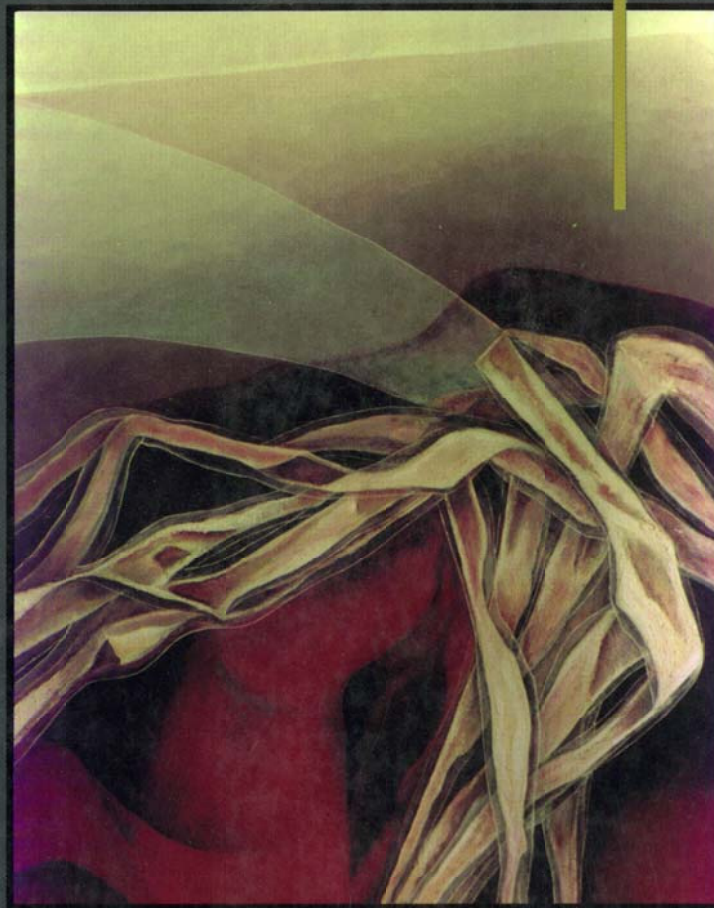


Carma McClure

CASE
la automatización del software



serie paradigma



ADDISON-WESLEY IBEROAMERICANA



carma

C A S E

**La automatizacion
del software**

A Constance Sue

C A S E

La automatización del software

Carma McClure

Traducción: José Manuel Orenge y Ortega
Licenciado en Informática
Miembro de A.L.I. y de O.A.I.

Revisión Técnica:
Manuel Rodríguez García.
Departamento de Lenguajes en la
Escuela Politécnica Superior de la
Universidad Carlos III de Madrid.



ADDISON-WESLEY IBEROAMERICANA



ca-ma

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo RA-MA Editorial y Addison-Wesley Iberoamericana, S.A. no asumen ninguna responsabilidad derivada de su uso, ni tampoco por cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir.

Edición original publicada por RA-MA Editorial. Madrid, España.

Traducción autorizada de la edición original en lengua inglesa:
CASE IS SOFTWARE AUTOMATION; publicado por:
Prentice Hall, Englewood Cliffs, New Jersey 07632, USA.

- © Carma McClure. MCMLXXXIX.
- © De la traducción y de la edición en lengua española: RA-MA 1992.
- © 1993 por Addison Wesley Iberoamericana, S.A.
Wilmington, Delaware, E.U.A.

Edición autorizada para venta en el continente americano.

Reservados todos los derechos.

Ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro, sin autorización previa y por escrito del editor.

✓ ISBN 0-201-62522-9

Impreso en E.U.A. *Printed in U.S.A.*

1 2 3 4 5 6 7 8 9 10-DO-96 95 94 93 92

Inv. # 6149-1001
Copia
Berna 12-28-1999
U.S.D.

Presentación de la serie PARADIGMA

La serie **PARADIGMA** trata de sacar a la luz la reflexión de cualificados autores acerca de los múltiples aspectos, tanto internos como externos, de la disciplina informática. Se caracteriza por su forma teórica de abordar los temas. No tienen en ella cabida los manuales o libros eminentemente enfocados hacia la enseñanza de alguna materia concreta. En cambio se alimenta de la reflexión teórica, de la investigación, de la opinión fundada.

Con **PARADIGMA** queremos aportar, desde editorial RA-MA, una herramienta de comunicación para todas aquellas ideas y nuevas metodologías que dinamizan continuamente la profesión del informático. El objetivo principal es conseguir una importante cota de calidad tanto en el contenido de lo publicado en la serie, como en la forma de presentación de la misma. De este modo, lo que se ofrece a los lectores es el fruto más esmerado de nuestro trabajo. Vaya hacia ustedes con los mejores deseos.

Dentro de la serie damos cabida a dos tipos distintos de obras, lo que da lugar a dos diferentes colecciones:

Informática: Fundamentos Teóricos

Informática Profesional

INDICE

Lista de cuadros	13
Prólogo	15
 PARTE 1. INTRODUCCION A LA INGENIERIA DE SOFTWARE ASISTIDA POR ORDENADOR	
 CAPITULO 1. Ingeniería de Software Asistida por Ordenador:	
 Actualidad	19
 Sólo imaginación	19
Ingeniería de Software Asistida por Ordenador	20
La automatización del software	21
Beneficios de la CASE	23
Las herramientas CASE	23
Ejemplos de herramientas CASE	25
La historia de la CASE	26
Objetivo de la CASE	33
Un cambio de actitud	34
La dirección de la automatización	35

PARTE 2. LOS COMPONENTES DE UN SISTEMA CASE

CAPITULO 2. El entorno de desarrollo de software CASE..	41
El nuevo entorno de desarrollo de software	41
El banco de trabajo CASE	42
Las capacidades gráficas	44
La necesidad de los diagramas	46
Ventajas de las técnicas de diagramación estructurada	48
Usos de los diagramas estructurados	48
Visión del sistema múltiple	57
El trío esencial de tipos de diagramas	59
Diagramación automática	59
Más allá del simple dibujo automático	64
La comprobación de errores	66
El depósito de la información de la CASE	80
Integración	94
Resumen	97
 CAPITULO 3. El soporte Case a los procesos de software ..	 101
El soporte de las metodologías y los procesos de software	101
El énfasis en las primeras fases del ciclo de vida	102
Los prototipos	103
La simulación	106
La generación de código	106
El soporte de la metodología estructurada	109
La clasificación de las metodologías estructuradas	113
Las metodologías estructuradas de utilización más extendida ..	124
El análisis estructurado	126
El diseño estructurado de Yourdon	134
La metodología de diseño de Jackson	143
Metodología de la ingeniería de la información de Martin	150
La metodología DSSD	158
 CAPITULO 4. Las plataformas hardware de la CASE	 165
La redefinición del entorno del soporte software	165
Las alternativas de las plataformas hardware	165
La plataforma hardware de niveles múltiples	168

Los componentes de un sistema CASE básico	171
Resumen	175
 CAPITULO 5. Las categorías de herramientas CASE	179
Las distintas herramientas CASE	179
Los juegos de herramientas CASE	180
Los juegos de herramientas de análisis	183
Juegos de herramientas para el diseño de los datos	188
Juegos de herramientas de programación	188
Juegos de herramientas de mantenimiento	191
Juegos de herramientas de gestión de proyectos	198
Framework	199
Los bancos de trabajo CASE	200
Compañeros de metodologías CASE	202
 PARTE III. LA UTILIZACION DE LA CASE	
 CAPITULO 6. Casos de estudio de la CASE	209
El incremento de la productividad del software	209
Experiencias con EXCELERATOR	211
Experiencias de productividad con APPLICATION FACTORY	214
Experiencias de productividad con INFORMATION ENGINEE- RING WORKBENCH	221
Resumen	225
 CAPITULO 7. Consideraciones de la implantación de la CASE	227
Problemas y soluciones del software	227
CASE o no CASE	229
Establecimiento de las necesidades	230
Tomar la decisión por una CASE	233
El plan de implantación de la CASE	234
Primero la metodología	234
El proyecto piloto de la CASE	238
Implantadores de la CASE	238
La selección de la herramienta CASE	239

Venta de la CASE	239
La evaluación de la CASE	242
 CAPITULO 8. Cambios en el ciclo de vida del software	 245
El ciclo de vida del software	245
Concepto unificador	246
Modelo de ciclo de vida tradicional	248
El ciclo de vida CASE del software	250
Modelo y solo uno	257
Una nueva visión del ciclo de vida	257
 CAPITULO 9. Relación de la CASE con otras tecnologías de software	 259
La combinación de tecnologías	259
La tecnología de cuarta generación	261
La tecnología de quinta generación	265
La cobertura del ciclo de vida	267
Ventajas de la CASE sobre la cuarta generación	268
Resumen	269
 PARTE 4. EL DESARROLLO DEL SOFTWARE EN LOS AÑOS 90	
 CAPITULO 10. Las características de la automatización del software	 275
Más automatización del software	275
Imagine de nuevo	277
Los cambios en el proceso del software	278
Los cambios en las herramientas de software	279
El shell inteligente	281
 CAPITULO 11. El entorno habitable	 285
El interfaz del usuario	285
Más allá de los entornos CASE	286

Amigable con el usuario	287
Orientado al usuario	293
De sensible a reactivo	302
Diagnóstico	303
Corrector	305
Ayuda	305
El tutor inteligente	312
Resumen	320
 CAPITULO 12. El conductor de metodología	 323
Conocimiento y experiencia en la metodología del software auto- mático	323
Los componentes de un conductor de metodología	326
El nuevo proceso de desarrollo del software	338
Resumen	342
 CAPITULO 13. La reusabilidad del software	 345
La reusabilidad con los sistemas CAD/CAM	345
Cambio revolucionario en el desarrollo de software	346
Librerías de software	347
Formas de la reusabilidad del software	348
Estudios de reusabilidad y proyectos	349
Los problemas de la reusabilidad del software	350
La representación de un componente reutilizable	352
Paquetes	352
Partes	355
Las limitaciones del código reutilizable	356
La CASE y el código reutilizable	358
El nivel más alto del software reutilizable	361
La reutilización de las especificaciones	361
La construcción del programa con componentes reutilizables ..	362
La reusabilidad incrementa la productividad	365
Resumen	366

PARTE 5. EPILOGO

CAPITULO 14. La CASE, caso cerrado	371
Ultimas consideraciones	371
Indice alfabético de términos	375

LISTA DE CUADROS

Cuadro 1.1.	Las ventajas de la tecnología CASE	23
Cuadro 1.2	Definiciones	36
Cuadro 2.1	Los objetivos de las estaciones de trabajo	43
Cuadro 2.2	Requisitos de una buena documentación	47
Cuadro 2.3	Técnicas de diagramación estructurada	49
Cuadro 2.4	Funciones de las herramientas de diagramación estructurada	61
Cuadro 2.5	Prestaciones de la diagramación automática	62
Cuadro 2.6	Tipos de correcciones de errores en los diagramas estructurados ...	68
Cuadro 2.7	Ejemplos de tipos de informes del depósito CASE	88
Cuadro 2.8	Prestaciones del depósito de la CASE	95
Cuadro 2.9	Niveles de integración	96
Cuadro 2.10	El CASE workbench: el nuevo entorno de desarrollo de software ..	97
Cuadro 3.1	Herramientas CASE de prototipos	103
Cuadro 3.2	Las premisas básicas de la ingeniería de software	114
Cuadro 3.3	Los diagramas de la ingeniería de software	115
Cuadro 3.4	El enfoque dirigido por los datos para el desarrollo de sistemas ...	118
Cuadro 3.5	Las premisas básicas de la ingeniería de la información	121
Cuadro 3.6	Características de los sistemas de tiempo real y de los de información	124
Cuadro 3.7	Las metodologías más utilizadas	126
Cuadro 3.8	Clasificación de las metodologías	127
Cuadro 3.9	Los pasos en los procesos de análisis estructurado	132
Cuadro 3.10	Los símbolos utilizados en los diagramas de flujo de datos	135
Cuadro 3.11	Pasos básicos en el proceso de diseño de Jackson	143
Cuadro 3.12	Los diagramas de la ingeniería de la información	153
Cuadro 4.1	Las funciones de la CASE	169
Cuadro 4.2	Consideraciones sobre las plataformas hardware CASE	176

Cuadro 5.1	Categorías de herramientas CASE	180
Cuadro 5.2	La clasificación de las herramientas	183
Cuadro 5.3	Los juegos de herramientas de análisis	186
Cuadro 5.4	Juegos de herramientas de programación	188
Cuadro 5.5	Beneficios del generador de código en COBOL	191
Cuadro 5.6	Juegos de herramientas de mantenimiento	192
Cuadro 5.7	Ejemplos de herramientas de mantenimiento	197
Cuadro 5.8	Juegos de herramientas de gestión de proyectos	198
Cuadro 5.9	Características de los juegos de herramientas frameworks CASE ...	199
Cuadro 5.10	Clasificación de los bancos de trabajo (workbench).....	201
Cuadro 5.11	Ejemplos de compañeros de metodología CASE	203
Cuadro 5.12	Clasificación de compañeros de metodología CASE.....	204
Cuadro 6.1	Cuestiones sobre la productividad	226
Cuadro 7.1	Causas del fracaso de la CASE	228
Cuadro 7.2	La definición de las necesidades de una organización para mejorar la tecnología actual del desarrollo de software	231
Cuadro 7.3	El plan de implantación de la CASE	235
Cuadro 7.4	Los pasos básicos para la implantación de la CASE.....	237
Cuadro 7.5	Consideraciones técnicas en la selección de las herramientas CASE.	240
Cuadro 7.6	Formas de vender la tecnología CASE	241
Cuadro 8.1	La CASE cambia el desarrollo de software	253
Cuadro 9.1	Comparación entre herramientas	259
Cuadro 9.2	Categorías en la cuarta generación	261
Cuadro 9.3	Cuarta generación frente a CASE	264
Cuadro 9.4	Las características de un sistema experto	266
Cuadro 9.5	¿Qué sucederá con las tecnologías de la tercera, cuarta y quinta genera- ción?	269
Cuadro 10.1	Las características de la automatización del software	278
Cuadro 11.1	Las características de un interfaz amigable con el usuario	292
Cuadro 11.2	Las características de un interfaz orientado al usuario	293
Cuadro 11.3	Métodos para integrar la ayuda en el sistema	308
Cuadro 11.4	Tipos de ayuda (Help)	309
Cuadro 11.5	Características de un buen tutor	313
Cuadro 11.6	El tutor inteligente	313
Cuadro 11.7	Las características de un entorno habitable	319
Cuadro 12.1	La tutela automática del sistema	325
Cuadro 12.2	Los principios de la ingeniería de software	328
Cuadro 12.3	Principios de la ingeniería de la información	330
Cuadro 12.4	La regla para los módulos de los programas	331
Cuadro 12.5	Las propiedades de un programa estructurado	332
Cuadro 12.6	Estrategias para guiar la descomposición funcional	337
Cuadro 12.7	El conductor de metodología	342
Cuadro 13.1	Las ventajas de la reusabilidad	348
Cuadro 13.2	Los problemas de la implantación práctica de lo que parece una buena idea	351
Cuadro 13.3	¿Qué es un componente reutilizable de software?	351
Cuadro 13.4	Ejemplos de paquetes de aplicaciones	354
Cuadro 13.5	Fallos de los paquetes de aplicaciones	355
Cuadro 13.6	Los pasos en la adquisición de un paquete de aplicaciones	357
Cuadro 13.7	Las propiedades de un componente reutilizable	358
Cuadro 13.8	Las propiedades de una librería de componentes reutilizables	360
Cuadro 13.9	Los conceptos de aproximación por partes frente a la aproximación en conjunto	366

PROLOGO

Revolución. Revolución ¡REVOLUCION! No. El tema de este libro no es otra revolución tecnológica. Más bien trata del cambio que está teniendo lugar en las herramientas de desarrollo del **software**. Como es un cambio tan lógico y que parece ser el próximo paso natural en el avance de la tecnología del **software**, ha captado el interés de la industria del **software** prácticamente de la noche a la mañana. En los comienzos del año 1987, este cambio podría describirse como un nuevo capricho para los MIS (**Management Information System**, Gestores de los Sistemas de Información). Como sucede a menudo con los caprichos, hizo que el mundo de los MIS dejase a un lado otras tecnologías, incluyendo las de cuarta y quinta generación. A finales del mismo año, el capricho se extendió a los profesionales del desarrollo de sistemas en tiempo real y a los sistemas comerciales de proceso de datos.

Este cambio en las herramientas de **software** es parte de una nueva tecnología denominada **computer aided software engineering** o **CASE** (**ingeniería de software asistida por ordenador**). Es una tecnología para automatizar el desarrollo y mantenimiento del **software**, siendo además una respuesta muy práctica a los últimos 25 años de crisis del **software**. Realmente, la tecnología CASE no es totalmente nueva ya que está construida sobre las técnicas estructuradas que se desarrollaron en la década de 1970. Esto es en parte una razón de su rápida y amplia aceptación.

Este libro está pensado para el profesional relacionado con el desarrollo y mantenimiento de los sistemas de **software**. Esta también pensado para los gestores de **software** y los estudiosos de la ingeniería de **software**.

El propósito del libro es introducirse en la tecnología CASE explorando lo que es, cómo se utiliza para mejorar la productividad del desarrollo del **software** y cómo se relaciona con otras tecnologías del **software**. Aunque mucha de la primitiva atención estaba centrada en las herramientas del **software**, la CASE abarca mucho más. En la tecnología CASE hay aspectos del **software**, del **hardware**, de la metodología y de la gestión. Cada uno de estos aspectos se discuten en este libro para ofrecerle al lector una comprensión completa de la tecnología actual de la CASE.

El objetivo principal del libro es presentar herramientas prácticas, métodos y técnicas de gestión que ayudan a resolver las siempre crecientes necesidades de mayor y mejor calidad del **software**: la CASE puede ayudar a conseguir tal objetivo.

En ocasiones un cambio puede resultar más excitante que una revolución súbita, que puede llevar también al caos y a la ruina. La CASE es este tipo de excitante cambio. Ha inyectado a la industria del **software** un alto nivel de esperanza y entusiasmo por la conquista de los problemas del **software**. La CASE puede ser muy bien el más profundo cambio en esta industria. Por lo menos es el paso mayor y más próximo hacia los más altos niveles de la automatización del **software**.

Mi agradecimiento especial a los distribuidores de CASE, que han contribuido con los ejemplos de los productos que aparecen en este libro. También mi más sincero agradecimiento a DEBBIE JACKSON por haber mecanografiado y editado el manuscrito del libro.

Carma McClure

PARTE 1

**INTRODUCCION A LA INGENIERIA DE
SOFTWARE ASISTIDA POR ORDENADOR**

INGENIERA DE SOFTWARE ASISTIDA POR ORDENADOR: ACTUALIDAD

SOLO IMAGINACION

Imagine el lector que está desarrollando **software** en su propia estación de trabajo (ver la Figura 1.1). Es posible que ésta sea un potente ordenador personal con pantalla de gráficos en color y con una impresora asociada. Una red de área local le conecta con su equipo de proyecto y con el resto de su departamento. Además su estación está conectada al ordenador principal, donde diccionarios y bases de datos corporativos le unen al resto de la organización. Cada una de sus tareas está asistida por herramientas que residen en su ordenador. Estas herramientas van desde las que simplemente le ayudan a mantener la cronología, hasta las que le ayudan a desarrollar modelos de prototipos del sistema que está construyendo. El lector se comunica con su estación por medio del teclado o pulsando los botones del ratón. La utiliza para comprobar el trabajo, para acceder a la biblioteca de “**chips de software**” reutilizables y realizar muchas de las tareas rutinarias que forman parte de todos los proyectos de desarrollo de **software**.

Hay muchas herramientas de **software** potentes y de muy extendido uso para estaciones de trabajo individuales, pero la clave está en que la estación de trabajo proporciona un “**ensamblaje de herramientas integradas**” (herramientas que son accesibles desde un puesto y de forma que puedan

llamarse, alimentarse y utilizarse desde otros). La diferencia está en las distintas herramientas individuales —ocultas a la vista del usuario— unidas por medio de un interfaz de usuario común. En otras palabras, la estación de trabajo proporciona un entorno completo de **hardware** y de **software** trabajando al unísono para soportar el desarrollo de **software**.

¿Es esto una fantasía? ¿Es una visión del presente o una visión del futuro entorno de desarrollo de **software**? Afortunadamente, esta visión ya es más una realidad que una fantasía.

