tener a los clientes satisfechos con los productos desarrollados, en cuanto a calidad, tiempo de entrega y precio. Muchas empresas desarrolladoras utilizan metodologías poco o nada rigurosas para ahorrarse los costes de las fases previas a la producción del software y entregar el producto en menos tiempo. Total, los clientes no quieren un montón de planos y el coste de su producción, sino un software que funcione cuanto antes.

Los clientes dan por hecho, que todo lo que ellos pidan, la empresa de software lo sabe hacer bien y con garantías, aunque sea la primera vez que se haga. Si no es así, buscan otra empresa más rápida, mejor y más barata, "que seguro que la encuentran". Estos planteamientos seguro que no los tienen si se trata de adquirir maquinaria, vehículos o construir edificios. El software es así.

El panorama puede resultar desolador, las alternativas son: mantener una metodología estricta, en pos de obtener un producto con garantías, aumentando costes y tiempo de entrega, con riesgo de perder el cliente o dejar la metodología a un lado y atender al cliente cuanto antes y a buen precio con unos riesgos evidentes. Existe otra alternativa: las metodologías de desarrollo ágil, entre las que se encuentra la XP.

La programación extrema es una nueva metodología introducida por Kent Beck [Beck00] con un único objetivo: ser capaz de responder de una forma rápida y de calidad a las exigencias de los clientes, por lo tanto la satisfacción del cliente. Si bien es cierto que éste debería ser el objetivo último de cualquier metodología, la progra-

mación extrema se centra en ello, dando por sentado que los requisitos del cliente cambian a lo largo del proceso y que hay que ser capaz de adaptarse a ellos de una forma muy ágil.

Según el propio Beck, la programación extrema es "un proceso ligero, de bajo riesgo, flexible, predecible, científico y divertido de desarrollar software". El equipo de desarrollo, para lograr este proceso, tiene que basarse en cuatro valores principales, los cuales están muy interrelacionados unos con otros:

SENCILLEZ. Sin ella no es posible crear un código ágil que se adapte de forma rápida a las necesidades cambiantes del cliente. Se debe programar sólo aquello que nos han pedido, sin pensar en lo que nos podrían pedir mañana. Si nos lo piden mañana lo programamos mañana. Además, para conseguir que el software se mantenga sencillo, utilizaremos la recodificación de forma continua, es decir reescribir código antiguo de otra forma que aporte más sencillez.

COMUNICACIÓN. En programación extrema se potencia el trabajo en equipo, tanto dentro del grupo de desarrollo como con el cliente. Algunas prácticas en este sentido son el código autodocumentado, programación por parejas, propiedad colectiva del programa, cliente in-situ o el empleo obligado de la utilización de estándares de codificación.

RETROALIMENTACIÓN. La retroalimentación funciona con el cliente y con el propio sistema. El cliente está integrado dentro del equipo de desarrollo y los ciclos del proceso de software son muy cortos, por lo que en muy poco tiempo se evalúa el diseño y se cambia si no cumple los requisitos del cliente. Por otro lado, las pruebas unitarias sistemáticas en el momento y la integración continua permiten tener conocimiento en tiempo real sobre la respuesta del sistema.

VALENTÍA. La valentía es la que permite a los programadores afrontar la recodificación continua del programa, aceptar las modificaciones de los requisitos, escuchar las peticiones del cliente a lo largo de su trabajo, realizar pruebas unitarias tras la programación de cada unidad o desechar un código obsoleto. La programación extrema fomenta que los miembros del equipo trabajen con confianza, sin esconderse, aportando valor y, por lo tanto, eliminando algunos de los problemas de los grandes grupos de trabajo bajo una metodología más "tradicional". En esta línea Kent Beck introdujo, posteriormente a su primera propuesta, el respeto hacia el trabajo de los otros miembros del equipo como otro valor clave dentro de la programación extrema.

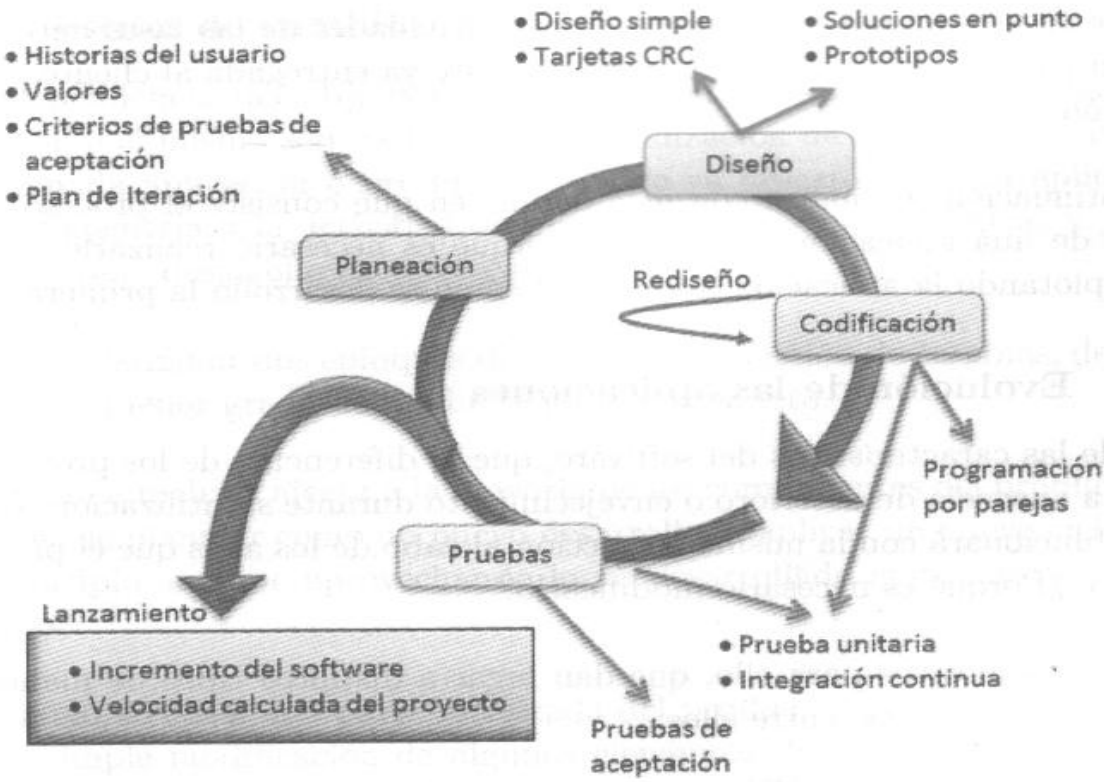


Figura 2.8: Programación extrema por www.codejobs.biz

Desde un punto de vista formal, la programación extrema propone ciclos del proceso de software muy cortos y rápidos, realizando pruebas de unidad inmediatas y una integración continua. De esta forma se van creando tantas pequeñas versiones o prototipos como sea posible, las cuales son testeadas antes de continuar. Cada vez que se considera preciso para eliminar código obsoleto o demasiado complejo, se recodifica, realizando las pertinentes pruebas a continuación para eliminar posibles errores o efectos secundarios del nuevo código. Igualmente se replanifica y rediseña con cada nueva versión, en un proceso iterativo hasta cumplir todos los requisitos del cliente. La figura 2.8 muestra un esquema básico del proceso.

No se entra más a fondo en esta metodología porque será abordada en asignaturas posteriores.

2.11. 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 capítulos. La fase de mantenimiento es algo particular, y se describe en este primer capítulo. Esta etapa final denominada mantenimiento, o explotación y mantenimiento, no suele representar una actividad específica, sino que consiste nor-

malmente en repetir o rehacer parte de las actividades de las fases anteriores par introducir cambios en una aplicación de software ya entregada al cliente y puesta e explotación.

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

2.11.1. Evolución de las aplicaciones

Una de las características del software, que lo diferencian de los productos hard ware, 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 arios, de manera que los elementos básicos hardware y software (máquina + sistema operativo) que constituyen la plataforma o entorno en que se ejecutan evoluciona 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.

2.11.2. 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.

2.11.3. 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 desarrollado 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.

2.12. 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.

En el capítulo 1 ya se revisaron los factores de calidad con los que se valora el producto. A continuación se analiza cómo debe organizarse el proceso de desarrollo para garantizar un nivel de calidad adecuado.

2.12.1. Plan de garantía de calidad

Una adecuada calidad del producto es inalcanzable sin una buena organización del desarrollo. Las técnicas de ingeniería de software tratan de formalizar el proceso de desarrollo evitando los inconvenientes de una producción artesanal.

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.

2.12.2. 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.
- 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.

2.12.3. 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 capítulo 8 se describen técnicas concretas para organizar y realizar pruebas de software.

2.12.4. Gestión de configuración

Para precisar el concepto de configuración se consultará, 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.

Se describe aquí brevemente los elementos básicos de la gestión de configuración. Estas ideas se encuentran más desarrolladas en otros textos generales de ingeniería de software.

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 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.

2.12.5. 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 conteniendo, 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 IS0-9001, 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.

CMMI: son las siglas de Integración de Modelos de Madurez de Capacidades (*Capability Maturity Model Integration).* Es un modelo desarrollado por el Instituto de la Ingeniería del Software *Software Engineering Institute* de EEUU para la mejora de los procesos de las empresas de software que califica las compañías según su nivel de madurez. Por proceso se entiende un conjunto de fases sucesivas que llevan a la obtención de un resultado, y por nivel de madurez, el grado de calidad que alcanzan los procesos. La versión actual es la 1.3.

CMMI establece una serie de buenas prácticas que las empresas deben cumplir para ser consideradas de un grado de madurez determinado a la hora de generar resultados. CMMI no establece como llevar a cabo estas prácticas, simplemente te indica qué se compromisos se deben cumplir.

Los niveles de madurez son 5:

- Nivel 0: Inexistente
- Nivel 1: Inicial
- Nivel 2: Repetible
- Nivel 3: Definido
- Nivel 4: Gestionado
- Nivel 5: Optimizado

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.

METRICA-3: La evolución natural del estandard anterior. En esta versión se tienen en cuenta la división en procesos. Descripción de las tareas de manera sistemática. Incorporación de nuevos estándares (como UML). Soporte para desarrollos orientados a objetos. Interfaces (tareas comunes a todos los procesos). Consideración del mantenimiento dentro de la norma.

2.13. Conclusiones

En este capítulo se ha presentado como se organiza por fases el desarrollo de un sistema software. El conjunto de estas fases se llama ciclo de vida. Esto no es algo particular del software sino que es habitual en cualquier proceso productivo y comercial. En el caso de la producción del software tiene sus particularidades. Se presentan diferentes alternativas de organización del ciclo de vida del software, incluso metodologías "extremas". Para concluir el capítulo se aborda la fase de mantenimiento y una breve presentación de la medida de la calidad del software.

2.14. Ejercicios propuestos

1. Plantéese el ciclo de vida de un nuevo producto, como una bebida, desde la óptica de su diseño, su fabricación y su comercialización.
2. Preguntado un alumno de primer curso de informática por lo primero que tendría que hacerse para construir una casa, respondió: "hacer un agujero". Después de leer este capítulo demuéstrese que eso no es lo más adecuado. Márquese las fases que seguiría.
3. Póngase un ejemplo de proyecto para cada uno de los tipos de ciclo de vida.
4. Búsquese un ejemplo de prototipo de usar y tirar y de prototipo reutilizable en la industria, en la construcción y en la producción de software.
5. Elabórese una lista con 5 posibles riesgos a valorar en un proyecto software.
6. Plantéese unas pautas que seguiría una empresa para ofertar el mantenimiento de una aplicación software que va a producir, pero que también quiere mantener.
7. Búsquese información sobre los niveles de madurez del modelo CMMI.
8. Indáguese sobre la norma Métrica 3.

Capítulo 3

Especificación de Requisitos

3.1. Introducción

Este capítulo 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 capítulo, 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.

3.2. Objetivos

El alumno después de leer este capítulo debe conocer el concepto de especificación de un sistema software, obtención, análisis y validación de los requisitos. Concretamente el lector deberá:

- Comprender la relevancia que para la realización de un sistema software tiene la obtención de los requisitos que tiene que cumplir para satisfacer su funcionamiento de manera correcta y las expectativas del cliente que lo encarga.

- Conocer y distinguir los diferentes tipos de requisitos que se presentan en la elaboración de un sistema software.
- Conocer las técnicas más relevantes para la captura de los requisitos de un sistema y ser capaz de elaborar un documento de especificación de requisitos, SRD.
- Reconocer diferentes notaciones para elaborar los diagramas que se emplean en la elaboración del SRD.

3.3. 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 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. 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 capítulo, 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 capítulo 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.) y 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.

3.3.1. 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 de 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.

3.3.2. 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.

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
- .....

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
- Pagos del IRPF
- .....

Evidentemente, para completar el modelo se tendrán que establecer las interfases entre las partes o subsistemas en que queda dividido, posibilitando 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.

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.

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 prolija 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.

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 desea obtener en el cuadro.

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 esta 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.

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
- .....

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,

la cual necesita un tiempo de acceso que no se puede lograr 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.

3.4. 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, pruebas 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 estar constituido por una o varias personas expertas, en todo o sólo en una parte del trabajo que se pretende automatizar con el sistema a especificar. Por ejemplo, cuando se desea 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 capítulo no se asociará al cliente con la persona o entidad que financia el proyecto. Aquí, el cliente será el encargado de elaborar, junto con el analista, las especificaciones del proyecto de software. 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 otras coincidirá con el cliente que propiamente financia el proyecto e incluso, en otras puede ser simplemente parte de una especificación de otro proyecto de mayor envergadura en el que está encuadrado el nuestro.

3.4.1. 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 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.

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:

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

   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 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 capítulo.

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. Siempre será un buen punto de partida.

3.4.2. 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. Las tareas a desarrollar son las siguientes:

- Estudio del sistema en su contexto
- Identificación de necesidades
- Análisis de las alternativas. Estudio de viabilidad
- Establecimiento del modelo del sistema
- Elaboración del documento de especificación de requisitos. SRD
- Revisión continuada del análisis

A continuación se detallan 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, subsistemas organizativos 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, CO_2 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 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.

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. EL 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 capítulo.

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.

3.5. 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 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.