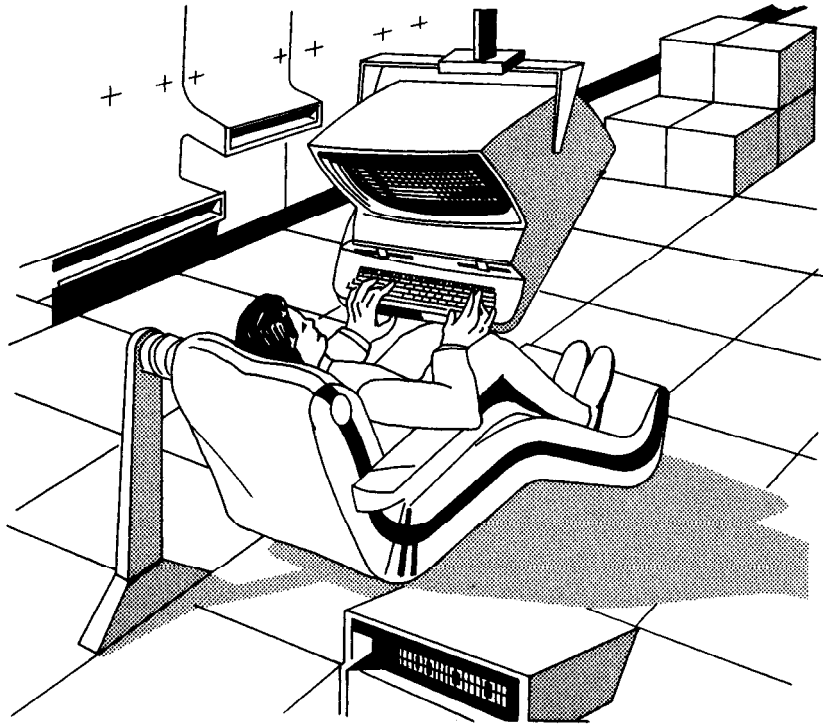


Figura 1.1. Imagine desarrollando el software en su propia y dedicada estación de trabajo

INGENIERIA DE SOFTWARE ASISTIDA POR ORDENADOR

Un drástico cambio en las herramientas de **software** está sucediendo hoy en día. Las herramientas de diseño para automatizar varias de las ta-

reas del ciclo de vida se crean para utilizarse en los micros. Muchas de tales herramientas se centran en el análisis del **software** y en las tareas de diseño. Con ayuda de estas herramientas, los profesionales del desarrollo pueden crear sistemas de **software** interactivamente en ordenadores personales y estaciones de trabajo.

Tanto los departamentos de MIS (management information systems) como los de desarrollo de sistemas de tiempo real, están recibiendo estos cambios con los brazos abiertos. Todos están entusiasmados ante la nueva tecnología de **software**, que puede resolver los problemas que desde hace 25 años aquejaban a la productividad. Esta nueva tecnología es la llamada **ingeniería de software asistida por ordenador**.

Definida de una forma simple, la ingeniería de **software** asistida por ordenador es la **automatización del desarrollo del software**, comúnmente referida como **tecnología CASE**. Hay más expectación y esperanza asociada a la CASE que a cualquiera de las tecnologías de **software** anteriores. La CASE está consiguiendo que los profesionales del desarrollo olviden las tecnologías de la tercera, la cuarta —e incluso quinta— generación pues la CASE reemplazó a los lenguajes de la cuarta generación como la más poderosa tecnología de desarrollo de **software**.

LA AUTOMATIZACION DEL SOFTWARE

La CASE propone una nueva formulación del concepto de ciclo de vida del **software**, basada en la automatización. La idea básica que subyace en la CASE es proporcionar un conjunto de herramientas bien integradas y que ahorren trabajo, enlazando y automatizando todas las fases del ciclo de vida del **software** (ver la Figura 1.2).

Las tecnologías tradicionales del **software** son de dos tipos: herramientas y metodologías. Estas categorías incluyen herramientas de tercera, de cuarta y (más recientemente) de quinta generación. La mayoría de estas herramientas son autónomas, basadas en un **mainframe** (ordenador principal) y dirigidas a la implantación del ciclo de vida del **software**.

En la categoría de metodologías de construcción de sistemas **software**, se incluyen las metodologías de desarrollo manual, como el análisis estructurado, el diseño estructurado y la programación estructurada.

Estas metodologías definen un disciplinado proceso para el desarrollo del **software** paso a paso.

La tecnología CASE es una **combinación** de herramientas de **software** y de metodologías. Más aún, la CASE es diferente de las primitivas tecnologías del **software** porque se centra en el problema de la productividad del **software** y no solamente en la implantación de soluciones. Extendiéndose a todas las fases del ciclo de vida del **software**, la CASE es la tecnología del **software** más completa hoy en día. La CASE aborda los problemas de la productividad del **software** durante todo el ciclo de vida, automatizando muchas de las tareas del análisis y del diseño como también las tareas de la implantación y el mantenimiento de los programas.

Como las metodologías estructuradas manuales son demasiado tediosas y de un trabajo muy intensivo, en la práctica raramente se siguen a un nivel más detallado. La CASE hace prácticas las metodologías estructuradas manuales al automatizar el dibujo de diagramas estructurados y la generación de la documentación del sistema.

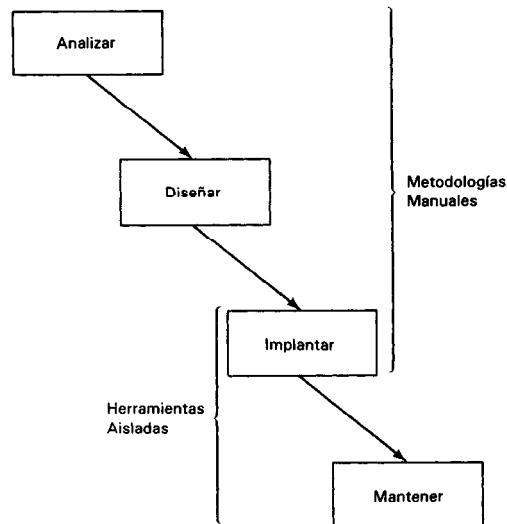


Figura 1.2. Las tecnologías tradicionales de software son de dos tipos: manuales y con herramientas aisladas. La tecnología CASE es una combinación de herramientas y metodologías totalmente integrada, con especial énfasis en la automatización del ciclo de vida del software

Cuadro 1.1. Las ventajas de la tecnología CASE

- Permite las técnicas estructuradas.
- Impone las ingenierías de software y de la información.
- Aumenta la calidad del **software** mediante comprobación automática.
- Favorece la realización de prototipos.
- Simplifica el mantenimiento del programa.
- Acelera el proceso de desarrollo.
- Libera al profesional de desarrollo de la principal parte creativa en el desarrollo de **software**.
- Anima al desarrollo evolucionado y gradual.
- Posibilita la reutilización de los componentes del **software**.

BENEFICIOS DE LA CASE

El cuadro 1.1 muestra las ventajas que la CASE ofrece a los profesionales del desarrollo de **software**. Esta exposición sugiere que la CASE no es enteramente una tecnología nueva, sino que está construida sobre técnicas y herramientas comprobadas en la práctica. En este sentido, la CASE puede definirse como un conjunto de **conceptos estructurados y de metodologías con un nuevo envase**. La novedad es la automatización.

LAS HERRAMIENTAS CASE

La columna vertebral de la tecnología CASE es una nueva generación de herramientas de **software**. Algunas de estas herramientas están basadas

en micros, tienen potentes entornos gráficos que hacen más amigable el interfaz del usuario y se incorporan a bases de trabajo donde pueden ser fácilmente llamadas, alimentadas y utilizadas por otros. Las herramientas CASE (CASE tools) se enfocan hacia la **productividad individual del profesional del desarrollo de software**. Con el uso de las estaciones de trabajo, de las redes de área local y de las herramientas basadas en los ordenadores personales, el profesional de desarrollo de **software** puede trabajar por primera vez desde un entorno de desarrollo de **software** fiable y dedicado.

Ya que las herramientas CASE se utilizan principalmente por los profesionales del desarrollo de **software** y no por los usuarios finales, la tecnología CASE no debe considerarse solamente como una mejora de las herramientas de cuarta generación. Además, las herramientas CASE difieren de las herramientas de la cuarta generación en otros dos aspectos. Primero, muchas herramientas CASE están diseñadas para estaciones de trabajo, mientras que muchas de las herramientas de cuarta generación están diseñadas para ordenadores principales (**mainframes**). Segundo, La CASE es una tecnología de **software** de propósito general para utilizarse en el desarrollo de todo tipo de sistemas (grandes y pequeños, comerciales y científicos, en línea y de tiempo real, etc.), mientras que las herramientas de cuarta generación se utilizan principalmente para desarrollar aplicaciones comerciales de tamaño pequeño o mediano.

En general, las herramientas CASE se diferencian de los primitivas herramientas de **software** en algunas cosas muy importantes. A saber, las herramientas CASE fueron diseñadas para:

- 1.— Soportar un entorno personal dedicado.
- 2.— Utilizar gráficos para especificar y documentar los sistemas.
- 3.— Juntar todas las fases del ciclo de vida del **software**.
- 4.— Capturar y juntar en el ordenador toda la información sobre el entorno del **software** de un sistema, desde los requerimientos iniciales hasta las actividades para un mantenimiento constante.
- 5.— Utilizar la inteligencia artificial para realizar automáticamente muchas de las rutinarias tareas del desarrollo y mantenimiento del **software**.

EJEMPLOS DE HERRAMIENTAS CASE

Aún más, las herramientas CASE realizan muchos más tipos de tareas que las herramientas de la tercera y cuarta generación. Soportan la automatización de una gran variedad de tareas de desarrollo y mantenimiento del **software**, incluyendo la gestión del proyecto. Como se muestra en la Figura 1.3, los ejemplos de herramientas CASE incluyen:

- Herramientas de diagramación para el dibujo de diagramas estructurados y la creación de especificaciones de sistemas gráficos.
- Pantallas de informes gráficos para la creación de las especificaciones del sistema y de formularios de prototipos simples.
- Diccionarios, sistemas de gestión de bases de datos y utilidades para almacenamiento información y consulta de la información técnica y de proyectos del sistema de gestión.
- Herramientas de validación de especificaciones para la detección automática de las que son incompletas, incorrectas sintacticamente o inconsistentes.
- Generadores de código para generar automáticamente código ejecutable a partir de las especificaciones obtenidas gráficamente.
- Generadores de documentación para producir la documentación técnica y del usuario requeridas por las técnicas estructuradas.

Aunque las herramientas son una parte muy importante de CASE, la tecnología CASE consta de mucho más. La CASE es una **redefinición del entorno completo de software**.

Una forma clave en la que CASE cambia el entorno de desarrollo del **software** es que muchas herramientas CASE operan en entornos de estación de trabajo. Esto cambia el desarrollo de **software** en un proceso altamente interactivo en el cual la detección de los errores comienza en los pasos iniciales de la definición de los requerimientos del sistema.

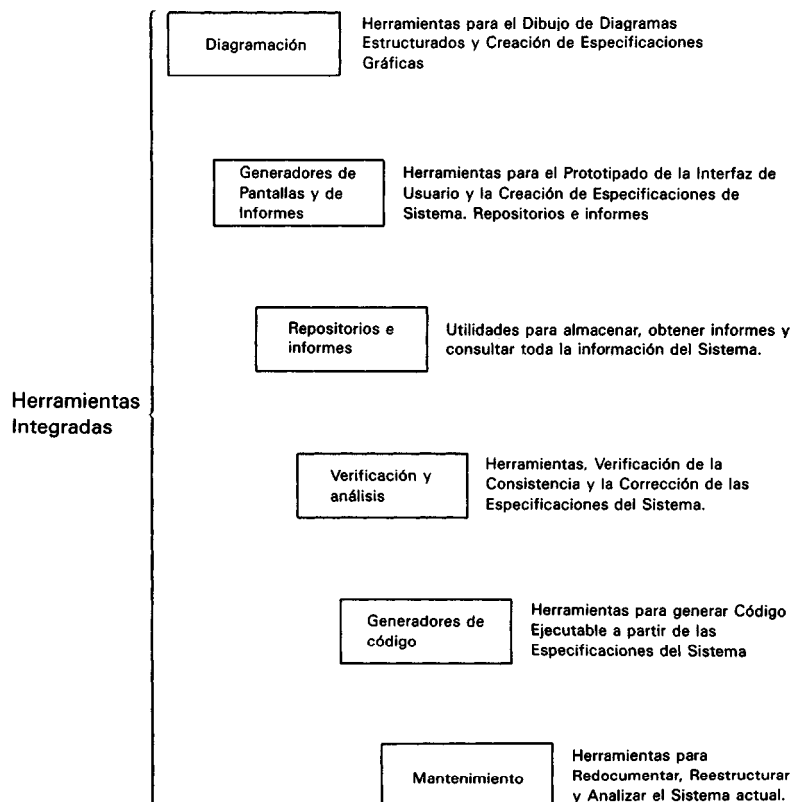


Figura 1.3. Las herramientas CASE cubren muchos tipos de las tareas del desarrollo y de la automatización del software

LA HISTORIA DE LA CASE

Como se muestra en la figura 1.4, la historia de la tecnología CASE comienza a principios de los años ochenta con la introducción de la documentación asistida por ordenador y de las herramientas de diagramación. Estas herramientas representan parte de las primeras herramientas de desarrollo de **software** basadas en ordenadores personales y los primeros intentos para automatizar el análisis y el diseño de tareas. Estas herramientas se idearon simplemente para el dibujo en puestos individuales. Se emplearon para crear diagramas estructurados (de flujo de datos, organigramas y de entidad/relación). Su propósito era producir automáticamente la documentación estructurada requerida por las distintas metodologías

de desarrollo estructurado. Las diferentes herramientas CASE soportan diferentes metodologías de desarrollo estructurado, como el desarrollo estructurado de análisis y de diseño de Yourdon o, el diseño estructurado de Jackson o la ingeniería de la información de Martin.

La figura 1.5 es una estructura gráfica manual del diseño de un programa de vigilancia de la temperatura utilizando la notación de Yourdon. La figura 1.6 es la estructura gráfica, también en la notación de Yourdon, pero dibujada automáticamente con una herramienta CASE. La figura 1.7 es un diagrama de flujo de datos utilizando la notación de Gane y Searson, dibujada por otra herramienta CASE, y la figura 1.8 es un diagrama entidad/relación dibujado automáticamente utilizando la notación de Martin.

Evolución de la tecnología CASE

Principios de los años Ochenta	Mediados de los años Ochenta	Ultimos años de los Ochenta	Principios de los años Noventa
Documentación asistida por ordenador Diagramación asistida por ordenador Herramientas de análisis y de diseño	Comprobación automática del análisis del diseño Depósito automático de la información del sistema	Generación automática del código a partir de las especificaciones de diseño Engarce del diseño automático y la programación automática	Conductor de metodología inteligente Interfaz amigable con el usuario La reutilización como metodología de desarrollo

Figura 1.4. La CASE es una tecnología con una buena sintonía de pocos años

Las primitivas herramientas CASE se dirigieron principalmente a la automatización de la documentación y de la comunicación como una mejora clave de la productividad del **software**. A mediados de los años ochenta, las herramientas CASE se mejoraron para proporcionar dos funciones muy importantes:

1. Comprobación automática de diagramas estructurados.
2. Almacenamiento de diagramas estructurados en librerías de diseño automático llamadas diccionarios, depósitos o enciclopedias.

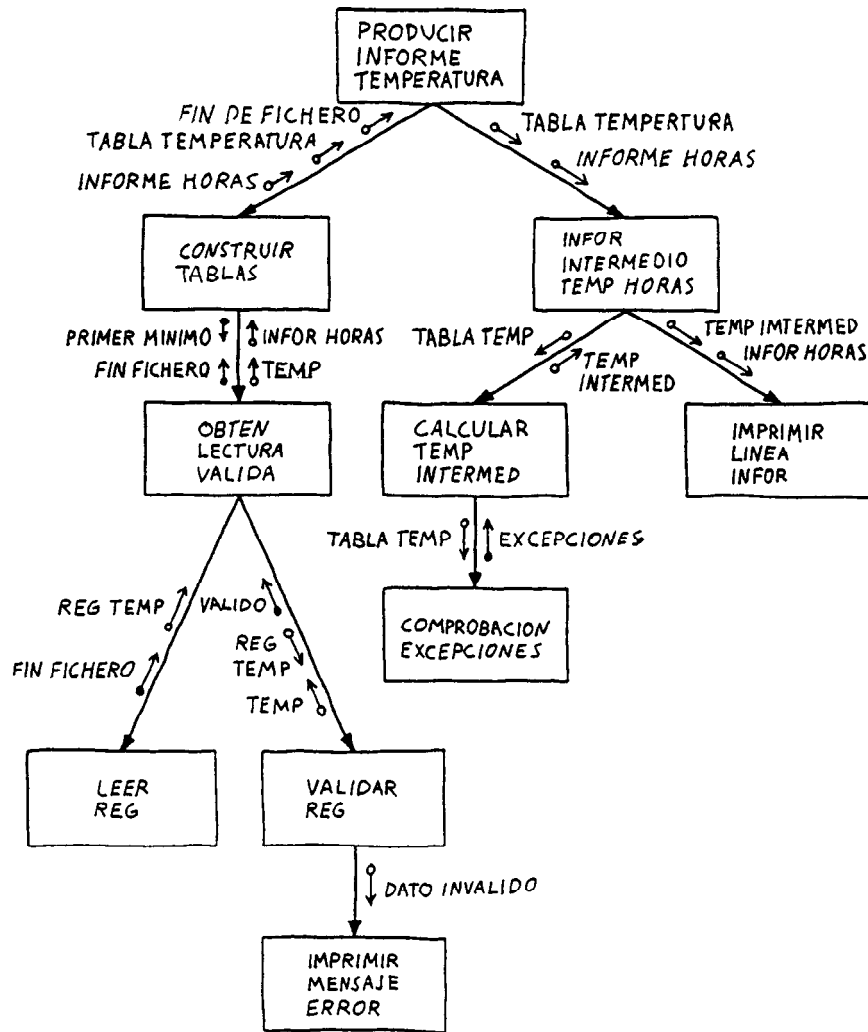


Figura 1.5. Diagrama de estructura manual utilizando la notación de Yourdon. Además de tediosa se tarda mucho dibujándola manualmente

Los diagramas se revisan para asegurar que están completos y correctos antes de utilizarlos como especificaciones de diseño de la implantación del programa. La comprobación se basa en las reglas de la metodología estructurada soportada. Además, se registran las referencias cruzadas de la información y se eliminan las redundancias.