3.5.1. 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

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:

```
Cuando se teclee la clave 3 veces mal, debe invalidarse 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.

3.5.2. 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 es un sistema procesador de datos. Recibe datos como entrada, los procesa, es decir los ordena, los cuenta, los utiliza para calcular expresiones.

Siendo así un sistema software se podría modelar reflejando 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 3.1 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.

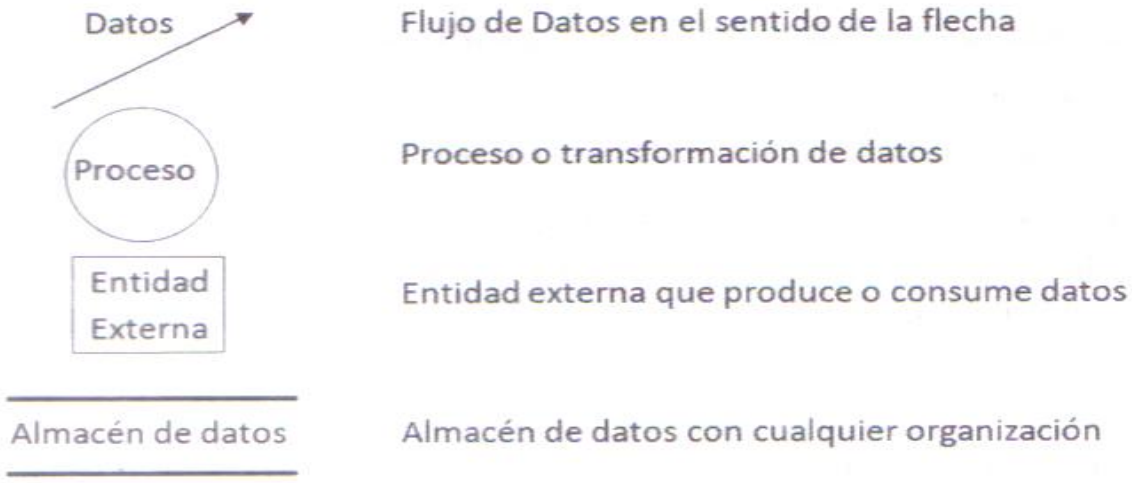


Figura 3.1: Notación DFD básica

Corno 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.

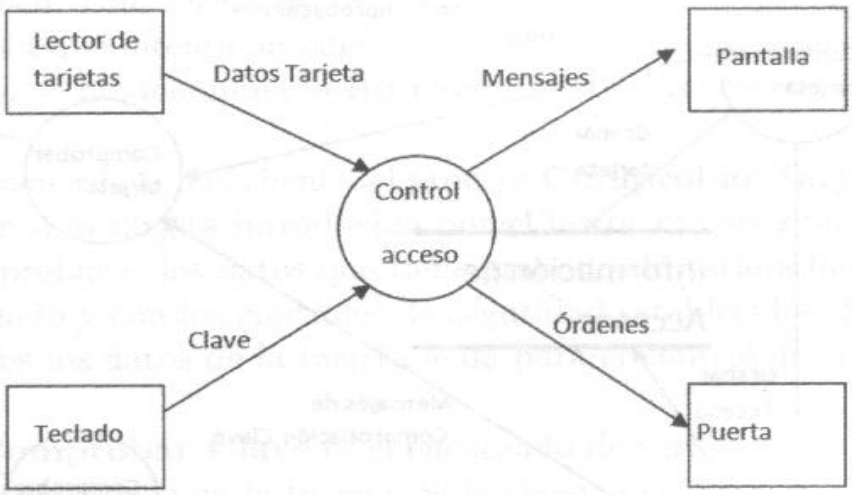


Figura 3.2: DFD de contexto para el sistema de control de accesos

En la figura 3.2 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 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.

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 O. 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 3.3 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 Órdenes.

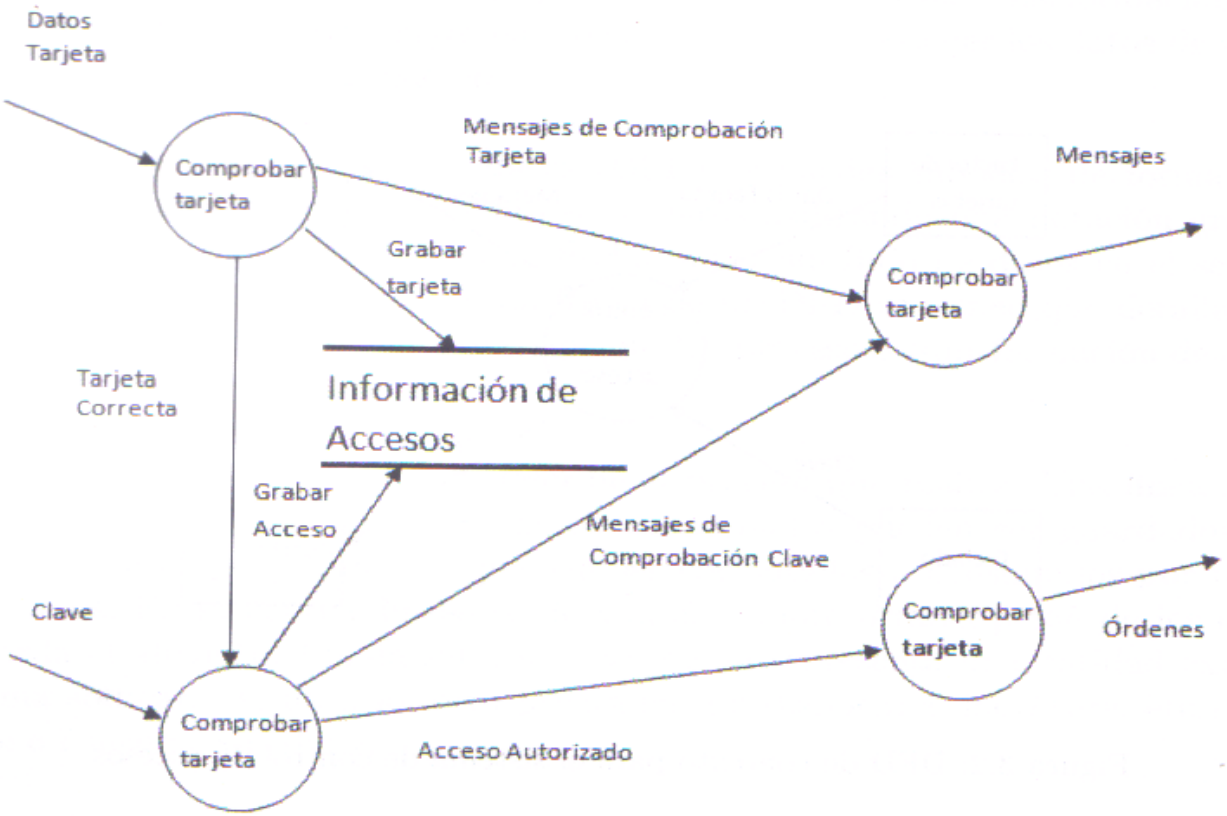


Figura 3.3: DFD de nivel 1 para el sistema de control de acceso

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 3.3, este almacen se ha denominado **Información de Accesos**.

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 ara guardar los datos de la tarjeta que intenta acceder, y si finalmente se consigue o no. En la figura 3.3, este almacén se ha denominado **Información de Accesos.**

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 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 O Diagrama de contexto
Nivel 1 DFD O
Nivel 2 DFD 1 hasta DFD n
de los procesos 1 hasta n de 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 1.1.1 hasta DFD 1.1.x
....
....

...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 3.3, 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 3.3 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.

3.5.3. Diagramas de transición de estados

Los sistemas software, como muchos otros que se modelan, no suelen ser estáticos. Aunque tengan partes integrantes que lo sean, una componente fundamental de los mismos será los cambios que se produce durante su funcionamiento. Suelen ser, por tanto, sistemas dinámicos. Cuando un sistema permanece inalterado durante un período de tiempo se dice que se encuentra en un estado. Permanece estático. Todos los elementos que pueden cambiar en el sistema, elementos variables, no cambiarán. Una manera de caracterizar un sistema dinámico es mediante la descripción de los diferentes estados por los que puede pasar y la transición entre los mismos.

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.

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.

La notación básica para elaborar los diagramas de estados que se emplea habitualmente se muestra en la figura 3.4 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.

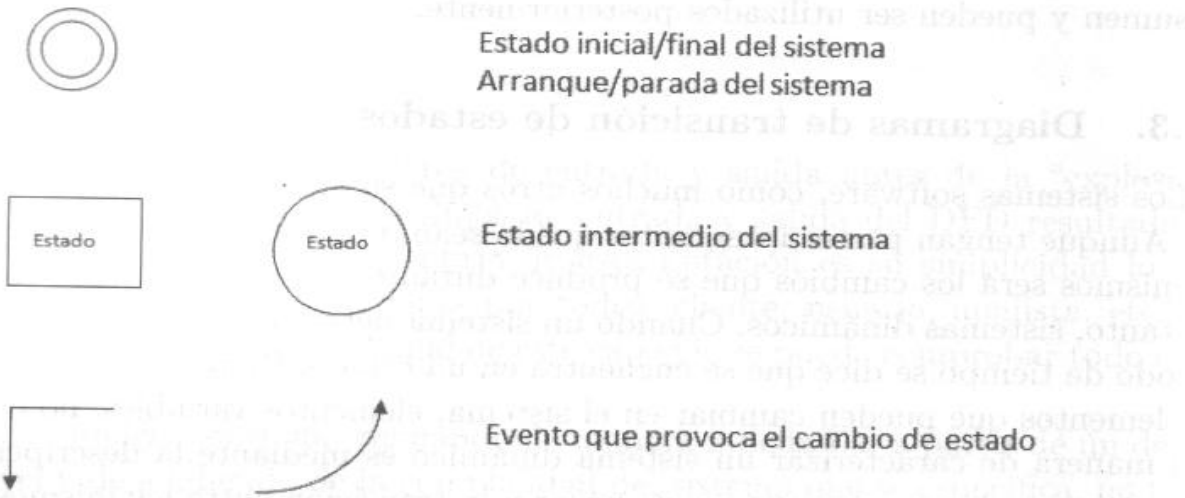


Figura 3.4: Notación básica de diagramas de estado

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 3.5 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.

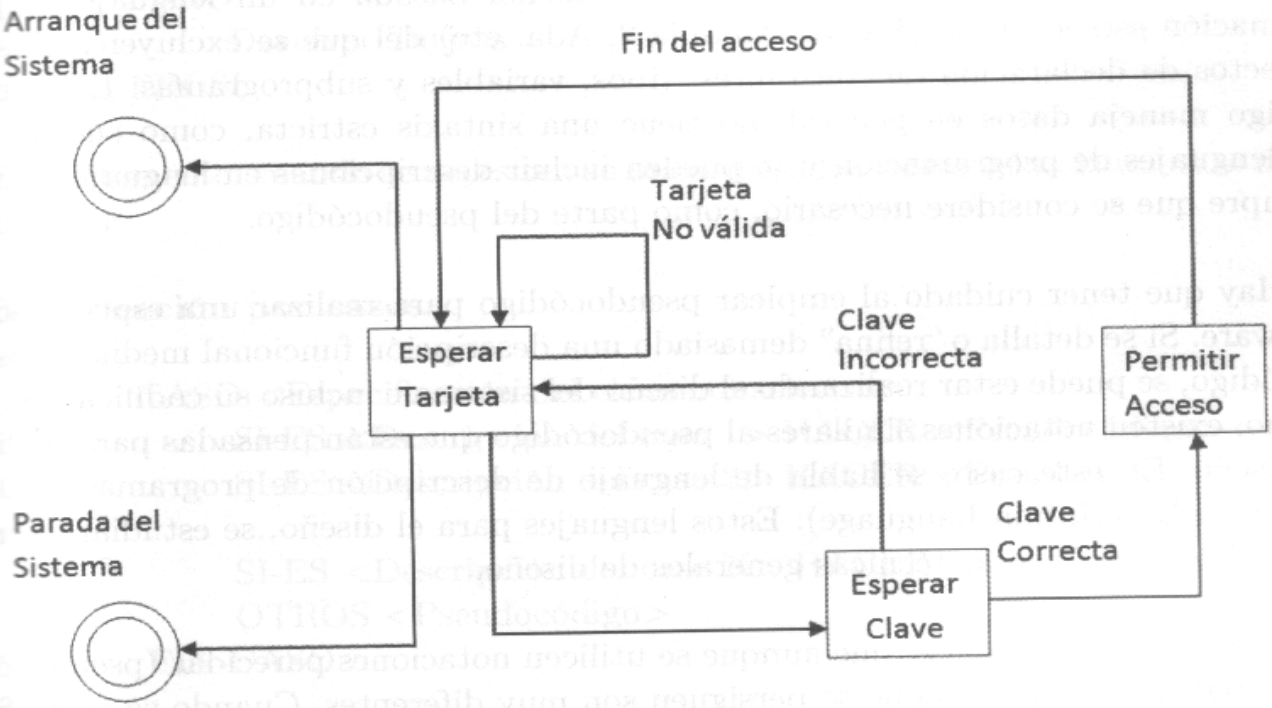


Figura 3.5: Diagrama de estado del sistema de acceso

En los ejemplos del final del capítulo se muestra otro diagram de estado empleando una notación alternativa.

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.

3.5.4. 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 capítulo 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 de los previstos. Por ejemplo, para seleccionar en un cajero automático el, tipo de operaciones que se pueden realizar dependiendo del tipo de tarjetal 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 indicad 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 efectú el <Pseudocódigo>. Por ejemplo, si se quieren listar todos los accesos realizados e el sistema de control de acceso, se podría indicar:

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

3.5.5. Descripción de datos

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

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 O al 9 /

3.5.6. 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 esta 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.

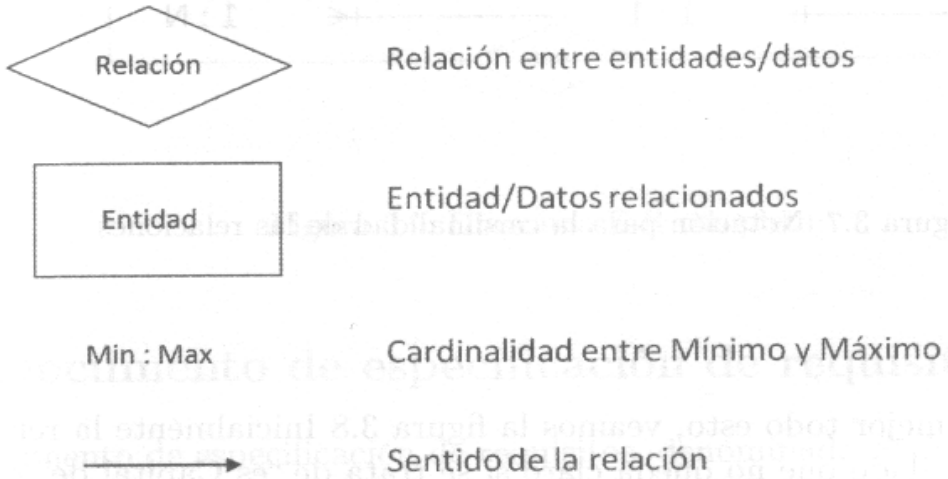


Figura 3.6: Notación básica de diagramas de datos

Existen diversas propuestas de notación para realizar los diagramas E-R, aunque entre todas ellas las diferencias son mínimas. En la figura 3.6 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 3.7 , 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.

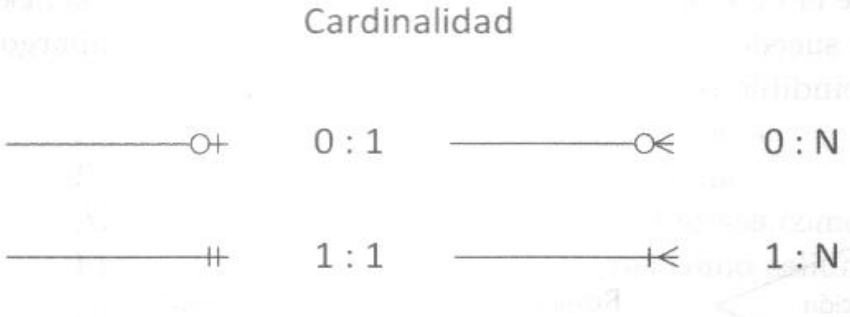


Figura 3.7: Notación para la cardinalidad de las relaciones

Para explicar mejor todo esto, veamos la figura 3.8 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.

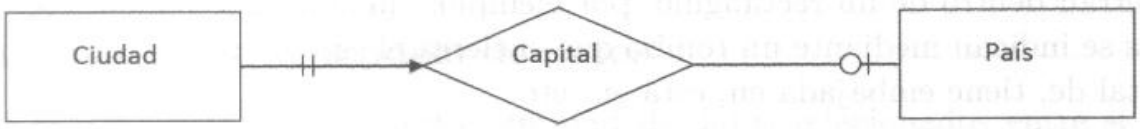


Figura 3.8: Diagrama E-R sencillo

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 3.9 se tiene el diagrama E-R del sistema completo, cuya interpretación resulta trivial.

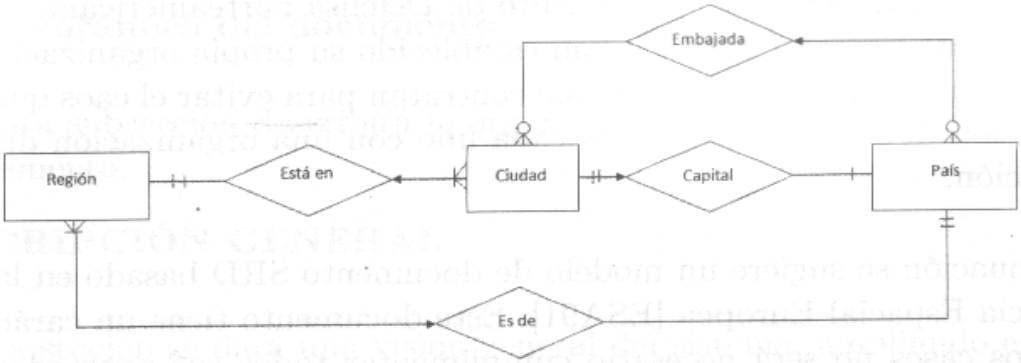


Figura 3.9: Diagrama E-R completo

3.6. 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 la sección 3.4. 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 ta-

les 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 documentó propuesto se recoge en el cuadro 3.1. 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.

   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.

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".

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".

BIBLIOTECA UNED-AVILA

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 (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.

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.

4. 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.

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 3.1: Índice de Documento SRD

3.7. Ejemplos de especificaciones

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

3.7.1. 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

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 Relaciones 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:	C++
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 realzará de forma aleatoria.

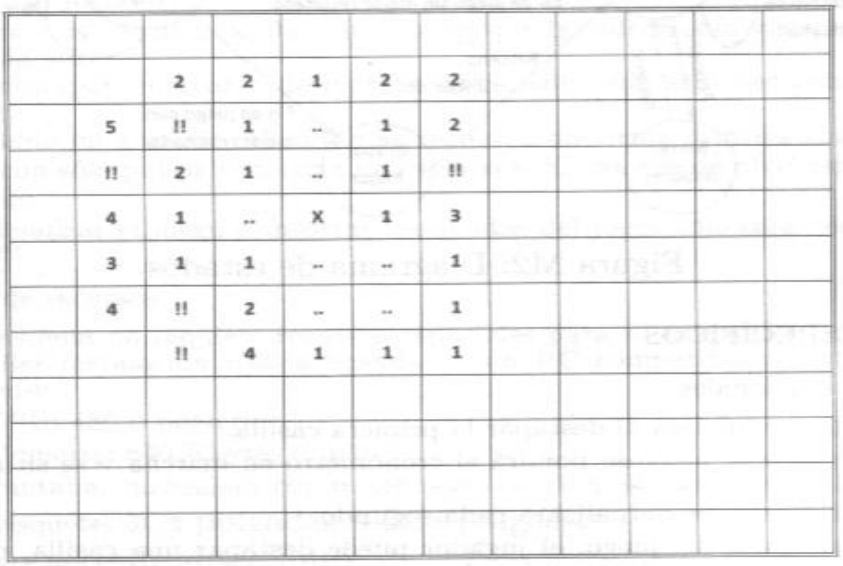


Figura M1: Tablero del juego de las minas

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 M1 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 desee 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 + n° 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 de 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 M2: Diagrama de estados

3. REQUISITOS ESPECÍFICOS

3.1 Requisitos funcionales

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.1.1.2 El cronómetro se actualizará cada segundo.

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 $\leq$ 0.1 segundo

R.2.3 Tiempo para situar inicialmente las minas $\leq$ 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 $\rightarrow$ medio $\rightarrow$ alto $\rightarrow$ 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 x 25 caracteres
- Disquete: 3 1/2 pulgadas o 5 1/4 pulgadas
- No necesita disco duro

3.6 Requisitos de verificación

R.6.1 (Requisito deseable) Se dispondrá de un modo auxiliar para depuración en el que el tablero sera "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 documentation

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.

3.7.2. 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.

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

2. DESCRIPCIÓN GENERAL

2.1 Relación con otros proyectos
Ninguna.

2.2 Relación con proyectos anteriores y posteriores
Ninguna.

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

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

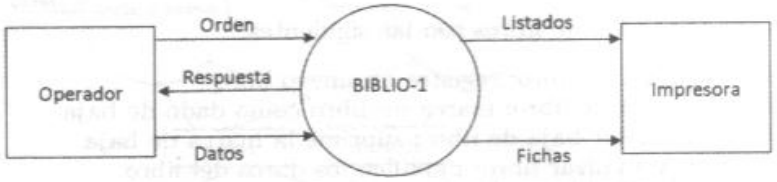


Figura B1: DC. Diagrama de contexto

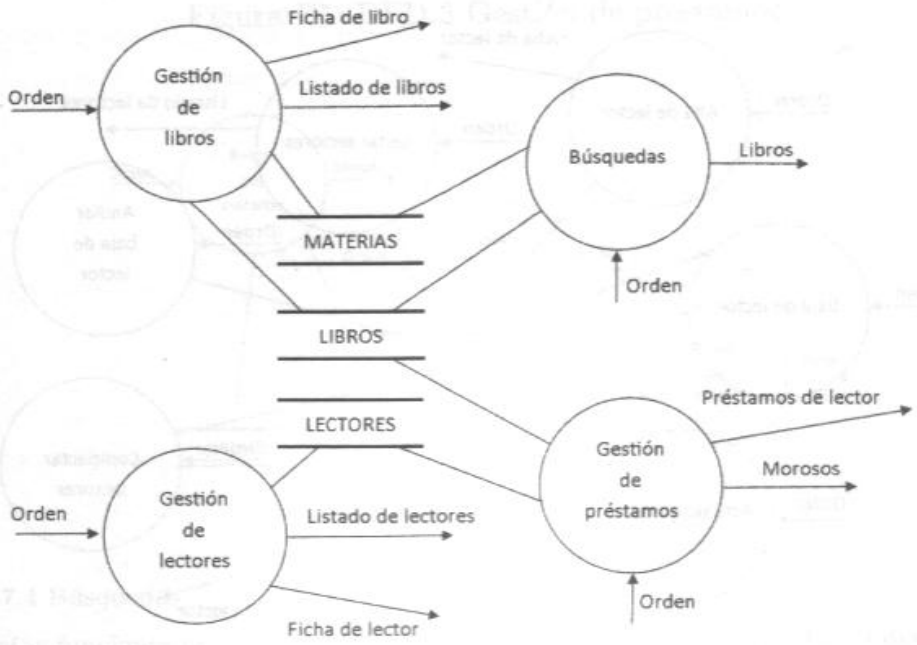


Figura B2: DFD.0 Gestión de la biblioteca

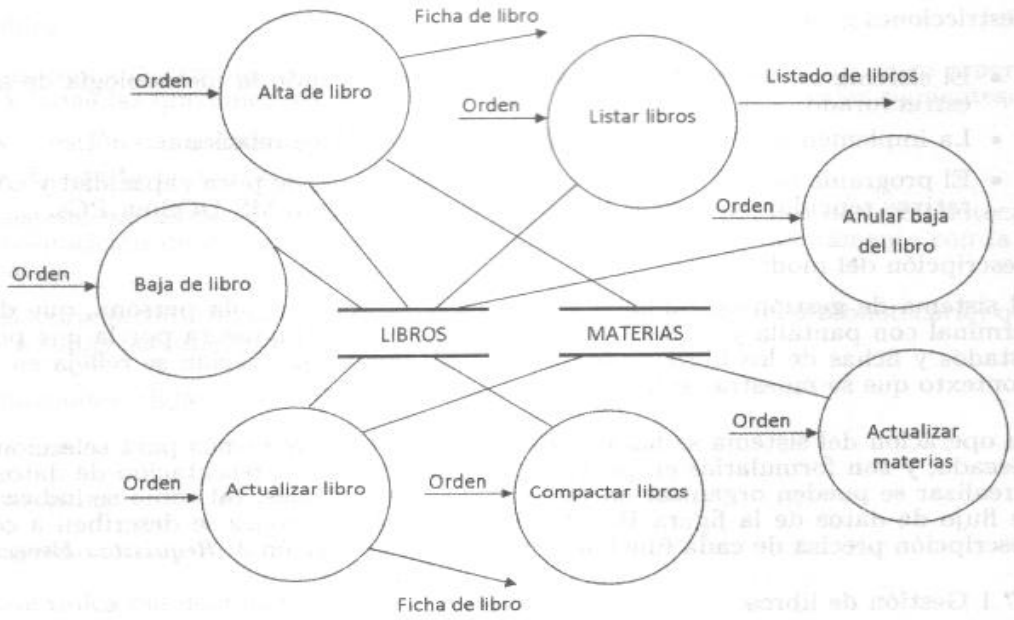


Figura B3: DFD.1 Gestión de libros

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

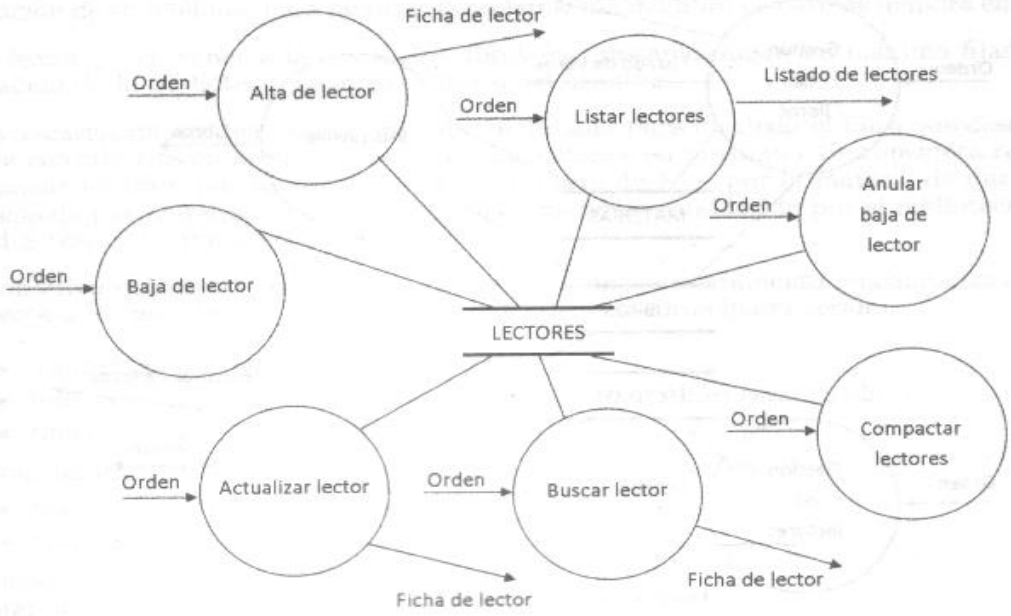


Figura B4: DFD.2 Gestión de lectores

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

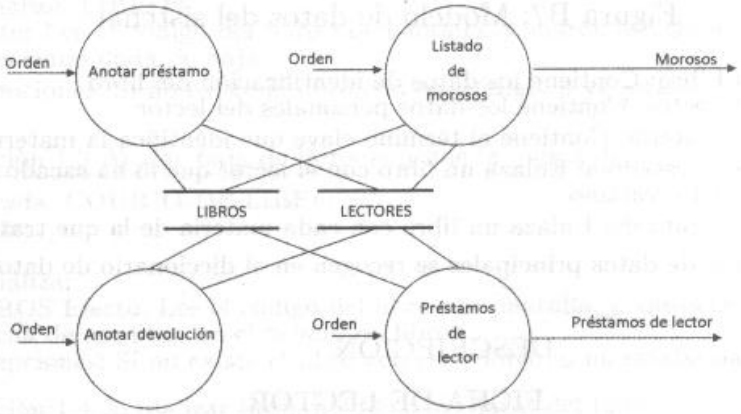


Figura B5: DFD.3 Gestión de préstamos

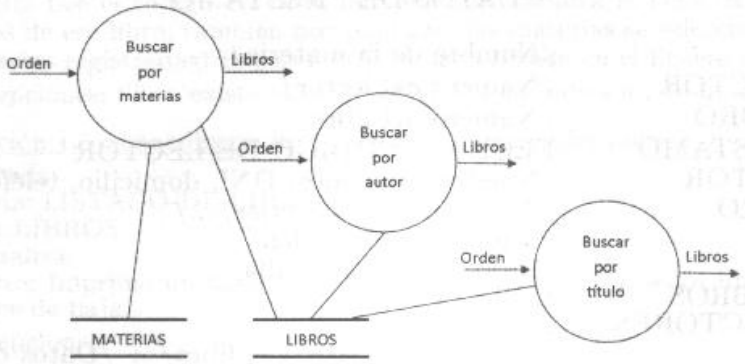


Figura B6: DFD.4 Búsquedas

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:

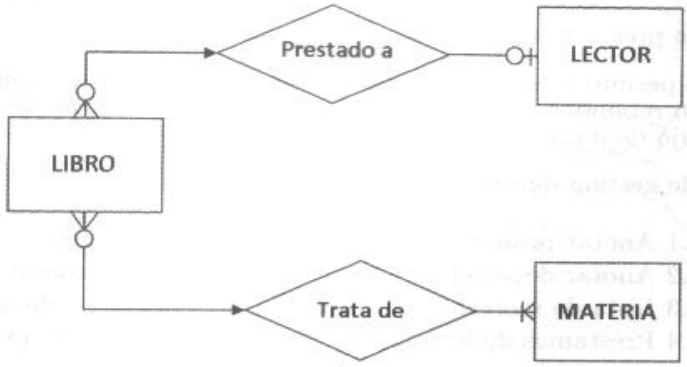


Figura B7: Modelo de datos del sistema

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.8

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 B8: Diccionario de datos del sistema

3. REQUISITOS ESPECÍFICOS

3.1 Requisitos funcionales

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

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:

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.
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.
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.

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: 31/2"720Kb o 51/4"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.

3.8. Conclusiones

A lo largo este capítulo se ha descrito la labor de análisis y definición de los requisitos que ha de cumplir un proyecto de software. Esta labor no es exclusiva de la producción del software. Cualquier actividad productiva que tenga como objetivo la creación de un sistema complejo debe pasar por esta fase.

El cliente, el mercado y la propia empresa requieren un sistema nuevo que debe ser descrito de manera precisa y específica. Eso es la especificación del sistema. La relación detallada de todas las características del nuevo sistema debe hacerse de manera rigurosa, no sólo teniendo en cuenta lo que ha encargado el cliente, sino también todo lo necesario para que lo encargado funcione correcta y legalmente. 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 de lo especificado también se realiza dentro de esta 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.

Una parte fundamental de este capítulo es la dedicada a las diferentes notaciones empleadas para redactar de manera precisa las particularidades del sistema a desarrollar.

Para concluir se ha elaborado, a modo de ejemplo, un documento de requisitos de un sistema software.

3.9. Ejercicios propuestos

1. Elabórese una lista de 5 requisitos funcionales y 5 no funcionales de tres posibles sistemas de software.
2. Realícese una descripción en lenguaje natural, diagrama de flujo de datos y diagrama de estados de los siguientes sistemas:
   - Una puerta manual
   - Una puerta automática con detector de presencia
   - Un ascensor con dos pisos
3. Elabórese un plan para la captura, validación y verificación de los requisitos de un proyecto.
4. Búsquese una herramienta de gestión de requisitos. Analícese sus funcionalidades, ventajas, inconvenientes, incluso su precio si fuera posible.
5. Elabórese el SRD de la práctica de fundamentos de programación que realizó el curso pasado. Detállense los diagramas necesarios para su completa especificación. ¿Con qué argumentos se podría convencer al cliente de que se ha cumplido con lo que se encargó?

Capítulo 4

Fundamentos del Diseño de Software

4.1. Introducción

Con este capítulo 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 capítulo se abordan los fundamentos de la etapa de diseño.

4.2. Objetivos

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 exclusivas 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.

4.3. ¿Qué es el diseño?

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. Cualquier producto que se crea en la ingeniería se describe de manera precisa para su producción mediante la redacción de los planos de su diseño. Estos planos del diseño permiten la comunicación del mismo a todas las partes implicadas en la producción del bien.

La elaboración del diseño permite, además, una clasificación del producto de acuerdo a las características que se plasmen en el documento resultante. Una tendencia bastante utilizada, y que se estudiará en profundidad en futuras asignaturas, es la elaboración del diseño de una aplicación de acuerdo con unas pautas previamente establecidas. Es lo que se conoce como *diseño con patrones 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 capítulo 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 alguna 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 interfases 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 la sección 3.5.4 del capítulo 3). 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 capítulo.

4.4. 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 capítulo 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 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.

4.4.1. Abstracción

Este concepto siempre resulta muy "abstracto" para los estudiantes. Con objeto de definirlo de manera precisa se irá al diccionario de la Real Academia de la Lengua:

1. adj. Que significa alguna cualidad con exclusión del sujeto.
2. adj. Dicho del arte o de un artista: Que no pretende representar seres o cosas concretos y atiende solo a elementos de forma, color, estructura, proporción, etc.

Si se consulta por la etimología [ETIMCHIL] de la palabra quizás se encuentre más luz.

La palabra **abstracto** viene del latín *abstractus*, compuesto del prefijo *abs* (separación, privación) y *tractus*. Éste es el participio del verbo *trahere*, que da palabras como trecho y tractor. Etimológicamente sería algo como "sin trecho", pero se aplica para referirse a cualidades o elementos que no son específicos. Por ejemplo: belleza, libertad, justicia, etc. El antónimo de abstracto es concreto.

Abstracto quiere decir etimológicamente "arrastrado o sacado a partir de sus límites externos". Abs-/ab- significa alejamiento a partir del límite exterior de algo. Es por eso que una abstracción o algo abstracto es cualquier cosa que partiendo de la apariencia externa de una realidad, se aleja de ella.

Para ilustrar el concepto se presenta un ejemplo "concreto". Supóngase un carta o naipe de una baraja española, "El dos de oros". La carta en sí no es más que un pedazo de cartón con un dibujo y unos números en una de las dos caras del cartón, la otra no tiene dibujo. Esta descripción es el objeto concreto. Sin embargo si se conoce el concepto de baraja de cartas y las funciones que tiene, se puede saber que el cartón representa algo más que lo descrito. La carta forma parte de una colección de elementos que están clasificados y también ordenados, tienen un valor implícito dentro de una escala establecida. También forma parte de un conjunto limitado de valores. Es una carta de la baraja española. Este es el concepto abstracto que se saca de sus límites físicos.

El mismo caso se da con una tarjeta de crédito. Su presencia física es un rectángulo de plástico con una banda magnética y unos datos grabados en su superficie. Sin embargo, el conceptp que abarca es un medio de pago universalmente aceptado, con unos límites máximos, una fecha de caducidad, etc.

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 se referiren a través de las palabras. Cada nueva palabra sirve para referirse a un nuevo concepto abstracto aceptado por todos sin necesidad de aclarar cada vez su significado.

Una **abstracción**, en el diseño de un sistema software, será cualquier elemento que forme parte del sistema que se quiere modelar, con la suficiente entidad o importancia para distinguirlo del resto y que esté dotado de funciones relevantes para el sistema. Cuando se identifiquen esos elementos "importantes" deberá hacerse teniendo en cuenta no sólo su forma "concreta" sino también todo lo que lo rodea y que permite distinguirlo como un elemento importante.

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.

En el ejemplo del control de acceso planteado en el capítulo anterior, la tarjeta de acceso sería una abstracción. Tiene una clave, se debe leer en sistema, se puede cambiar su clave, etc. Cuantas más características contemplemos mayor grado de abstracción conseguiremos. Otra abstracción podría ser la pantalla, y otra el teclado.

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 capítulo anterior, en un primer nivel aparecen abstracciones tales como: Tarjeta, Mensajes, Órdenes, 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.

Una buena fuente de abstracciones aplicables al diseño proviene del análisis del dominio del sistema. Como ya se comentó en el capítulo anterior.

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 de control de acceso utilizado como ejemplo se puede ver como una máquina abstracta si se consigue formalizar todo su comportamiento. En todos estos casos el diseño consiste en definir formalmente la máquina abstracta que se necesita.

4.4.2. 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 interfases entre todos ellos queden completamente definidas y correctamente diseñadas. En el próximo capítulo 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 interfases 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.

Cualquier actividad productiva que se desarrolle de manera discontinua en el tiempo por un grupo de personas debe ser descompuesta y desarrollada de manera modular.

Para los estudiantes de primer curso de Ingeniería informática resulta una tarea adicional y no siempre cómoda la descomposición modular de cualquier aplicación desarrollada. Es en esta asignatura donde los alumnos deben adquirir el hábito y la metodología para realizar la descomposición modular del trabajo a realizar para así poder beneficiarse de las ventajas expuestas de la misma.

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

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

4.5.1. 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.

4.5.2. 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 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.

4.5.3. 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.

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, sí 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 PonerenCola( X: in COLA);
```

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

```
procedure PonerEntero is new PonerenCola( INTEGER );
procedure PonerReal is new PonerenCola( REAL);
```

```
procedure PonerCaracter is new PonerenCola( CHARACTER );
procedure PonerOtroTipo is new PonerenCola( OtroTipo );
```

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

En el Modelo Orientado a Objetos se maneja el concepto de clase. Las clases se construyen en base a tipos de datos (enteros, reales, cadenas de caracteres, sectores, pilas, etc.) Si se sustituyen algunos de los tipos de datos mencionados por parámetros formales que los representan, y que pueden materializarse en cualquiera de estos tipos de datos, podemos considerar que estos parámetros formales son tipos genéricos, es decir, tipos de datos no concretos.

Las clases de objetos genéricos se pueden denominar clases genéricas, ya que representan a todas las clases que se pueden construir cuando se sustituye los tipos genéricos por tipos de datos existentes. Cuando hablamos de genericidad en el lenguaje C++, estamos hablando de clases y funciones genéricas, es decir, de clases y funciones que están definidas en base a tipos de datos genéricos, parámetros formales que son tipos de datos que se traducen en tiempo de compilación. En el caso del lenguaje C++ se habla de *templates* o *plantillas*.

El uso de templates permite definir funciones y clases genéricas mediante la utilización de datos genéricos, que se escriben como parámetros formales. El compilador sustituye el parámetro formal por el tipo real y genera el código necesario cuando algún fragmento de código hace uso de una instanciación de un template.

4.5.4. 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 combi-

nada 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++
- JAVA

Todos ellos disponen del mecanismo de herencia. En [Budd94] y [NAUGHTON97] se detallan las particularidades del mecanismo de herencia propuesto por cada uno de los lenguajes citados.

4.5.5. 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 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. 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 ristras 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 rellenar 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. 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.

4.5.6. 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:

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 Ti se encarga de obtener un resultado que debe utilizar otra T2, ambas se deben sincronizar. La tarea T2 esperará hasta que Ti tenga disponible un nuevo resultado y en el caso de que T2 no haya utilizado el anterior resultado es Ti 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 [BenAri90] y en gran número de casos la solución finalmente adoptada resulta ser completamente a la medida.

4.6. 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 comunmente. 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 cuasi 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 capítulo 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:

- 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 capítulo anterior y en todo caso, se procurará utilizar solamente como complemento a otras notaciones.

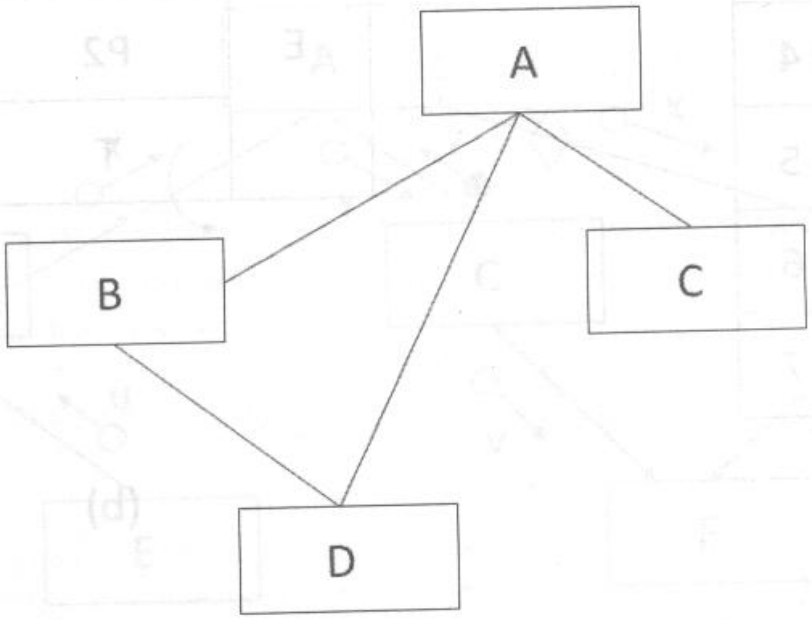


Figura 4.1: Diagrama de bloques

4.6.1. 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 4.1 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.

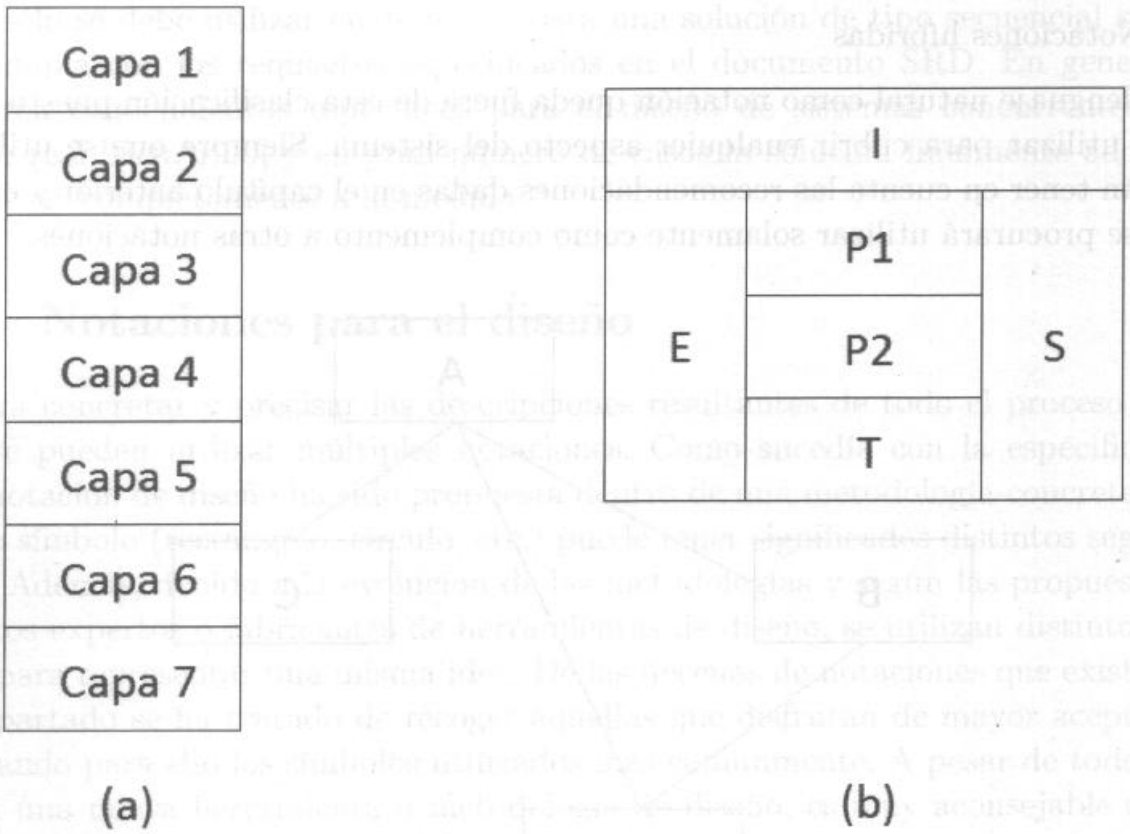


Figura 4.2: Diagramas de cajas adosadas

Otra notación informal para estructurar un sistema se muestra en la figura 4.2. 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 4.2 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.

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 4.2 (b). Los subsistemas generales propuestos en la plantilla son: E (Entrada), S (Salida), T (Test para la autocomprobación y el mantenimiento), (Interfase de usuario) y P 1 , P2, ... (Procesado y control del sistema). A continuación se presentan algunas notaciones más formales para describir la estructura de un sistema.

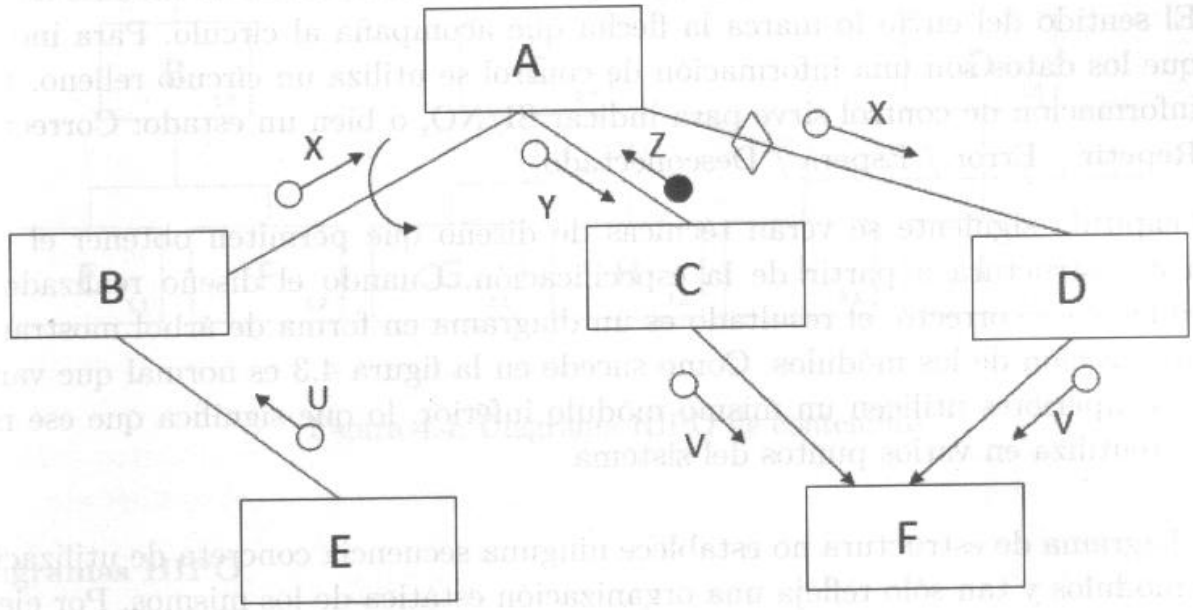


Figura 4.3: Diagrama de estructura

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 4.3 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 capítulo 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 4.3 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 4.3 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 4.3 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.
- 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 P1 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.

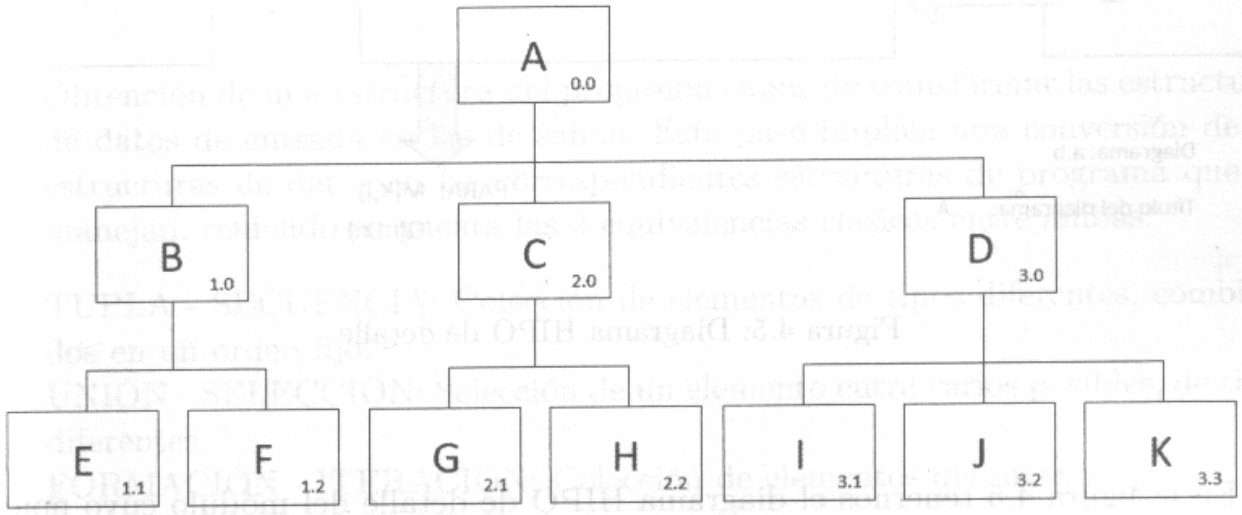


Figura 4.4: Diagrama HIPO de contenidos

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).

La figura 4.4 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, ...).