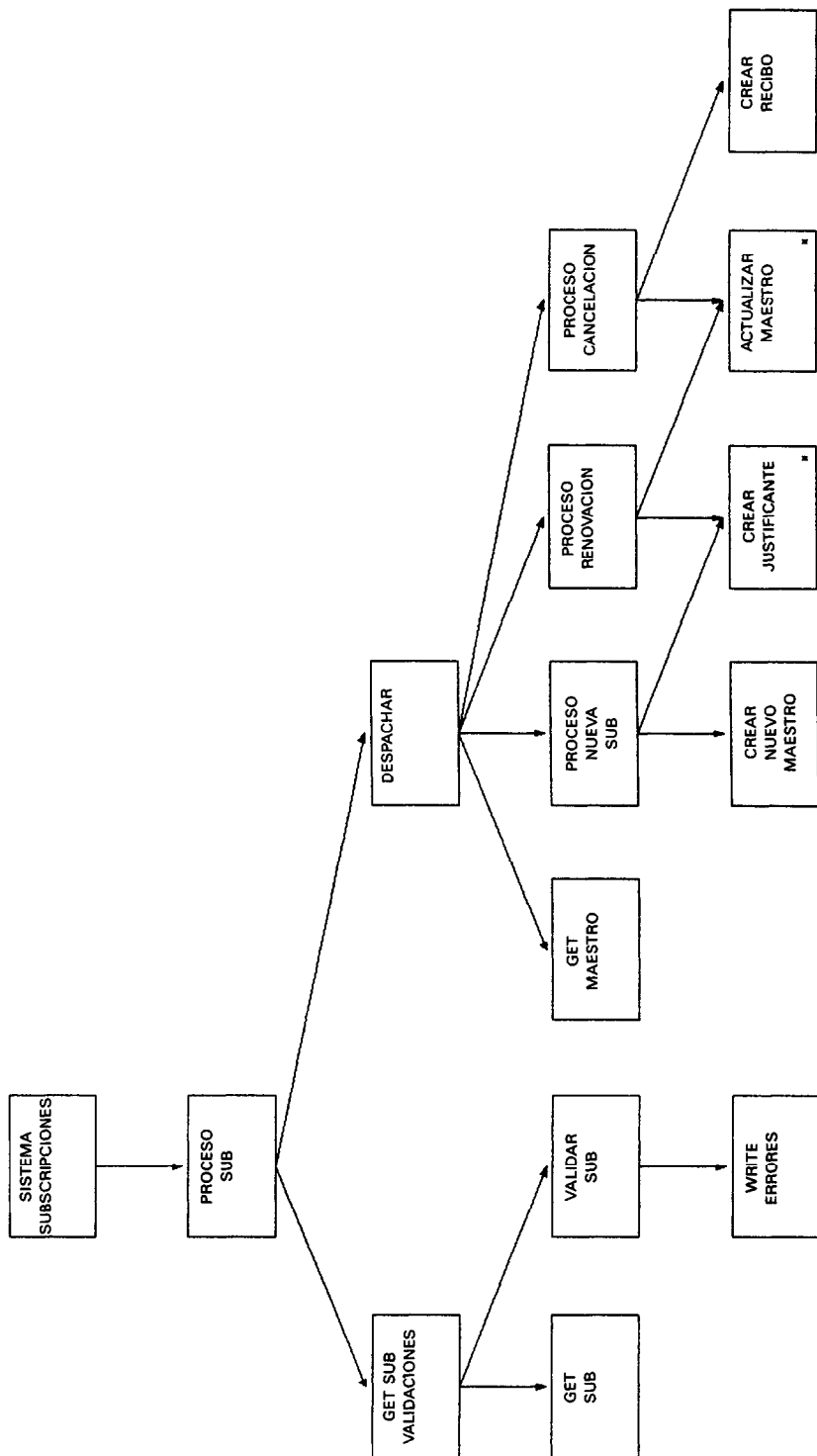


Figura 1.6. Diagrama de estructura obtenido con el INFORMATION ENGINEERING WORKBENCH de KnowledgeWare

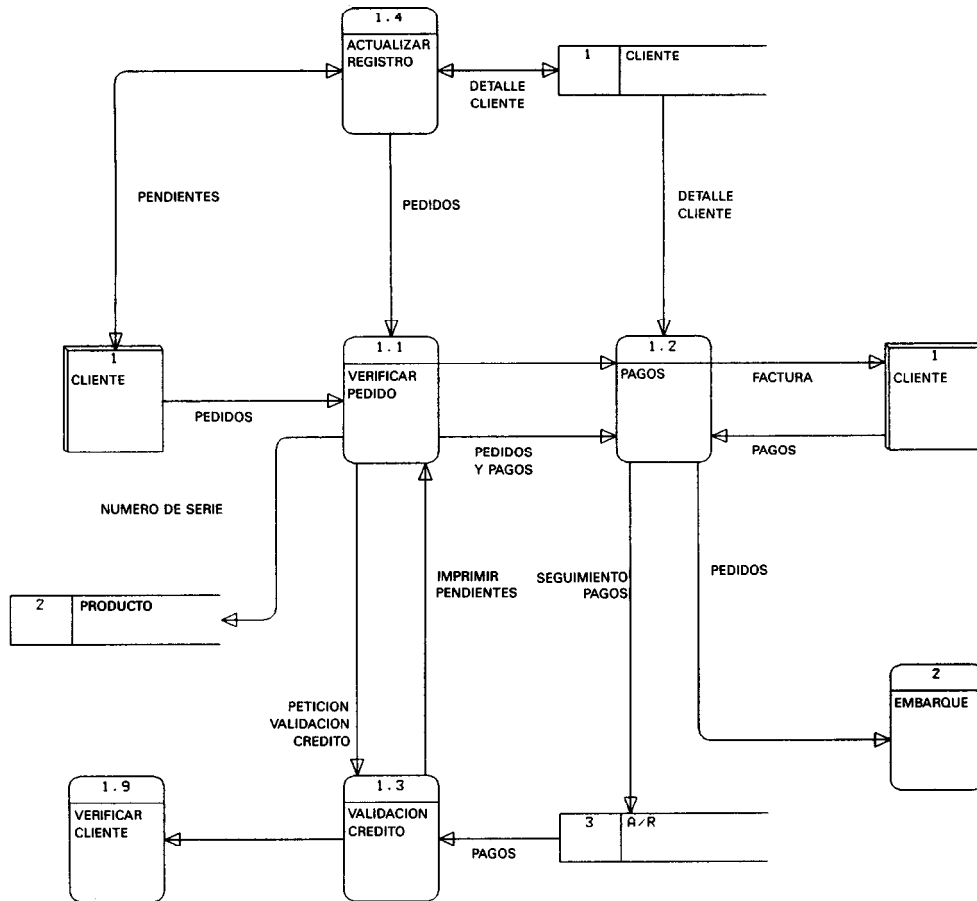


Figura 1.7. Diagrama de flujo de datos dibujado automáticamente por EX-CELERATOR de Index Technology Corporation

Después de la validación de los diagramas, estos se almacenan en diccionarios o en bases de datos, llamados depósitos CASE. Una vez almacenados los diagramas se pueden actualizar fácilmente compartiéndolos con todo el equipo del proyecto y más tarde utilizados en otros proyectos de desarrollo de **software**.

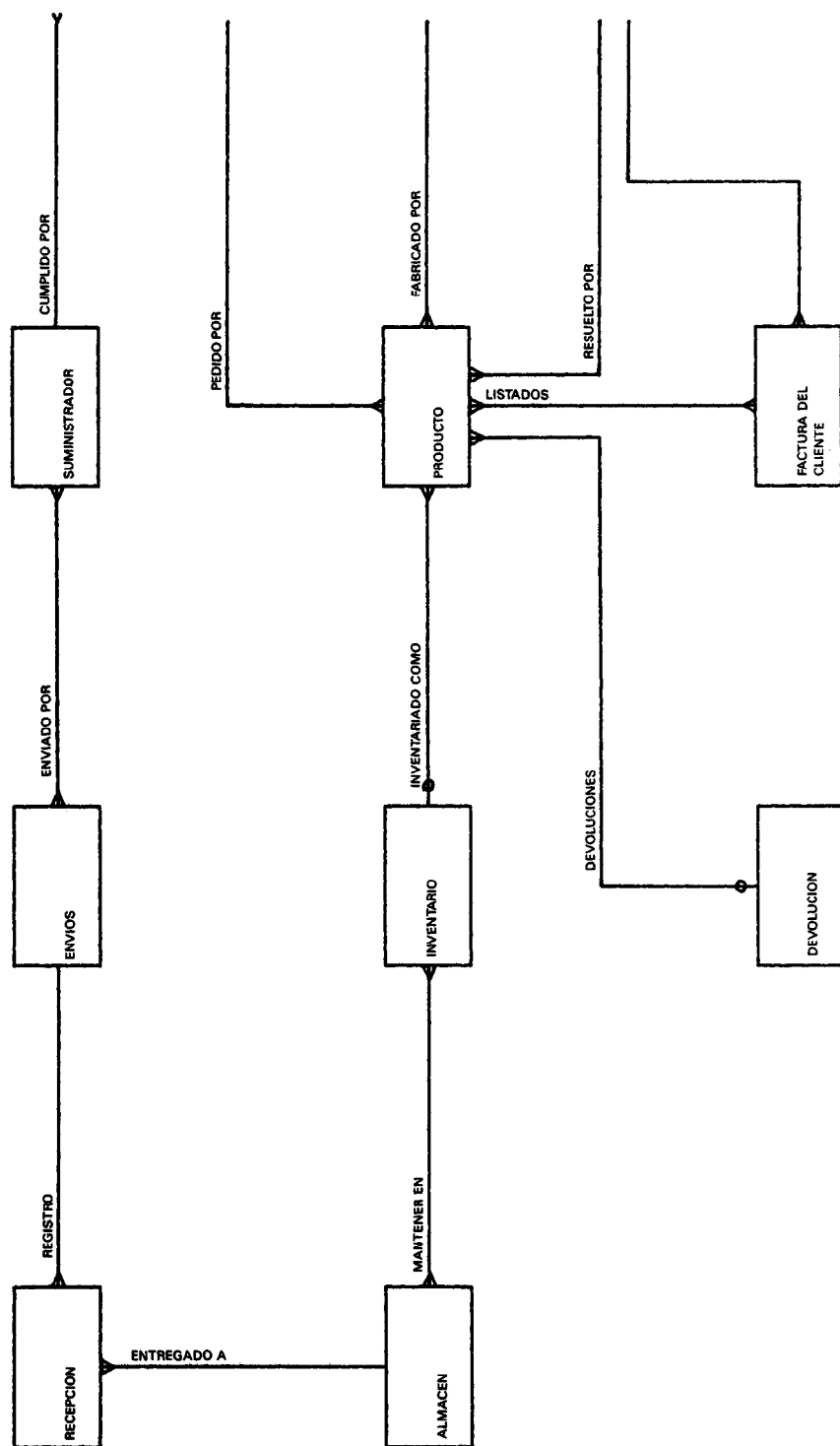


Figura 1.8. Diagrama de entidad/relación obtenido con INFORMATION ENGINEERING
FACILITY TM de Texas Instruments Incorporated

En este estadio, el foco de la CASE está en el diseño automático, porque el análisis y el diseño se consideran como las fases iniciales del ciclo de vida del **software**. Realizando un trabajo mejor en las fases iniciales del ciclo de vida redundará favorablemente en la calidad y costo del sistema. Con la asistencia automática de las herramientas CASE, el tiempo de diseño se reduce substancialmente, a la vez que se producen especificaciones de diseño de alta calidad y sin errores.

El paso siguiente en la evolución de la tecnología CASE fue la unión del diseño automático con la programación automática. Mientras que el diseño automático se relaciona con la asistencia por ordenador a las tareas de análisis y diseño, programación automática significa generadores automáticos de código. La unión implica que del 80 al 90 por ciento del sistema del **software** puede generarse a partir del diseño de diagramas estructurados (ver figura 1.9).

El diseño automático lo soportan las herramientas CASE basadas en ordenadores personales desarrolladas durante la primera mitad de la década de los años ochenta. La programación automática la proporcionan los generadores de aplicaciones y de código de la cuarta generación, la mayoría de los cuales están basados en **mainframes**. Para unir el diseño

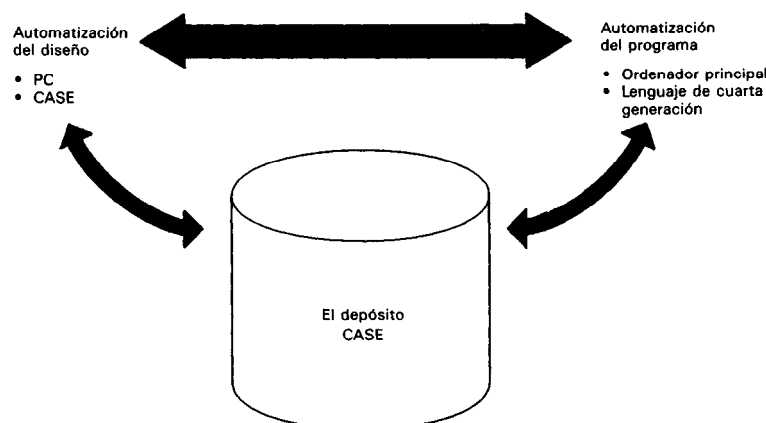


Figura 1.9. El engarce de la automatización del diseño con la automatización del programa significa que puede generarse automáticamente todos los sistemas del software desde las especificaciones gráficas a alto nivel del sistema. Una parte muy importante de esta unión se debe al depósito CASE

automático y la programación automática se requiere un puente entre dos entornos **hardware** (ordenadores personales **mainframes**) y dos tecnologías de **software** (CASE y cuarta generación). Una parte muy importante de este puente será añadir un depósito al entorno CASE.

El depósito CASE es un mecanismo para almacenar y organizar toda la información sobre un sistema de **software**. Parte de esta información es necesaria para la gestión del proyecto, y parte, para la generación automática de código. Incluye información sobre el problema que se va a resolver, sobre el dominio del problema, sobre los procesos del **software** que están siendo utilizados, sobre modelos de datos, sobre modelos de procesos, sobre prototipos, sobre la historia y los recursos del proyecto, y sobre el contexto organizativo.

Además de la unión entre el diseño automático y la programación automática, el depósito CASE convierte por primera vez, el concepto del **software** reutilizable en algo práctico. Los componentes del **software** reutilizable son la clave del vertiginoso aumento de la productividad del **software**. En lugar de ir creando cada vez nuevo **software** o de mejorar el dudoso, los profesionales del desarrollo de **software** pueden utilizar los componentes del **software** reutilizable almacenado en los depósitos CASE. No solamente podrán los profesionales reutilizar los módulos de código fuente, sino que también podrán reutilizar planes de proyectos, modelos de prototipos, modelos de datos y especificaciones de diseño.

OBJETIVO DE LA CASE

Para resumir, la CASE ha cambiado radicalmente la forma de construir los sistemas de **software** al proporcionar tres avances principales:

1. Un entorno de desarrollo interactivo con un tiempo de respuesta rápido, recursos dedicados y una comprobación de errores desde el principio.
2. La automatización de muchas tareas de desarrollo y mantenimiento del **software**.
3. Una programación visual proporcionada por potentes interfaces gráficos.

El fin último de la tecnología CASE es automatizar todo el ciclo de vida del **software** mediante un conjunto de herramientas de **software** integradas. No obstante, para el estado tecnológico actual se trata de una meta a conseguir, no de una realidad.

UN CAMBIO DE ACTITUD

Quizá la mejor forma de describir hoy la tecnología CASE es como un cambio de actitud. La CASE representa un cambio fundamental en la actitud hacia el desarrollo del **software**, que es un paso necesario para ir preparando el camino de los avances revolucionarios de las tecnologías del **software**.

La CASE no es el primer cambio significativo en la actitud hacia el proceso del **software**. A finales de los años sesenta, la introducción de las técnicas estructuradas ya representó un cambio en la actitud. Reaccionando ante el nacimiento de la crisis del **software**, entonces el punto de vista era que el desarrollo de **software** era un proceso complicado, tedioso y propenso al error. No debería ser tratado como arte privado de cada programador individual, sino como un proceso disciplinado y mecánico. El concepto de estructuración hacía hincapié en los estándares y procedimientos de programación y en el control de la gestión. La actitud era que la productividad y calidad del **software** podrían controlarse mejor a través de la disciplina, la formalidad y la estandarización.

Después, a finales de los años setenta volvió a suceder otro cambio de actitud en el proceso del **software**. Esta vez el punto de vista era que la productividad del **software** podría incrementarse con la utilización de herramientas que simplificasen el desarrollo de **software**. La crisis del **software** podría aliviarse si muchas personas (y menos preparadas técnicamente) pudieran desarrollar ciertos tipos de **software** para aplicaciones. Este cambio estuvo marcado por la introducción de las herramientas de cuarta generación y la programación dirigida al usuario final.

Actualmente, desde finales de los años ochenta, vemos otro cambio de actitud ante el proceso del **software**. En un sentido, este cambio representa un paso atrás, hacia una visión más conservadora del desarrollo de **software**. De nuevo, el punto de vista es que el desarrollo de **software** es una tarea difícil que requiere una formulación disciplinada y mecánica, y

que para la realización de sistemas complejos son necesarios profesionales del desarrollo de **software** altamente preparados. Aunque la programación orientada al usuario final y las herramientas de cuarta generación han realizado una valiosa contribución, la experiencia ha demostrado que tienen un puesto limitado en el desarrollo de **software**. En su mayoría son útiles en la construcción de aplicaciones comerciales de tamaño pequeño y medio. No son de propósito general.

El cambio más reciente es un redescubrimiento de los principios fundamentales de la ingeniería del **software** definidos durante los últimos 25 años. Aparecen de nuevo las técnicas estructuradas pero esta vez reforzadas con el concepto de la automatización del proceso de **software**. En este estadio, la CASE es un renacimiento de las técnicas estructuradas.

La actitud de la CASE es que el desarrollo y el mantenimiento del **software** debería verse como una actividad formal y disciplinada capaz de una comprobación mejor de la exactitud y automatización del proceso. La única forma real para incrementar significativamente la calidad y la productividad del **software** es a través de la automatización.

LA DIRECCION DE LA AUTOMATIZACION

El tema de este libro es la automatización del **software**. Se tratará la tecnología CASE, el vehículo con el que se mueve el proceso del **software** en dirección a la automatización. Exploraremos lo que es la tecnología CASE, lo que no es y lo que puede ser en el futuro. El cuadro 1.2 presenta las definiciones básicas de la tecnología CASE.

En la parte I hemos introducido las materias de la automatización del **software** y de la ingeniería de **software** asistida por ordenador. En la parte II definimos los componentes de un sistema CASE en los capítulos 2 y 3 y describimos los tipos básicos de herramientas CASE en el capítulo 5. En el capítulo 4 se tratan las plataformas **hardware** para los sistemas CASE. En la parte III exploramos el impacto de la CASE en el proceso del **software**. En el capítulo 6 presentamos las experiencias de algunos usuarios de herramientas CASE. En el capítulo 7 se sugieren métodos para introducir la CASE en las organizaciones. En el capítulo 8 se analiza cómo la tecnología CASE cambia el ciclo de vida del **software** tradicional. La relación

entre la tecnología CASE y las tecnologías de tercera, cuarta y quinta generación se aclaran en el capítulo 9.

Cuadro 1.2. Definiciones

CASE: computer aided software engineering, ingeniería de software asistida por ordenador.

Tecnología CASE: una tecnología de software que proporciona una disciplina automatizada para el desarrollo, mantenimiento y gestión del software; incluye las metodologías estructuradas automatizadas y las herramientas automatizadas.

Herramienta CASE: una herramienta de software que automatiza (al menos en parte) una tarea en particular del ciclo de vida del software.

Sistema CASE: un conjunto integrado de herramientas CASE que comparten un interfaz común de usuario y se procesan en un entorno común de ordenador.

Juego de herramientas CASE: un conjunto de herramientas integradas diseñadas para trabajar conjuntamente y automatizar total o parcialmente una fase del ciclo de vida del software o una clase particular de función del software.

Banco de trabajo CASE: un conjunto integrado de herramientas CASE diseñadas para trabajar conjuntamente y automatizar (o proporcionar ayuda automatizada) totalmente el ciclo de vida del software, que incluye el análisis, el diseño, la codificación y la prueba.

Metodología CASE: una metodología estructurada automatizable que define una formulación técnica y disciplinada para todos o algunos de los aspectos del desarrollo y del mantenimiento del software.

Cuadro 1.2. Definiciones (continuación)

Compañero de metodología CASE: un conjunto de herramientas CASE que automatizan las tareas en una metodología CASE particular y/o automatiza la producción de la documentación y otros requisitos exigidos por la metodología.

Estación de trabajo CASE: una estación de trabajo o un ordenador personal de 32 bits equipada con herramientas CASE que automatiza varias funciones del ciclo de vida.

Plataforma hardware CASE: sistema de arquitectura hardware que proporciona una plataforma de operación a las herramientas CASE.

En la parte IV exploramos cómo será el desarrollo de **software** a finales de esta década. El entorno de desarrollo de **software** en los años noventa se presenta en el capítulo 10. En el capítulo 11 describimos el entorno amigable, que es un interfaz inteligente de usuario para el entorno de desarrollo de **software** de los años noventa. El conductor de la metodología, un sistema experto que proporciona una guía metodológica a través del ciclo de vida del **software** de los años noventa se describe en el capítulo 12. En el capítulo 13 explicamos cómo y por qué la reutilización del **software** será la estrategia del desarrollo de **software** más importante de los años noventa. El libro se cierra en la parte V con un corto epílogo en el capítulo 14.

PARTE 2

LOS COMPONENTES DE UN SISTEMA CASE

EL ENTORNO DE DESARROLLO DE SOFTWARE CASE

EL NUEVO ENTORNO DE DESARROLLO DE SOFTWARE

Según se han ido sucediendo los nuevos avances tecnológicos en **hardware** y **software**, el entorno de desarrollo ha ido cambiando desde un entorno individual en los años cincuenta y principios de los sesenta hacia el modo de proceso por lotes (*batch*) en los años sesenta y principios de los setenta, y después al modo de tiempo compartido (*time sharing*) a finales de los años setenta y comienzos de los ochenta [1]. A finales de los ochenta, un nuevo modo, basado en las *estaciones de trabajo personales* se ha impuesto como entorno de desarrollo de **software**. La introducción de potentes herramientas en las estaciones de trabajo con el acompañamiento de metodologías estructuradas transforma y automatiza el proceso del desarrollo de **software**.

La **estación de trabajo CASE** es un entorno completo, que incluye **hardware** y **software**, cuya función es proporcionar asistencia por ordenador para la producción, mantenimiento y gestión de los sistemas de **software**.

Además, la estación de trabajo proporciona el procesamiento de textos, el almacenamiento y la recuperación de la información, el correo electrónico y las funciones de ayuda y calendario. Es un ordenador personal

dedicado a proporcionar el soporte máximo posible individualmente al profesional del desarrollo de **software**.

El objetivo principal de la utilización de las estaciones de trabajo CASE es incrementar la productividad del **software**. Inmediatamente después, un segundo objetivo es incrementar la calidad del **software**. Otros objetivos frecuentemente citados se encuentran en el cuadro 2.1.

EL BANCO DE TRABAJO CASE

Un amplio conjunto de herramientas de **software** inteligentes e integradas llamadas **banco de trabajo CASE** compone el **software** del entorno de la estación de trabajo. Los hay a la medida de las preferencias individuales del programador de desarrollo; de asistencias a tareas especializadas, como la gestión del proyecto, el diseño y el mantenimiento, y reemplazan a las herramientas tradicionales basadas en el proceso por lotes y a los lenguajes de la tercera generación de los años sesenta y setenta.