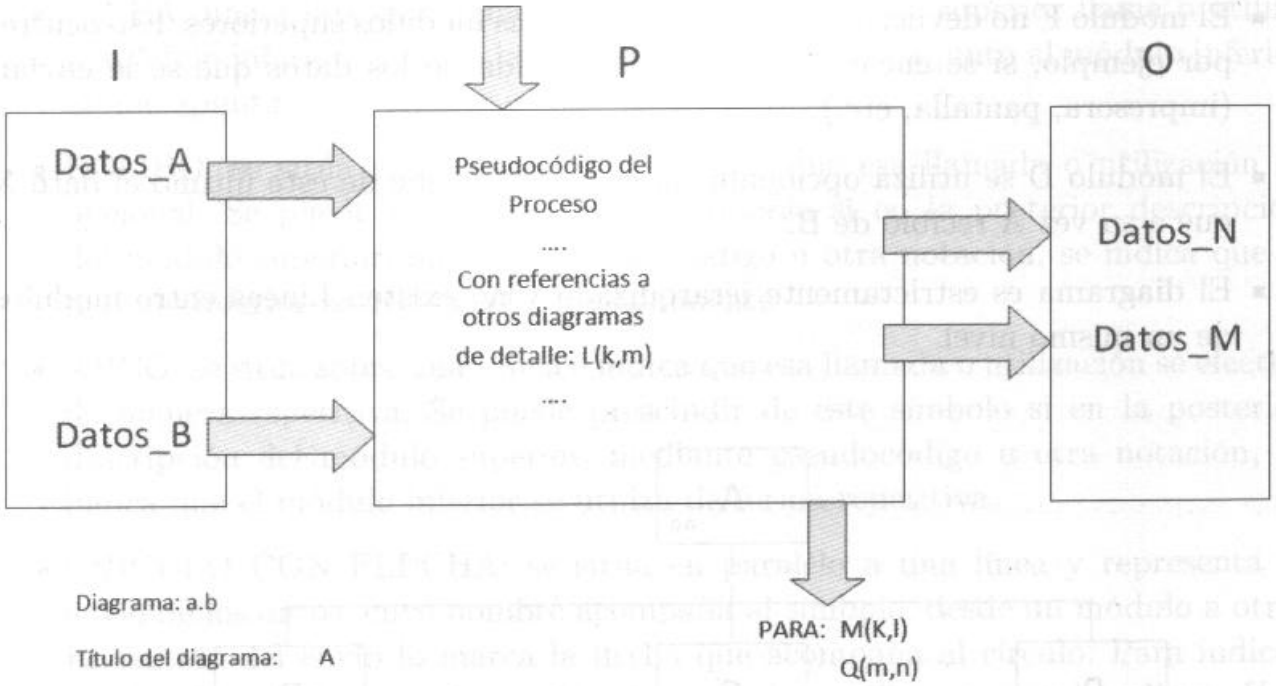


Figura 4.5: Diagrama HIPO de detalle

En la figura 4.5 tenernos 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 (0). 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 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).

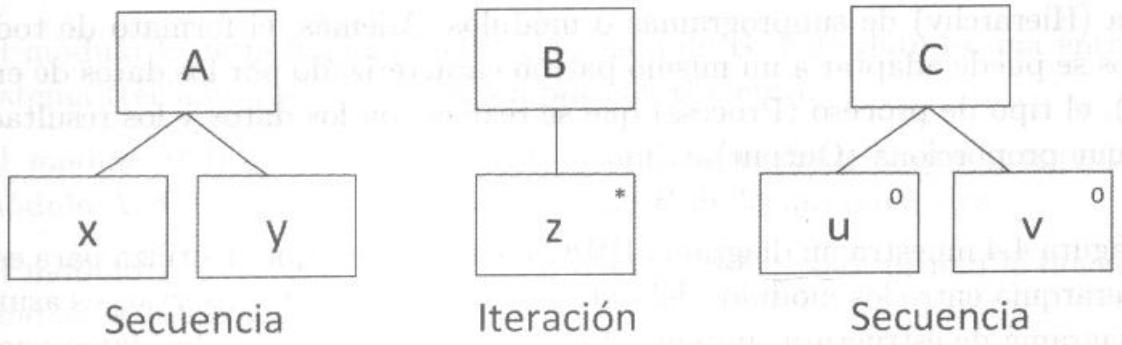


Figura 4.6: Diagrama de Jackson

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:

1. Especificación de las estructuras de los datos de entrada y salida.

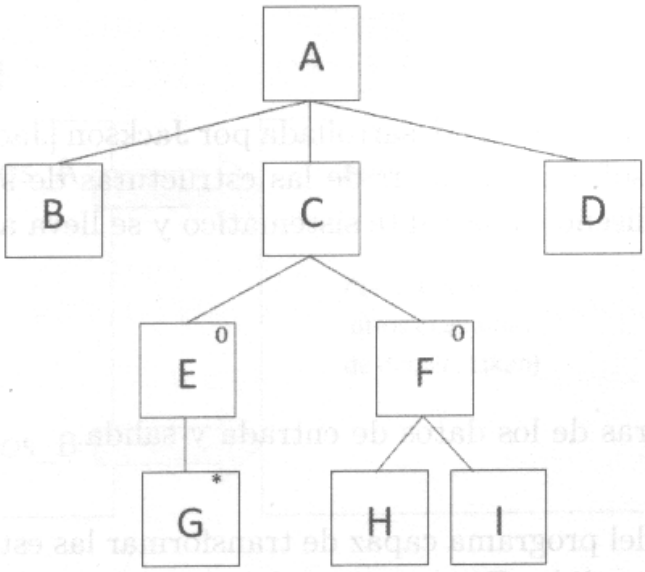
2. Obtención de una estructura del programa capaz de transformar las estructuras de datos de entrada en las de salida. Este paso implica una conversión de las estructuras de datos en las correspondientes estructuras de programa que las manejan, teniendo en cuenta las 2 equivalencias clásicas entre ambas:

   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 4.6 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 4.6, 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").




```
B;
IF Condicion THEN
  FOR i:=1 TO N DO
    G
  END
ELSE
  H;I
END
D;
```

Figura 4.7: Diagrama de Jackson

En la figura 4.7 se muestra un diagrama de Jackson y la estructura de programa equivalente. Si los elementos del diagrama representan datos, la estructura equivalente, expresada en notación de diccionario de datos, es la siguiente:

A= B+C+D
C= [E | F]
E = G
F = H + I

4.6.2. 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.

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 capítulo 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 capítulo se estudiarán las técnicas para realizar esta labor.

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 capítulo 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.

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.

Diagramas de flujo de datos

Las características de esta notación están detalladas en el capítulo 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.

Digramas de transición de estados

Esta notación está descrita en el capítulo 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.

Lenguaje de Descripción de Programas (PDL)

La notación PDL ha sido presentada en el apartado dedicado al pseudocódigo del capítulo 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 apropiados es Ada.

La notación denominada Ada-PDL permite declarar estructuras de datos e incluso existen compiladores para verificar el 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.

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 aglutinan los aspectos estáticos (datos), dinámicos (operaciones) y de estructura del sistema.

Diagramas de abstracciones

Como ya hemos visto en este mismo capítulo, 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.

Según se muestra en la figura 4.8 (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 estaríaformado por las definiciones de funciones y/o procedimientos declaradas en el módulo de definición.

Figura 4.8: Estructura de una abstracción (a) y de un Objeto (b)

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.

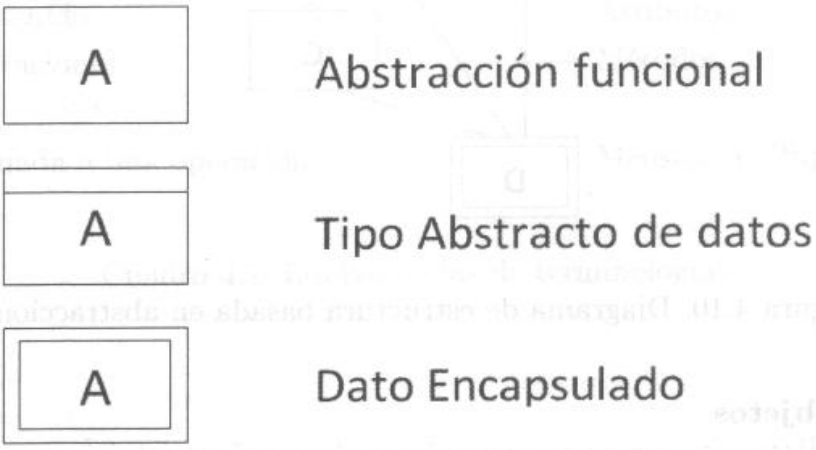


Figura 4.9: Abstracciones: notación básica

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;

```
LeerTarjeta;
...
ComprobarTarjeta;
```

En la figura 4.9 se muestra la notación gráfica para indicar la forma de abstracción que se está empleando. En la figura 4.10 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.

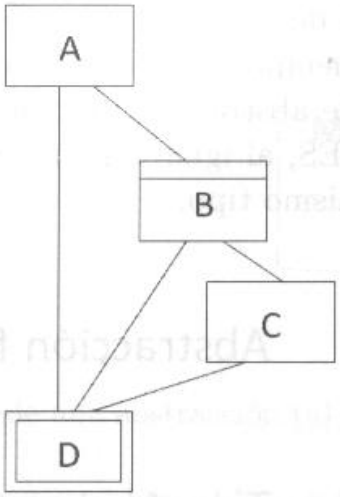


Figura 4.10: Diagrama de estructura basada en abstracciones

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 4.8 (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 4.1

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 capítulo utilizaremos la terminología de objetos, debido a la mayor aceptación

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 4.1: 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".

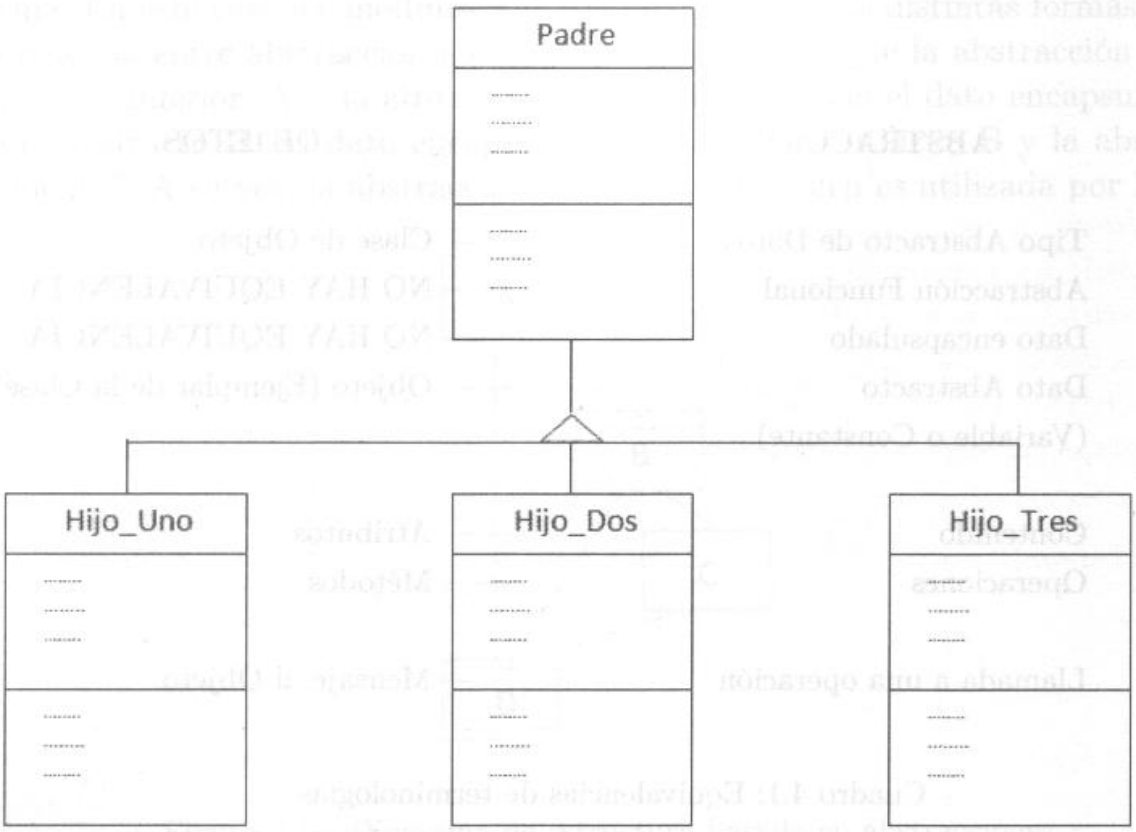


Figura 4.11: Clasificación, especialización o herencia entre objetos

En la figura 4.11 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 4.8 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.

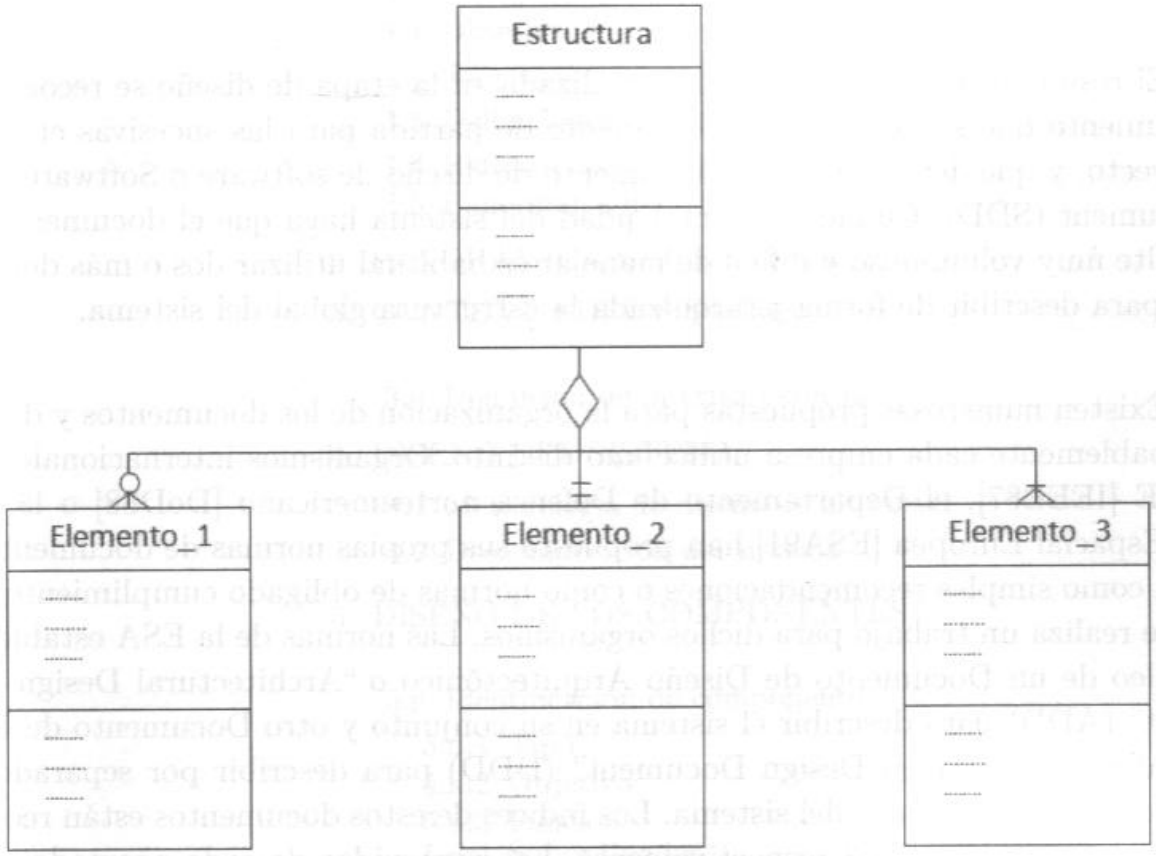


Figura 4.12: Composición de objetos

B.- COMPOSICIÓN (También válida entre abstracciones)

La relación de composición permite describir un objeto mediante los elementos que lo forman. En la figura 4.12 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 4.12, 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.

4.7. 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 dificil 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 [IEEE87], 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 4.2 y 4.3 respectivamente. Los contenidos de cada apartado son los siguientes:

4.7.1. 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 capítulo 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
   - 1.5 Panorámica del documento
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 Identificación de 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 DE RECURSOS ESTIMADOS
7. MATRIZ DE REQUISITOS COMPONENTES

Cuadro 4.2: Índice de Documento ADD

3. CONTEXTO DEL SISTEMA

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

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.

4.7.2. 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 4.3, 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 el 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 del documento
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 4.3: 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 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.

4.8. Conclusiones

En este capítulo se ha presentado en qué consiste el diseño de un sistema software y los conceptos necesarios para realizar un diseño, como son:

- Abstracción
- Modularidad
- Refinamiento
- Estructuras de datos
- Ocultación
- Genericidad
- Herencia
- Polimorfismo
- Concurrencia

También se presentan notaciones para realizar el diseño de los sistemas software. Tanto desde un punto de vista estático, qué elementos tiene el sistema, como desde un punto de vista dinámico, qué evolución tienen esos elementos cuando el sistema funciona.

En un principio sirven también para mostrar a los lectores qué tipo de cosas se quieren representar en el diseño y qué diferentes alternativas de representación hay. Se presentan diferentes notaciones.

En el capítulo 6 de este libro se presentará la notación universalmente aceptada para el diseño de un sistema software: el lenguaje UML.

4.9. Ejercicios propuestos

1. Póngase 5 ejemplos de abstracciones, de modularidad, de herencia, de ocultación y de polimorfismo empleadas en la vida diaria.
2. Modélese mediante abstracciones un autobus urbano.
3. Modélese mediante diagramas de Jackson la función factorial.
4. Modélese mediante objetos el autobus del ejercicio 2. Empléese el concepto de herencia.
5. Explíquese brevemente como aplicar la técnica de refinamientos sucesivos en la descripción de una receta de cocina.

Esta colección de ejercicios propuestos tiene como objeto que el lector se olvide de la programación para obtener los resultados pedidos y se concentre en el desarrollo del diseño de los sistemas.

Capítulo 5

Técnicas Generales de Diseño de Software

5.1. Introducción

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.

5.2. Objetivos

En este capítulo 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 metodologias 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 capítulo 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
- 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.

5.3. 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 como las siguientes para que se pueda considerar válida, a saber:

5.3.1. 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.

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 capítulo 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 redunda 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 en [Stevens74] dentro de la metodología de diseño estructurado, pero son igualmente validos para realizar la descomposición funcional de un sistema utilizando cualquier otro tipo de técnica o metodología.

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:

	↑	Acoplamiento por Contenido
FUERTE	|	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 5.1 se muestran gráficamente situaciones que dan lugar a la existencia de un acoplamiento fuerte entre módulos.

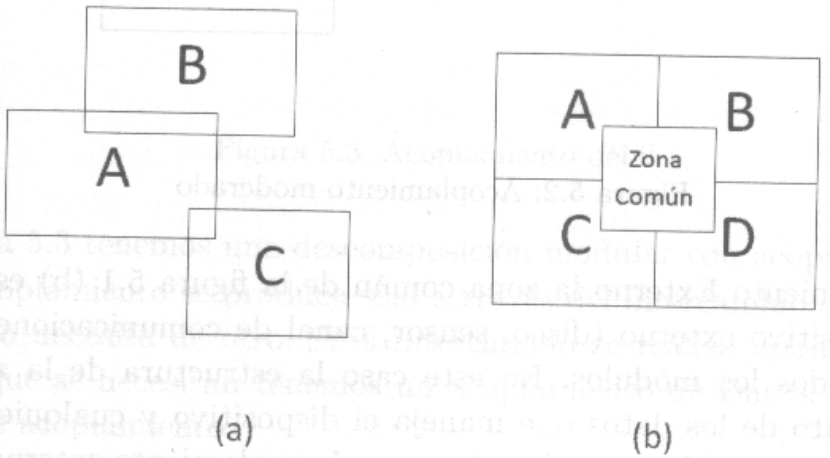


Figura 5.1: Acoplamiento fuerte: (a) por contenido; (b) común o externo

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 5.1 (a), en realidad no existe una separación real 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 5.1 (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 dificil y siempre que se pueda se debe evitar.

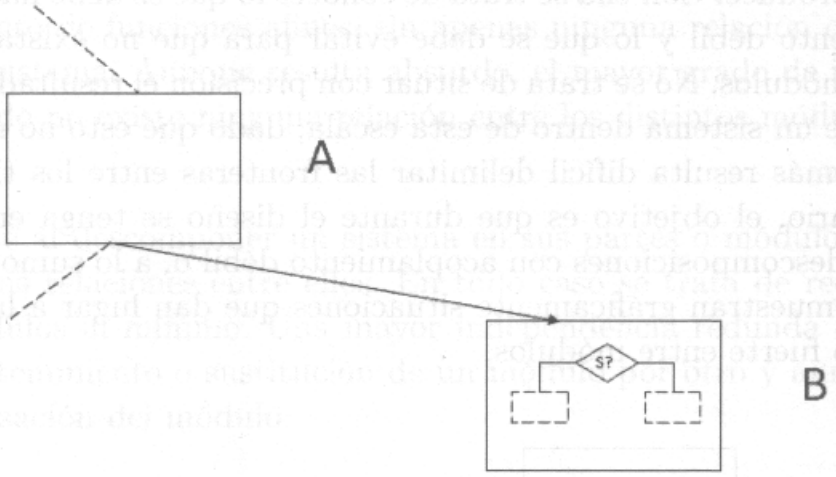


Figura 5.2: Acoplamiento moderado

En el acoplamiento Externo la zona común de la figura 5.1 (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 5.2 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.

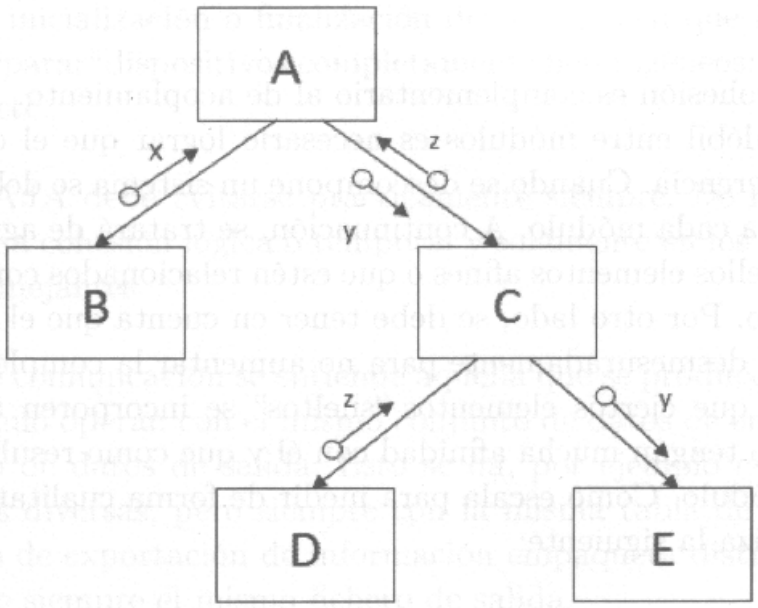


Figura 5.3: Acoplamiento débil

En la figura 5.3 tenemos una descomposición modular con acoplamiento débil. 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 5.3 entre los que no existe ningún tipo de acoplamiento directo.

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.

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 serial 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.

Comprensibilidad

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 a 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 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 prolija 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 dificil de comprender.

5.3.2. 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 mediatizada 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 tremendamente 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 que 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.

5.4. 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.

5.4.1. 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 capítulo 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
```

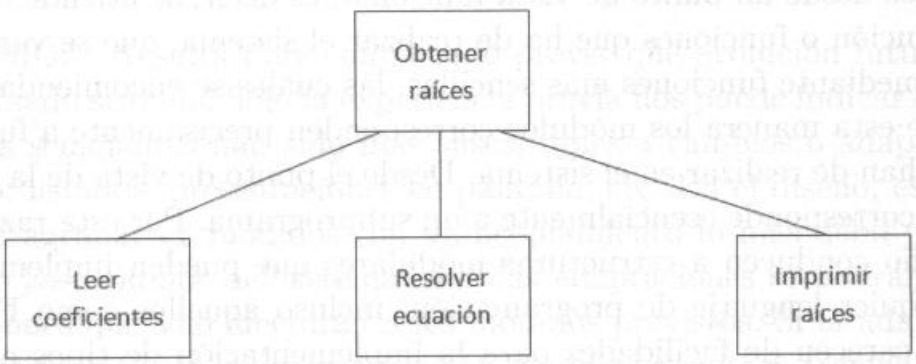


Figura 5.4: Diseño por refinamiento

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 figura 5.4. 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.

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 capítulo 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 1 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 1 palabra]

El diagrama equivalente aparece a la izquierda de la figura 5.5. En el caso del texto de salida, la escritura ha de hacerse al completar las líneas y justificadas, en caso de que no sean la última del párrafo. Además se usan líneas en blanco corno separación. La estructura de salida podría ser una iteración de líneas, cada una del tipo apropiado.

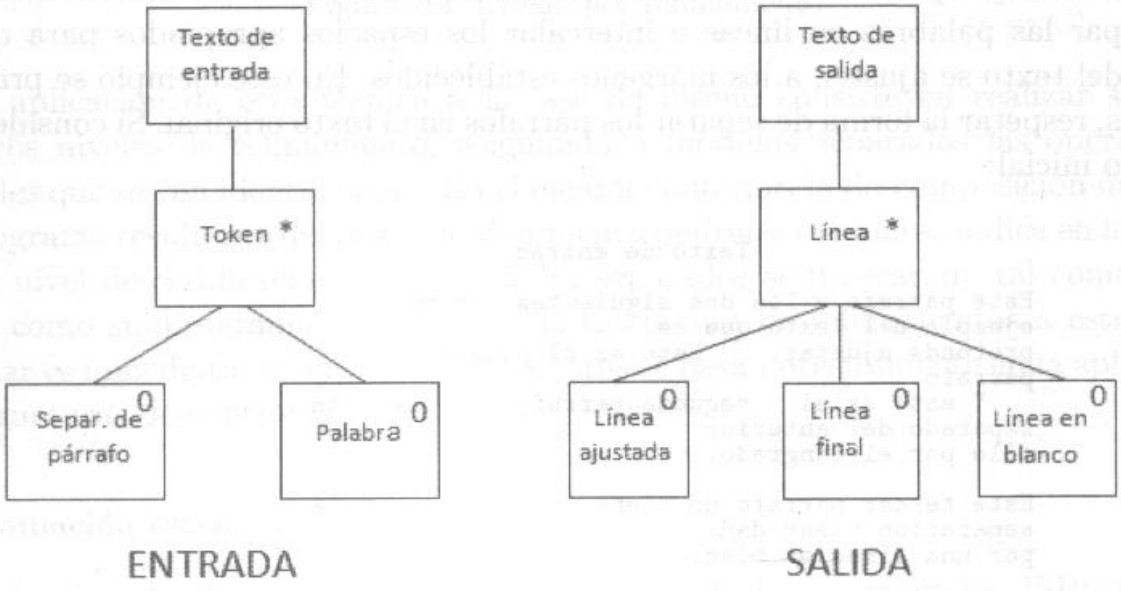


Figura 5.5: Ejemplo de estructuras de entrada y salida

texto de salida = [línea ajustada 1 línea final 1 línea en blanco]

El diagrama de estructura aparece a la derecha de la figura .

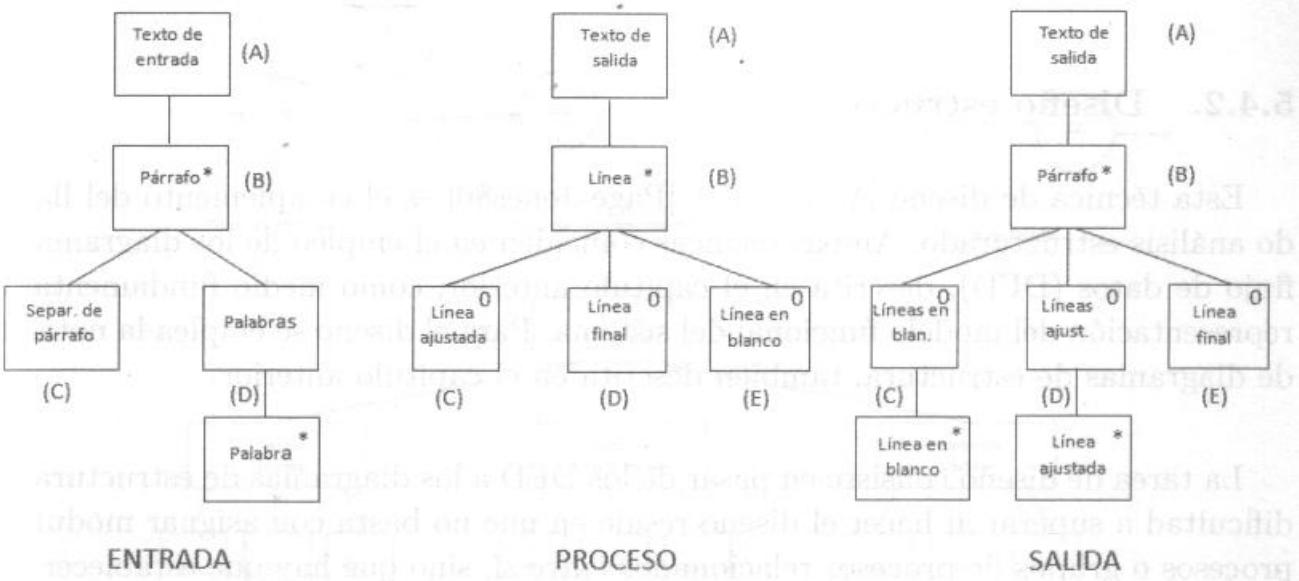


Figura 5.6: Ejemplo de diseño usando JSP

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 5.6. 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 5.6, 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.

5.4.2. 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 capítulo 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 capítulo 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 (Figura5.6 (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.

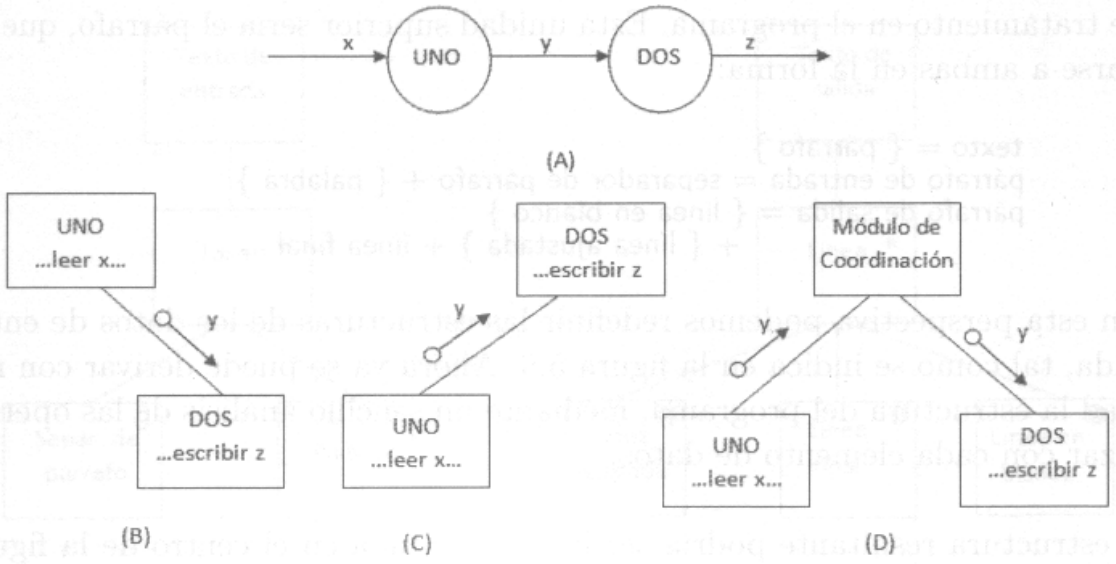


Figura 5.7: Ejemplo de Estructuras alternativas

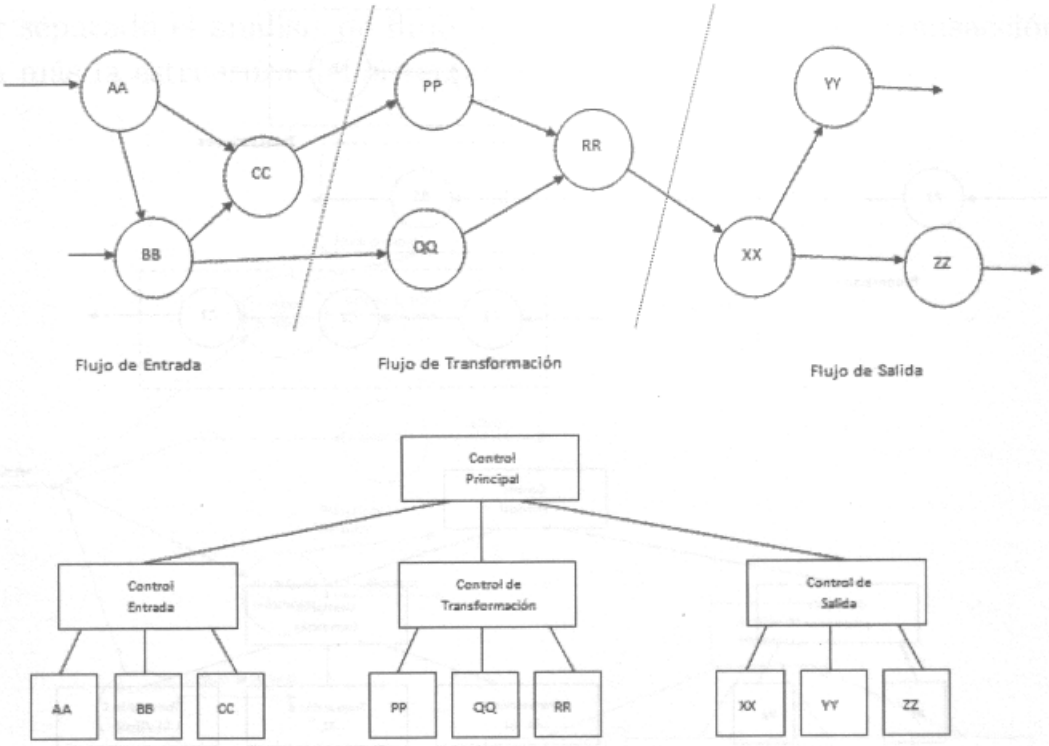


Figura 5.8: Diseño basado en el flujo de transformación

En la figura 5.7-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 5.7-C es el módulo de la segunda operación quien tiene el control sobre la primera. En el último caso, de la figura 5.7-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.

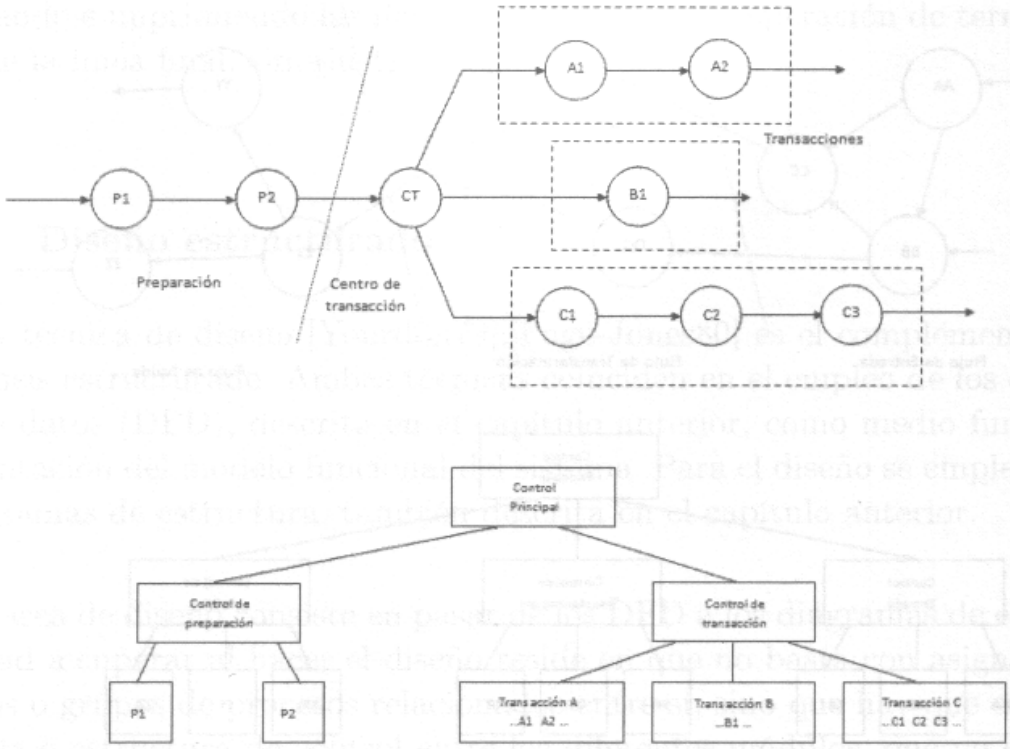


Figura 5.9: Diseño basado en el flujo de transacción

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 5.8. 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 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 5.9 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.

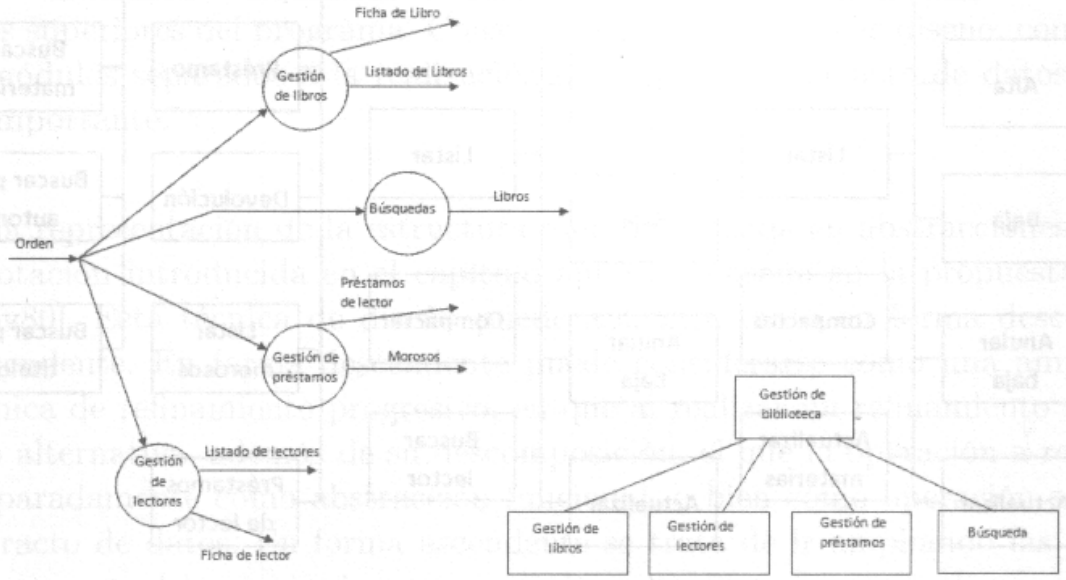


Figura 5.10: Gestión de biblioteca. Diseño inicial

5.4.3. Ejemplo: Sistema de gestión de biblioteca

Se describe aquí como ejemplo un posible diseño del caso práctico especificado en el capítulo 3 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 5.10. 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.

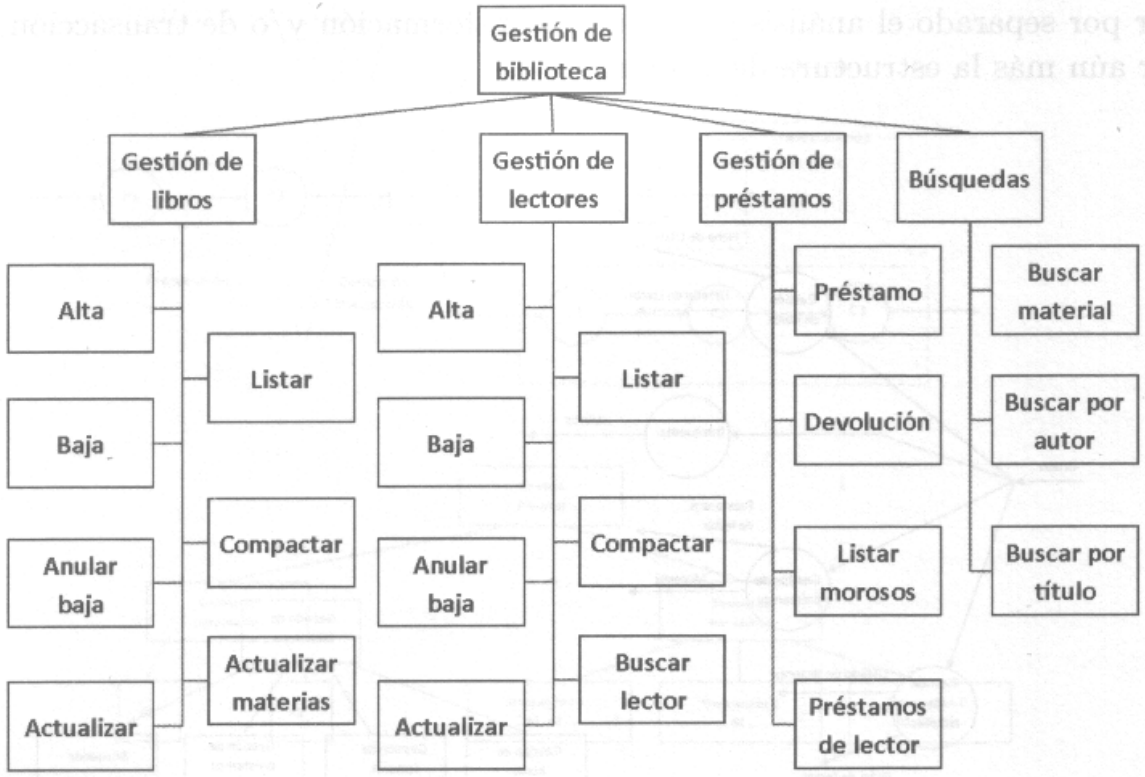


Figura 5.11: Diseño final del sistema de gestión de biblioteca

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 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 5.11, en que aparece un segundo nivel de descomposición del sistema.

5.5. 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 capítulo 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.

5.5.1. 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 capítulo 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 corno 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 2° grado, usado en la sección 5.4.1, podemos identificar los tipos abstractos correspondientes a un número complejo, y a una ecuación de 2° grado. Sabiendo que la fórmula que da las raíces es:

$$Ax^2 + Bx + C = 0 \Longrightarrow X = \frac{-B \pm \sqrt{(B2 - 4AC)}}{2A}$$

podemos definir sobre dichos tipos abstractos las siguientes operaciones necesarias para la aplicación:

Ecuación de 2° grado:	Número complejo:
Leer ecuación	Escribir
Escribir ecuación	Sumar, Restar, etc
Obtener raíces	Raíz cuadrada

Para obtener las raíces se usarán las operaciones definidas sobre números complejos. La estructura modular del programa podría la representada en la figura 5.12

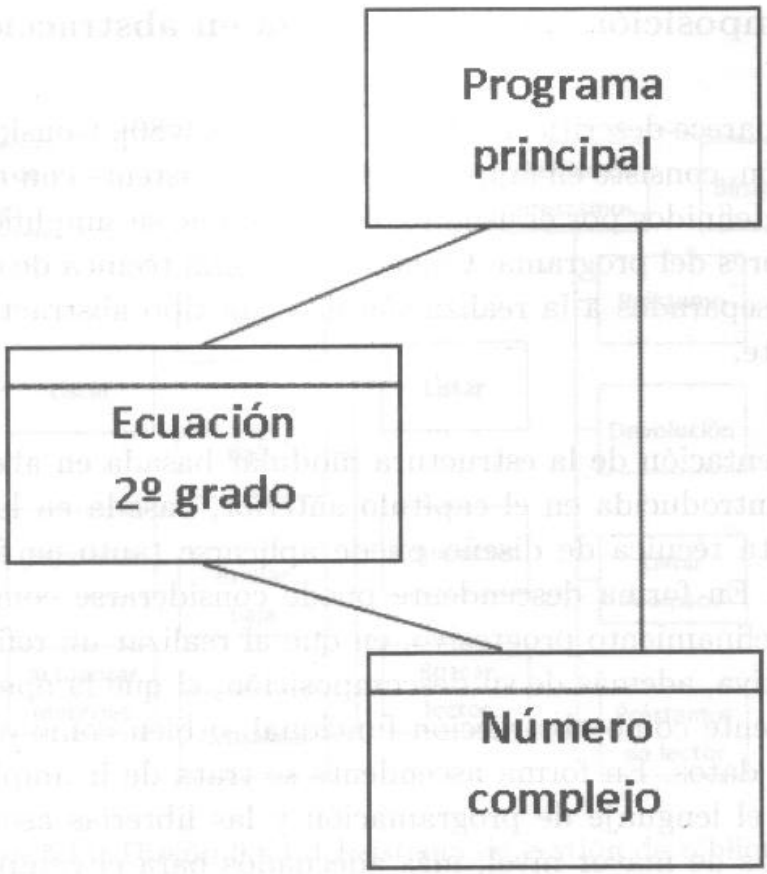


Figura 5.12: Ejemplo de diseño basado en abstracciones

5.5.2. 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 capítulo. 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 segunda 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
textos impresos
= salida
margen derecho
= margen
palabras
texto de entrada = texto inicial
separación en párrafos
= 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)

OPERACIONES
ajustado
= ajustar
= ajuste
= intercalando (espacios)
recomponen
quepan
ser impreso

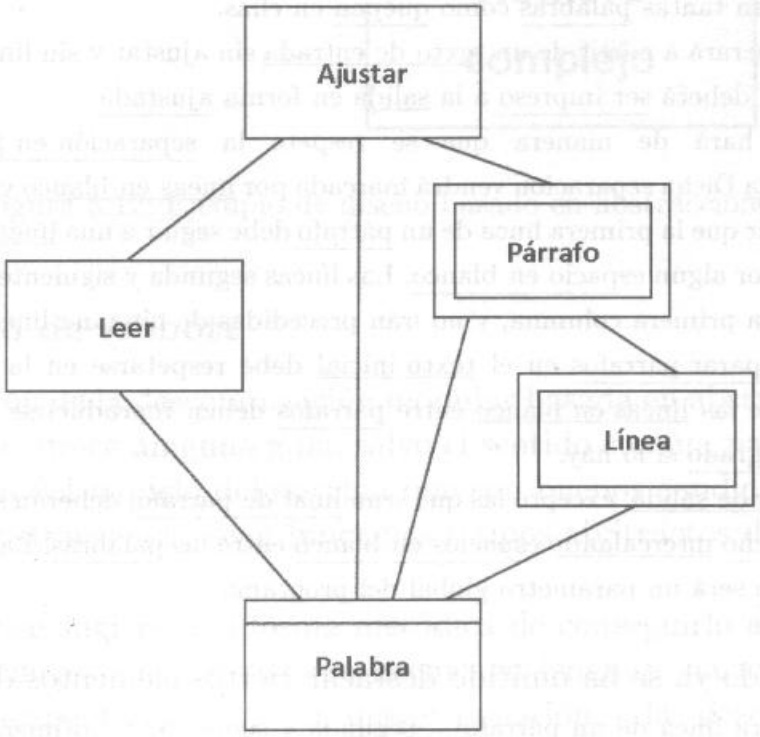


Figura 5.13: Programa de ajuste: diseño mediante abstracciones

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: Párrafo (de salida)
Atributos:
separador de párrafo
líneas de salida
Operaciones:
iniciar párrafo
poner palabra (= recomponer)
terminar párrafo

DATO: Separador de párrafo
Atributos:
líneas en blanco
sangrado
Operaciones:

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 5.13. 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).

5.5.3. Ejemplo: Videojuego de las minas

Este ejemplo ha sido descrito en el capítulo 3, 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
Atributos:
tamaño
datos de casillas
nº de minas
nº de casillas destapadas
Operaciones:
iniciar tablero
mover cursor
destapar
poner/quitar marca

DATO: Casilla
Atributos:
hay mina
nº de minas vecinas
está destapada
esta marcada

OPERACIÓN: Minas (juego)

DATO: Tabla de resultados
Atributos:(por nivel y entrada)
texto
tiempo
Operaciones:
anotar resultado
presentar 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 capítulo.

5.6. 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.

5.6.1. 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:

   3.1 Identificar los objetos (o clases), sus atributos y componentes.

   3.2 Identificar las operaciones sobre los objetos y asociarlas a la clase u objeto adecuado.

   3.3 Aplicar herencia donde sea conveniente.

   3.4 Describir cada operación en función de las otras, y subsanar posibles omisiones.

   3.5 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. 3.1 **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.

   3.2 **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.

   3.3 **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.

   3.4 **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 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.

3.5 **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.

5.6.2. 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 desvíen 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:

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 desvíen 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:

DATOS	OPERACIONES
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 desvíen	recibir
dispositivo de medida	=explorar
mensaje	transmitir
fin de transmision	=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

Dispositivo de medida
leer
iniciar medición
obtener máximo
obtener mínimo
obtener media

Mensaje
enviar
recibir

Reloj
leer
poner en hora

Estación
arrancar
detener

3.c Aplicar herencia. En este caso se puede observar que no todas la 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 desvían
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 desvían
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, mínino: 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
         iniciar la estación
         REPETIR
            SI hay mensaje ENTONCES
               recibir mensaje
               poner en hora le reloj
               componer mensaje con datos de la hora anterior
               enviar el mensaje
            FIN-SI

            SI ha pasado un minuto ENTONCES
               leer medidas de temperatura, viento y presión
            FIN-SI

            SI ha pasado un minuto ENTONCES
               leer pluviómetro
               iniciar lectura de 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 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 5.14 Las relaciones de herencia de la figura 5.14 se traducen ahora a relaciones de uso. Una subclase utiliza el código de su superclase.

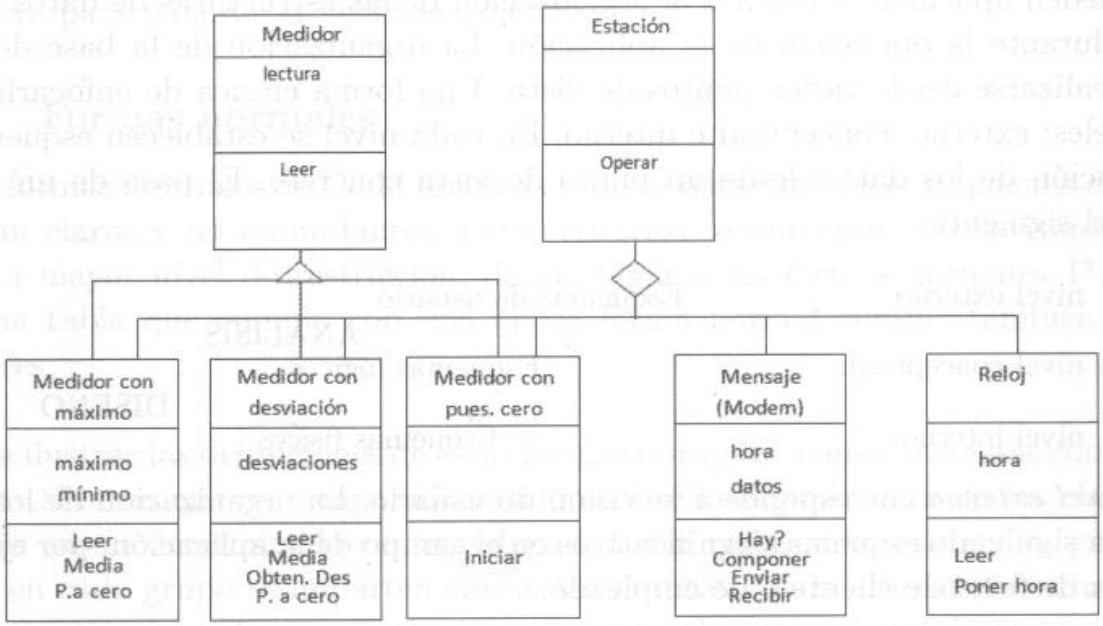


Figura 5.14: Modelo de objetos de la estación meteorológica

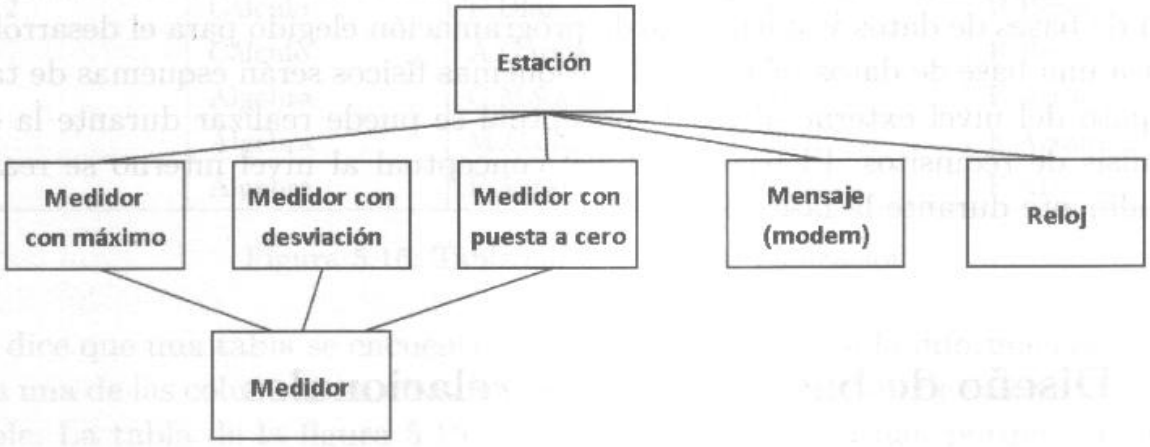
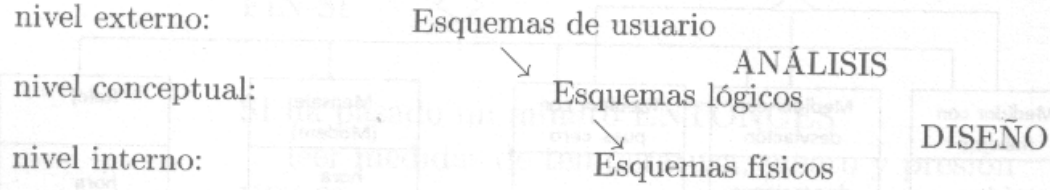


Figura 5.15: Arquitectura del software de la estación meteorológica

5.7. 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 dé 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. Esa organización se resume en un diagrama de modelo de datos, bien del tipo Entidad-Relación (descrito en el capítulo 3) o como diagrama de Modelo de Objetos (descrito en el capítulo 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.

5.8. 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.

5.8.1. 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 1ª, 2ª, 3ª, 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 5.16 se recoge toda esta información en una sola tabla.

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 5.16: Tabla sin ninguna normalización

Se dice que una tabla se encuentra en 1ª *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 5.16 no está en la 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 está en la 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 5.17.

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 5.17: Tabla en 1ª forma normal pero no en 2ª

Se dice que una tabla está en 2ª *forma normal* si está en 1ª 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 5.17 la clave primaria es la combinación de las dos primeras columnas. La tabla no está en 2ª 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 5.18

Grupo	Asignatura	Profesor	Despacho
11,	Cálculo	F.Díaz	D-23
12,	Cálculo	F.Díaz	D-23
13	Cálculo	A.García	D-27
11	Álgebra	C.Morales	D-34
12	Álgebra	M.Campos	D-21
13	Álgebra	F.Arranz	D-32

Asignatural	Coordinador
Cálculo	R.Pérez
Álgebra	F.Arranz

Figura 5.18: Tabla en 2ª forma normal pero no en 3ª

Se dice que una tabla está en 3ª forma normal si satisface el criterio de la 2ª 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 5.18 no está en 3ª 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 5.19

Grupo	Asignat.	Prof.
11,	Cálculo	F.Díaz
12,	Cálculo	F.Díaz
13	Cálculo	A.García
11	Álgebra	C.Morales
12	Álgebra	M.Campos
13	Álgebra	F.Arranz

Asignat.	Coord.
Cálculo	R.Pérez
Álgebra	F.Arranz

Prof.	Desp.
F.Díaz	D-23
A.García	D-27
C.Morales	D-34
M.Campos	D-21
F.Arranz	D-32

Figura 5.19: Tabla en 3ª forma normal

Existen criterios aún más restrictivos que definen las formas normales 4ª y siguientes, que se aplican poco en la práctica.

5.8.2. 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.

5.8.3. 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 corno 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 5.20 (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 5.20 (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 5.20 (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.

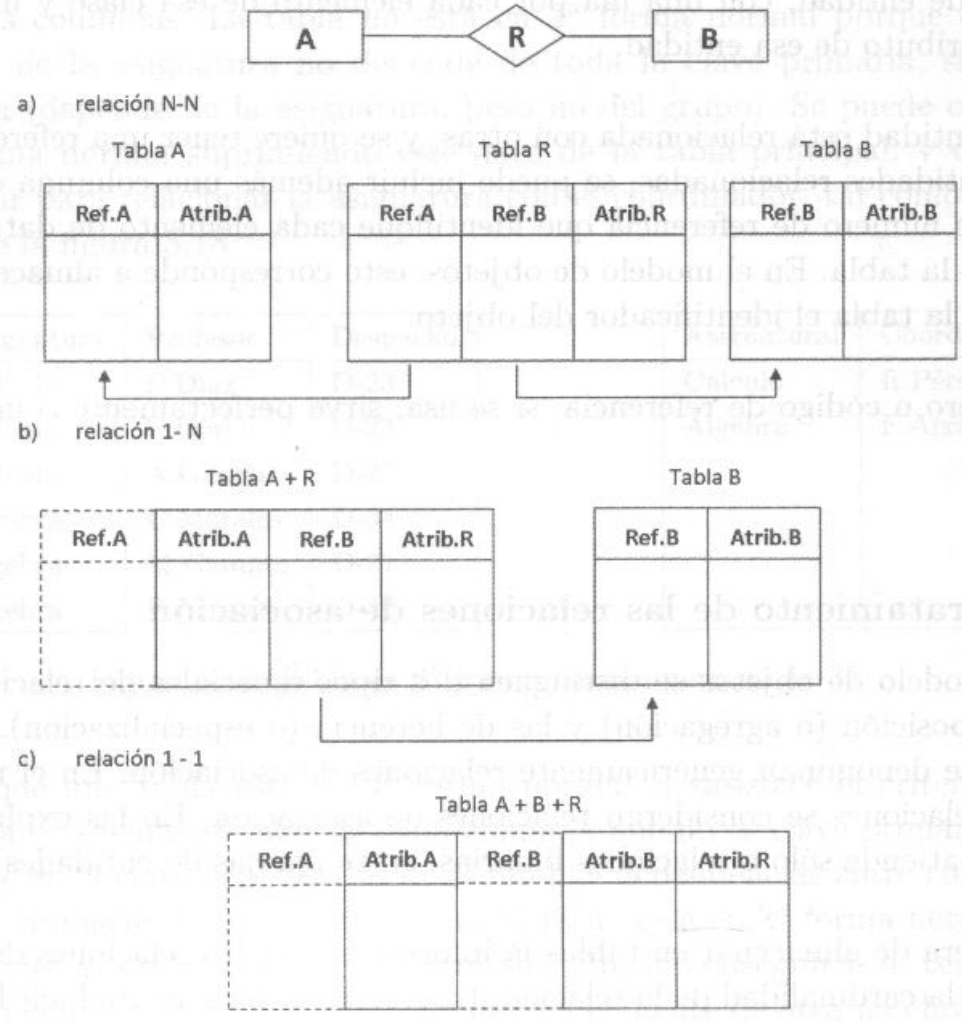


Figura 5.20: Tablas para relaciones de asociación

5.8.4. 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.

5.8.5. 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 5.21. 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.

5.8.6. Diseño de Índices

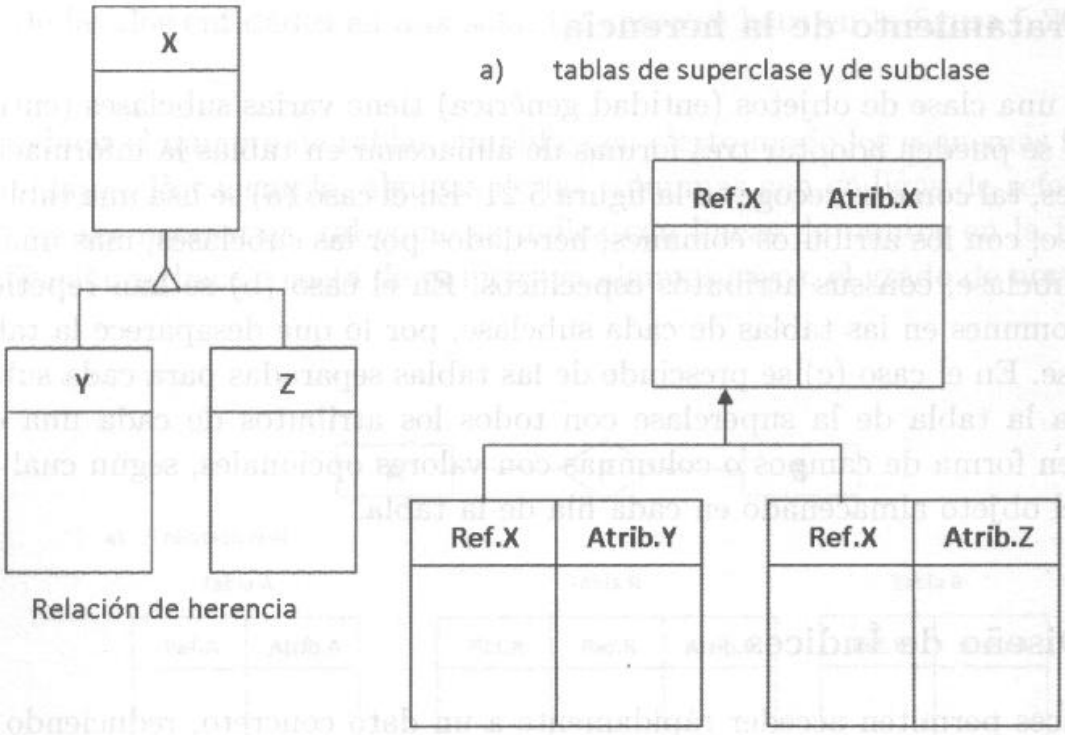
Los índices permiten acceder rápidamente a un dato concreto, reduciendo así el tiempo de acceso. Pero esto es a costa de aumentar el espacio necesario de almacenamiento y, lo que es más importante, aumentar el tiempo necesario para almacenar cada nuevo dato y más aún para modificar el valor de un atributo 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.

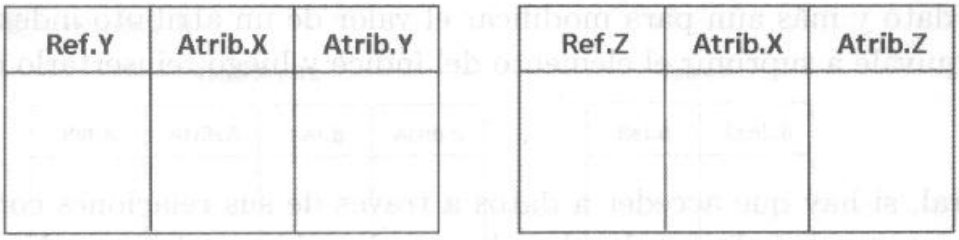
5.8.7. Ejemplo: Diseño de datos para la gestión de biblioteca

La visión externa de los datos está constituida por los elementos "Ficha de libro" y "Ficha de lector", con los siguientes contenidos:

Ficha de lector:	Ficha de libro:	
Nombre	Autor(es)	Materia(s)
Apellidos	Título	Lector (si está prestado)
Teléfono	Editorial, etc.	Fecha del préstamo (si está prestado)
Domicilio		



b) sólo tablas de subclases



c) sólo tabla de superclase

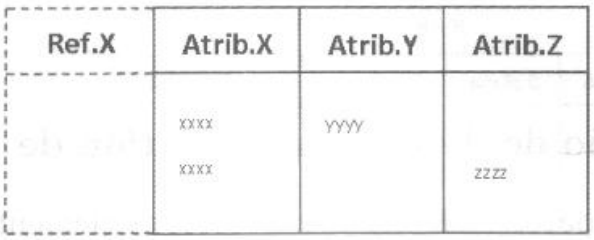


Figura 5.21: Tablas para relaciones de herencia

En el documento de especificación de requisitos correspondiente a este ejemplo, que aparece en el capítulo 3, 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 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 1ª forma normal, ya que en el mismo campo tenemos una colección de datos en número variable. Para conseguir la 1ª 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

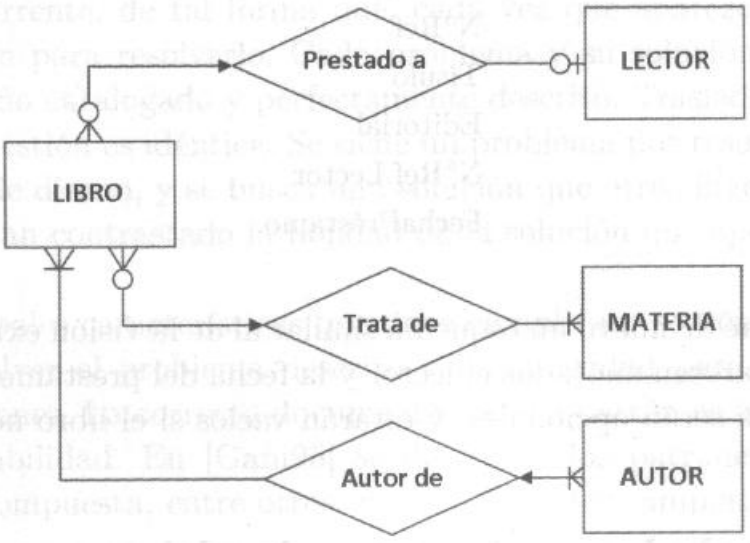


Figura 5.22: Modelo de datos revisado

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 N° de referencia para acceso mediante las relaciones.

LIBRO	LECTOR	AUTOR	MATERIA
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.

TRATA-DE	AUTOR-DE
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.

LIBRO
NºRef
Título
Editorial
NªRef.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.

5.9. 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.

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.

5.10. Diseño de software con patrones

El diseño con patrones fue introducido en el mundo científico dentro de la arquitectura por Christopher Alexander [Ale77] y adoptado con posterioridad en la ingeniería de software, [Gam95], siendo por tanto, la primera referencia obligada en este campo. El concepto de patrón se puede definir como una solución probada a un problema recurrente, de tal forma que, cada vez que aparezca dicho problema, se aplica el patrón para resolverlo. Cada problema y su solución está descrito por un patrón de diseño catalogado y perfectamente descrito. Trasladado a la ingeniería del software, la cuestión es idéntica. Se tiene un problema por resolver, fundamentalmente en la fase de diseño, y se busca una solución que otros ingenieros de software ya resolvieron y han contrastado la bondad de la solución que aportaron.

Las dos principales características que debe cumplir un patrón son la efectividad a la hora de resolver el problema descrito y la capacidad para ser reutilizado en multitud de ocasiones. La correcta documentación del patrón es imprescindible para lograr la reutilizabilidad. En [Gam95] se describen los patrones a través de una plantilla común compuesta, entre otros, de los siguientes campos:

- Nombre del patrón
- Propósito
- Motivación
- Aplicabilidad
- Desarrollo del patrón

- Usos conocidos
- Ejemplos

Cada vez son más las aplicaciones de los patrones dentro de la ingeniería de software, algunas de ellas son:

- Interfaces de usuario u hombre-computador
- Patrones de diseño en programación orientada a objetos
- Patrones para la integración de sistemas heterogéneos que deben comunicarse y sincronizarse
- Patrones de flujo de datos dentro de la gestión de procesos empresariales

El empleo de patrones permite crear aplicaciones de gran tamaño en un periodo de tiempo muy corto, lo que puede llevar a que aparezcan problemas para la elección, organización y clasificación de los mismos. En [Mic04] se propone la creación de una tabla organizadora de patrones, la cual permite analizar los posibles patrones a utilizar para cada parte del programa que se está diseñando y elegir el más adecuado. La variedad de repositorios de patrones y la multiplicidad de lenguajes de programación hacen compleja esta tarea.

Dentro de los patrones de diseño en programación orientada a objetos, [Gam95] clasifica los patrones en tres grandes grupos:

- Patrones creacionales, son los que resuelven problemas en la creación de instancias.
- Patrones estructurales, son los que resuelven problemas de composición de clases u objetos para dar lugar a otras más complejas.
- Patrones de comportamiento, son lo que resuelven problemas en la interacción, comunicación y responsabilidad entre clases u objetos.

En www.wikipedia.org se describen varios ejemplos de los distintos tipos de patrones. A continuación se reproduce el ejemplo de un patrón de comportamiento, la clase Iterador (Iterator). Ver figura 5.23.

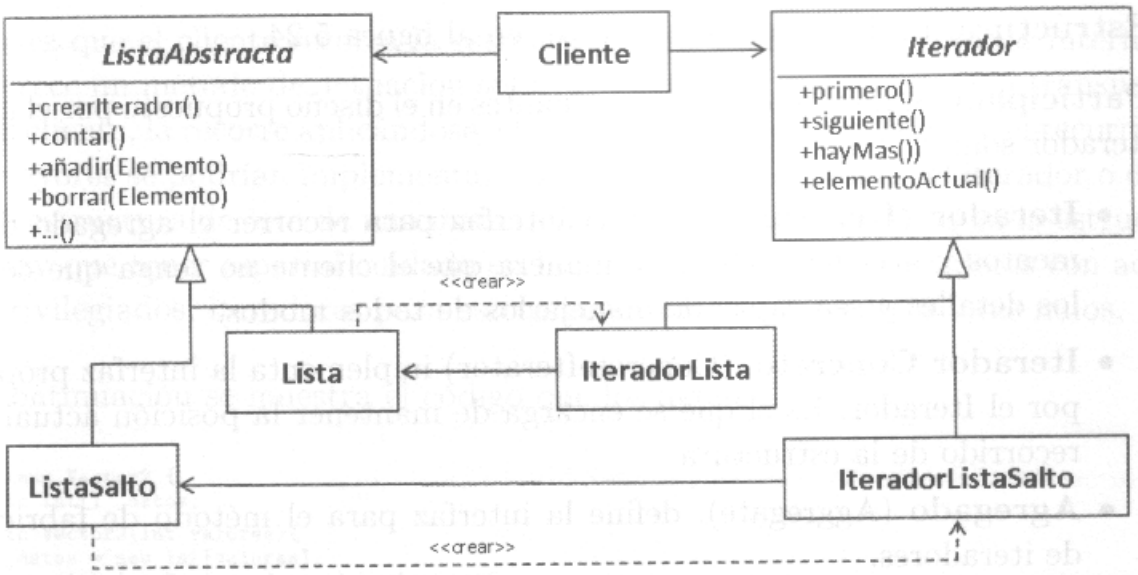


Figura 5.23: Esquema de uso de patron Iterador

- **Propósito**. El patrón Iterador es un mecanismo de acceso a los elementos que constituyen una estructura de datos para la utilización de estos sin exponer su estructura interna. Ver figura 5.23

- **Motivación**. El patrón surge del deseo de acceder a los elementos de un contenedor de objetos (por ejemplo, una lista) sin exponer su representación interna. Además, es posible que se necesite más de una forma de recorrer la estructura siendo para ello necesario crear modificaciones en la clase. La solución que propone el patrón es añadir métodos que permitan recorrer la estructura sin referenciar explícitamente su representación. La responsabilidad del recorrido se traslada a un objeto iterador.

  El problema de introducir este objeto iterador reside en que los clientes necesitan conocer la estructura para crear el iterador apropiado. Esto se soluciona generalizando los distintos iteradores en una abstracción y dotando a las estructuras de datos de un método de fabricación que cree un iterador concreto.

- **Aplicabilidad**. El patrón iterator permite el acceso al contenido de una estructura sin exponer su representación interna. Además diferentes iteradores pueden presentar diferentes tipos de recorrido sobre la estructura (recorrido de principio a fin, recorrido con saltos...). Por otro lado los iteradores no tienen por qué limitarse a recorrer la estructura, sino que podrían incorporar otro tipo de lógica (por ejemplo, filtrado de elementos). Es más, dados diferentes tipos de estructuras, el patrón iterador permite recorrerlas todas utilizando una interfaz común uniforme.

- **Estructura**. El diagrama se puede ver en al figura 5.24
- **Participantes**. Las entidades participantes en el diseño propuesto por el patrón iterador son:
  - **Iterador (Iterator)**. define la interfaz para recorrer el agregado de elementos y acceder a ellos, de manera que el cliente no tenga que conocer los detalles y sea capaz de manejarlos de todos modos.
  - **Iterador Concreto**. (ConcreteIterator) implementa la interfaz propuesta por el Iterador. Es el que se encarga de mantener la posición actual en el recorrido de la estructura.
  - **Agregado** (Aggregate). define la interfaz para el método de fabricación de iteradores.
  - **Agregado Concreto**. (ConcreteAggregate) implementa la estructura de datos y el método de fabricación de iteradores que crea un iterador específico para su estructura.

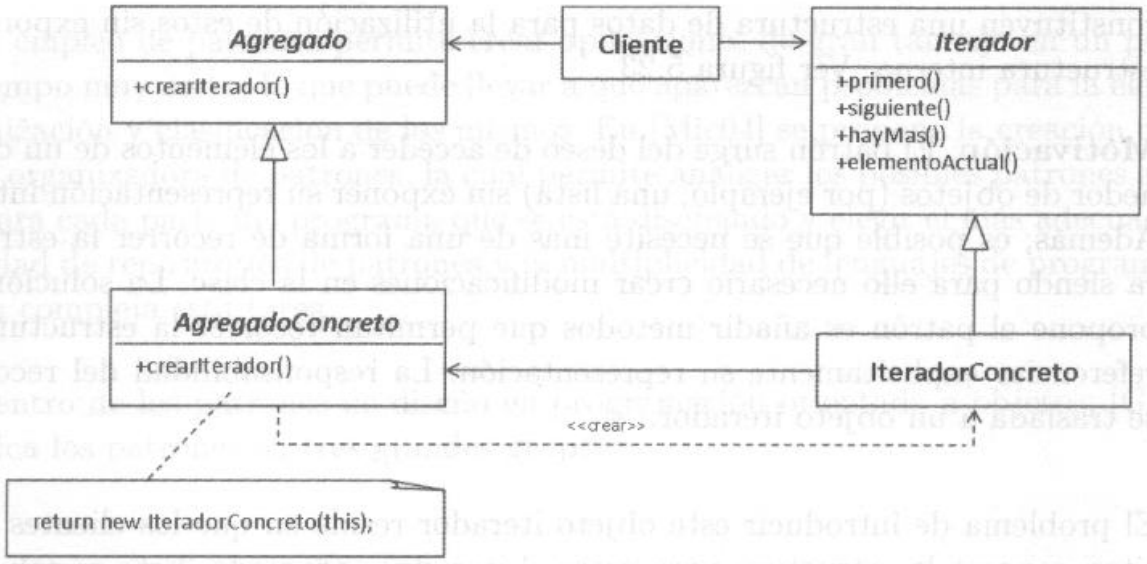


Figura 5.24: Estructura genérica de patron Iterador

- **Colaboraciones**. Un iterador concreto es el encargado de guardar la posición actual dentro de la estructura de datos, interactuando con esta para calcular el siguiente elemento del recorrido.
- **Consecuencias**. El patrón Iterador permite por tanto diferentes tipos de recorrido de un agregado y varios recorridos simultáneos, simplificando la interfaz del agregado.
- **Implementación**. Para la creación del patrón iterador debe implementarse el control de la iteración (pudiendo ser un iterador externo que ofrece los métodos

para que el cliente recorra la estructura paso a paso, o un iterador interno que ofrece un método de actuación sobre la estructura que, de manera transparente al cliente, la recorre aplicándose a todos sus elementos) y definirse el recorrido. A mayores se podrían implementar operaciones adicionales en el iterador o definir la estructura de éste de una manera más robusta ante cambios en la estructura. Hay que tener especial cuidado en la implementación de iteradores con accesos privilegiados, iteradores para estructuras compuestas o iteradores nulos.

A continuación se muestra el código que los define