Los bancos de trabajo CASE difieren de los primitivos entornos de programación, como UNIX e INTERLISP, en su amplia cobertura del ciclo de vida del **software**. El UNIX es un sistema operativo de tiempo compartido en el cual un formato uniforme de fichero es el principal medio de integración en un amplio rango de herramientas de programación [2]. El UNIX es un entorno de programación de propósito general que no soporta un lenguaje de programación en particular ni una metodología de desarrollo de **software**. El INTERLISP, por otra parte, es un entorno de programación que soporta solamente la programación en lenguaje LISP. Este da a INTERLISP la ventaja de adecuar y optimizar las herramientas a un único lenguaje de programación [3]. Todas las herramientas INTERLISP están escritas en LISP y proporcionan un sistema operativo fuertemente integrado, funciones de utilidad, editores y correctores de errores.

Mientras que los entornos de programación de los años setenta y ochenta se concentraron en herramientas para las fases de codificación e implantación, los bancos de trabajo CASE proporcionan potentes herramientas para la especificación, el diseño, la implantación, la comprobación y la documentación. Son un entorno de soporte de **software** de propósito general destinado a soportar toda la gama de tareas del **software** y su gestión.

Cuadro 2.1. Los objetivos de las estaciones de trabajo

- Maximizar la productividad del programador .
- Aumentar la calidad del **software** y reducir los errores.
- Simplificar el proceso de desarrollo del **software**.
- Posibilitar un empleo más eficiente de los recursos del ordenador.
- Reducir los costos del **software**.
- Aumentar la fiabilidad del entorno de desarrollo.
- Proporcionar un entorno de desarrollo que tenga un interfaz humano mejor con el sistema.
- Revolucionar los procesos de desarrollo de **software**.
- Automatizar la generación de la documentación del **software**.
- Automatizar la generación de código ejecutable.
- Soportar unos prototipos rápidos.
- Automatizar la comprobación durante el proceso de desarrollo.
- Automatizar la gestión del proyecto.
- Formalizar y estandarizar el proceso de desarrollo de **software**.
- Normalizar la documentación del **software**.
- Integrar las herramientas de desarrollo.
- Promover la reutilización del **software**.
- Promover un control general sobre el desarrollo y el mantenimiento del **software**.
- Mejorar la portabilidad del software a través de los entornos.

Para cumplir los objetivos de mejorar la productividad y la simplicidad de los procesos de desarrollo de **software**, el banco de trabajo CASE no puede consistir solamente en una colección con las mejores herramientas de los años setenta y ochenta. Esas herramientas no se diseñaron para utilizar en los entornos de los ordenadores personales permitiendo la particularización individual para cada usuario de la estación de trabajo. Tampoco fueron diseñadas para emplear potentes utilidades gráficas que potencien el interfaz humano. Ni fueron diseñadas para utilizarse en cooperación con otras para conseguir unidas todos los aspectos del proceso de **software**. Ni fueron diseñadas para capturar la información sobre el progresivo desarrollo de **software** y la evolución de los productos del **software**. Finalmente, no fueron diseñadas como herramientas inteligentes capaces de realizar muchas de las tareas del desarrollo por sí mismas.

Lo que hace el banco de trabajo CASE es ensamblar las herramientas que pueden proporcionar un soporte funcional completo al proceso del **software**:

- * Creación de los requerimientos gráficos de sistema y de las especificaciones de diseño.
- * Validación, análisis y referencias cruzadas de la información del sistema.
- * Almacenamiento, gestión y comunicación de la información del sistema y de la gestión del proyecto.
- * Construcción del prototipo de sistemas y simulación de sistemas.
- * Generación de código de sistemas y de documentación.
- * Imposición de los estándares y de los procedimientos.
- * Prueba, validación y análisis de los programas.
- * Interfaces para la salida a los diccionarios y las bases de datos.

Para proporcionar un soporte total de **software**, un banco de trabajo CASE completo debe tener las siguientes prestaciones:

- * Gráficos.
- * Comprobación de errores.
- * Depósito de información.
- * Juego de herramientas totalmente integradas.
- * Cobertura total del ciclo de vida.
- * Soporte de prototipos.
- * Generación automática de código.
- * Soporte de metodología estructurada.

En este capítulo trataremos las cuatro primeras competencias de un banco de trabajo CASE, y en el siguiente, las restantes.

LAS CAPACIDADES GRAFICAS

Cuando se utiliza una estación de trabajo CASE, lo primero que impresiona es el editor gráfico. La facilidad con la que se consigue que los objetos aparezcan, desaparezcan y se muevan por la pantalla de la estación de trabajo con sólo apretar un botón es increíble. Sin embargo, tras los gráficos hay una prestación muy importante. El mejor interfaz gráfico es el que puede hacer al usuario más productivo.

Definición del programa del sistema de suscripciones

El sistema a suscripciones procesa las transacciones de suscripciones contra el fichero de suscripciones. Las transacciones son de tres tipos: nuevas suscripciones, renovaciones y cancelaciones. Cada transacción primero se valida y después se procesa contra el fichero maestro. En las sucesivas suscripciones, se crea un registro de cliente y se genera una anotación en el balance. En las renovaciones, se actualiza la fecha de expiración y se genera una anotación en el balance. En las cancelaciones se marca el registro como cancelado y se obtiene un saldo.

Figura 2.1. La descripción textual de las especificaciones del sistema de suscripciones

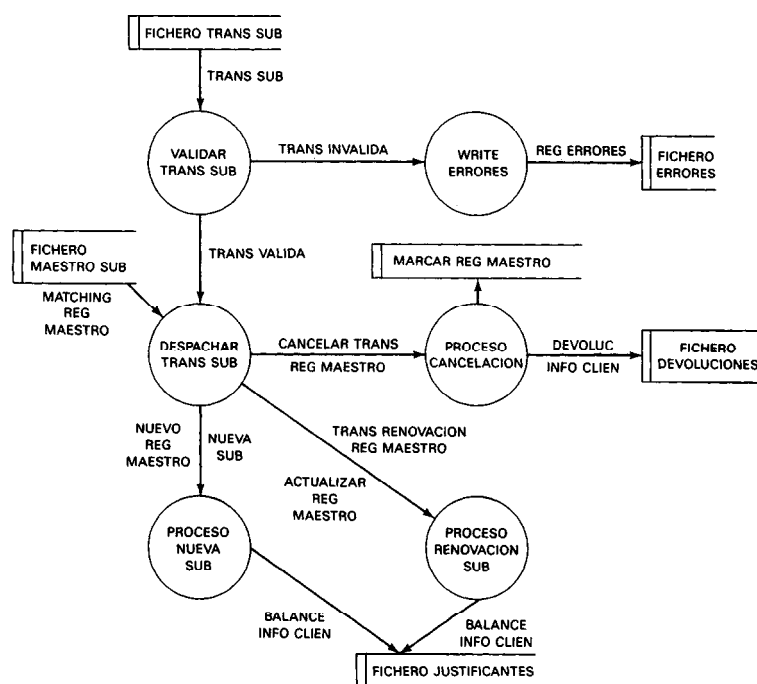


Figura 2.2. Diagrama de flujo de datos utilizado para explicar gráficamente las especificaciones del sistema de suscripciones

La figura 2.1 es una descripción narrativa de las especificaciones de un sistema de suscripciones. La figura 2.2 también describe las especificaciones de sistemas de suscripciones, pero de una forma gráfica (un diagrama de flujo de datos, DFD). Mucha gente prefiere la forma gráfica, ya que la mente humana está orientada a la imagen. Mientras la narración es unidimensional, la imagen es multidimensional, tomando prestadas algunas de las propiedades del mundo físico, como el tamaño, la forma y el color. Ya que el lenguaje de las imágenes es más rico que el de los textos, se puede representar mucha más información en una imagen que en un texto y el lector puede comprender el significado más rápidamente [4].

LA NECESIDAD DE LOS DIAGRAMAS

Las representaciones gráficas (o diagramas) siempre han desempeñado un papel muy importante en el desarrollo del software. Los diagramas se utilizan para definir las especificaciones de un programa y representar el diseño de los programas. Proporcionan el proyecto original para la implantación del diseño en código y además son una forma importante de documentación del software.

Los diagramas claros son una parte esencial en el diseño de sistemas complejos y en el desarrollo de programas. Si sólo hay una persona desarrollando el diseño de un sistema o de un programa, los diagramas ayudan a un razonamiento claro. Una técnica de diagramación pobre en presentaciones puede inhibir el razonamiento, en cambio una buena puede acelerar el trabajo y aumentar la calidad de los resultados.

Cuando trabajan muchas personas en un sistema de software, los diagramas son una herramienta esencial de comunicación. Se necesita una técnica de diagramación formal para que los desarrolladores intercambien ideas y reúnan con precisión los distintos componentes del sistema.

Una vez corregidos los errores, los diagramas son una herramienta muy valiosa para saber cómo otro programador intentaba que funcionara el programa y para localizar el origen de los errores y el impacto del cambio.

Cuanto más grande sea el equipo de personas, mayor es la necesidad de precisión en la diagramación. Es muy difícil, casi imposible, que los componentes de un gran equipo puedan comprender en detalle el trabajo de

los demás. Para resolver este problema, cada componente del equipo deberá estar familiarizado con el sistema, al menos con una visión general, y ver donde encaja su parte en el todo. Deberá tener claros y perfectamente definidos sus contactos y deberá ser capaz de ver el sistema o programa con varios grados de detalle. Es más, cada parte del sistema deberá describirse en la misma forma estandarizada para facilitar la comunicación y evitar las confusiones que pueden conducir a error.

Los diagramas son realmente un lenguaje de modelado de **software**, porque ofrecen para describirlo una forma concisa y sin ambigüedades de representación. Son fundamentales en una buena documentación de sistemas (ver cuadro 2.2). Además, son tan fundamentales para el análisis y el diseño del **software** que las diferentes metodologías pueden caracterizar a partir de las técnicas de diagramación utilizadas para modelar un sistema de **software**.

Cuadro 2.2. Requisitos de una buena documentación

Una buena documentación del sistema y de los programas es un componente esencial de la filosofía estructurada. La buena documentación cumple los siguientes requisitos:

- * Mejora la comprensión de los sistemas de **software**.
- * Es fácil y poco costosa de realizar y de actualizar, y puede producirse automáticamente.
- * Proporciona una visión de alto nivel del sistema (programa), explicando su propósito y las relaciones entre los diversos componentes (datos y procedimientos).
- * Describe detalladamente los componentes de los datos y procedimientos.
- * Proporciona un proyecto para representar el problema de los requerimientos del diseño de un sistema y del traslado del diseño al código de programa.

VENTAJAS DE LAS TECNICAS DE DIAGRAMACION ESTRUCTURADA

Los formalismos gráficos utilizados en la construcción de los DFD y en el modelo entidad/relación, cumplen las condiciones básicas para producir una buena documentación. Primero combinan las anotaciones gráficas y de texto para facilitar la comprensión de los sistemas. Los gráficos son especialmente útiles porque tienden a ser menos ambiguos que las descripciones por escrito. Además, al tender a ser más concisos, los gráficos pueden dibujarse en mucho menos tiempo de lo que se tardaría en escribir un documento conteniendo la misma cantidad de información.

Segundo, las técnicas de diagramación estructurada son de jerarquía descendente (**Top-down**). Esto significa que pueden soportar una formulación de desarrollo estructurado “arriba-abajo”. Pueden describir un sistema, un programa o una estructura de datos variando el grado de detalle durante cada paso de la descomposición del proceso proporcionando una forma estandarizada de describir la lógica del proceso y las estructuras de los datos.

Finalmente, las técnicas de diagramación estructurada ayudan a los programadores a tratar con grandes volúmenes de detalles generados durante el proceso de desarrollo del programa. Proporcionan una representación lógica más que una orientada a la programación, lo que es mucho más significativo para los usuarios y para los directivos no técnicos.

USOS DE LOS DIAGRAMAS ESTRUCTURADOS

Las técnicas de diagramación se han desarrollado a la par que las metodologías de programación. En los años cincuenta y sesenta, se utilizaron los organigramas (**flowcharts**) para planificar detalladamente la complicada lógica de los programas. Perdieron terreno al no poder ofrecer una visión a alto nivel de la estructura de los programas. En los años setenta se extendieron las técnicas estructuradas y con ellas se introdujeron las técnicas estructuradas de diagramación (como los DFD y los diagramas estructurados).

Unas técnicas de diagramación se utilizan como herramientas de análisis; otras, como herramientas de diseño de ficheros y de bases de datos.

Cuadro 2.3. Técnicas de diagramación estructurada

	Tipo de documentación	
	de procedimiento	de datos
Herramientas de análisis	×	
Diagrama de flujo de datos	×	
Diagrama de flujo de control	×	
Tabla de decisión, árboles	×	×
Matrices	×	
Diagrama de dependencia	×	
Diagrama de descomposición		
Herramientas de diseño		
Diagrama de estructura	×	
Diagrama de acción	×	×
Diagrama de Warnier-Orr	×	
Diagrama de transmisión de estado	×	
Tablas de decisión, árboles	×	
Pseudocódigo	×	
Organigramas	×	
Diseños de pantalla		×
Flujo de diálogo		×
Diseño de ficheros y de bases de datos		
Modelo de datos		×
Diagrama de entidad/relación		×
Estructura de datos		×
Registros lógicos		×
Ficheros y bases de datos físicos		×

En el cuadro 2.3 se exponen algunos ejemplos de técnicas de diagramación estructuradas de acuerdo a estas categorías.

Los DFD, los diagramas de descomposición y los diagramas de flujo de control son ejemplos de herramientas de análisis (ver figuras 2.3, 2.4 y 2.5). Por ejemplo, el DFD se utiliza para describir la transformación que los datos experimentan en su flujo a través del sistema. El diagrama de flujo de control añade la información de control necesaria para describir los

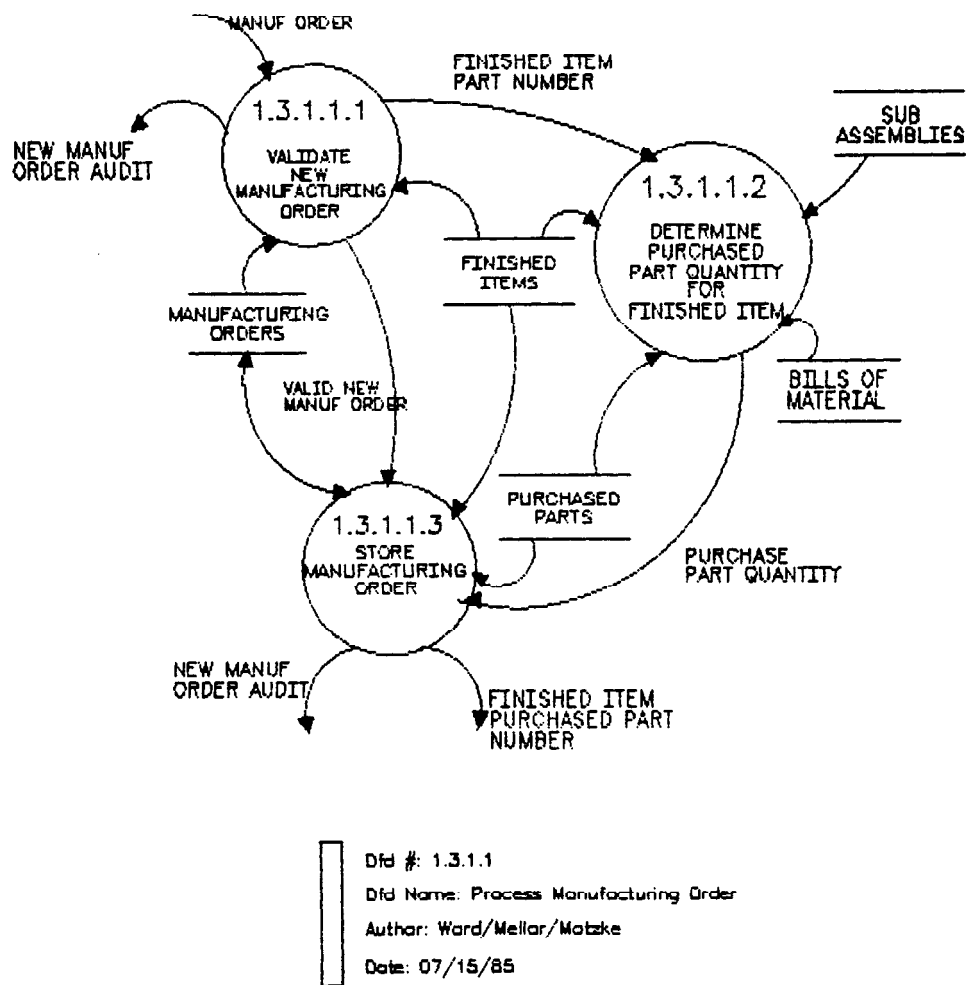


Figura 2.3. Diagrama de flujo de datos dibujado por el ANALIST DE-SIGNER TOOLKIT de Yourdon Software Engineering Company

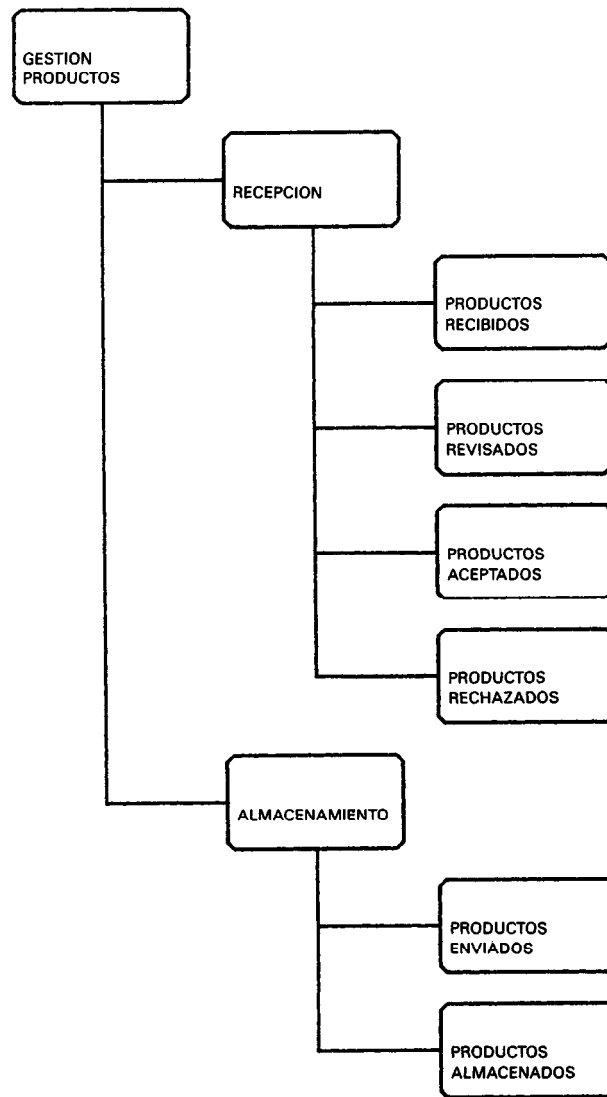


Figura 2.4. Diagrama de descomposición dibujado utilizando el INFORMATION ENGINEERING FACILITY™ de Texas Instruments Incorporated