```
public class Vector2 {
    public int[] _datos;
    public Vector2(int valores){
        _datos = new int[valores];
        for (int i = 0; i < _datos.length; i++){
            _datos[i] = 0;
        }
    }
    public int getValor(int pos){
        return _datos[pos];
    }
    public void setValor(int pos, int valor){
        _datos[pos] = valor;
    }
    public int dimension(){
        return _datos.length;
    }
    public IteradorVector iterador(){
        return new IteradorVector(this);
    }
}
```

Definición del iterador concreto.

```
public class IteradorVector{
private int[] _vector;
    private int _posicion;
    public IteradorVector(Vector2 vector) {
        _vector = vector._datos;
        _posicion = 0;
    }
    public boolean hasNext(){
        if (_posicion < _vector.length)
 return true;
 else
 return false;
    }
    public Object next(){
 int valor = _vector[_posicion];
 _posicion++;
 return valor;
    }
}
```

Ejemplo de uso

```
    public static void main(String argv[]) {
     Vector2 vector = new Vector2(5);
     //Creación del iterador
     IteradorVector iterador = vector.iterador();
     //Recorrido con el iterador
     while (iterador.hasNext())
             System.out.println(iterador.next());
    }
```

En java ya existe una interfaz iterator que hace a las veces de abstracción. Todos los iteradores utilizados en sus librerías cumplen esta interfaz, lo que permite tratarlos a todos ellos de manera uniforme. Hacer que nuestro IteradorVector implementase Iterator permitiría que fuese utilizado por otras librerías y programas de manera transparente.

5.11. Ejemplos de diseños

En este apartado se recogen los documentos completos de diseño de los dos sistemas que se han ido desarollando a lo largo del libro.

5.11.1. Ejemplo: Videojuego de las minas

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

DOCUMENTO DE DISEÑO DEL SOFTWARE (AAD)

Proyecto: JUEGO DE LAS MINAS (Versión simple)

Autores: J.A. Cerrada y R.Gómez Fecha: Enero 1999

Documento: MINAS-ADD-99

CONTENIDO

1. INTRODUCCIÓN
   1.1. Objetivo
   1.2. Ámbito
   1.3. Definiciones, siglas y abreviaturas
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 REQUISISTOS/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.

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	..	X	1	3					
	3	1	1	..	..	1					
	4	!!	2	..	..	1					
		!!	4	1	1	1					

Figura M1: 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 O 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 desee 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

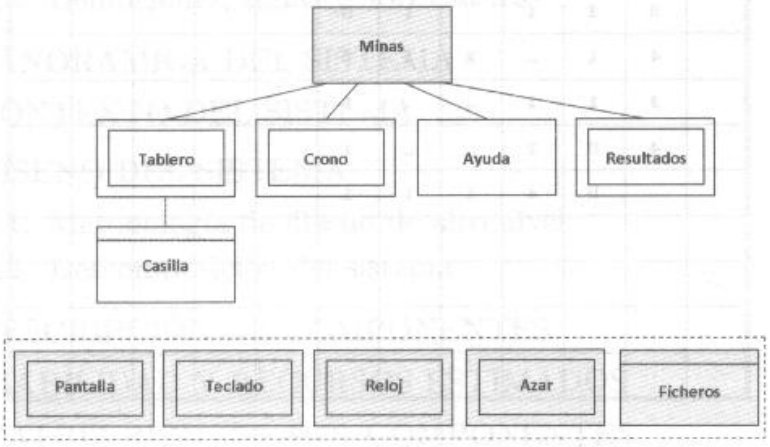


Figura M2: Diagrama de estructura

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

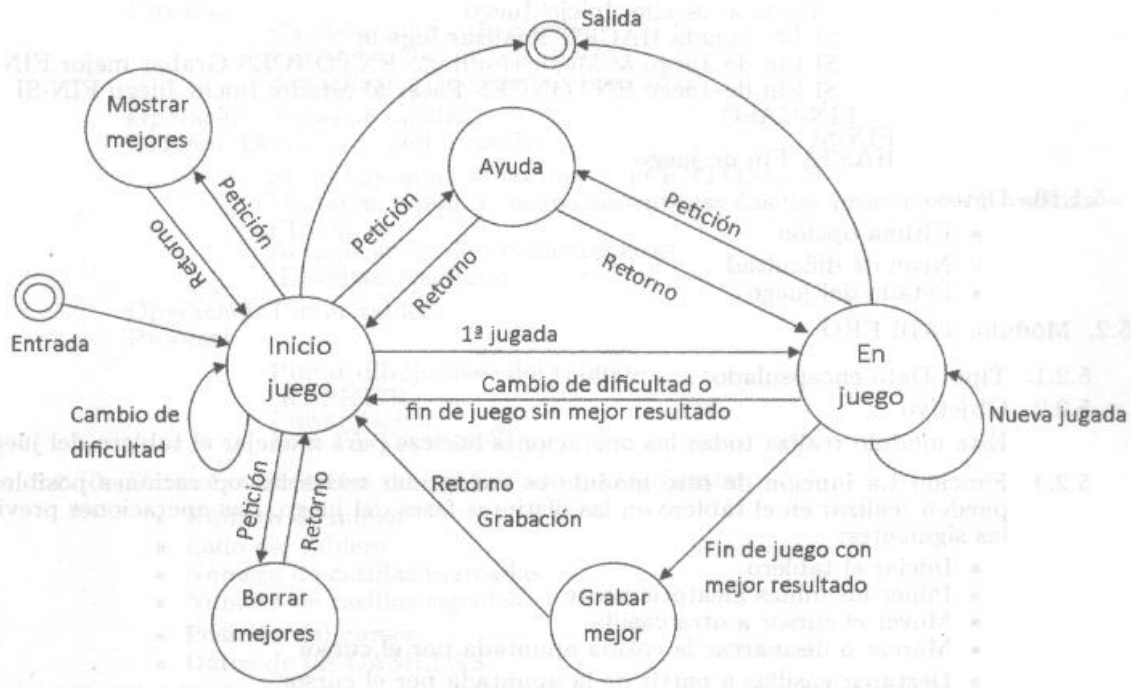


Figura M3: Diagrama de estados del sistema

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. intezfases: Ninguna

5.1.7. Recursos: Ninguno

5.1.8. Referencias: Ninguna

5.1.9. Proceso: De acuerdo con el diagrama de la figura M.3 será:

```
Inicializar 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:I/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 n° de minas que rodean a las casillas vecinas
HASTA Total de minas

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 n° de casillas marcadas

Operación:Destapar casillas
Proceso: Destapar y pintar casilla
SI no hay mina & vecinas = O ENTONCES
Destapar y pintar recursivamente las casillas próximas con vecinas = O
FIN-SI
Actualizar n° 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
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

5.5.6. Intetfases
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. intetfases: 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, posicion 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

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 cursar
- 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 linea
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 cursar

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 leida

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

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.

	MÓDULOS					
REQUISITOS	MINAS	TABLERO	CRONO	AYUDA	RESULTADOS	CASILLA
R.1.1	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 M4: Matriz REQUISITOS-COMPONENTES

5.11.2. 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 capítulo 3. El documento ADD para el sistema de gestión de biblioteca es el siguiente:

DOCUMENTO DE DISEÑO DEL SOFTWARE (AAD)

Proyecto: SISTEMA DE GESTIÓN DE BIBLIOTECA

Autores: M. Collado y J.F. Estívariz Fecha: Noviembre 1999

Documento: BIBLIO-ADD-99

CONTENIDO

1. INTRODUCCIÓN
   1.1. Objetivo
   1.2. Ámbito
   1.3. Definiciones, siglas y abreviaturas
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 REQUISISTOS/COMPONENTES

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 incluida 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 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

Corno 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 corno 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
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.

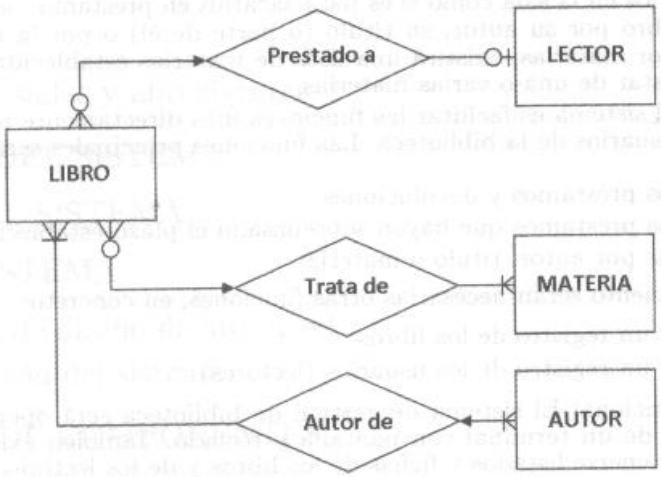


Figura B1: Modelo de datos ENTIDAD-RELACIÓN

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:

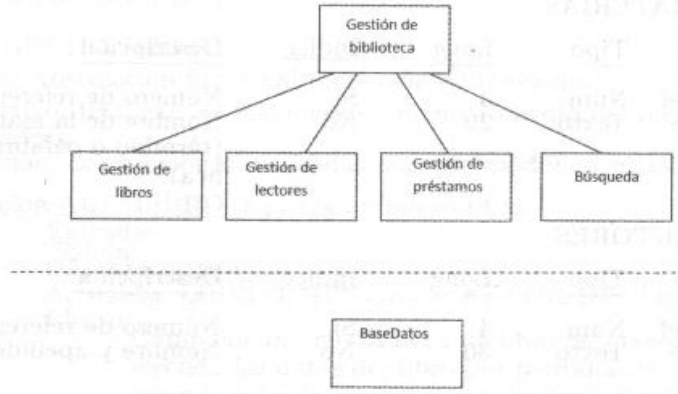


Figura B2: Arquitectura del sistema

- 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

Campo	Tipo	Long.	Indice	Descripción
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

Campo	Tipo	Long.	Indice	Descripción
NumRef	Num	4	Sí	Número de referencia del autor
nombre	Texto	30	No	Nombre y apellidos del autor

- Tabla: AUTOR DE

Campo	Tipo	Long.	Indice	Descripción
Libro	Num	4	Sí	Número de referencia del libro
Autor	Num	4	No	Número de referencia del autor

- Tabla: TRATA-DE

Campo	Tipo	Long.	Indice	Descripción
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

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 ALTALIBRO: 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 corno 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
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 varías 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

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					
REQUISITOS	BaseDatos	Biblio	GestionLibros	GestionLectores	GestionPrestamos	Busquedas
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 B3: Matriz REQUISITOS-COMPONENTES

5.12. Conclusiones

En este capítulo se hace un repaso a las técnicas más importantes de diseño que se emplean habitualmente:

- Técnicas por refinamiento progresivo

- Técnicas de diseño basado en abstracciones
- Técnicas de diseño orientado a objetos
- Técnicas de diseño de datos

Todas ellas tratan de facilitar la realización del diseño. Basadas en una o varias de estas técnicas se han desarrollado metodologías completas de diseño. En cursos posteriores se imparten asignaturas donde se desarrollan en profundidad estas técnicas y otras que se basan en ellas.

5.13. Ejercicios propuestos

1. Búsquese dos ejemplos en los que ponga de manifiesto la propiedades de Acoplamiento y Cohesión dentro de un diseño de un sistema.
2. Realícese el diseño del "autobus" del capítulo pasado utilizando las técnicas de diseño basado en abstracciones.
3. Diséñese las tablas de la base de datos que modeliza una escuela con profesores, asignaturas, alumnos y departamentos. Extiéndase a la universidad con otras facultades.
4. Hágase un diseño del sistema de gestión de biblioteca orientada objetos.
5. Plantéese como conseguiría en su diseño un acoplamiento débil en los módulos.

Capítulo 6

UML, Lenguaje Unificado de Modelado

6.1. Introducción

En los capítulos anteriores se han presentado las metodologías y herramientas para la realización de las fases del ciclo de vida del software, tanto para la especificación de requisitos, como análisis y modelado de los sistemas software.

En este capítulo se presenta UML, un lenguaje universal de modelado de sistemas que se emplea para especificar y modelizar los sistemas software.

6.2. Objetivos

El objetivo de este capítulo es que el lector se familiarice con el lenguaje UML y su utilización para la especificación y el modelado. En primer lugar se hace un breve recorrido por la historia de su génesis. Esto permite contemplar cómo es un proceso de normalización de herramientas conceptuales, en el que están implicadas tanto la industria del sector como la comunidad científica, y cómo se lleva a cabo con total normalidad y se llega a un estandard de aplicación mundial. En segundo lugar se presenta los elementos de lenguaje UML, relaciones, clases y diagramas. Algunos de estos elementos se salen fuera del alcance de la asignatura pero se presentan para completar la visión de todo el lenguaje UML.

6.3. ¿Qué es UML?

El lenguaje unificado de modelado (UML, por sus siglas en inglés, Unified Modeling Language) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad. Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocio, funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y compuestos reciclados.

Es importante remarcar que UML es un "lenguaje de modelado" para especificar o para describir métodos o procesos. Se utiliza para definir un sistema, para detallar los artefactos en el sistema y para documentar y construir. En otras palabras, es el lenguaje en el que está descrito el modelo.

Se puede aplicar en el desarrollo de software gran variedad de formas para dar soporte a una metodología de desarrollo, pero no especifica en sí mismo qué metodología o proceso usar.

UML no puede compararse con la programación estructurada, pues UML significa Lenguaje Unificado de Modelado, no es programación, sólo se diagrama la realidad de una utilización en un requerimiento. Mientras que, programación estructurada, es una forma de programar como lo es la orientación a objetos. La programación orientada a objetos viene siendo un complemento perfecto de UML, pero no por eso se toma UML sólo para lenguajes orientados a objetos.

UML es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software. Se usa para entender, diseñar, configurar, mantener y controlar la información sobre los sistemas a construir.

UML presenta información sobre la estructura estática y el comportamiento dinámico de un sistema. Un sistema se modela como una colección de objetos discretos que interactúan para realizar un trabajo que finalmente beneficia a un usuario externo.

El lenguaje de modelado pretende unificar la experiencia pasada sobre técnicas de modelado e incorporar las mejores prácticas actuales en un acercamiento estándar.

Existen herramientas que pueden ofrecer generadores de código a partir de los diagramas UML para una gran variedad de lenguaje de programación, así como construir modelos por ingeniería inversa a partir de programas existentes.

6.4. Orígenes de UML

La notación UML deriva y unifica las tres metodologías de análisis y diseño más extendidas: metodología de Grady Booch [Booch94] para la descripción de conjuntos de objetos y sus relaciones, técnica de modelado orientada a objetos de James Rumbaugh (OMT: Object - Modelling Technique)[Rumbaugh91] y aproximación de Ivar Jacobson (OOSE: Object- Oriented Software Engineering)[Jacobson92] mediante la metodología de casos de uso (use case).

El desarrollo de UML comenzó a finales de 1994 cuando Grady Booch y Jim Rumbaugh de Rational Software Corporation empezaron a unificar sus métodos. A finales de 1995, Ivar Jacobson y su compañía Objectory se incorporaron a Rational en su unificación, aportando el método OOSE.

De las tres metodologías de partida, las de Booch y Rumbaugh pueden ser descritas como centradas en objetos, ya que sus aproximaciones se enfocan hacia el modelado de los objetos que componen el sistema, su relación y colaboración. Por otro lado, la metodología de Jacobson es más centrada a usuario, ya que todo en su método se deriva de los escenarios de uso. UML se ha ido fomentando y aceptando como estándar desde el Object Management Group (OMG).

Diversas organizaciones y empresas colaboraron en la elaboración de la primera versión del lenguaje. Entre ellas Digital Equipment Corporation, Hewlett-Packard, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments y Unisys.

En 1997 UML 1.1 fue aprobada por la OMG convirtiéndose en la notación estándar de facto para el análisis y el diseño orientado a objetos. Como prueba de su validez como lenguaje de modelado, UML es el primer método en publicar un meta-modelo en su propia notación, incluyendo la notación para la mayoría de la información de requisitos, análisis y diseño. Se trata pues de un meta-modelo autoreferencial (cualquier lenguaje de modelado de propósito general debería ser capaz de modelarse a sí mismo).

Desde el año 2005, UML es un estándar aprobado por la ISO como ISO/IEC 19501:2005 Information technology, Open Distributed Processing Unified Modeling Language (UML). Con el paso de los años el lenguaje ha ido evolucionando, incorporando nuevas funcionalidades para conseguir una mayor unificación integrando todas las posibles tecnologías de desarrollo de software. La versión más actual es la 2.5.

6.5. Objetivos de UML

Durante el desarrollo del UML sus autores tuvieron en cuenta los siguientes objetivos:

- Conseguir la capacidad de modelar sistemas, desde el concepto hasta los artefactos ejecutables, utilizando técnicas orientadas a objetos.
- Cubrir las cuestiones relacionadas con el tamaño inherentes a los sistemas complejos y críticos.
- Crear un lenguaje de modelado utilizable tanto por las personas como por las máquinas.

Mediante el fomento del uso de UML, OMG pretende alcanzar los siguientes objetivos:

- Proporcionar a los usuarios un lenguaje de modelado visual expresivo y utilizable para el desarrollo e intercambio de modelos significativos.
- Proporcionar mecanismos de extensión y especialización.
- Ser independiente del proceso de desarrollo y de los lenguajes de programación.
- Proporcionar una base formal para entender el lenguaje de modelado.
- Fomentar el crecimiento del mercado de las herramientas OO.
- Soportar conceptos de desarrollo de alto nivel como pueden ser colaboraciones, frameworks, patterns, y componentes.
- Integrar las mejores prácticas utilizadas hasta el momento.

El UML debe entenderse como un estándar para modelado y no como un estándar de proceso software. Aunque UML debe ser aplicado en el contexto de un proceso, la experiencia ha mostrado que organizaciones diferentes y dominios del problema diferentes requieren diferentes procesos.

UML es un lenguaje que se puede utilizar no sólo para modelar aplicaciones software, también se puede utilizar en otros campos como:

- Sistemas de información de la empresa
- Bancos y servicios financieros
- Telecomnicaciones
- Transporte

- Defensa/industria aerospacial
- Comercio
- Electrónica médica
- Ámbito científico
- Servicios distribuidos en la Web

6.6. Estructura de UML

Los conceptos y modelos de UML pueden agruparse en las siguientes áreas conceptuales:

1. Estructura estática:

   Cualquier modelo preciso debe primero definir su universo, esto es, los conceptos clave de la aplicación, sus propiedades internas, y las relaciones entre cada una de ellas. Este conjunto de construcciones es la estructura estática.

   Los conceptos de la aplicación son modelados como clases, cada una de las cuales describe un conjunto de objetos que almacenan información y se comunican para implementar un comportamiento.

   La información que almacena es modelada como atributos; La estructura estática se expresa con diagramas de clases y puede usarse para generar la mayoría de las declaraciones de estructuras de datos en un programa.

2. Comportamiento dinámico:

   Hay dos formas de modelar el comportamiento, una es la historia de la vida de un objeto y la forma como interactúa con el resto del mundo y la otra es por los patrones de comunicación de un conjunto de objetos conectados, es decir la forma en que interactúan entre sí.

   La visión de un objeto aislado es una máquina de estados que muestra la forma en que el objeto responde a los eventos en función de su estado actual. La visión de la interacción de los objetos se representa con los enlaces entre objetos junto con el flujo de mensajes y los enlaces entre ellos. Este punto de vista unifica la estructura de los datos, el control de flujo y el flujo de datos.

3. Construcciones de implementación:

   Los modelos UML tienen significado para el análisis lógico y para la implementación física. Un componente es una parte física reemplazable de un sistema y es capaz de responder a las peticiones descritas por un conjunto de interfaces. Un nodo es un recurso computacional que define una localización durante la ejecución de un sistema. Puede contener componentes y objetos.

4. Mecanismos de extensión:

   UML tiene una limitada capacidad de extensión pero que es suficiente para la mayoría de las extensiones que requiere el día a día sin la necesidad de un cambio en el lenguaje básico.

   Un estereotipo es una nueva clase de elemento de modelado con la misma estructura que un elemento existente pero con restricciones adicionales.

5. Organización del modelo:

   La información del modelo debe ser dividida en piezas coherentes, para que los equipos puedan trabajar en las diferentes partes de forma concurrente. El conocimiento humano requiere que se organice el contenido del modelo en paquetes de tamaño modesto. Los paquetes son unidades organizativas, jerárquicas y de propósito general de los modelos de UML. Pueden usarse para almacenamiento, control de acceso, gestión de la configuración y construcción de bibliotecas que contengan fragmentos de código reutilizable.

6. Elementos de anotación:

   Los elementos de anotación son las partes explicativas de los modelos UML. Son comentarios que se pueden aplicar para describir, clasificar y hacer observaciones sobre cualquier elemento de un modelo. El tipo principal de anotación es la nota que simplemente es un símbolo para mostrar restricciones y comentarios junto a un elemento o un conjunto de elementos.

7. Relaciones:

Existen séis tipos de relaciones entre los elementos de un modelo UML. Dependencia, asociación, generalización, realización, agregación y composición estas se describen a continuación.

- **Dependencia**

Es una relación semántica entre dos elementos en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (elemento dependiente). Se representa como una línea discontinua, posiblemente dirigida, que a veces incluye una etiqueta.

En el diagrama de la figura 6.1 se puede ver una relación de dependencia entre dos clases. La clase A usa la clase B. La clase impresora usa la clase documento.

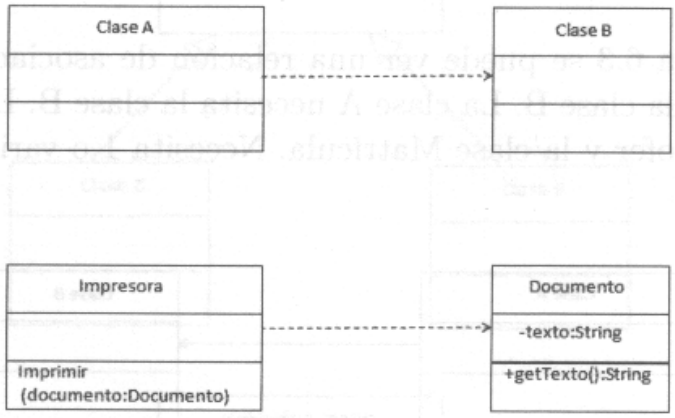


Figura 6.1: Relación de dependencia

- **Asociación**

Es una relación estructural que describe un conjunto de enlaces, los cuales son conexiones entre objetos. La agregación es un tipo especial de asociación y representa una relación estructural entre un todo y sus partes. La asociación se representa con una línea continua, posiblemente dirigida, que a veces incluye una etiqueta. A menudo se incluyen otros adornos para indicar la multiplicidad y roles de los objetos involucrados.

La multiplicidad de una asociación determina cuantos objetos de cada tipo intervienen en la relación, o sea, el número de instancias de una clase que se relacionan con una instancia de la otra. Cada asociación tiene dos multiplicidades, una para cada extremo de la relación. Para especificar la multiplicidad de una asociación hay que especificar la multiplicidad mínima y la multiplicidad máxima, (mínima..máxima). En la figura 6.2 se pueden ver los diferentes índices de multiplicidad y su significado.

Multiplicidad	Significado
1	uno y sólo uno
0..1	cero o uno
N..M	desde N hasta M
*	cero o varios
0..*	cero o varios
1..*	uno o varios
	(almenos uno

Figura 6.2: Multiplicidades de asociación

En la figura 6.3 se puede ver una relación de asociación. La clase A está asociada a la clase B. La clase A necesita la clase B. La clase Taxi necesita la clase Chofer y la clase Matrícula. Necesita 1 o varios chóferes y un sola matrícula.

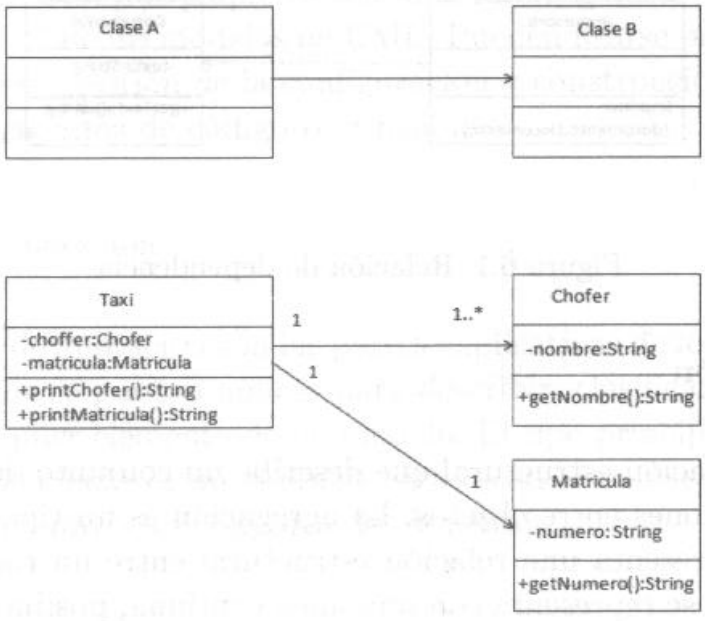


Figura 6.3: Relación de asociación

- **Generalización**

Es una relación de especialización / generalización en la cual los objetos del elemento especializado (el hijo) pueden sustituir a los objetos del elemento general (el padre). De esta forma, el hijo comparte la estructura y el comportamiento del padre. Gráficamente, la generalización se representa con una línea con punta de flecha vacía.

En la figura 6.4 se puede ver una relación de generalización. La clase A es una generalización de la B o de la clase C. También se puede comtemplar a la inversa y decir que la clase B y la clase C son especializaciones de la clase A. La clase Moto y la clase Coche son especializaciones de la la clase general Vehículo a Motor.

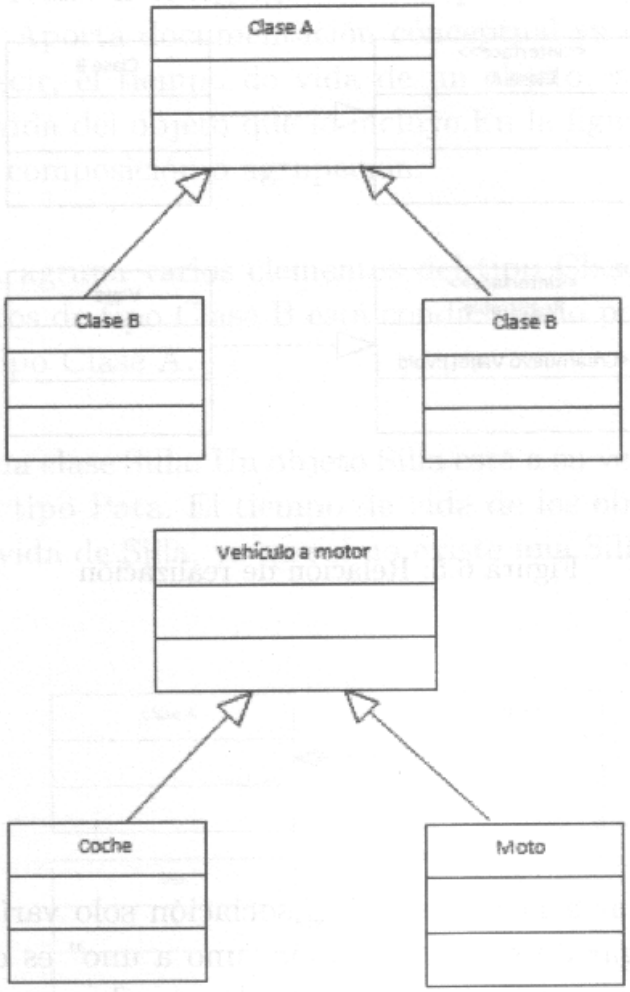


Figura 6.4: Relación de generalización

- **Realización**

Es una relación semántica entre clasificadores, donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Se pueden encontrar relaciones de realización en dos sitios: entre interfaces y las clases y componentes que las realizan, y entre los casos de uso y las colaboraciones que los realizan. La realización se representa como una mezcla entre la generalización y la dependencia, esto es, una línea discontinua con una punta de flecha vacía .

En la figura 6.5 se puede ver una relación de realización. La clase B es una realización de la A. La clase Viaje realiza la interface Registrable.

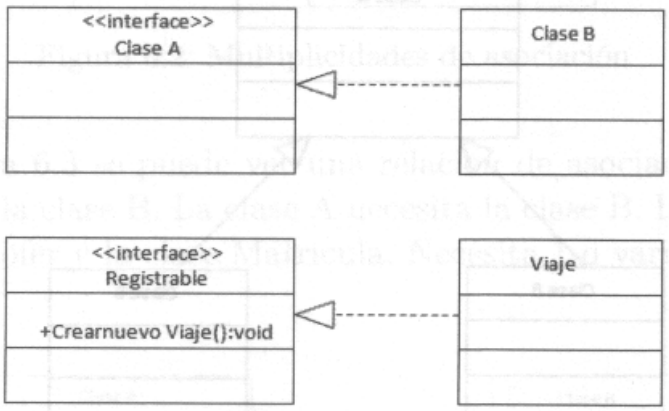


Figura 6.5: Relación de realización

- **Agregación**

Es muy similar a la relación de Asociación solo varía en la multiplicidad ya que en lugar de ser una relación "uno a uno" es de "uno a muchos".Se representa con una flecha que parte de una clase a otra en cuya base hay un rombo de color blanco. En la figura 6.5 se puede ver una relación de agrupación. La Clase A agrupa varios elementos del tipo Clase B. La clase Agenda agrupa a varios Contactos.

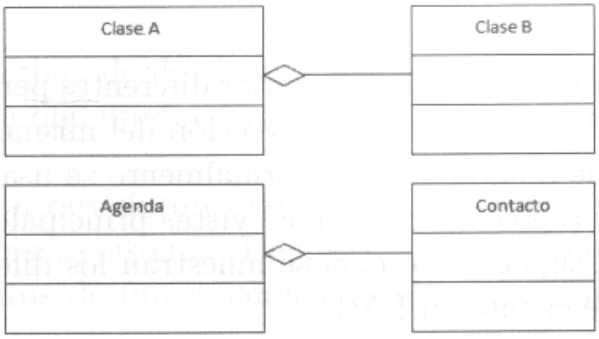


Figura 6.6: Relación de agregación

- **Composición**

Similar a la relación de Agregación solo que la Composición es una relación mas fuerte. Aporta documentación conceptual ya que es una "relación de vida", es decir, el tiempo de vida de un objeto está condicionado por el tiempo de vida del objeto que lo incluye.En la figura 6.7 se puede ver una relación de composición o agrupación.

La Clase A agrupa varios elementos del tipo Clase B. El tiempo de vida de los objetos de tipo Clase B está condicionado por el tiempo de vida del objeto de tipo Clase A.

Tenemos una clase Silla. Un objeto Silla está a su vez compuesto por cuatro objetos del tipo Pata. El tiempo de vida de los objetos Pata depende del tiempo de vida de Silla, ya que si no existe una Silla no pueden existir sus Patas.

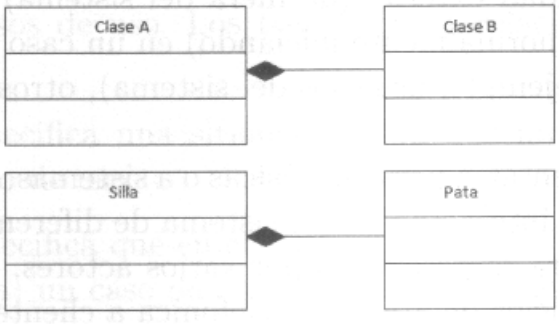


Figura 6.7: Relación de composición

8. Diagramas

Los diagramas se utilizan para representar diferentes perspectivas de un sistema de forma que un diagrama es una proyección del mismo. UML proporciona un amplio conjunto de diagramas que normalmente se usan en pequeños subconjuntos para poder representar las cinco vistas principales de la arquitectura de un sistema. En la siguiente sección se muestran los diferentes tipos de diagramas con los que se cuenta en UML

6.7. Diagramas UML

En esta sección se presentan los diagramas utilizados en UML.

6.7.1. Diagramas de casos de uso

Los diagramas de casos de uso ilustran la funcionalidad proporcionada por una unidad del sistema. Los diagramas de casos de uso describen las relaciones y las dependencias entre un grupo de casos de uso y los actores participantes en el proceso.

Es importante resaltar que los diagramas de casos de uso no están pensados para representar el diseño y no puede describir los elementos internos de un sistema. Los diagramas de casos de uso sirven para facilitar la comunicación con los futuros usuarios del sistema, y con el cliente, y resultan especialmente útiles para determinar las características necesarias que tendrá el sistema. En otras palabras, los diagramas de casos de uso describen qué es lo que debe hacer el sistema, pero no cómo.

Actor

Un actor es una entidad externa (de fuera del sistema) que interacciona con el sistema participando (y normalmente iniciando) en un caso de uso. Los actores pueden ser gente real (por ejemplo, usuarios del sistema), otros sistemas o eventos.

Los actores no representan a personas físicas o a sistemas, sino su rol. Esto significa que cuando una persona interactúa con el sistema de diferentes maneras (asumiendo diferentes papeles), estará representado por varios actores. Por ejemplo, una persona que proporciona servicios de atención telefónica a clientes y realiza pedidos para los clientes estaría representada por un actor "equipo de soporte" y por otro actor "representante de ventas".

Caso de uso

Un caso de uso describe, desde el punto de vista de los actores, un grupo de actividades de un sistema que produce un resultado concreto y tangible.

Los casos de uso son descriptores de las interacciones típicas entre los usuarios de un sistema y ese mismo sistema. Representan el interfaz externo del sistema y especifican qué requisitos de funcionamiento debe tener éste (únicamente el qué, nunca el cómo).

Cuando se trabaja con casos de uso, es importante tener presentes algunas sencillas reglas:

- Cada caso de uso está relacionado como mínimo con un actor.
- Cada caso de uso es un iniciador (es decir, un actor).
- Cada caso de uso lleva a un resultado relevante (un resultado con "valor intrínseco").

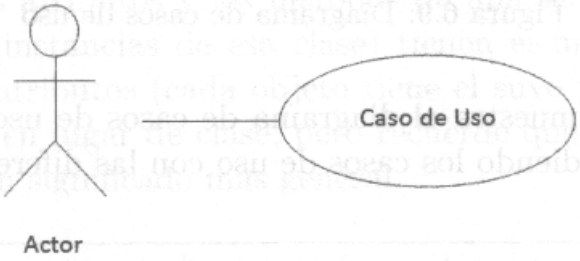


Figura 6.8: Actor y caso de uso

En la figura 6.9 se puede ver un diagrama de casos de uso de un cajero automático con dos actores: un cliente y un empleado de banco. Los casos de uso pueden tener relaciones con otros casos de uso. Los tres tipos de relaciones más comunes entre casos de uso son:

- «include» que especifica una situación en la que un caso de uso tiene lugar dentro de otro caso de uso.
- «extends» que especifica que en ciertas situaciones, o en algún punto (llamado punto de extensión) un caso de uso será extendido por otro.
- Generalización que especifica que un caso de uso hereda las características del "super" caso de uso, y puede volver a especificar algunas o todas ellas de una forma muy similar a las herencias entre clases.

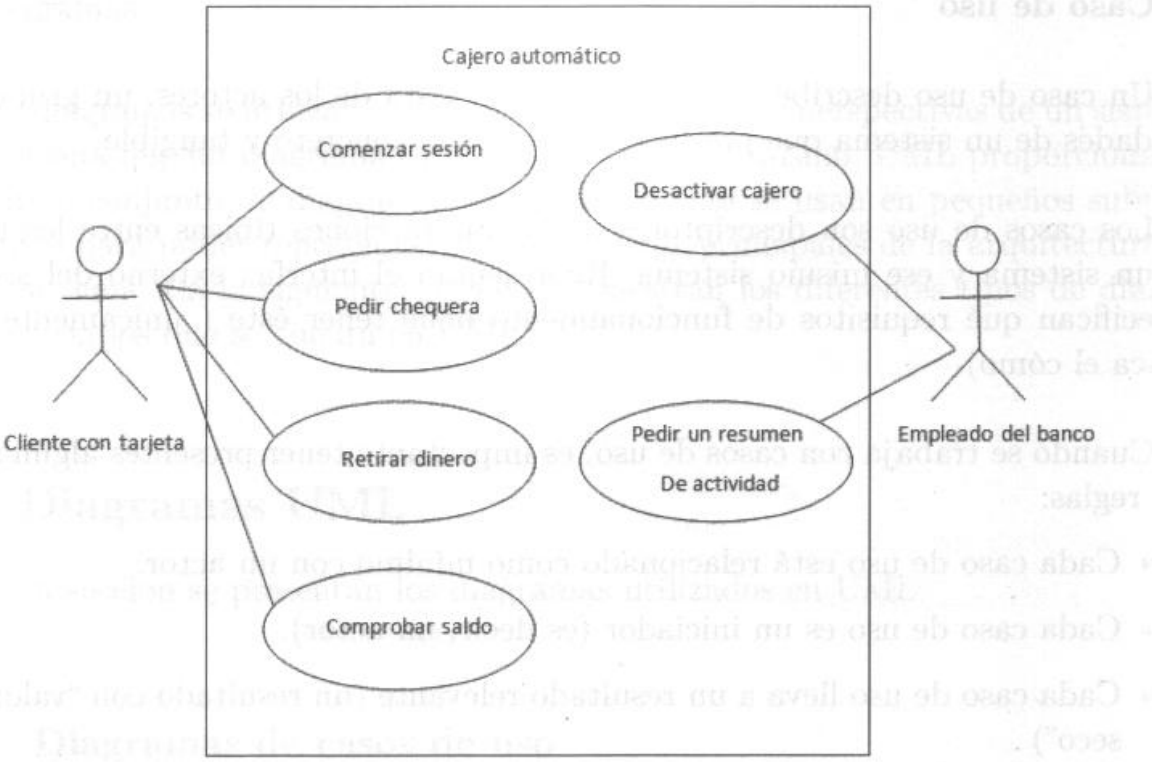


Figura 6.9: Diagrama de casos de uso

En la figura 6.10 se muestra el diagrama de casos de uso del cajero automático presentado en 6.9 extendiendo los casos de uso con las diferentes relaciones.

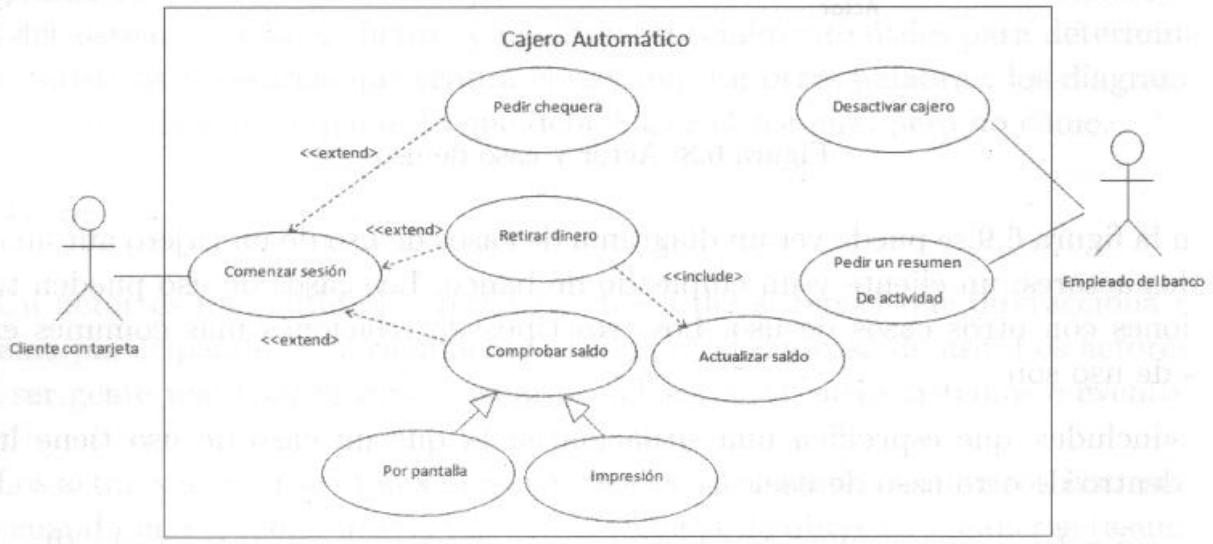


Figura 6.10: Diagrama de casos de uso

Descripción de casos de uso

Las descripciones de casos de uso son reseñas textuales del caso de uso. Normalmente tienen el formato de una nota o un documento relacionado de alguna manera con el caso de uso, y explica los procesos o actividades que tienen lugar en el caso de uso.

6.7.2. Diagrama de clases

Los diagramas de clases muestran las diferentes clases que componen un sistema y cómo se relacionan unas con otras. Se dice que los diagramas de clases son diagramas **estáticos** porque muestran las clases, junto con sus métodos y atributos, así como las relaciones estáticas entre ellas: qué clases **conocen** a qué otras clases o qué clases **son parte** de otras clases, pero no muestran los métodos mediante los que se invocan entre ellas.

Clase

Una clase define los atributos y los métodos de una serie de objetos. Todos los objetos de esta clase (instancias de esa clase) tienen el mismo comportamiento y el mismo conjunto de atributos (cada objeto tiene el suyo propio). En ocasiones se utiliza el término tipo en lugar de clase, pero recuerde que no son lo mismo, y que el término tipo tiene un significado más general.

En las clases están representadas por rectángulos, con el nombre de la clase, y también pueden mostrar atributos y operaciones de la clase en otros dos «compartimentos» dentro del rectángulo.

La clase esta formada por atributos y operaciones o métodos

- Atributos

  En UML, los atributos se muestran al menos con su nombre, y también pueden mostrar su tipo, valor inicial y otras propiedades. Los atributos aparecen calificados en el diagrama dependiendo de su acceso como:

  - + Indica atributos públicos
  - # Indica atributos protegidos
  - - Indica atributos privados

- Operaciones

  Las operaciones (métodos) también se muestran al menos con su nombre, y pueden mostrar sus parámetros y valores de retorno. Las operaciones, al igual que los atributos, aparecen calificados en el diagrama dependiendo de su acceso como::

  - + Indica operaciones públicas
  - # Indica operaciones protegidas
  - - Indica operaciones privadas

- Plantillas

  Las clases pueden tener plantillas o *templates*, un valor usado para una clase no especificada o un tipo. El tipo de plantilla se especifica cuando se inicia una clase (es decir cuando se crea un objeto).

Asociaciones de clases

Las clases se puede relacionar, (estar asociadas), con otras de las siguientes posibles formas:

- Generalización
- Asociación
- Realización
- Agregación
- Composición

Los diagramas de clases pueden contener más componentes aparte de clases. Estos pueden ser:

- Interfaces

  Las interfaces son clases abstractas, esto es, instancias que no pueden ser creadas directamente a partir de ellas. Pueden contener operaciones, pero no atributos. Las clases pueden heredarse de las interfaces pudiendo así realizarse instancias a partir de estos diagramas.

CULTURAL SAN TERESA BIBLIOTECA UNED-AVILA

- Tipo de datos

  Los tipos de datos son primitivas incluidas en algunos lenguajes de programación. Algunos ejemplos son: bool y float. No pueden tener relación con clases, pero las clases sí pueden relacionarse con ellos.

- Enumeraciones

  Las enumeraciones son simples listas de valores. Un ejemplo típico de esto sería una enumeración de los días de la semana. Al igual que los tipos de datos, no pueden relacionarse con las clases, pero las clases sí pueden hacerlo con ellos.

- Paquetes

  Los paquetes, en lenguajes de programación, representan un espacio de nombres en un diagrama se emplean para representar partes del sistema que contienen más de una clase, incluso cientos de ellas.

En la figura 6.11 se presenta un diagrama de clases.

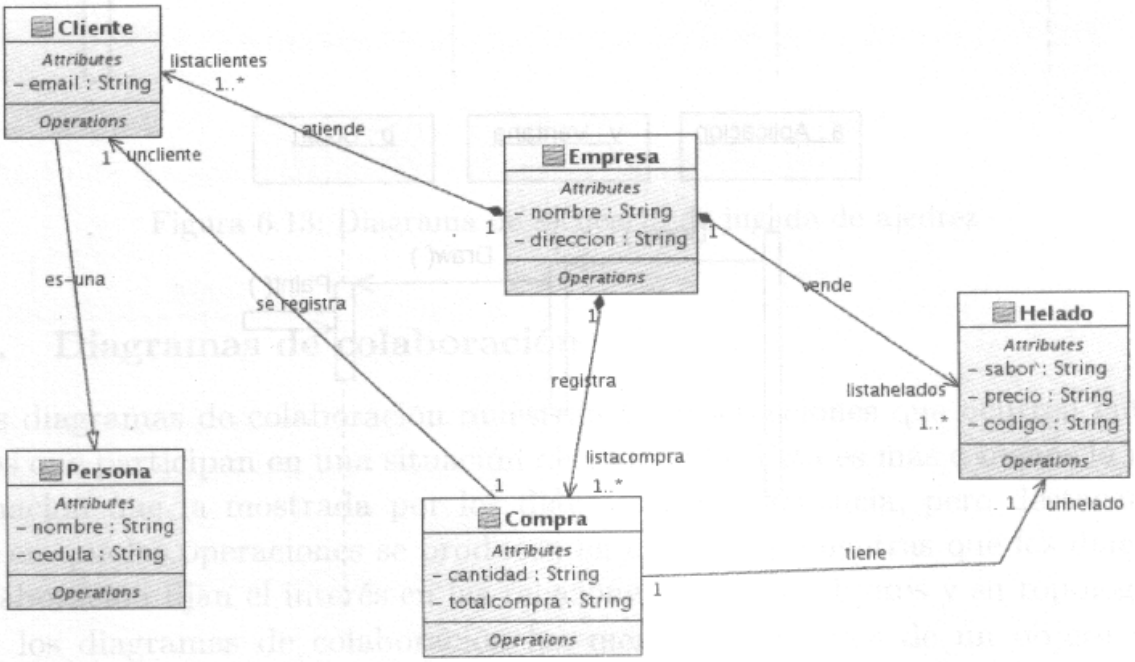


Figura 6.11: Diagrama de clases

6.7.3. Diagramas de secuencia

Los diagramas de secuencia muestran el flujo de mensajes (es decir la forma en que se invocan) entre objetos para un determinando caso de uso. Los diagramas de secuencia ponen especial énfasis en el orden y el momento en que se envían los mensajes a los objetos. El propio diagrama explica la secuencia de llamadas que se producen entre los objetos que intervienen. Pueden tener mayor o menor detalle y representar diferentes llamadas a diferentes objetos.

Los diagramas de secuencia tienen dos dimensiones: la vertical muestra la secuencia de llamadas ordenadas en el tiempo en el que ocurren. La dimensión horizontal muestra las diferentes instancias de objetos a las que son enviadas las llamadas.

En la figura 6.12 puede verse una secuencia de invocaciones para el dibujo de el botón de una ventana de una aplicación.

En los diagramas de secuencia, los objetos están representados por rectángulos con el nombre del objeto dentro y por líneas intermitentes verticales, con el nombre del objeto en la parte más alta. El eje de tiempo también es vertical, incrementándose hacia abajo, de forma que los mensajes son enviados de un objeto a otro en forma de flechas con los nombres de la operación y los parámetros. Es conveniente dibujar una flecha de retorno de la llamada.

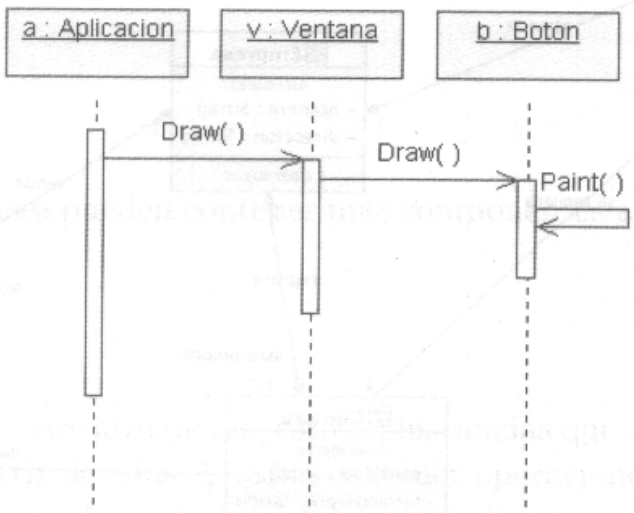


Figura 6.12: Diagrama de secuencia simple de dibujo de aplicación

Los mensajes pueden ser o bien síncronos, el tipo normal de llamada del mensaje

donde se pasa el control a objeto llamado hasta que el método finalize, o asíncronos donde se devuelve el control directamente al objeto que realiza la llamada. Los mensajes síncronos tienen una caja vertical en un lateral del objeto invocante que muestra el flujo del control del programa.

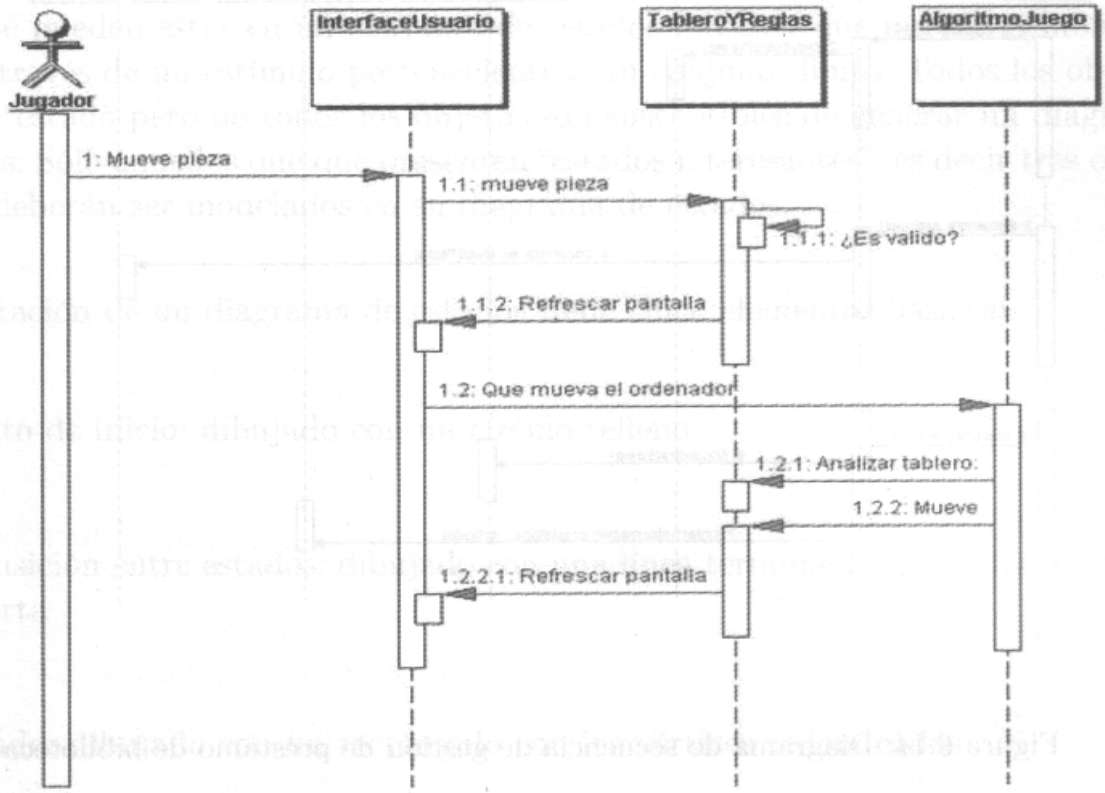


Figura 6.13: Diagrama de secuencia de jugada de ajedrez

6.7.4. Diagramas de colaboración

Los diagramas de colaboración muestran las interacciones que ocurren entre los objetos que participan en una situación determinada. Esta es más o menos la misma información que la mostrada por los diagramas de secuencia, pero destacando la forma en que las operaciones se producen en el tiempo, mientras que los diagramas de colaboración fijan el interés en las relaciones entre los objetos y su topología.

En los diagramas de colaboración los mensajes enviados de un objeto a otro se representan mediante flechas, mostrando el nombre del mensaje, los parámetros y la secuencia del mensaje. Los diagramas de colaboración están indicados para mostrar una situación o flujo programa específicos y son unos de los mejores tipos de diagramas para demostrar o explicar rápidamente un proceso dentro de la lógica del programa.

En las figuras 6.14 y 6.15 pueden verse los diagrams de secuencia y el respectivo de colaboración del caso de uso de alta de un préstamo en una biblioteca.