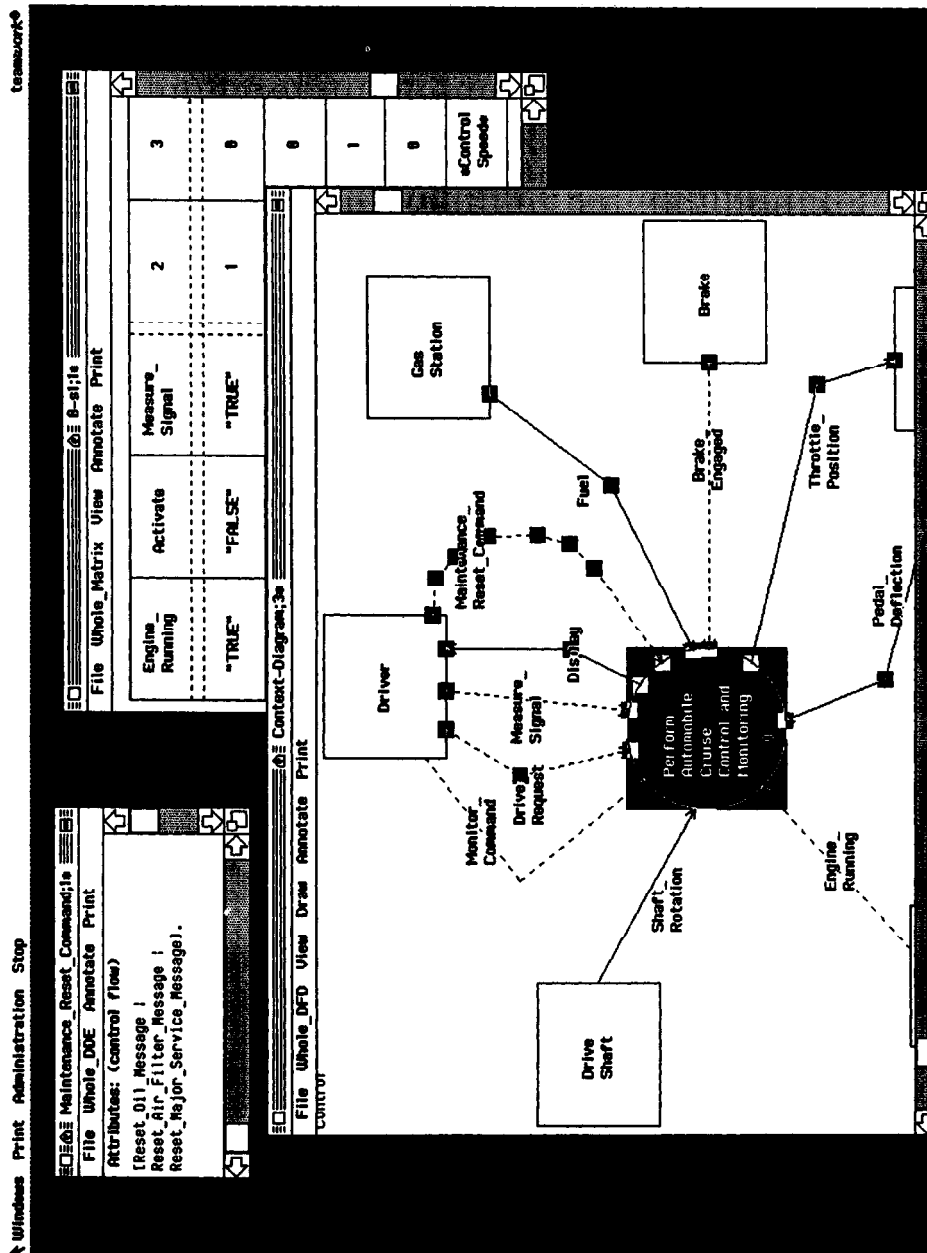


Figura 2.5. Una pantalla con ventanas múltiples de TEAMWORK de Cadre Technologies. Muestra un diagrama de contexto que es un tipo de diagrama de flujo de control utilizado en el desarrollo de los sistemas de tiempo real

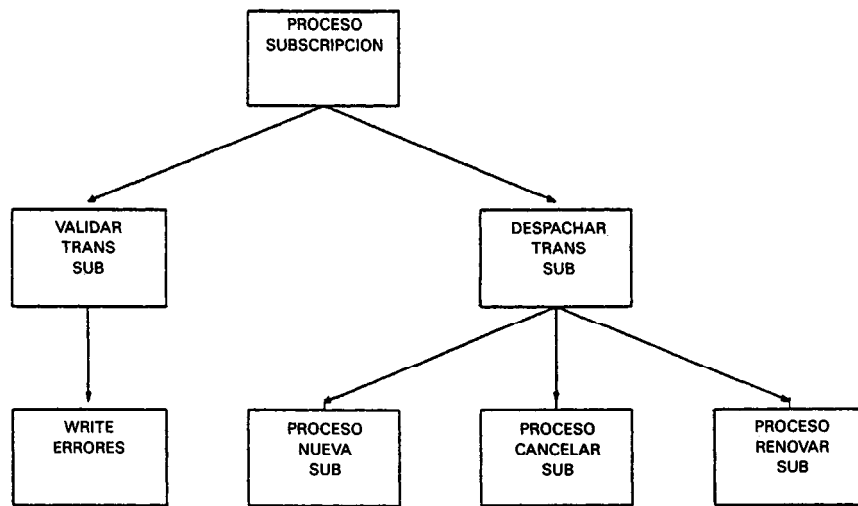


Figura 2.6. Diagrama de estructura del sistema de suscripciones obtenido con el INFORMATION ENGINEERING WORKBENCH de KnowledgeWare

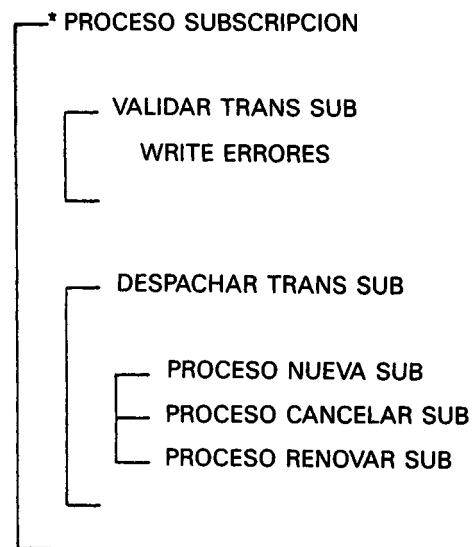


Figura 2.7. Diagrama de acción del sistema de suscripciones, obtenido con el INFORMATION ENGINEERING WORKBENCH de KnowledgeWare. Es un tipo de diagrama estructurado en árbol. El diagrama de estructura de la figura 2.6 y el diagrama de acción muestran la misma información aunque de forma distinta

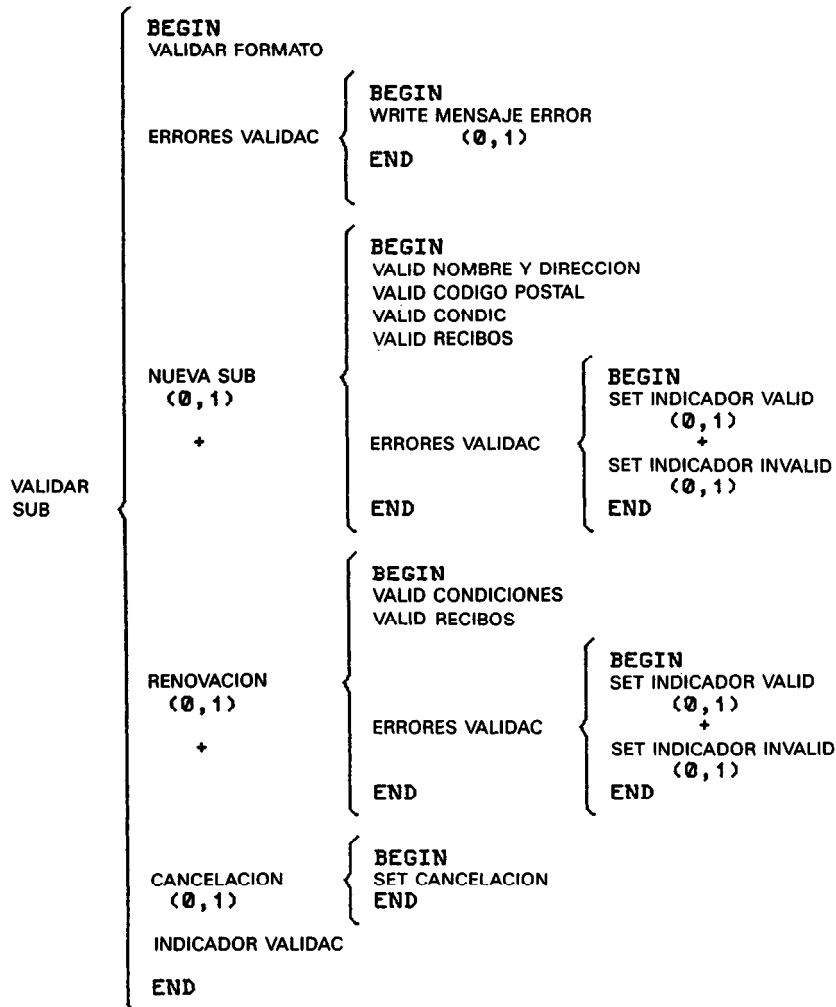


Figura 2.8. Diagrama de Warnier-Orr obtenido con DESIGNAID de Nastec Corporation

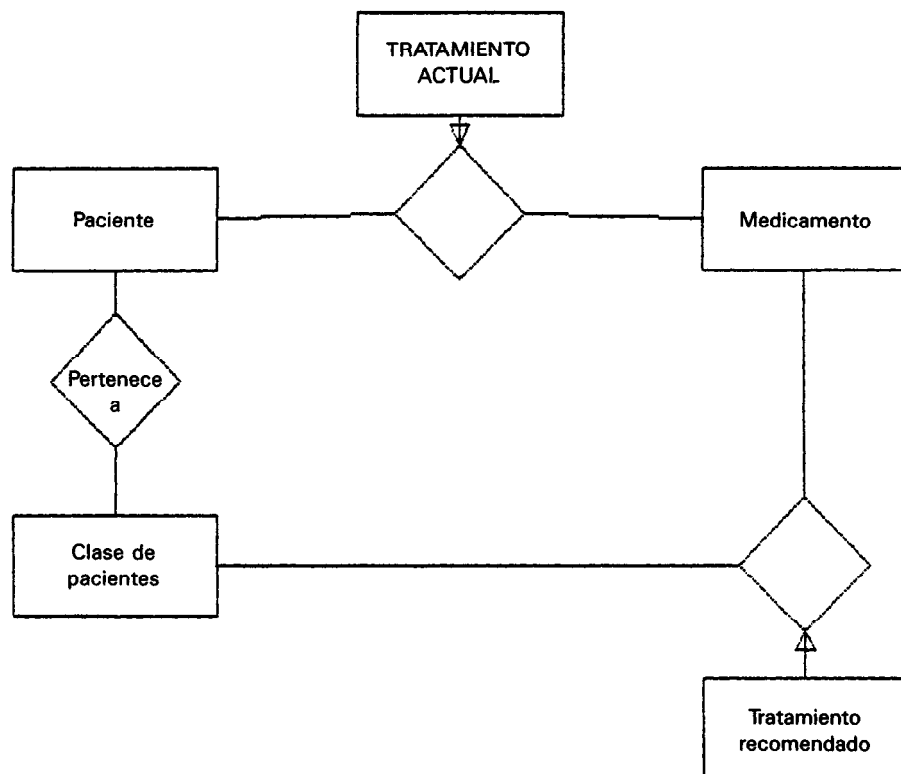


Figura 2.9. Diagrama de entidad/relación obtenido con el ANALYST/DESIGNER TOOLKIT de Yourdon Software Engineering Company

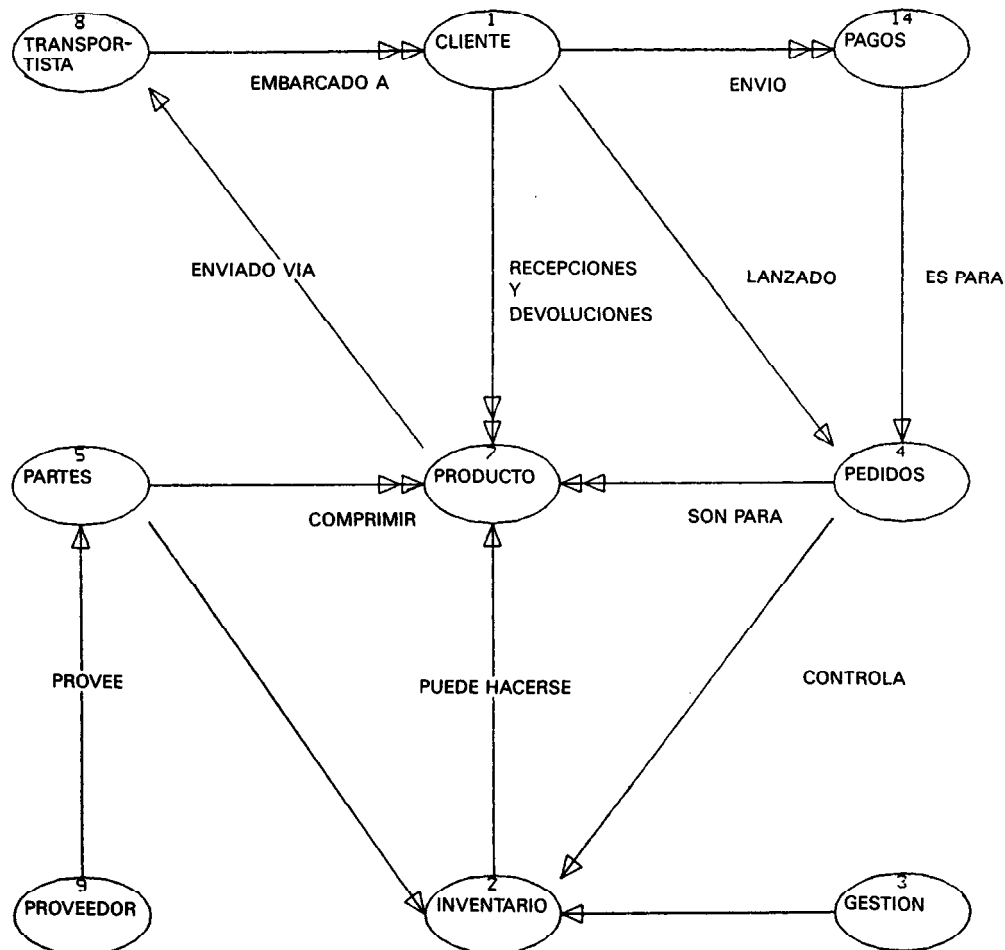


Figura 2.10. Diagrama de modelo de datos obtenido con el EXCELERATOR de Index Technology Corporation

sistemas de tiempo real. El diagrama de descomposición se utiliza para identificar la estructura de una organización y sus funciones comerciales básicas.

El diagrama de estructura (**structure chart**) de la figura 2.6, se utiliza como una herramienta de diseño para definir la organización jerárquica de un programa, incluyendo sus componentes de procedimiento y los interfaces de datos que los conectan. Los diagramas de acción y los de Warnier-

Orr (ver figuras 2.7 y 2.8) pueden utilizarse para mostrar la organización jerárquica del programa también como una lógica detallada de procedimiento.

Los diagramas de entidad/relación y los de modelo de datos se utilizan para describir estructuras de datos (ver figuras 2.9 y 2.10). Cada entidad de datos, sus atributos y sus relaciones con otras entidades de datos puede definirse con estos diagramas. Además, la cardinalidad podría ser un indicativo para distinguir las relaciones opcionales y obligatorias cuando se capturen las reglas de las actividades del negocio.

VISION DEL SISTEMA MULTIPLE

Las técnicas de diagramación se utilizan para dar una visión de alto nivel y detallada de un sistema. Por ejemplo, los diagrama de estructura dan una panorámica de alto nivel de un programa. Pueden utilizarse para explicar en términos generales las principales funciones que realizan el programa y que datos de alto nivel y procedimientos componen el programa. Por otra parte, los diagramas de acción y de pseudocódigo ofrecen una visión del programa a nivel de instrucción (ver figura 2.11). Muestran donde la variable se inicializa, se comprueba o se referencia en el programa.

Una visión de alto nivel y/o detallada de un programa es importante según el propósito del lector. Si el lector está buscando un error, la documentación detallada puede guiarle a la localización exacta del error. Si la persona desea determinar en cual de varios programas se realiza una determinada función, una visión de alto nivel puede ser la mejor ayuda.

Los diferentes **diagramas** son más adecuados para la representación de alto nivel o la detallada. Por ejemplo, los diagramas de estructura son los más adecuadas para la representación global o de alto nivel que introduce al lector en el programa. Pero a un nivel de detalle, los diagramas de estructura están demasiado llenos de cajas y errores para presentar claramente los detalles.

Para representar un sistema completo con diagramas, se necesitan una estructura del proyecto detallada y de alto nivel, y una estructura de datos detallada y de alto nivel. La visión estructurada del proceso (o del sistema) identifica los componentes del proceso y las relaciones entre ellos mostrando

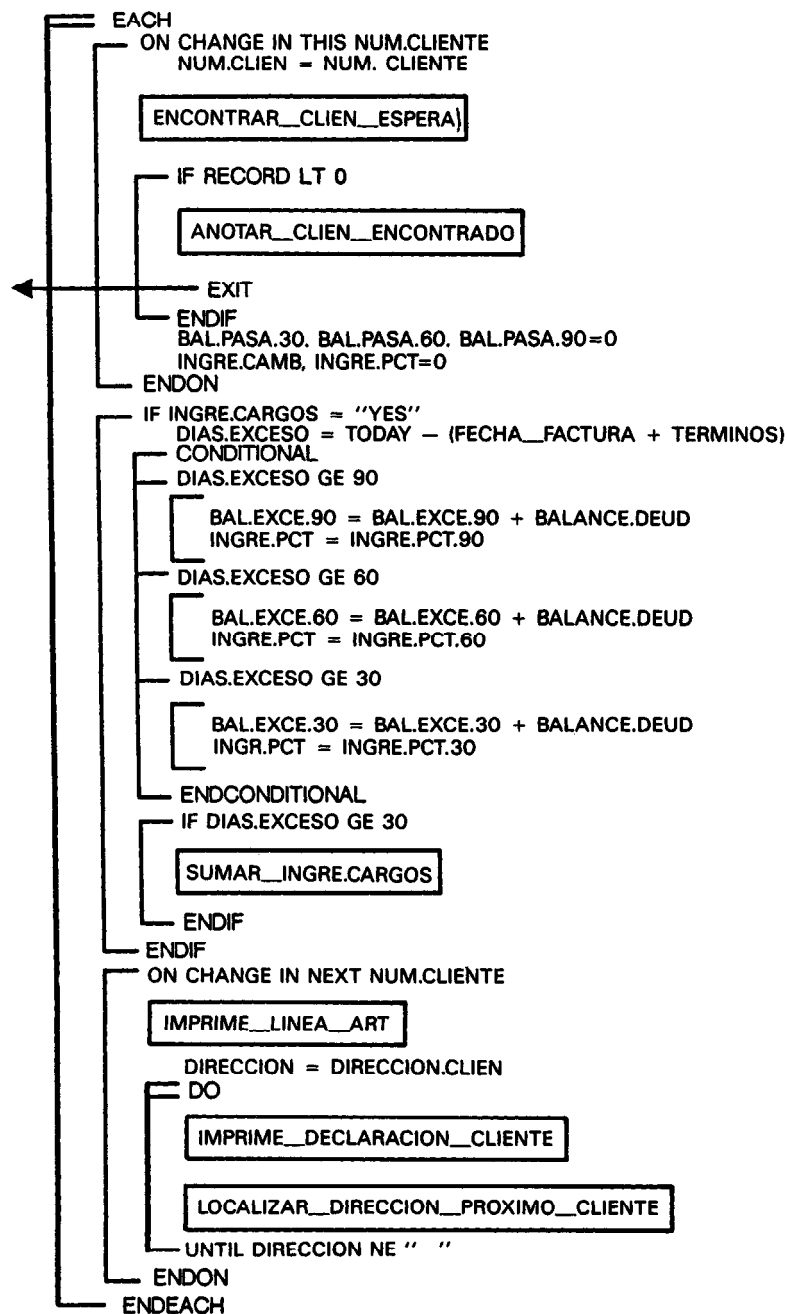


Figura 2.11. Diagrama de acción utilizando el BUILDER LANGUAGE a Cortex Corporation

el flujo de los datos, el flujo de control, y el tiempo y la secuencia de la información. La visión de la estructura de los datos describe las entidades y sus atributos, así como las relaciones entre las entidades. Con la ayuda de ambas visiones, el programador puede ver cómo los datos se derivan o utilizan en el proceso; y en el caso de sistemas en tiempo real, cómo los hechos externos (señales) ocasionan los cambios de estado del sistema.

EL TRIO ESENCIAL DE TIPOS DE DIAGRAMAS

Aunque hay en uso muchas técnicas distintas de diagramación estructurada, hay tres tipos de diagramación necesarios para representar un sistema de **software** [5, págs. 1-22]:

- 1.—**Diagrama de flujo de datos, DFD:** un diagrama conocido y familiar utilizado durante el análisis para definir los componentes del problema y diseñar un primer esbozo de los componentes del programa y del paso de datos entre ellos.
- 2.—**Diagrama de los modelos de datos:** un diagrama utilizado durante el proceso de modelado de los datos para representar los registros y entidades de los datos y las asociaciones lógicas entre ellos.
- 3.—**Diagramas de estructura de árbol:** un diagrama jerárquico creado durante un diseño del sistema/programa para definir la arquitectura global del sistema para mostrar los modelos/programas y las relaciones entre ellos.

Como mínimo, un banco de trabajo CASE debe proporcionar la posibilidad de dibujar automáticamente una versión actualizada de cada uno de estos tres tipos de diagramas. Para representar sistemas embebidos de tiempo real, se necesitan otros tipos de diagramas (como los de flujo de control, árboles de decisión, de estado finito) para mostrar los requisitos de control, de tiempos y de secuencias.

DIAGRAMACION AUTOMATICA

El problema con la representación gráfica es el tiempo que lleva dibujarla manualmente y el tiempo que lleva cambiarla manualmente. La auto-

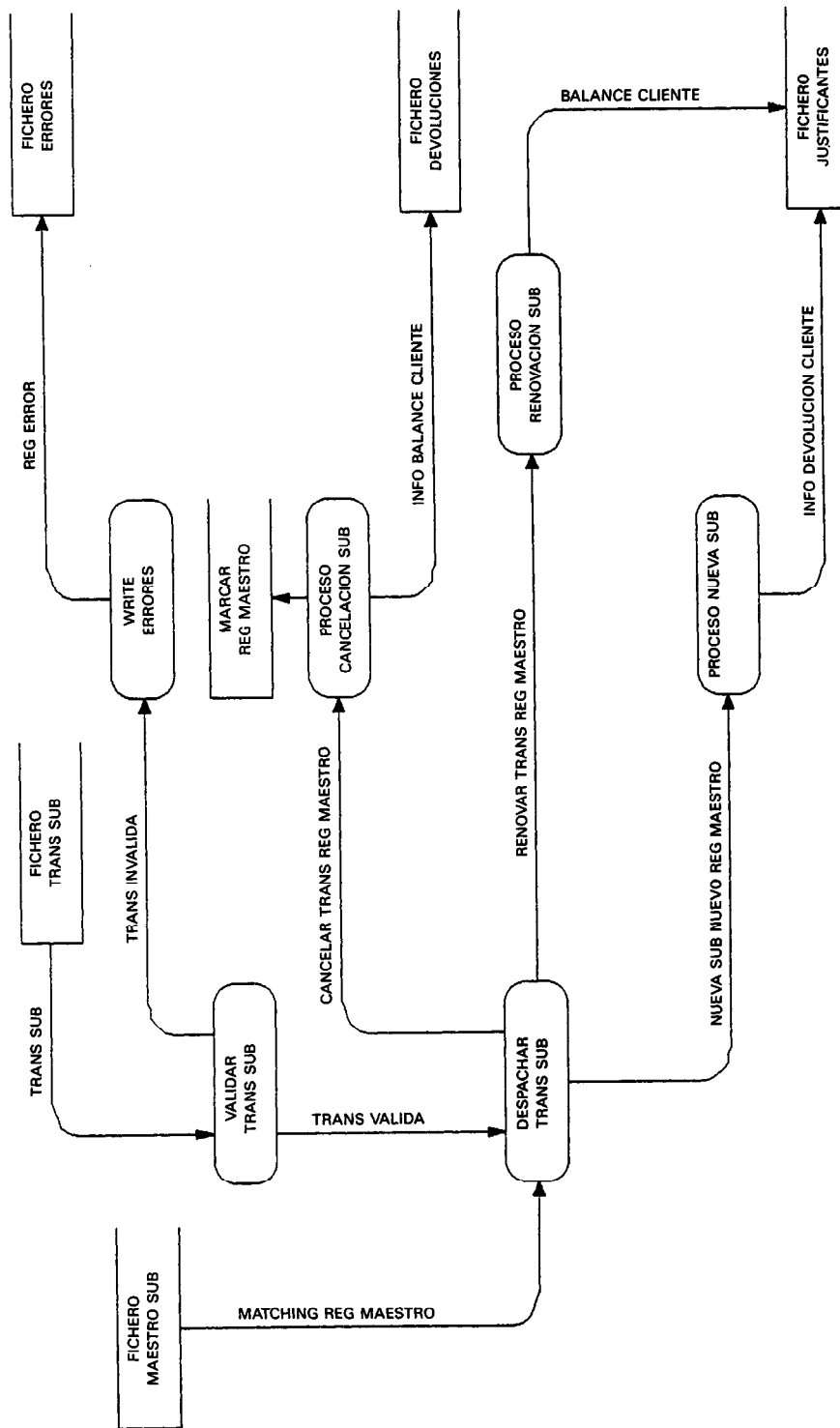


Figura 2.12. Diagrama de flujo de datos dibujado con el INFORMATION ENGINEERING WORKBENCH de KnowledgeWare. Es equivalente al diagrama dibujado manualmente de la figura 2.2

matización elimina este problema. Cuanto menos tiempo se emplee en dibujar más productivo será el programador.

El diagrama de datos dibujado manualmente en la figura 2.2 se muestra en la figura 2.12, pero dibujado automáticamente con una **CASE workbench** (banco de trabajo).

La diagramación interactiva en la pantalla de un ordenador tiene importantes ventajas. Acelera enormemente el proceso; fuerza la estandarización, y puede realizar automáticamente la documentación del proceso. En el cuadro 2.4 se resumen las ventajas de las herramientas de diagramación estructurada automática.

Cuando un analista o programador está creando un procedimiento, continuamente está realizando modificaciones al diseño según las ideas van tomando forma. El analista continuamente está refinando las estructuras de control, la lógica y las estructuras de los datos; inserta nuevas estructuras entre las existentes; añade nuevas condiciones; mueve o duplica los procesos lógicos; guarda algunos procedimientos para utilizarlos más tarde; etc. Para hacer esto correcta y rápidamente es necesario un ordenador, como se muestra en la figura 2.13. El ordenador guía cada uno de estos pasos del proceso de diseño y puede aplicar muchos tipos de comprobación. Por ejemplo, ¿están todas las variables definidas en el depósito? De ser así, ¿están siendo utilizados apropiadamente? Algunas de las más importantes cuestiones que se preguntan sobre la capacidad de dibujar automáticamente de un **CASE Workbench** se muestran el cuadro 2.5.

Cuadro 2.4. Funciones de las herramientas de diagramación estructurada

Las herramientas de diagramación estructurada automática proporcionan las siguientes importantes funciones:

- * Una ayuda para aclarar el conocimiento y la solución del problema.
- * Precisar y registrar la comunicación entre los componentes del equipo de desarrollo, los usuarios y los directivos.
- * Normaliza la representación de las estructuras de las arquitecturas y de los datos del programa.
- * Una ayuda para la detección de los errores.

Cuadro 2.4. Funciones de las herramientas de diagramación estructurada (continuación)

- * Una ayuda para analizar y comprender la existencia de los programas.
- * Una ayuda para cambiar los sistemas y programas (mientras se están creando y durante su mantenimiento).
- * Un desarrollo más rápido motivado por la diagramación asistida por ordenador.
- * Un mecanismo que capacita a los usuarios para revisar los requerimientos del programa y las especificaciones de su diseño.
- * Un mecanismo de producción automática de la documentación del programa.

Cuadro 2.5. Prestaciones de la diagramación automática

- * ¿Qué tipos de diagramas pueden dibujarse?
- * ¿Es posible crear símbolos definidos por el usuario?
- * ¿Cómo pueden manipularse los diagramas en la pantalla de la estación de trabajo (mover, borrar, copiar, cortar, empalmar, aumentar/disminuir, suprimir detalles, etiquetar objetos, conectar objetos, etc.)?
- * ¿Hay color? ¿Cuál es la información que el color añade al diagrama? ¿Puede el usuario cambiar el color?
- * ¿Puede guardarse fácilmente un diagrama y utilizarse más tarde?
- * ¿Puede cambiarse y redibujarse rápidamente un diagrama?
- * ¿Puede un objeto de un diagrama enlazarse con otro diagrama que lo describe con más detalle? ¿En cuántos niveles puede enlazarse?
- * ¿Cada vez que se realiza un cambio en un diagrama, se cambian automáticamente todos los diagramas relacionados con él?
- * ¿Puede el usuario definir nuevos tipos de diagramas, su rotación y sus reglas de sintaxis?
- * ¿Hay una conversión automática de una versión de tipos de diagramas a otra?
- * ¿Puede el usuario seleccionar una versión de un tipo de diagrama por ser un estándar de la organización?
- * ¿Soporta caracteres gráficos?
- * ¿Proporcionan ventanas y menús flotantes?
- * ¿Con qué facilidad se puede añadir a los diagramas información textual (etiquetas de los objetos) y comentarios de formato libre?
- * ¿El editor de textos es de un interfaz cerrado o totalmente integrado al editor de gráficos?

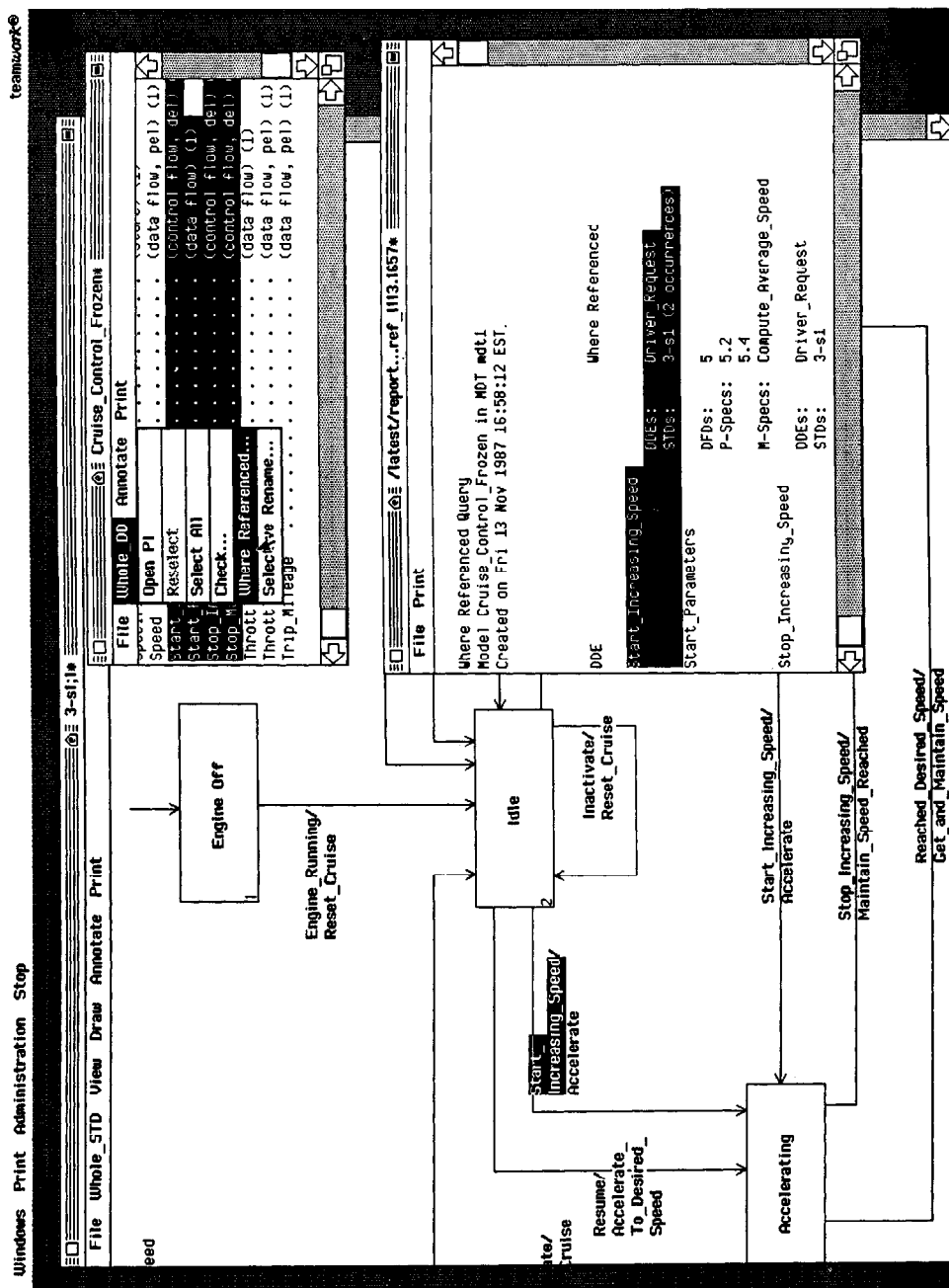


Figura 2.13. Las ventanas múltiples de TEAMWORK de Cadre Technologies proporcionan al usuario la oportunidad de ver simultáneamente variados tipos de información acerca del sistema

MAS ALLA DEL SIMPLE DIBUJO AUTOMATICO

En su mayor parte, las cuestiones expuestas en el cuadro 2.5 tratan de las prestaciones de edición de gráficos y de textos de un **CASE workbench**. Aunque esto es un elemento muy importante para incrementar la productividad, en realidad sólo es el principio. Es el primer nivel del soporte de la diagramación. Para incrementar significativamente la productividad, las funciones de la diagramación automática del **CASE workbench** deben ir más allá en el siguiente nivel de soporte de la diagramación. En este nivel, los diagramas se almacenan en línea (**on-line**) en el depósito y se analizan para detectar los errores, las omisiones y las inconsistencias.

Para descubrir cuanto es lo que un **CASE workbench** va más allá de una simple diagramación automática, la primera cuestión es preguntar: ¿Qué es —si lo hay— lo que se representa con un objeto dibujado en la pantalla del ordenador? En otras palabras, ¿hay algún significado computacional asociado con el objeto gráfico que ha sido comprendido y capturado por el **workbench**? Esta es una cuestión muy importante por tres razones. Primero, porque cuando se captura el significado asociado con el gráfico, la integridad y la consistencia del diagrama se comprueban de una forma más cuidadosa. Se incluyen algunos ejemplos en la sección titulada “La comprobación de errores”. Segundo, cuando se captura el significado de los símbolos gráficos, la misma información puede representarse de diferentes formas, pero equivalentes. Por ejemplo, un DFD puede **convertirse automáticamente** en un diagrama de estructura de árbol (es decir, diagrama de descomposición) después de haber identificado la raíz (ver figuras 2.14 y 2.15). Un diagrama de estructura de árbol (por ejemplo, un **diagrama de estructura**) también puede convertirse automáticamente en un diagrama de acción [6] (ver figuras 2.6 y 2.7). En ambos diagramas hay la misma información, solamente cambia la forma de representación. De este modo el **workbench** puede acomodarse a las preferencias de los diferentes profesionales del desarrollo de **software** y puede adaptarse a las diagramaciones normalizadas de las diferentes organizaciones. Tercero, cuando se almacena el significado de los símbolos, se captura del diagrama la información necesaria para la **generación automática de código**.

La segunda pregunta depósito es: ¿Qué es —si lo hay— lo que se almacena cuando se dibuja un diagrama? Muchas herramientas **workbench** almacenan la información que representa el diagrama en alguna clase de diccionario o depósito CASE. Cada símbolo, toda su información relativa

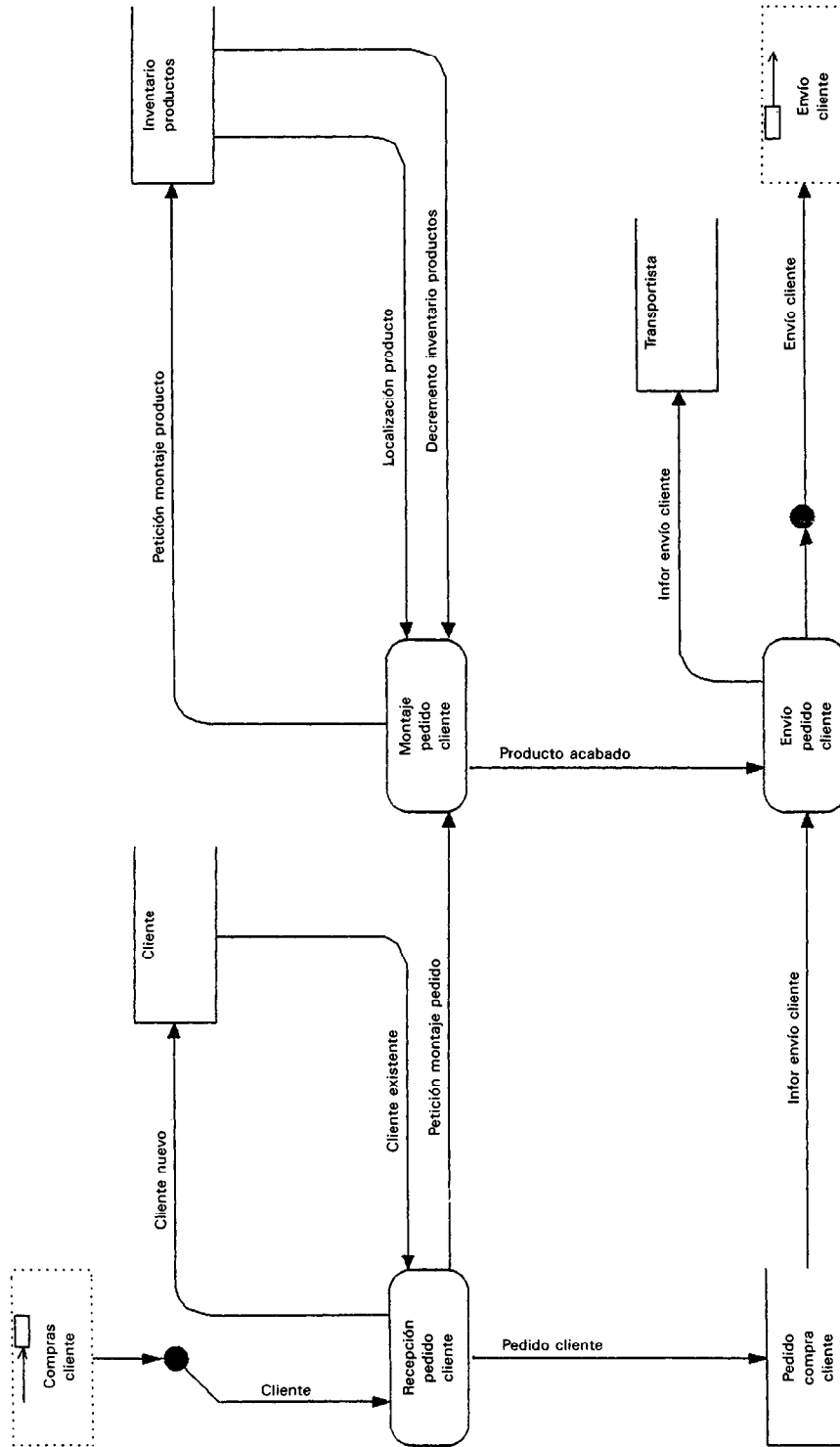


Figura 2.14. Diagrama de flujo de datos para el programa del Proceso de Pedidos de Clientes, obtenido con el INFORMATION ENGINEERING WORKBENCH de KnowledgeWare

y sus relaciones lógicas con los demás símbolos y su significado se almacenan en el depósito CASE. Esto es esencial para proporcionar la capacidad de cambiar y redibujar rápidamente un diagrama, así como poder cambiar automáticamente todos los diagramas asociados afectados por el cambio. Esta potente prestación posibilita que todos los diagramas asociados puedan actualizarse automáticamente de forma sincronizada. Sabemos por la experiencia pasada que esto, además de tedioso, es imposible de hacer manualmente. Y también le da al programador la capacidad de gestionar mejor toda la información relativa al sistema. La prestación de almacenamiento de un **CASE workbench** se estudia más adelante en este capítulo en la sección titulada “El depósito CASE de la información”.

La tercera cuestión es: ¿Qué tipo de corrección de errores se realiza automáticamente? Los diagramas deben estar completos y ser bastantes rigurosos para poder utilizarse como fuente de información primaria para la generación de código y de las bases de datos.

LA COMPROBACION DE ERRORES

La comprobación de errores es una de las prestaciones más importantes de un **CASE workbench**. Detectar los errores desde el principio es un modo excelente de reducir los costos del **software**. La comprobación automática de errores ayuda al programador a encontrar los errores mucho antes en el ciclo de vida del **software**. La calidad de un sistema de **software** depende de sus especificaciones, y la calidad de las especificaciones de un sistema dependen de su integridad y de su consistencia. El cuadro 2.6 muestra cinco tipos de comprobación de errores en la diagramación de las especificaciones de un sistema. La comprobación automática de errores comienza tan pronto como el programador empieza a dibujar un diagrama estructurado en la pantalla de la estación de trabajo CASE.