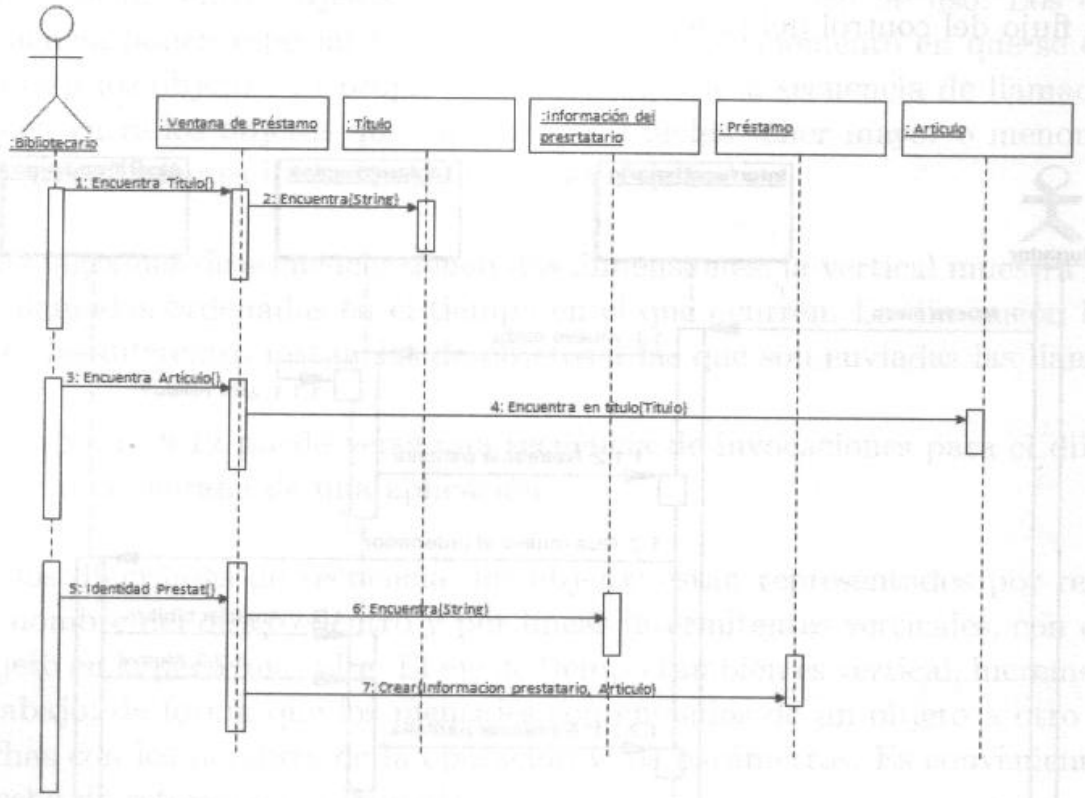


Figura 6.14: Diagrama de secuencia de gestión de préstamo de biblioteca

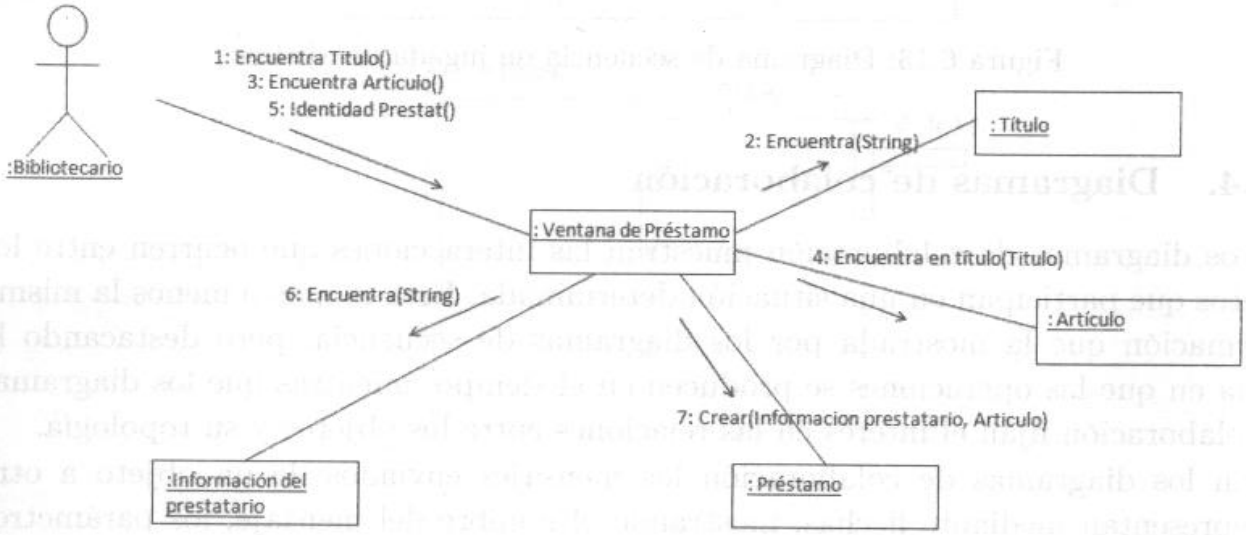


Figura 6.15: Diagrama de colaboración de gestión de préstamo de biblioteca

6.7.5. Diagrama de estado

Los diagramas de estado muestran los diferentes estados de un objeto durante su vida, y los estímulos que provocan los cambios de estado en un objeto.

Los diagramas de estado ven a los objetos como máquinas de estado o autómatas finitos que pueden estar en un conjunto de estados finitos y que pueden cambiar su estado a través de un estímulo perteneciente a un conjunto finito. Todos los objetos tienen un estado pero no todos los objetos son susceptibles de generar un diagrama de estados. Sólo aquellos que que presenten "estados interesantes", es decir tres o más estados, deberán ser modelados en su diagrama de estados.

La notación de un diagrama de estados tiene cinco elementos básicos:

- Punto de inicio: dibujado con un círculo relleno
- Transición entre estados: dibujado con una línea terminada con punta de flecha abierta
- Estado: dibujado con un rectángulo con los vértices redondeados
- Punto de decisión: dibujado con un círculo no relleno
- Puntos de terminación: dibujados con un círculo con otro relleno en su interior

Para dibujar un diagrama de estados se debe comenzar por el punto de inicio y una línea de transición que lleve hasta el primer estado del objeto. Después se dibujan cada uno de los estados distribuidos por el diagrama y se conectan con las línea de transición de estados.

Hay dos tipos especiales de estados: inicio y fin. Son especiales en el sentido de que no hay ningún evento que pueda devolver a un objeto a su estado de inicio, y de la misma forma no hay ningún evento que pueda sacar a un objeto de su estado de fin.

En las figura 6.16 puede verse el diagrama de transición de estados de un teléfono.

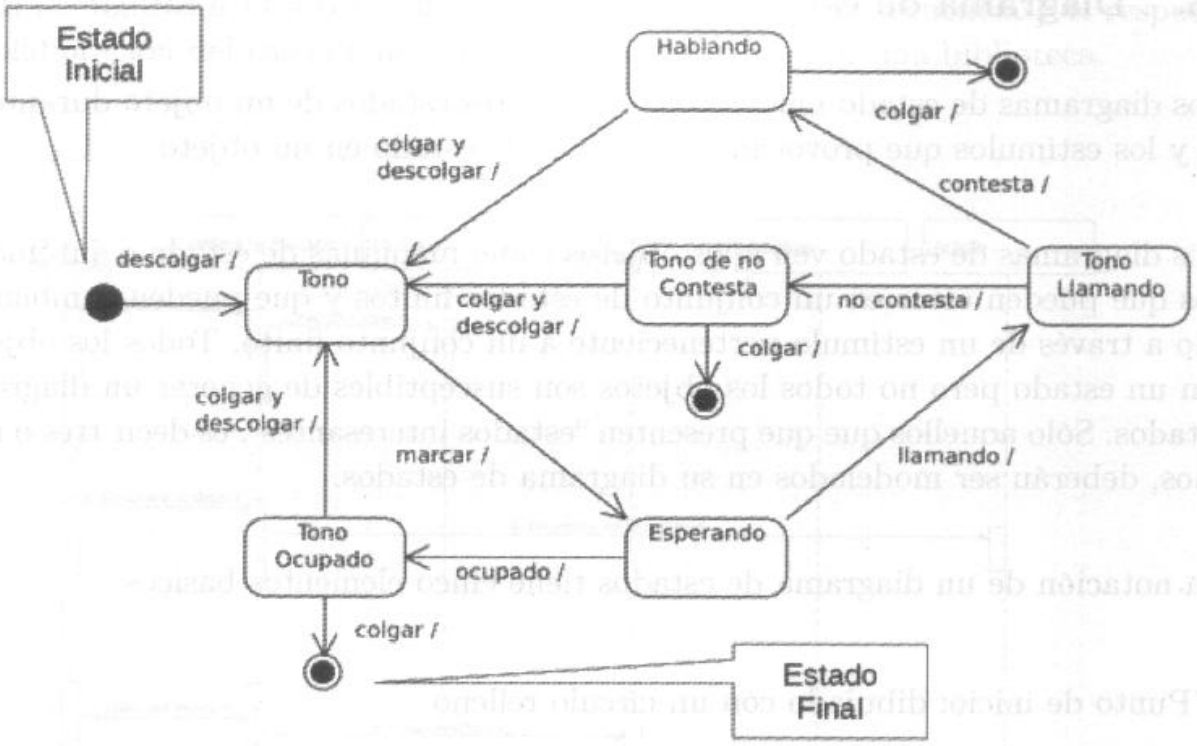


Figura 6.16: Diagrama de transición de estados de un teléfono

En las figura 6.17 puede verse el diagrama de transición de estados del acceso con tarjeta con dos puntos de bifurcación.

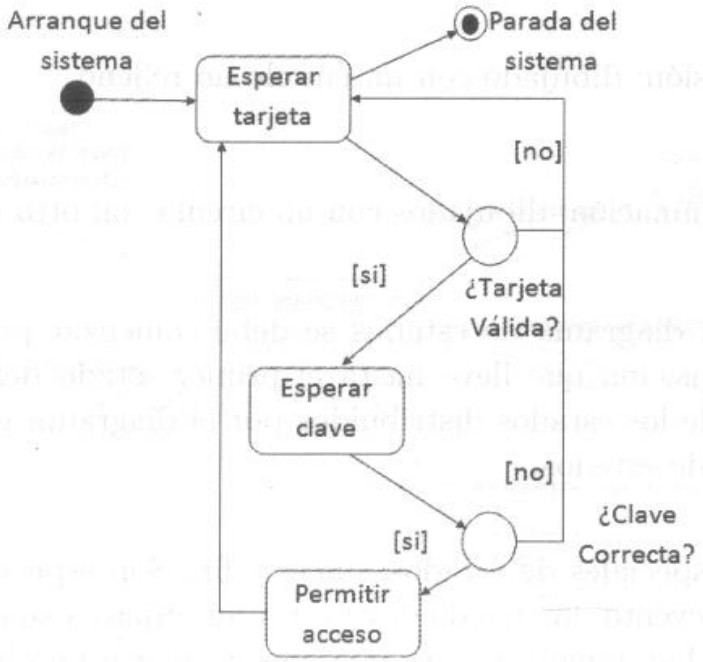


Figura 6.17: Diagrama de transición de estados de sistema de accesos

6.7.6. Diagrama de actividad

Los diagramas de actividad describen la secuencia de las actividades en un sistema. Los diagramas de actividad son una forma especial de los diagramas de estado, que únicamente contienen actividades. Los diagramas de actividad son similares a los diagramas de flujo procesales, con la diferencia de que todas las actividades están claramente unidas a objetos. Los diagramas de actividad siempre están asociados a una clase, a una operación o a un caso de uso.

Los diagramas de actividad soportan actividades tanto secuenciales como paralelas. La ejecución paralela se representa por medio de iconos de fork/espera, y en el caso de las actividades paralelas, no importa en qué orden sean invocadas (pueden ser ejecutadas simultáneamente o una detrás de otra).

Actividad

Una actividad es un único paso de un proceso. Una actividad es un estado del sistema que tiene actividad interna y, al menos, una transición saliente. Las actividades también pueden tener más de una transición saliente, si tienen diferentes condiciones.

Las actividades pueden formar jerarquías, lo que significa que una actividad puede estar formada de varias actividades "de detalle", en cuyo caso las transiciones entrantes y salientes deberían coincidir con las del diagrama de detalle.

Los diagramas de actividad son indicados para modelar procesos de alto nivel. Son indicados para modelizar el proceso de negocio de una compañía. Su complejidad técnica es menor y son más fáciles de entender por quien no tiene muchos conocimientos técnicos de computación.

Los elementos para definir un diagrama de actividades son similares a los usados en los diagramas de secuencia. Círculo relleno de inicio, rectángulos con los bordes redondeados para determinar las actividades, flechas conectoras para unir las actividades, puntos de decisión con círculo huecos. Cómo elemento distinto aparecen las líneas divisorias o calles para establecer las responsabilidades entre los distintos objetos del sistema.

En la figura 6.18 se muestra un diagrama de actividades para la gestión de pedidos en una empresa

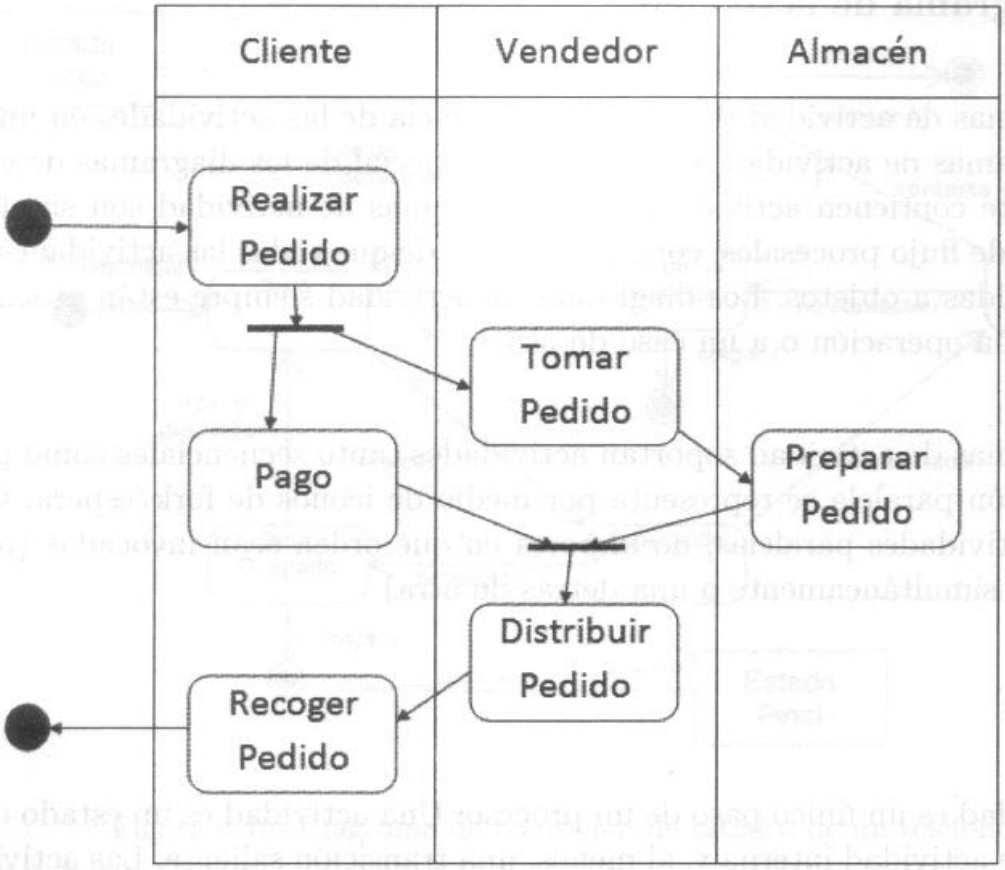


Figura 6.18: Diagrama de secuencia de actividad

6.7.7. Diagramas de componentes

Los diagramas de componentes muestran los componentes del software (ya sea las tecnologías que lo forman como Kparts, componentes CORBA, Java Beans o simplemente secciones del sistema claramente distintas) y los artilugios de que está compuesto como los archivos de código fuente, las librerías o las tablas de una base de datos.

Los componentes pueden tener interfaces (es decir clases abstractas con operaciones) que permiten asociaciones entre componentes.

Los Diagramas de Componentes ilustran las piezas del software, controladores embebidos, etc. que conformarán un sistema. Un diagrama de Componentes tiene un nivel más alto de abstracción que un diagrama de clase. Usualmente un componente se implementa por una o más clases (u objetos) en tiempo de ejecución. Estos son bloques de construcción, como eventualmente un componente puede comprender una gran porción de un sistema.

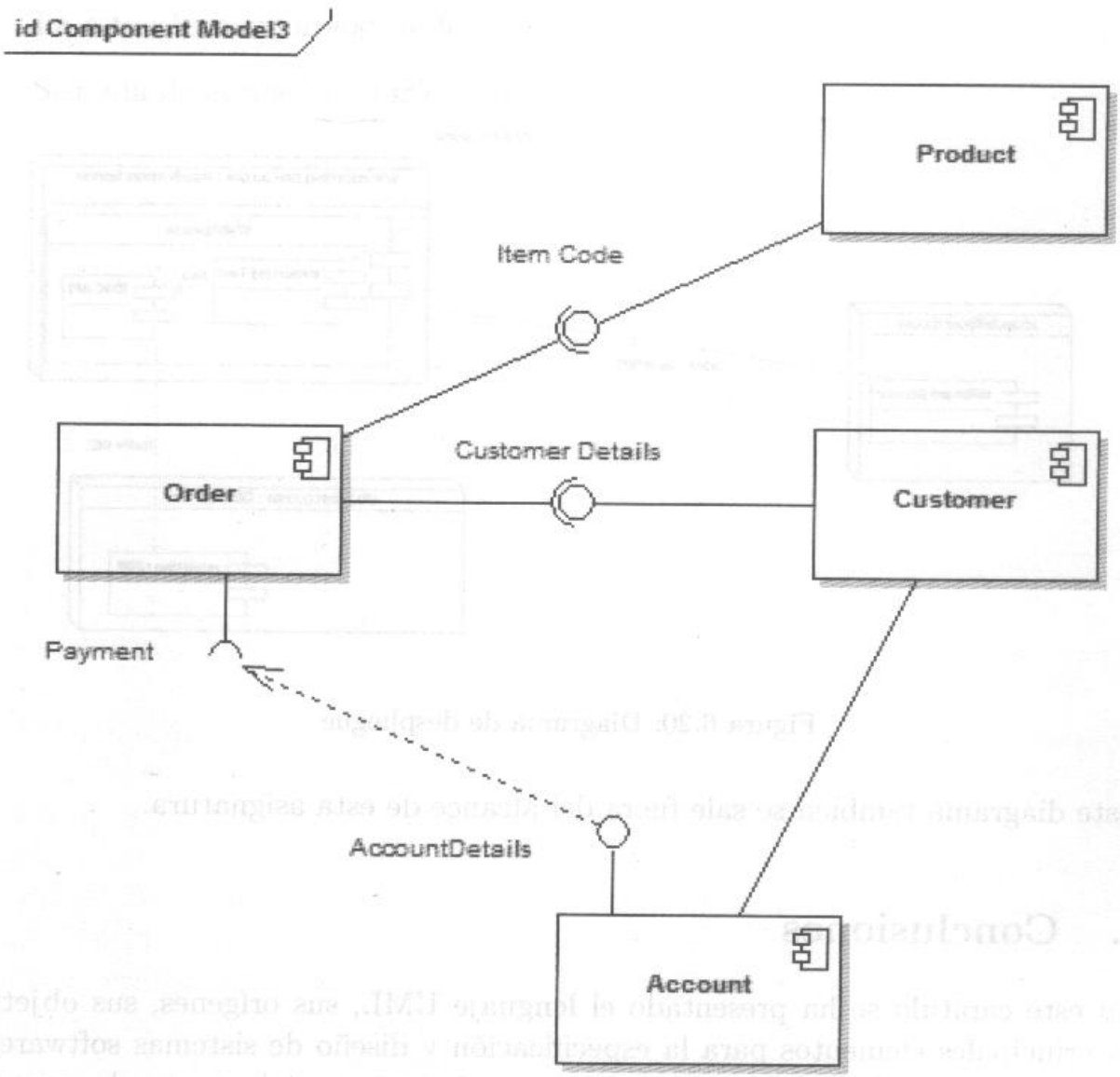


Figura 6.19: Diagrama de componentes

Este diagrama queda fuera del alcance del programa de la asignatura.

6.7.8. Diagrama de despliegue

El diagrama de despliegue muestra cómo el sistema se asentará físicamente en el entorno hardware que lo acompaña. Su propósito es mostrar dónde los componentes del sistema se ejecutarán y cómo se comunicarán entre ellos. Este será un diagram muy utilizado por el las personas que produzcan el sistema. La notación que utiliza es la misma que el diagrama de componentes. Se puede ver en la figura 6.20

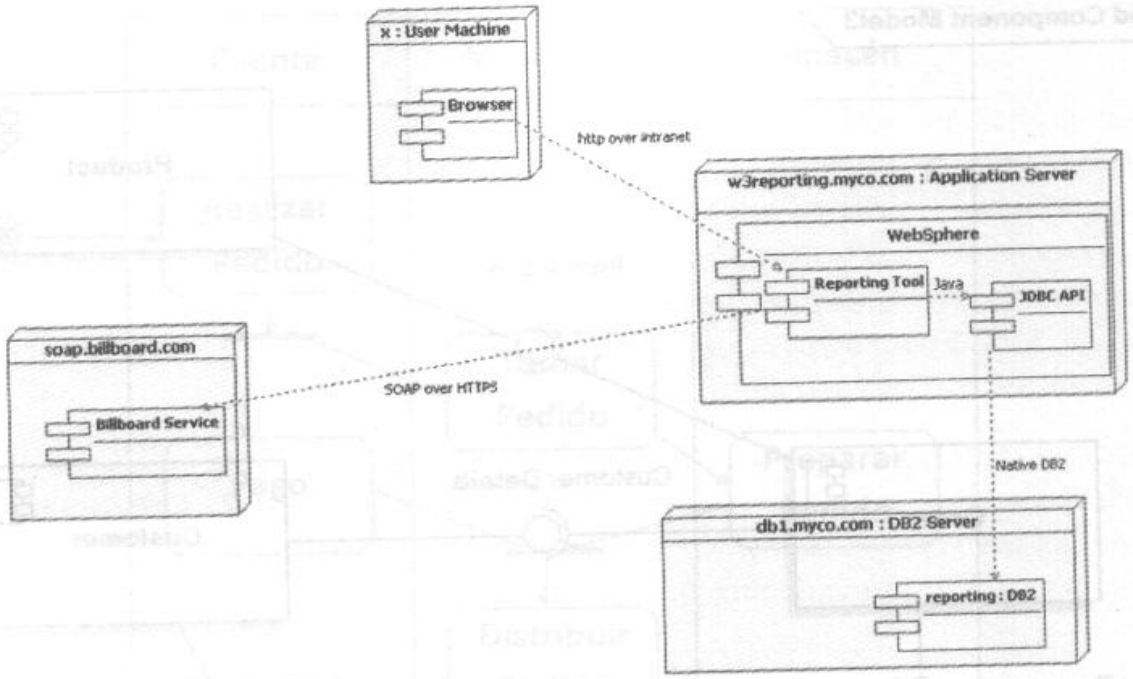


Figura 6.20: Diagrama de despliegue

Este diagrama también se sale fuera del alcance de esta asignatura.

6.8. Conclusiones

En este capítulo se ha presentado el lenguaje UML, sus orígenes, sus objetivos y sus principales elementos para la especificación y diseño de sistemas software. El recorrido realizado facilita al lector unos conocimientos mínimos que le permitan aplicar esta notación para las tareas de especificación y diseño que se han abordado en los capítulos anteriores.

6.9. Ejercicios propuestos

Realícense todos los diagramas UML necesarios para la completa especificación y diseño de los siguientes sistemas software:

1. El programa "hola mundo"
2. La práctica del calendario de la asignatura Fundamentos de Programación
3. Un sistema de gestión de clientes, y proveedores de una empresa
4. Videojuego de las minas del capítulo anterior

5. La estación meteorológica del capítulo anterior
6. Sistema de acceso con tarjeta, teclado y pantalla

Capítulo 7

La Codificación del Software

7.1. Introducción

La fase codificación constituye el núcleo central de cualquiera de los modelos de desarrollo de software: ciclo de vida, uso de prototipos, modelo en espiral. Una vez especificado y diseñado, de forma parcial o total, un determinado programa llega el momento de construir el código, es decir de programar. Hasta el 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.

7.2. Objetivos

Codificar es el proceso de expresar un programa de computadora en un lenguaje de programación. Sin embargo construir un programa no es sólo codificar, también implica las tareas de verificación, depuración y pruebas de ese programa. En este capítulo nos centraremos en estudiar únicamente la tarea de codificación. Los objetivos específicos son los siguientes:

- Analizar los lenguajes de programación más utilizados y agruparlos por sus características comunes.

- Estudiar los recursos de programación que nos ofrecen los leguajes y las técnicas de implementación a las que dan lugar.
- Repasar algunos de los criterios habitualmente utilizados para elegir un determinado lenguaje de programación.

7.3. Los lenguajes de programación

Existen diferentes formas de comunicación entre los humanos y las computadoras que de una forma u otra nos permiten modificar el comportamiento de una computadora y, por lo tanto, de programarla. Según la guía Swebok se pueden diferenciar:

- Lenguajes de configuración, en los que el ingeniero de software elige un conjunto de parámetros entre una serie de opciones predefinidas. Los ficheros de configuración de Unix o Windows son un ejemplo.
- Lenguajes de caja de herramientas o "toolkits" que se utilizan para construir aplicaciones a partir de la combinación de un conjunto de piezas preprogramadas, llamados "toolkit". También se les llama lenguajes de programación de aplicación.
- Lenguajes "script" o también llamados macros o ficheros por lotes ("batch").
- Lenguajes de programación propiamente dichos, que nos permiten construir una aplicación completa y son muy flexibles. Nos centraremos en este apartado en ellos.

Aunque la historia de los computadores es relativamente reciente, son cientos y cientos los lenguajes de programación 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 de ellos se utiliza o has sido utilizado en el desarrollo de software a escala industrial. El estudio detallado de tan sólo los dos o tres más representativos queda fueras 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 de programación en 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 leguajes de las nuevas generaciones coexisten con los de las generaciones anteriores hasta que se imponen por completo los de mejores prestaciones.

7.3.1. Primera generación de lenguajes

Los primeros computadores eran máquinas de válvulas tremendamente costosas y los programas que podían ejecutar eran muy pequeños. La programación se ha-

cía 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 esta que estaba reservada a un número reducido de personas.

Durante la década de los 50 el número de computadores disponible 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 esté 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, solo se programaran 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.

7.3.2. 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 diversas versiones: FORTRAN-IV, FORTRAN-66, FORTRAN-77, FORTRAN-9O 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 rama de aplicaciones de todo tipo que incluyen la realización de sistemas de gestión de bases dc datos, sistemas en tiempo real, etc.

- **COBOL** (COmmon Business-Oriented Lenguage):

  Es el lenguaje con el que están escritos casi todos los sistemas para el procesamiento de información (contabilidad, nominas, 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: ALGOL6O, 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.

7.3.3. 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 las 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 coma 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 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 coma 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 packages 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 computo (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 las 80 las estaciones de trabajo con pantallas gráficas y procesadores más potentes permitieron 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.

- **JAVA**

  El lenguaje de programación Java fue originalmente desarrollado en 1995 por Sun Microsystems (la cual fue adquirida posteriormente por la compañía Oracle). Su sintaxis deriva mucho de C y C++, pero tiene menos facilidades de bajo nivel que cualquiera de ellos. Las aplicaciones de Java son generalmente compiladas a un pseudocódigo que puede ejecutarse en cualquier máquina virtual Java (JVM) sin importar la arquitectura de la computadora subyacente. Es un lenguaje de programación de propósito general, concurrente, orientado a objetos y basado en clases que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido en inglés como WORA, o "write once, run anywhere"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra. Java es, a partir de 2012, uno de los lenguajes de programación más populares en uso, particularmente para aplicaciones de cliente-servidor de web.

- **EIFFEL**

  Este lenguaje de finales de las 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 105 lenguajes funcionales. Las primeras versiones aparecieron junto a los lenguajes de 2ª 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 dc hechos y reglas crece, el proceso de inferencia puede resultar tremendamente costoso y lento.

  Existen en el mercado una gran variedad de lenguajes de marcas. Estos lenguajes no son propiamente lenguajes de programación, aunque suele confundirse con ellos. Estos lenguajes permiten codificar un documento con una serie de etiquetas o marcas que aportan información adicional o estructural al documento. El ejemplo más conocido es el HTML, que permite crear páginas web a través de documentos con marcas que son interpretadas por el navegador. Existen muchos otros tipos, especialmente para la edición de textos como el TEX, pero también para aplicaciones gráficas, matemáticas, de música, etc.

7.3.4. 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 casos 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 prototipos que posteriormente serán mejorados con lenguajes de la tercera generación.

Desde hace más de veinte años han venido surgiendo lenguajes o herramientas. La amplia difusión de los computadores personales de altas prestaciones (potencia de cálculo, pantalla gráfica, etc), así como el uso generalizado de Internet y la consiguiente necesidad de aplicaciones "web", han sido determinantes para su desarrollo más amplio. Sería tremendamente 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 las 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, también 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++ ó 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.

7.4. 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 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++, Java, 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 que 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 cuento 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 cuente 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.

- EXPERIENCIA PREVIA

Aunque se supone que el equipo de programadores 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 a varios lenguajes.

7.5. Aspectos metodológicos

En esta sección 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.

7.5.1. 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 tra-

bajo 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' espacio
  - 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.

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 (+=, -=, *=, \=, =,<<=, >>=, &=), ya que pueden resultar confusos.

7.5.2. 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.
- 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 organiza 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 sean correctos y que el programa no tenga 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

1. El resultado correcto, si los datos son válidos, o bien
2. 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.

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.

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, sobrepresión, 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 manejo de excepciones.

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.

7.5.3. 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 consultarlo 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).
- Evitar las operaciones en coma flotante y realizarlas preferiblemente en coma fija.
- Evitar las conversiones innecesarias de tipos de datos.

7.5.4. 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 deriva 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 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.

7.6. Conclusiones

La programación o construcción del software es la etapa en la que, después de análisis, especificación y diseño, elaboramos el producto. La elección del lenguaje de programación más adecuado no es sencilla, depende de múltiples factores como el tipo de aplicación, el bagaje de la empresa u organización que lo construye, la plataforma hardware elegida, etc. En este capítulo se han dado algunos consejos para la correcta elección del lenguaje, y se ha dado un breve repaso a los distintos lenguajes de programación. Existen cientos de lenguajes disponibles en el mercado. La calidad final del producto una vez terminado depende mucho de haber seguido una correcta metodología durante la programación. En las organizaciones más desarrolladas existen estrictas normas sobre el uso de lenguajes y compiladores. Aquí se han revisado algunos de los aspectos metodológicos más importantes que sirven de base para su elaboración.

7.7. Ejercicios propuestos

1. Elabórese un conjunto de reglas para la creación de los nombre de los identificadores de un programa. Elíjase el lenguaje de programación que mejor conozca. ¿Cree que obligar a cumplir dichas reglas es una pérdida de tiempo? Razone su respuesta.

2. Escríbase una rutina en C que calcule términos de la sucesión de números enteros conocida como la serie de Fibonaci sin utilizar la sentencia GOTO. Vuelva a escribirla utilizando la sentencia GOTO. ¿Qué ventajas o inconvenientes ve entre ambos métodos?

3. Piénsese en los dos lenguajes de programación que mejor conozca. Elabórese una lista de ventajas e inconvenientes que ve en cada uno de ellos para la programación de una aplicación que gestione una gasolinera.

4. Búsquese información sobre la popularidad de los diferentes lenguajes de programación e intente elaborar una lista de los 20 más utilizados.

5. Consúltese la guía SWEBOK y descubra, según ella, cuáles son los principios fundamentales de la construcción del software.

Capítulo 8

Pruebas de Software

8.1. Introducción

A lo largo del proceso de elaboración del software 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 andes 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 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), donde la gravedad de las consecuencias de un fallo es altísima, el costo de las pruebas puede ser la partida más importante del costo de todo el desarrollo.

Para evitar el caos de una prueba 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.

Las pruebas permiten valorar la calidad de un programa, es decir, descubrir sus errores, sin embargo no deben verse como una red de seguridad. La calidad se incorpora a lo largo de todo el proceso de ingeniería del software y las pruebas deben confirmarnos que se ha logrado un buen programa, a través de la adecuada aplicación de métodos y herramientas.

Con frecuencia, la fase de pruebas requiere más esfuerzo que cualquier otra de la ingeniería del software. Una correcta planificación puede ahorrarnos mucho tiempo dentro del proyecto y evitar que un menor porcentaje de errores pase desapercibido.

8.2. Objetivos

Se analiza a lo largo de este capítulo los distintos tipos de pruebas que se deben o pueden realizar a un software para garantizar de forma razonable su correcto funcionamiento. Se revisan además las técnicas más utilizadas en función del tipo de lenguaje de programación.

8.3. Tipos de pruebas

Las pruebas de software tienen un doble objetivo: la verificación y la validación. La verificación persigue comprobar que se ha construido el producto correctamente. La validación comprueba que el producto se ha construido de acuerdo a los requerimientos del cliente.