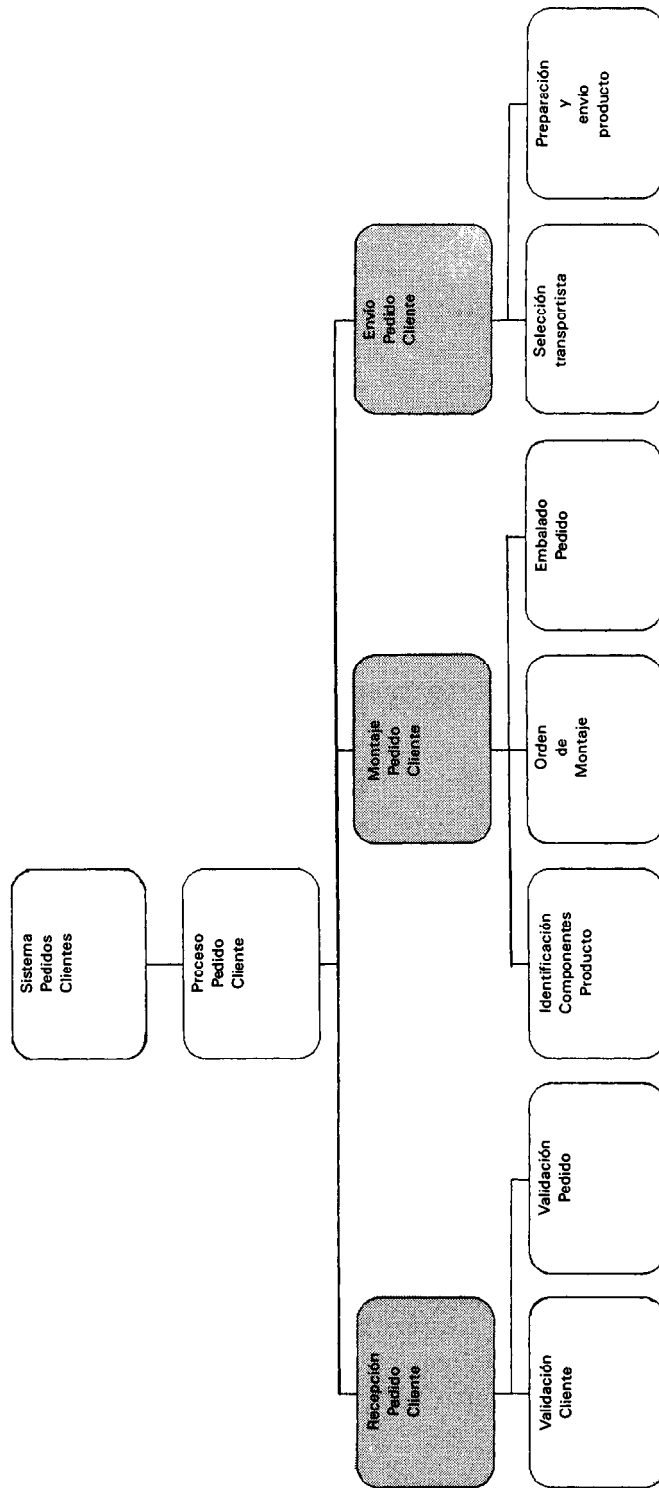


Figura 2.15. Diagrama de descomposición del Programa del Proceso de Pedidos Clientes, obtenido con el INFORMATION ENGINEERING WORKBENCH de KnowledgeWare. El diagrama de descomposición se deriva del diagrama de flujo de datos de la figura 2.14. Observe que los tres procesos sombreados en el diagrama de estructura se derivaron de los tres procesos correspondientes mostrados en el diagrama de flujo de datos

Comprobación de sintaxis y de tipo

La primera clase de comprobación es la de **errores de tipo y de sintaxis**. A modo de ejemplo, consideremos el DFD de la figura 2.16. Una regla de sintaxis de un DFD es que cada burbuja de proceso debe tener por lo menos un flujo de datos de entrada y por lo menos un flujo de datos de salida. Obsérvese que el PROCEDIMIENTO DE ACEPTACION DEL PEDIDO de la figura 2.16 es incorrecto, porque no hay flujo de datos de entrada. Otro ejemplo de regla de sintaxis de un DFD es que dos datos de almacenamiento no pueden conectarse directamente. Obsérvese que el flujo que conecta el FICHERO DE INVENTARIO y el FICHERO DE CLIENTE es incorrecta.

En otros tipos de diagramas estructurados, como el diagrama de entidad/relación o el de flujo de control, también tienen reglas de sintaxis que pueden comprobarse.

Cuadro 2.6. Tipos de correcciones de errores en los diagramas estructurados

- **Comprobación** de sintaxis y de tipo.
- **Comprobación** de integridad y de consistencia.
- **Comprobación** de descomposición funcional.
- **Comprobación** cruzada de consistencia a través de todos los niveles y vistas.
- **Comprobación** de rastreo de requerimientos.

Las reglas del tipo también gobiernan la construcción de los diagramas estructurados. Por ejemplo, en un DFD, un símbolo de proceso debe usarse siempre para representar un componente de procedimiento (como una función comercial, una tarea del proceso, un módulo del programa). De forma similar, un flujo de datos debe representar un componente de datos (una variable, un registro). Las reglas de tipo se introducen para eli-

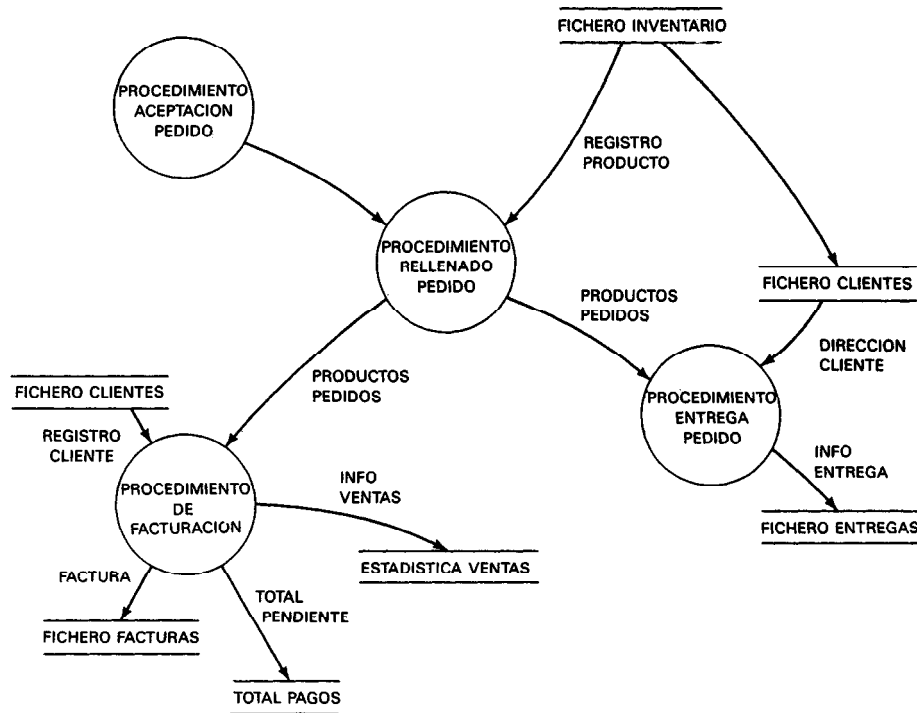


Figura 2.16. Diagrama de flujo de datos mostrando los cuatro subprocesos en el sistema de distribución de ventas

minar las ambigüedades que puedan causar confusión cuando se intenta comprender una especificación o la transformación en código.

La comprobación de la sintaxis y del tipo se suele realizar interactivamente, según se va creando el diagrama en la estación de trabajo CASE. El propósito de la comprobación interactiva es evitar que el personal de desarrollo introduzca información incorrecta o no razonable en las especificaciones.

La comprobación de la integridad y de la consistencia

El segundo tipo de comprobación es el de **la integridad y la consistencia** de un diagrama. Como su nombre sugiere, el propósito de la compro-

bación de la integridad y de la consistencia es asegurar que el diagrama ha sido completado y que contiene toda la información necesaria. Esta clase de comprobación está controlada por el usuario y se activa normalmente cuando el usuario ha terminado el diagrama y solicita que se compruebe automáticamente antes de formar parte de las especificaciones del sistema.

Consideremos de nuevo el DFD de la figura 2.16. En un diagrama de flujo de datos completo, todas las flechas de flujo de datos entre los procesos están etiquetadas con el dato que pasa de un proceso a otro. Obsérvese que este diagrama no está completo, porque la flecha entre el PROCEDIMIENTO DE ACEPTACION DEL PEDIDO y el PROCEDIMIENTO DE RELLENADO DEL PEDIDO no está etiquetada. Otra regla para completar un DFD es que debe haber por lo menos un dato de entrada y otro de salida en cada burbuja de proceso. El diagrama de la figura 2.16 no está completo, porque no hay dato de entrada en el PROCEDIMIENTO DE ACEPTACION DEL PEDIDO.

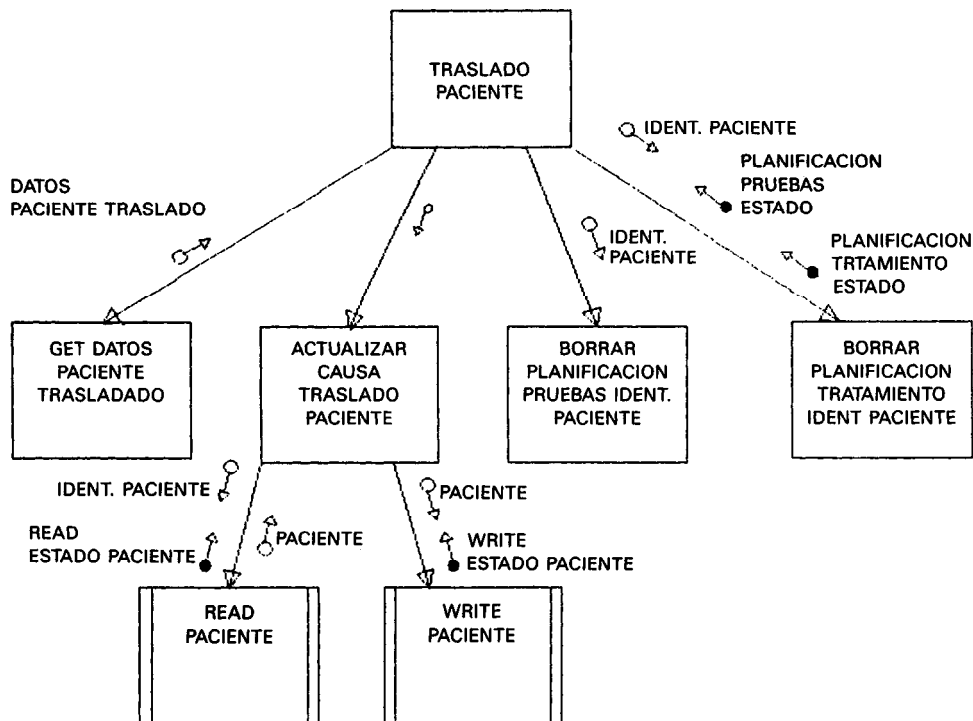


Figura 2.17. Diagrama de estructura obtenido con el ANALYST/DESIGNER TOOLKIT de Yourdon Engineering Company

La comprobación de la integridad debe ir mucho más allá de la simple comprobación de que todos los componentes del diagrama han sido etiquetados apropiadamente. También debe comprobar exhaustivamente si la definición de cada objeto existe en el depósito CASE y si esta definición es consistente con cualquier información sobre el objeto almacenada previamente.

Como ejemplo de comprobación de la integridad de un diagrama con estructura de árbol jerárquico, considérese el **diagrama de estructura** mostrado en la figura 2.17. Este diagrama no está completo, porque la flecha que une SALIDA DE PACIENTE con ACTUALIZACION CAUSA DE SALIDA DE PACIENTE no está etiquetada.

La comprobación de la integridad y de la consistencia puede aplicarse a cualquier tipo de diagrama (flujo de control, entidad/relación, etc.) que el profesional de desarrollo cree con un **CASE workbench**.

La comprobación de la descomposición funcional

El siguiente tipo de comprobación de un diagrama es la relativa a la **estructura de árbol jerárquico**, que es una forma de descomposición funcional. El diagrama de estructura de árbol consta de un conjunto de componentes funcionales ordenados jerárquicamente. Los componentes que realizan funciones de más alto nivel se colocan en los niveles superiores de la jerarquía, mientras que los componentes que realizan funciones de nivel más bajo (o más detallado) aparecen en los niveles inferiores. Observando la jerarquía, los componentes de cada nivel sucesivo contienen funciones que definen (refinan) las funciones residentes en el nivel anterior.

Las metodologías estructuradas contienen reglas para la descomposición funcional. Por ejemplo, la metodología de diseño estructurado de Yourdon permite que un componente funcional pueda descomponerse hasta en siete subfunciones [7, págs. 148-170]. Además, la programación estructurada estricta no permite a una función invocarse a sí misma [8, págs. 45-65].

La comprobación de la descomposición funcional es una forma de control de calidad que puede realizarse en un diagrama de estructura de árbol. El análisis de complejidad es otro control de calidad para los diagramas de estructura de árbol. La métrica de complejidad de McCabe puede utili-

zarse como medida cuantitativa de la complejidad del diseño de un programa estructurado. Para estar dentro del rango de complejidad aceptable, cada componente funcional de un diagrama de estructura de árbol debe tener un valor de complejidad de McCabe menor que 10. La complejidad de un componente funcional se calcula contando el número de comparaciones en la lógica de ejecución de la función [8, págs. 67-82]. Contar el número de acoplamientos pasados entre cada par de funciones en una **diagrama de estructura** es otro método de calcular la complejidad de un diseño [8, págs. 67-82].

Una clase especial de comprobación de descomposición funcional se denomina **refinamiento semántico**, en ella cada descomposición de una función en subfunciones se comprueba para asegurar que proporciona información más detallada sobre la función. Por ejemplo, si en un nivel una función se describe como realizadora de una operación de entrada/salida, en el siguiente nivel debe definirse el tipo de operación de entrada/salida (lectura o escritura) [9].

La comprobación en las metodologías específicas

Los tres primeros tipos de comprobación eran para los errores en un diagrama. Estas son las comprobaciones más simples y lo mínimo que puede pedirse a un **CASE workbench**. Sin embargo, las comprobaciones específicas que se realicen dependerán de las particularidades de la(s) metodología(s) estructurada(s) soportada(s) en el **CASE workbench**. Así pues, se realizan diferentes comprobaciones en los diagramas de estructura de Yourdon, en los diagramas de estructura de árbol de Jackson, en los diagramas de Warnier-Orr y en los diagramas de acción de Martin. Aunque todos ellos son diagramas estructurados en árbol jerárquico, pertenecen a diferentes metodologías, tienen diferentes reglas de sintaxis y tienen diferentes comprobaciones para asegurar la calidad. De modo similar, los DFD de DeMarco y los DFD de Gane-Searson, los diagramas de dependencia de Martin y los diagramas de flujo de control de Ward-Mellor, tienen diferentes reglas de sintaxis, aunque todos ellos son formas del diagrama de flujo de datos básico.

La comprobación a través de una familia de diagramas

Las metodologías estructuradas son una formulación de desarrollo descendente (**top - down**) en las cuales un diseño se crea empezando por una panorámica general del sistema y progresivamente se va afinando en los detalles en los niveles sucesivos. Cada nivel se representa en un diagrama (o en un conjunto de diagramas) y todos los niveles están conectados para soportar esta “explosión” de detalles. Por ejemplo, cuando el programador desea observar una visión del proceso con más detalle en un DFD, puede “expandirlo” en el siguiente nivel de detalle, que puede representarse en un DFD, en un **diagrama de estructura**, en un diagrama de acción, en un pseudocódigo (o miniespecificación) o en código fuente (ver figuras 2.18, 2.19 y 2.20). Un almacén de datos, un flujo de datos o un acoplamiento de datos pueden expandirse en un diagrama de modelo de datos, en un diseño de registro o en una entrada de diccionario.

El cuarto tipo de comprobación de diagramas no atañe sólo a una clase de diagramas, sino a una **familia de diagramas** relacionados. Un ejemplo de familia de diagramas es un conjunto de DFD estratificados por niveles, en los cuales cada nivel sucesivo describe los procesos con mayor detalle. La figura 2.16 identifica el PROCESO DE ACEPTACION DEL PEDIDO en el SISTEMA DE DISTRIBUCION DE VENTAS. La figura 2.21 muestra con detalle todo lo que acontece dentro del PROCEDIMIENTO DE ACEPTACION DEL PEDIDO y la figura 2.22 muestra también con detalle uno de los cinco procesos anteriores, el proceso de VALIDACION DEL CLIENTE.

La comprobación de la consistencia a través de una familia de diagramas revisa la coherencia de la información de un nivel a otro. A esta comprobación también se denomina **análisis de equilibrio** o **análisis de conservación**.

Por ejemplo, si en el primer nivel (ver figura 2.16) del FICHERO DE PEDIDOS DE CLIENTES se requiere una entrada al PROCEDIMIENTO DE ACEPTACION DE PEDIDO, para que sea consistente por las dos vistas, el fichero debe también aparecer como entrada requerida en el siguiente nivel (figura 2.21). Como otro ejemplo de consistencia, si el REGISTRO DE CLIENTE es un flujo de datos de las figuras 2.21 y 2.22, no puede utilizarse también como nombre de un procedimiento en el diagrama de la figura 2.22.

La comprobación de la consistencia y de la integridad debe extenderse a todos los diagramas relacionados de un sistema (programa o subconjunto) según diseñe el programador. Esta comprobación se produce a petición del usuario del **workbench**, quien especifica su alcance.

La comprobación de la consistencia y de la integridad a través de una familia de diagramas, elimina las ambigüedades para hacer posible la utilización de esta información como fuente para la generación automática de código o para la comunicación entre los distintos componentes del equipo del proyecto. Este tipo de prueba puede ser de gran ayuda para encontrar los errores más sutiles (y también los más costosos) en las especificaciones.

La comprobación del rastreo de requerimientos

El **rastreo de requerimientos** es un método que permite comprobar si el sistema satisface los requerimientos del usuario. Generalmente se demuestra por pasos, mostrando que el trabajo producido por el paso en curso puede ser rastreado hasta el trabajo producido en el paso previo. Por ejemplo, desde la especificación de diseño podemos rastrear hacia atrás las especificaciones de requerimientos, o desde el código fuente, la especificación de diseño. Un tipo de requerimientos de rastreo y el nivel de diseño hacia atrás se muestra creando automáticamente un **diagrama de estructura** desde el código fuente y comparándolo con el **diagrama de estructura** creado durante la fase de diseño. Cualquier discrepancia entre los dos **diagrama de estructura** será notificada y debe ser reconciliada mostrando los dos códigos fuente cuyos diseños no coinciden. Esta técnica puede ayudar a la identificación de los requerimientos erróneos y no implantados, las entradas no utilizadas y los errores de código.

De esta forma se puede saber si el código generado a partir de un conjunto de especificaciones de diseño, que se derivó de las especificaciones de requerimiento, satisfacen un requerimiento en particular. Después, al comprobar un sistema o cambiado un requerimiento, el profesional del desarrollo puede localizar fácil y exactamente qué partes del sistema deben estudiarse.

El Departamento de Defensa de EE.UU. en su Estándar DOD 2167 ha establecido que la documentación del rastreo de requerimientos es crítica y requiere que sea un componente de desarrollo en los sistemas críticos de defensa.