Los procesos de verificación y validación llevan a realizar pruebas a diferentes niveles, como se muestra en la figura 8.1, en los que se revisa la calidad de cada etapa del proceso de elaboración del software:

- Pruebas de unidades
- Pruebas de integración
- Pruebas de validación
- Pruebas de sistema

Figura 8.1: Tipos de pruebas

Las pruebas de unidad se centran en comprobar la correcta codificación de las partes más pequeñas con entidad propia en las que podemos descomponer un programa: unidades o módulos de software. Estas pruebas comprueban la lógica del procesamiento interno del módulo y su estructura de datos.

Podríamos pensar que si todos los módulos funcionan correctamente, lo cual ya hemos comprobado una vez realizadas las pruebas de todas las unidades, el software que resulta de juntar todos esos módulos deberá igualmente funcionar correctamente. Nada más lejos de la realidad, la variedad de problemas que pueden surgir como resultado de juntar los módulos es enorme. Las pruebas de integración revisan que la estructura de programa que resulta de la unión de los diferentes módulos es consiste con el diseño y está libre de errores.

Una vez ensamblados todos los módulos correctamente y eliminados los errores de interacción entre ellos, comienza la fase de verificación de que software realiza aquello para lo que ha sido especificado. Estamos en las pruebas de validación. Ahora lo relevante es la interacción del usuario con el programa, el cual debe obtener los resultados deseados ante las diferentes peticiones que pueda requerir.

El software una vez terminado y funcionando correctamente, es decir según se ha especificado y diseñado, debe montarse sobre una plataforma hardware y comunicarse tanto con los usuarios como con otros sistemas o equipos, incluso formando parte de un sistema más grande. Las pruebas de sistema revisan por completo el sistema basado en computadora, yendo un paso más allá del proceso de software.

8.4. Pruebas de unidades

Aunque puede 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 al 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 y el 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 8.2, 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 tener las más graves consecuencias.

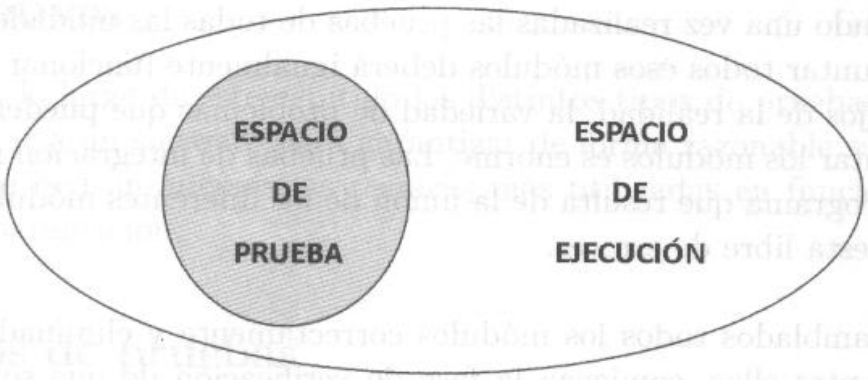


Figura 8.2: Dominio de pruebas

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 otros parámetros de 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.

8.4.1. Pruebas de caja negra

Según muestra la figura 8.3, la estrategia de caja negra (black box) ignora por completo la estructura interna del programa y se basa exclusivamente en la compro-

bación 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.

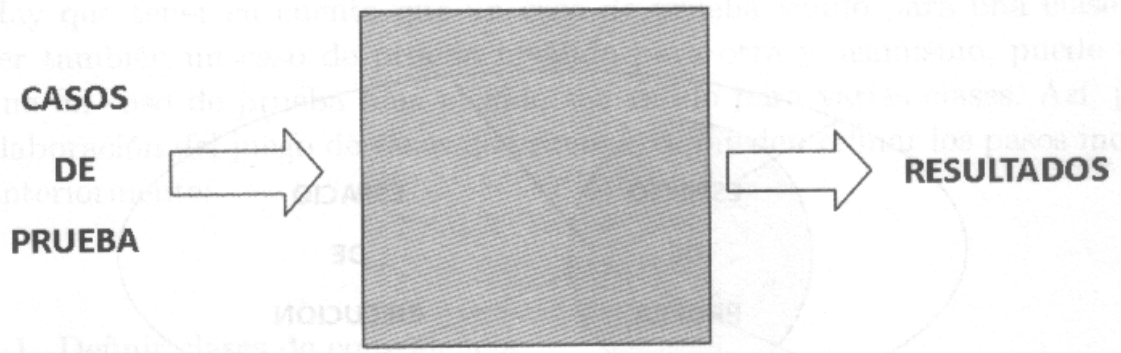


Figura 8.3: Pruebas de caja negra

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 llevarlo 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
- Análisis del valor límite
- Comparación de versiones
- Empleo de la intuición

PARTICIÓN EN CLASES DE EQUIVALENCIA

Según se muestra en la figura 8.4, se trata de dividir el espacio de ejecución del programa en varios subespacios. Cada subespacio o clase de equivalencia 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 de equivalencia y de otra cinco.

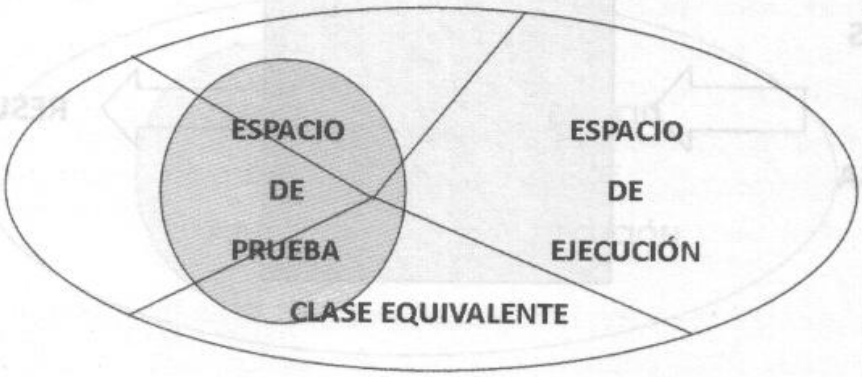


Figura 8.4: Clases de equivalencia

Los pasos que se deben seguir con este método son los siguientes:

1. Determinar las clases de equivalencia apropiadas.
2. Establece 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 el defecto. Según cómo se caractericen las clases de equivalencia, se pueden emplear las siguientes directrices:

A Rango de valores.
Ejemplo: 0 <= edad <120años
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 clases de equivalencia.
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 los valores límite (en inglés boundary analysis), como se muestra en la figura 8.5, 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 válidos de prueba serían:

Límite inferior: -1 0 1
Límite superior: 119 120 121

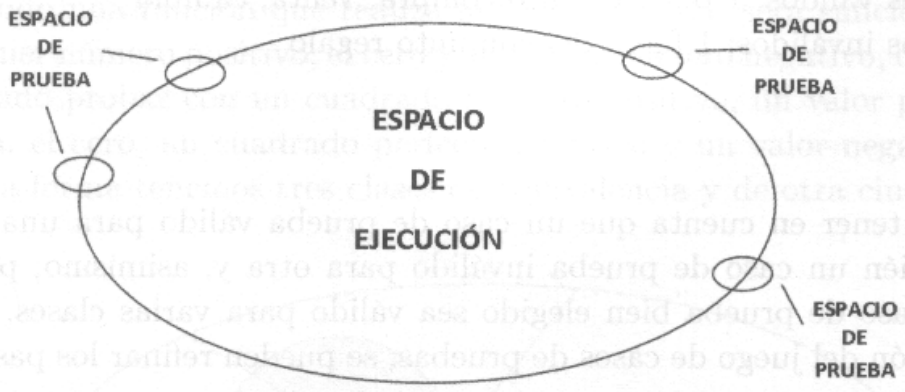


Figura 8.5: Pruebas de valores límite

Los errores en los límites pueden tener consecuencias más graves de lo normal, debido a que se puede 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, comprobando que 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 la pruebas.

Ente 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 funcione correctamente y que cumpla las especificaciones. Sin embargo, no es un criterio infalible puesto que un error en la especificación se trasladará a todas las versiones.

EMPLEO DE 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.

8.4.2. Pruebas de caja transparente

Según se muestra en la figura 8.6, en la estrategia de prueba de caja transparente (existen diversas denominaciones en inglés como white box, clear box, glass box) se conoce y se tiene en cuenta la estructura interna del módulo. En la elaboración de los casos de pruebas 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.

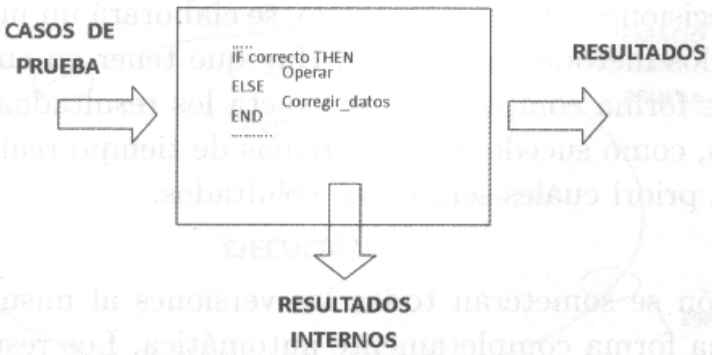


Figura 8.6: Pruebas de caja transparente

En principio puede parecer que las únicas pruebas que realmente son 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 o electrónico, para los que si 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 diferentes.

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 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.

Los métodos más ampliamente utilizados en la elaboración de pruebas de caja transparente son los siguientes:

- Cubrimiento lógico
- Pruebas de bucles
- Empleo de la intuición

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 8.7, denominaremos camino básico a cualquier recorrido que, siguiendo las flechas de las líneas de flujo, nos permita ir desde el punto inicias (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.

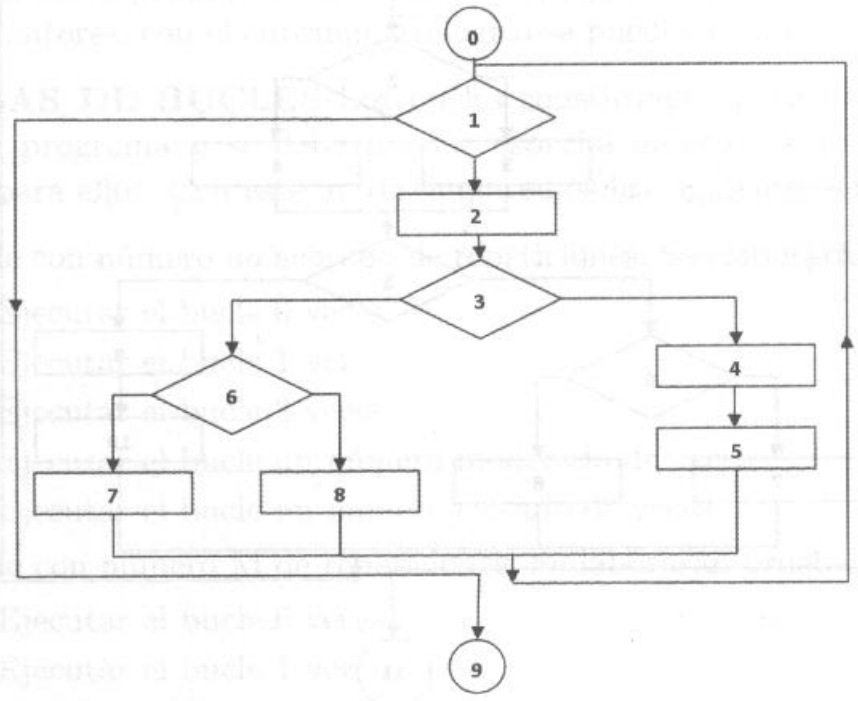


Figura 8.7: Diagrama de flujo con 3 predicados lógicos simples

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

N° máximo de caminos = N° predicados + 1

En el caso de la figura 8.7 tenemos N° 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:

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.

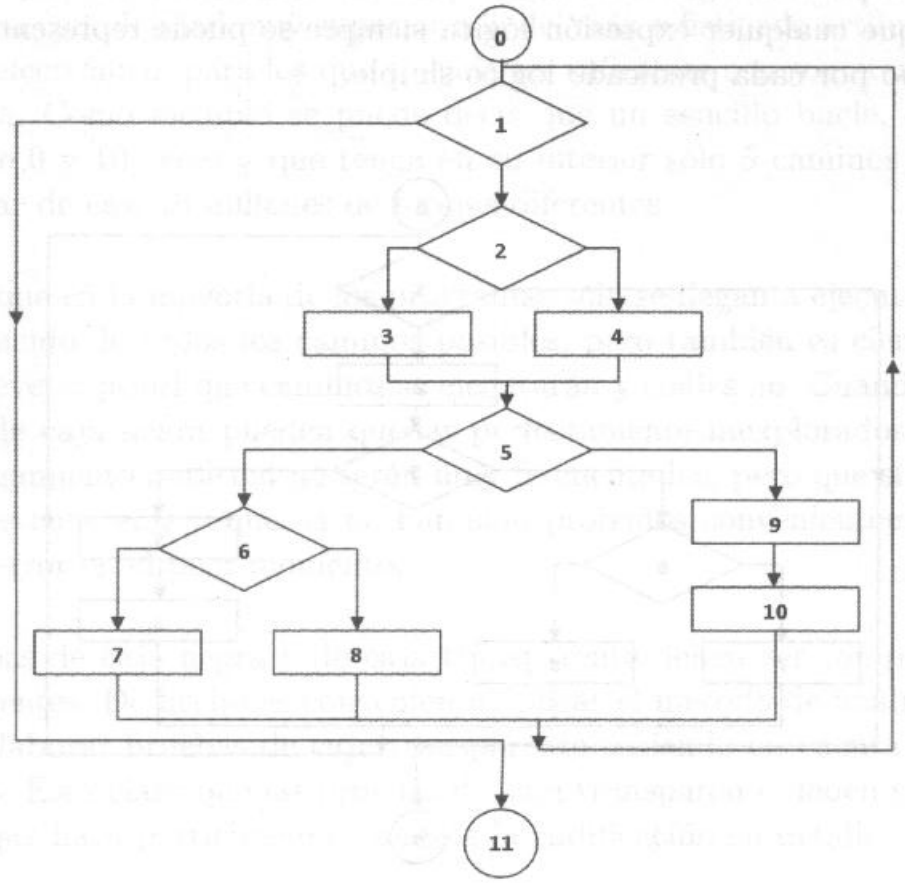


Figura 8.8: Diagrama de flujo con 4 predicados lógicos simples

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 prueba 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 8.8 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 la pruebas de cada módulo deben garantizas el nivel I. Según los distintos autores, con el cubrimiento lógico se pueden quedar sin detectar el 50

- **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 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
  - Bucle con número M de repeticiones. Se laborarán 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 utiliza 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, por ejemplo el índice final de uno es el inicial del siguiente, es empleará un enfoque similar 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 parar ello es necesario conocer muy en detalle la estructura del módulo y tener alguna experiencia previa.

8.4.3. Estimación de errores no detectados

Aunque sea constatable que no aparecen nuevos errores y ha 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 puede 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 EI (por ejemplo EI = 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 EA (por ejemplo EA = 10) 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 ED (por ejemplo ED = 95).

EE = (EA - ED) * (EI / ED) = (100 - 95) * (56 / 95) = 3 errores

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: 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.

8.5. Pruebas de unidades en programación orientada a objetos

En programación orientada a objetos la unidad se convierte en las clases u objetos, que encapsulan tanto datos como operaciones. Dentro de una clase son las operaciones o métodos de esta clase los elementos más pequeños que podemos probar. El problema surge cuando un método es heredado por alguna subclase. El hecho de haber probado dicho método dentro de la superclase no nos garantiza que funcione correctamente para la subclase, ya que el contexto varía sutilmente. A diferencia del software convencional en el que se prueba el algoritmo interno de un determinado procedimiento o función con su interfaz correspondiente, en programación orientada a objetos debemos probar la clase como unidad, en la que las operaciones van modificando los datos encapsulados en ella y por lo tanto el comportamiento de la clase.

Las secuencias de prueba de una clase se diseñan para probar las operaciones más relevantes de ella, y es examinando los valores de los atributos de la clase donde comprobaremos si existen errores. Existen varios métodos para probar una clase:

- Pruebas basadas en fallo, el objetivo es descubrir fallos que durante el diseño se hayan detectado o valorado como probables. Los casos de prueba se crean con el fin de detectar dichos posibles fallos. Obviamente la efectividad dependerá del esfuerzo e interés dedicado durante el diseño en analizar las posibles causas de fallo.

- Pruebas aleatorias, se eligen de forma aleatoria una serie de métodos dentro de la clase para probarlos y se diseñan las secuencias de prueba correspondientes.

- Pruebas de partición, de forma similar a las clases de equivalencia utilizadas en las pruebas de software convencional, se pueden crear una serie de categorías en las entradas y salidas para probar una clase de forma selectiva. Las particiones las podemos hacer con distintos criterios:

- En base la capacidad de las operaciones de cambiar el estado de la clase, es decir a modificar un determinado atributo.
- En base a los atributos que utiliza cada operación. Una categoría la compondrían todas las operaciones que utilizan un determinado atributo.

8.6. Estrategias de Integración

Las unidades o módulo 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:

- Desacuerdosde interfaz
- Interacción indebida entre módulos
- Imprecisiones acumuladas

entre otras. 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 (top-down)
- Integración ascendente (bottom-up)

Estas estrategias pueden emplearse independientemente o en forma combinada.

8.6.1. 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.

8.6.2. Integración descendente

Como se muestra en la figura 8.9, 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.

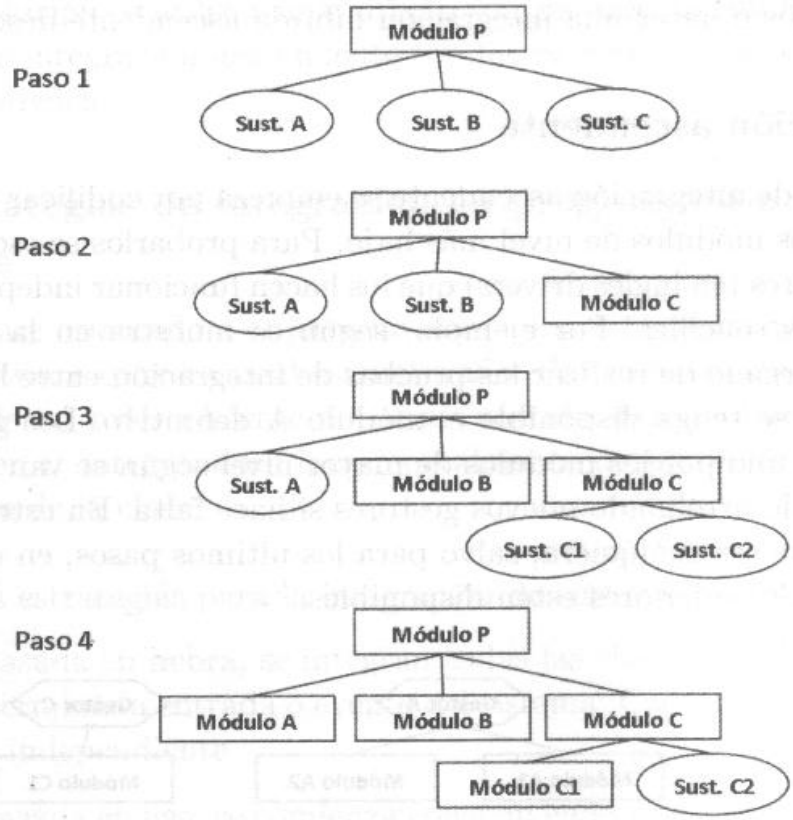


Figura 8.9: Integración descendente.

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 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

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 concluir 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.

8.6.3. 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 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 8.10, 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 A definitivo. 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.

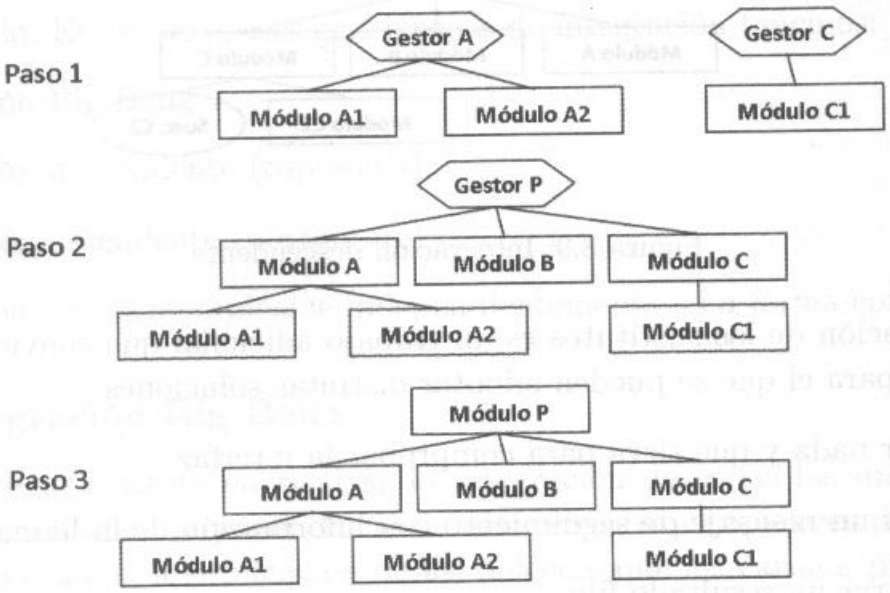


Figura 8.10: Integración ascendente

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.

8.6.4. Estrategias de integración en programación orientada a objetos

En el software orientado a objetos no existe una estructura de control jerárquica, como en el software convencional estructurado. Por lo tanto las estrategias de integración ascendente y descendente no tienen sentido. Las operaciones dentro de una clase no pueden ir integrándose incrementalmente debido a las interacciones entre los componente de la clase.

Existen dos estrategias para la integración de software orientado a objetos:

- Prueba basada en hebra, se integran todas las clases necesarias para responder a una determinada entrada o evento del sistema. Cada hebra se integra y prueba de forma independiente.
- Prueba basada en uso, se comienza construyendo y probando las clases llamadas independientes, es decir las que menos relación tienen con las posibles clases servidor. A continuación se prueban las clases dependientes, que usan clases independientes. De esta forma se siguen construyendo el sistema, añadiendo y probando más clases dependientes.

Los módulos gestores y sustitutos también se utilizan pero de forma diferente a la integración convencional. Por ejemplo se puede usar un módulo gestor para sustituir el interfaz de usuario y probar el resto del sistema antes de la implementación del interfaz. Los módulos sustitutos pueden utilizarse en situaciones donde se requiere la colaboración de dos clase y una de ellas no está todavía implementada.

8.7. Pruebas de validación

Una vez culmina la integración y sus pruebas asociadas, y con el software completamente ensamblado, el siguiente paso consiste en asegurar que el producto obtenido cumple la especificación bajo la que se construyó. La validación del software se consigue a través de una serie de pruebas que comprueban la conformidad con los requerimientos, no sólo en cuanto a funcionalidad, sino también en cuanto a rendimiento, ergonomía y documentación.

Más allá incluso de la propia especificación, 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. Para un desarrollador de software es imposible prever cómo usará el cliente realmente el programa. Algunos problemas típicos son:

- Mensajes ininteligibles para el usuario
- Presentación inadecuada de resultados
- Deficiencias en el manual de usuario
- Procedimientos de operación inusuales

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 las 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 presenta. 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.

8.8. Pruebas del sistema

El software es un elemento de un sistema basado en computadora más grande. Las pruebas de sistema tienen como objetivo probar el sistema completo basado en computadora. A continuación se describen diferentes tipos de pruebas de sistema, todas ellas encaminadas a asegurarse que el software se ha integrado de forma correcta con su entorno.

Pruebas de recuperación

Sirven para comprobar la capacidad del sistema para recuperarse antes 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.

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, normalmente en sistemas de tiempo real o en sistemas embebidos. Para medir los tiempos se suelen necesitar equipos de instrumentación externos (osciloscopios, emuladores, analizadores lógicos,...). Por ejemplo se pueden forzar al sistema provocando múltiples interrupciones simultáneamente para medir el tiempo máximo que emplea en tratarlas.

Pruebas de despliegue o de configuración
Sirven para comprobar el funcionamiento de software que debe ejecutarse en varias plataformas y sistemas operativos distintos. Además permiten comprobar la correcta instalación del programa y la documentación para ello.

8.9. Conclusiones

Probar es parte inevitable del desarrollo de cualquier sistema de software, de hecho, las pruebas de software representan la parte más importe dentro del esfuerzo técnico en el proceso de software. Incluso en los sistemas más pequeños es necesario marcar la estrategia para planificar y ejecutar las pruebas sistemáticas que garanticen la calidad del producto final.

De forma global, la estrategia de pruebas comienza por las partes más pequeña con entidad propia o unidades, las cuales se integran hasta componer el programa completo. Una vez integrado, se comprueba que se cumplen los requisitos que se especificaron. Finalmente se realizan pruebas para mejorar otros aspectos tales como el rendimiento, la integrabilidad o la documentación.

8.10. Ejercicios propuestos

1. Analícese las soluciones comerciales para la planificación y gestión de pruebas que ofrecen las empresas u organizaciones de las páginas web que se listan a continuación:

   1.1. www.testmanagement.com

   1.2. www.compuware.com

   1.3. www.soft.com

   1.4. www.opensourcetesting.org

2. Consúltese la guía SWEBOK y revise la propuesta de clasificación de pruebas que esta hace.

3. A partir de un algún programa convencional que se haya implementado, realícese pruebas de caja negra y caja transparente.

4. A partir de un algún programa orientado a objetos que se haya implementado, realícese las siguientes pruebas:
   - Pruebas de partición de sus clases
   - Pruebas de integración basadas en hebra y basadas en uso

5. ¿Quién realiza las pruebas de validación? Razónes la respuesta.

Bibliografía

[Abbott83] R.J. Abbott: Pro gram Design by Informal English Descriptions. Comm.ACM, V.26, N.11, Nov.1983,

[Ale77] ALEXANDER, C, *A Pattern Language*, Oxford University Press, 1977.

[Aho83] A.V. AHO, J. HOPEROFT, J. ULLMANN, *Data Structures and Algorithms.* Addison-Wesley, 1983.

[Beck00] BECK, KENT, *Extreme Programming Explained: Embrace Change*, Addison-Wesley Professional, 2000.

[Boehm88] BOEHM, B.W., *A Spiral Model of Software Development and Enhancement*, IEEE Computer, Mayo 1988.

[Booch94] G. BOOCH, *Object-Oriente d Analysis and Design with Applications (2nd .Ed).* Benjamín/Cummings, 1994.

[Booch99] G. BOOCH, J. RUMBAUGH, I. JACOBSON, *El Lenguaje Unificado de Modelado*, Addison Wesley, 1999.

[BOOTH67] TAYLOR L. BOOTH, *Sequential Machines and Automata Theory* New York: Wiley, 1967.

[BenAri90] M. BEN-ARI, *Principies of Concurrent and Distributed Pro gramming.* Prentice-Hall, 1990.

[Budd94] T. BUDD, *Introducción a la Programación Orientada a Objetos.* Addison-Wesley Iberoamericana, 1994.

[Cerrada00] J.A. CERRADA, M. COLLADO Y OTROS, *Fundamentos de Programación con Módula-2.* Editorial Universitaria Ramón Areces, Madrid, 2000.

[Coad90] P. COAD, E. YOURDON, *Object-Oriented Analysis*, Prentice-Hall, 1990.

[Collado87] M. COLLADO, R. MORALES, J.J. MORENO, *Estructuras de Datos: Realización en Pascal*, Ediciones Diaz de Santos, 1987.

[Date90] C.J. DATE, *An Introduction to Database Systems*, Addison-Wesley, 1990. Addison-Wesley Iberoamericana, 1993.

[DeMarco79] T.DEMARCO, *Structured analysis and System specification*, Prentice Hall, 1979.

[Dahl72] O. DAHL, E. DIJKSTRA, C. HOARE, *Structure Programming*, Academic Press, 1972.

[Dijkstra68] E. W. DIJKSTRA, *A Constructive Approach to the Problem of Program Correctness.* BIT N° 8, pp. 174-186, 1968.

[DoD88] DOD-STD-2167A, *Military Standard: Defense System Software Development*, USA Dep. of Defense, Feb 1988.

[DeMiguel93] A. DE MIGUEL, M.G. PIATINI, *Concepción y diseño de bases de datos: del modelo E-R al modelo relacional.* Ra-Ma, 1993.

[ESA91] ESA BOARD FOR SOFTWARE STANDARDISATION AND CONTROL (BSSC), *ESA Software Engineering Standars. Issue 2*, USA Dep. of Defense, FEb 1991.

[ETIMCHIL] *http://etimologias.dechile.net*

[Gam95] GAMMA, E ET AL, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Hatley87] D. J. HATLEY, I. A. PIRBHAI, *Strategies for Real-Time Systems Specification.* Dorset House, 1987.

[IEEE87] IEEE STD 1016-1987, *IEEE Recommended Practice for Software Design Descriptions.*IEEE Computer Society Press, 1987.

[IEEE84] IEEE STD 830-1984,*IEEE Guide to Software Requirements Specifications*, 1984.

[IEEE93] IEEE SOFTWARE ENGINEERING STANDARDS COLLECTION,IEEE Computer Society Press, 1993.

[ISO1694908] INTERNATIONAL STANDARD ORGANIZATION , *Quality management systems – Particular requirements for the application of ISO 9001:2008 for automotive production and relevant service part organizations*, 2008.

[Jacobson92] IVAR JACOBSON, MAGNUS CHRISTERSON, PATRIK JONSSON, GUNNAR OVERGAARD, *Object-Oriented Software Engineering: A Use Case Driven Approach*,(ACM Press) Addison-Wesley, 1992,

[Jackson75] M.A. JACKSON *Principies of Pro gram Design*, Academic Press,1975.

[Jackson83] M.A. JACKSON *System Development Prentice-Hall*, 1983.

[Liskov80] B. LISKOV *Modular Program Construction Using Abstractions*, Lectures Notes in Computer Science N° 86.

[McCall78] J.A. MCCALL, J.O. CAVANO, *A Framework for Measurement of Software Quality*,ACM Software Quality Assurance WorkShop, Nov 1978.

[Mic04] MICROSOFT, *Prescriptive Architecture: Integration and Patterns*, MSDN, 2004.

[Myers78] G. MYERS *Composite Structured Design.* Van Nostrand, 1978.

[NAUGHTON97] JPATRICK NAUGHTON; HERBERT SHILDT, *Java, Manual de Referncia*,McGraw-Hill / Interamerica, S.A., 1997.

[Orr81] K. ORR, *Structured Requierements Definition*, Ken Orr & Associates, 1981.

[Parnas72] D. L. PARNAS, *On Criteria to be used in Decomposing Systems into Modules*,Comm. ACM, V.14, N.1, Abril.1972, pp.221-227.

[Page-Jones80] M. PAGE-JONES, *The Practical Guide to Structured System Design*, Yourdon-Press, 1980.

[Pratt84] T. PRATT, *Programming Languages. 2ª Edición*, Prentice-Hall, 1984.

[Pressman10] R.S. PRESSMAN, *Ingeniería del Software: Un Enfoque Práctico*, McGraw-Hill,2010.

[Rumbaugh91] RUMBAUGH, M BLAHA, W. PREMERLANI, F.EDDY, W. LORENSEN, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[Stevens74] W. STEVENS, G. MYERS, L. CONSTANTINE, *Structured Design*, IBM Systems Journal, V. 13 n° 2, 1974, pp. 115-139.

[SWEBOK04] A. ABRAN, J. W. MOORE; EDITORS, P. BOURQUE, R. DUPUIS, *Guide to the Software Engineering Body of Knowledge*, Version IEEE Computer Society,2004.

[Warnier81] J.D. Warnier: *Logical Construction of Systems.* Van Nostrand Reinhold, 1981.

[Wirth71] WIRTH N., *Program Development by Stepwise Refinement*, Comm.ACM, V.14, N.4, 1971, pp.221-227.

[Wirth80] WIRTH N., *Algoritmos + Estructuras de Datos = Programas*, Editorial Castillo, 1980.

[Yourdon79] E. N. YOURDON, L. CONSTANTINE *Structured Design.* Prentice-Hall, 1979.

[Yourdon90] YOURDON, E.N., *Modern Structured Analysis, Prentice-Hall*,1990.

ISBN-13: 978-84-9961-093-1

www.