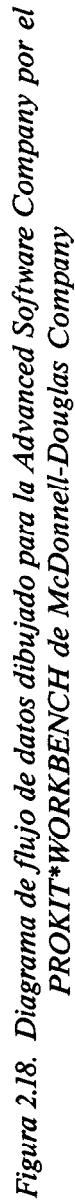


Figura 2.18. Diagrama de flujo de datos dibujado para la Advanced Software Company por el PROKIT*WORKBENCH de McDonnell-Douglas Company

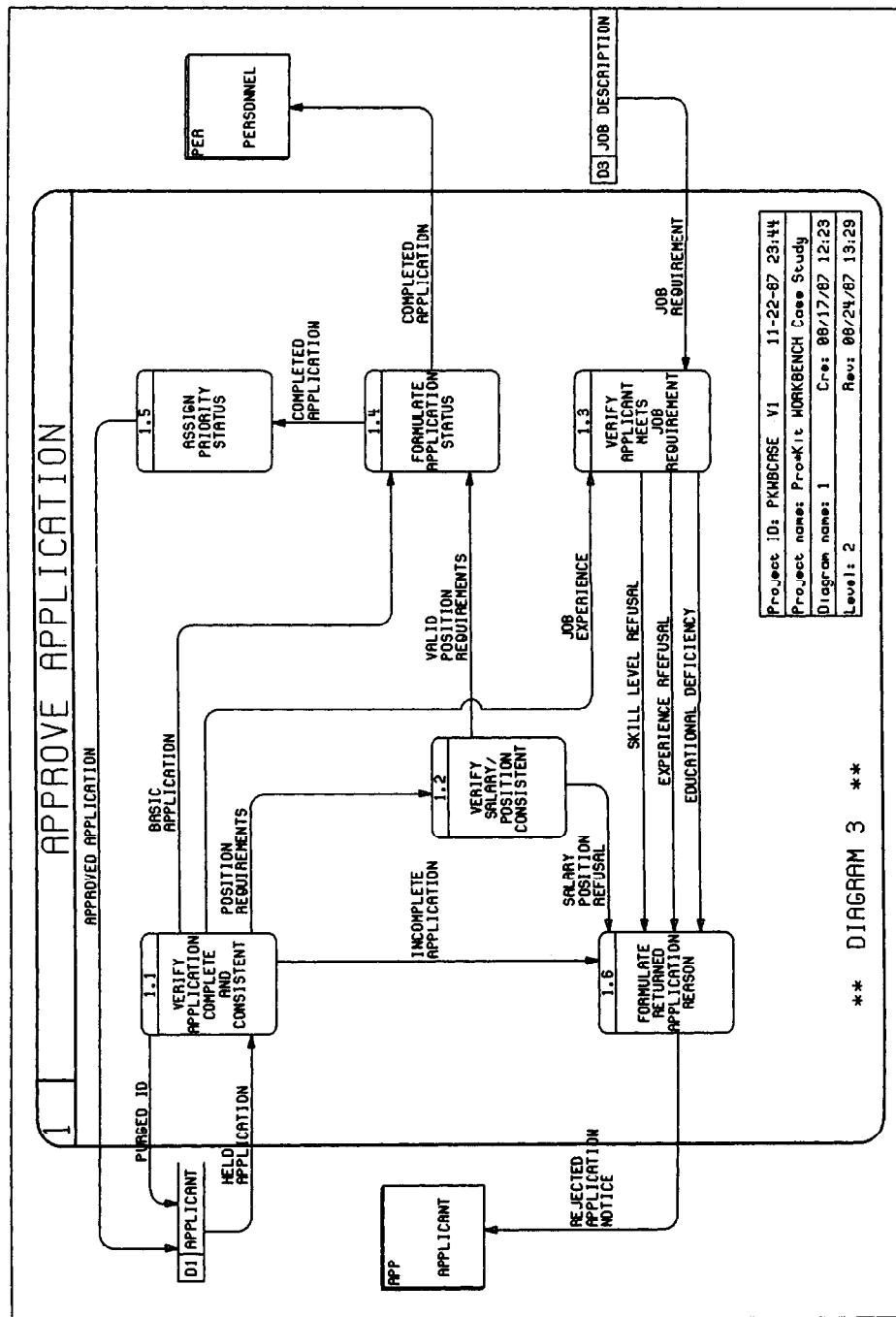


Figura 2.19. Diagrama de flujo de datos ampliado para el proceso Approve Application definido en la figura 2.18. Se obtuvo con el PROKIT*WORKBENCH de McDonnell-Douglas Company

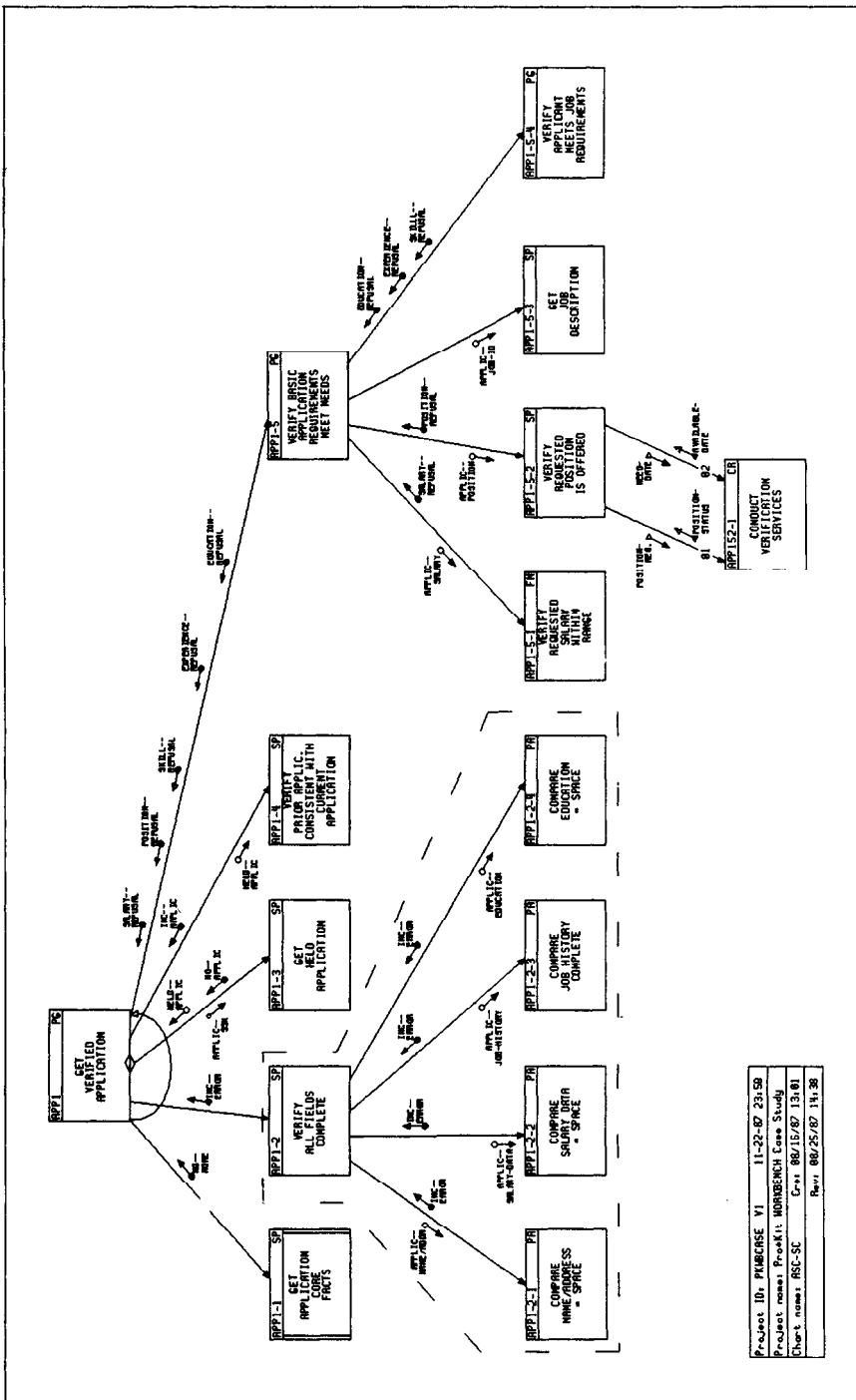


Figura 2.20. Diagrama de estructura del proceso Approve Application mostrado en la figura 2.19. Se obtuvo con el PROKIT*WORKBENCH de McDonnell-Douglas Company. Los tres diagramas que ilustran las figuras 2.18, 2.19 y 2.20 pertenecen a una familia de diagramas relativos a los sistemas de la Advanced Software Company

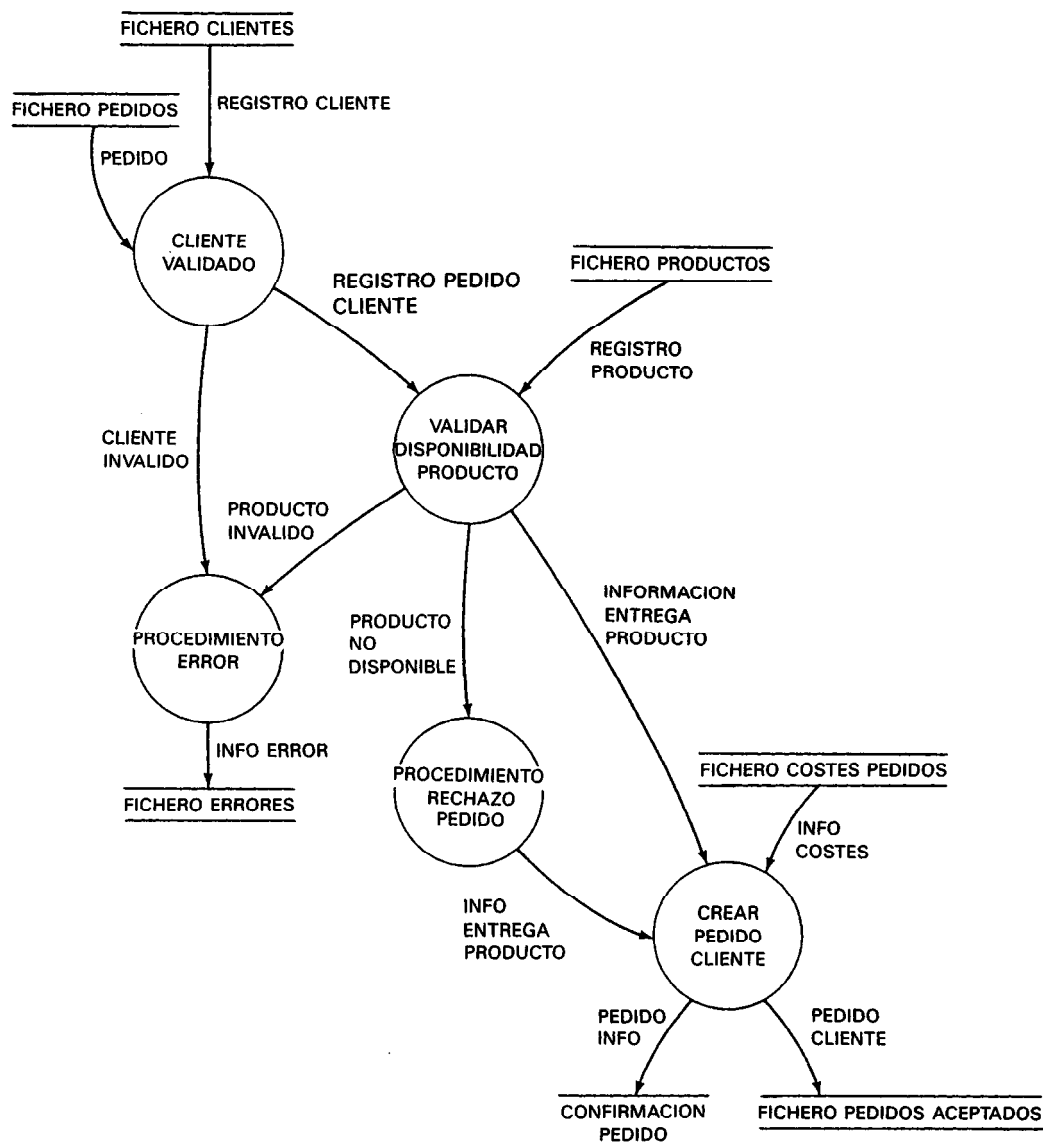


Figura 2.21. Diagrama de flujo de datos para el PROCEDIMIENTO ACEPTACION PEDIDOS mostrando el flujo de los datos y los procesos realizados en el tratamiento de los pedidos de los clientes

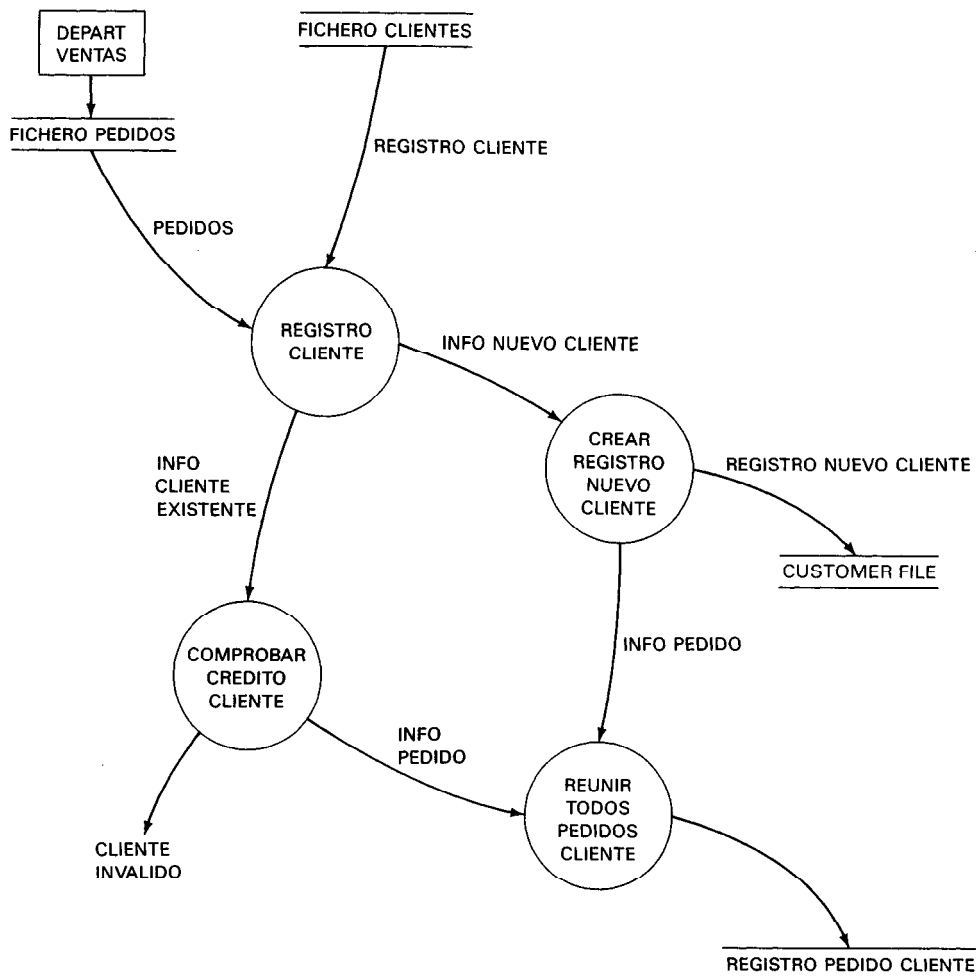


Figura 2.22. Visión ampliada del procedimiento VALIDACION CLIENTE, mostrando cuatro subprocedimientos: GET REGISTRO CLIENTE, CREAR REGISTRO NUEVO CLIENTE, COMPROBAR CREDITO CLIENTE y REUNIR TODOS PEDIDOS CLIENTE

Sin embargo, debería considerarse como un componente crítico en cualquier proyecto de desarrollo de sistemas, pues la posibilidad de rastrear la correspondencia entre los requerimientos y el código derivado y la estructura de los datos es necesaria y muy valiosa en la comprobación minuciosa de un sistema. El rastreo de los requerimientos es un método para asegurar que un sistema de **software** no solamente es correcto, sino que también es

completo. Proporciona un camino de los requerimientos al código que el programador puede seguir en cualquier dirección.

El informe de errores

La comprobación del diagrama incluye una función de informe. El programador puede pedir informes tanto en línea como fuera de línea que relacionen los cinco tipos de errores. También pueden conseguirse informes de las referencias cruzadas y la documentación de rastreo de los requerimientos para mostrar en qué diagramas cada componente de procedimiento y componente de datos está el error. Las figuras 2.23, 2.24, 2.25 y 2.26 muestran diversos informes de comprobación de errores.

EL DEPOSITO DE LA INFORMACION DE LA CASE

Aunque no causa tanta impresión visual como las prestaciones gráficas, el depósito de la información es realmente el corazón del **CASE workbench**. Es la base para:

- La integración de las herramientas CASE
- El control de la consistencia y de la integridad de las especificaciones del sistema.
- La distribución de la información compartida del sistema.
- La normalización de la documentación.
- La generación de la documentación del sistema.
- La generación de código.
- La reutilización del **software**.
- El control y la gestión del proyecto.

El depósito CASE es un mecanismo para almacenar y organizar toda la información relativa al sistema **software**, incluyendo la planificación, el

IEW® Case Study Connectivity Analysis
(excerpted from longer report)

Information Engineering Workbench Report
Connectivity Analysis Report
April 29, 1986 2:19:26

Data Flow Assembled Order

Location
Activity Assemble Customer Orders

Immediate Source
Activity Prod. Construct. & Order Assemb.
in context of Activity Assemble Customer Orders

Immediate Destination
Activity Package Order
in context of Activity Assemble Customer Orders

The data flow has no unconnected ends

Data Flow BO Order Assembly Request

Location
Activity Assemble Customer Orders
in context of Activity Process Customer Orders

Immediate Source
Activity Process Backorders
in context of Activity Process Customer Orders

Immediate Destination
Activity Assemble Customer Orders

The data flow has no unconnected ends

Data Flow BO Order Assembly Request

Location
Activity Assemble Customer Orders

Immediate Source
Internal Junction in
Activity Assemble Customer Orders

Immediate Destination
Activity Prod. Construct. & Order Assemb.
in context of Activity Assemble Customer Orders

The data flow has no unconnected ends

© KnowledgeWare™, Inc.

Figura 2.23. El Connectivity Analysis Report obtenido con el INFORMATION ENGINEERING WORKBENCH de KnowledgeWare lista los errores de sintaxis en los diagramas de flujo de datos

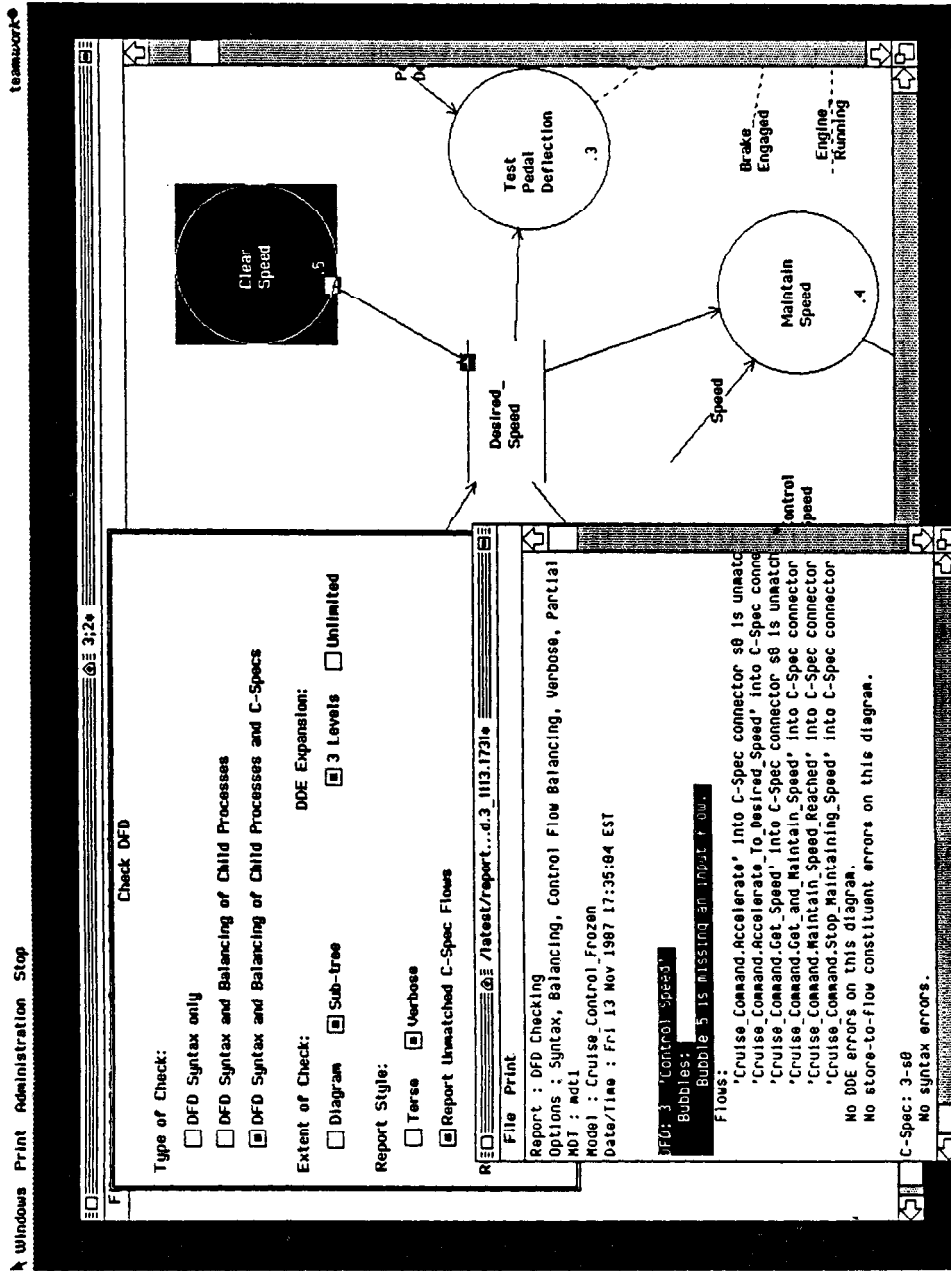


Figura 2.24. El informe de TEAMWORK de Cadre Technologies sobre errores variados en los diagramas de flujo de datos y de flujo de control


```

ANALYST Case Study
ID:          VERSION: 6
DATA ELEMENT INCOMPLETE DEFINITIONS
ANALYST101, PAGE 1
06/03/86, 13:30:37

=====
-----DATA ELEMENT NAME-----DATA ELEMENT DESCRIPTION-----
CUSTOMER-ADDRESS      This includes the customer street address, city, state and
                      zip code.
                      DATA ELEMENT TYPE: C
                      SIZE: 0090
                      DECIMALS: 0
                      CLASS: _
                      UNIQUE: _

                      -----DATA ELEMENT EDIT RULES-----DATA ELEMENT OWNER-----
                      No editing required.
                      William H. Threlkeld
                      (222) 222-2222

                      -----DATA ELEMENT DERIVATION RULES-----
                      -----
                      -----
                      -----
=====

-----DATA ELEMENT NAME-----DATA ELEMENT DESCRIPTION-----
PART NO              The PART NO is used to identify the equipment or service
                      requested by the customer.
                      DATA ELEMENT TYPE: N
                      SIZE: 0007
                      DECIMALS: 0
                      CLASS: _
                      UNIQUE: Y

                      -----DATA ELEMENT EDIT RULES-----DATA ELEMENT OWNER-----
                      The PART NO consists of a base number and a check digit.
                      The check digit will occupy the units position of the
                      number. The check digit will be calculated using the
                      modulus 10 remainder method.
                      ( ) _ _ _ _

NOT CONTAINED IN ANY DATA STRUCTURE, DATA FLOW, DATA STORE AND NOT DERIVED BY ANY PROCESS.
=====

```

Figura 2.25. El PROKIT WORKBENCH de McDonnell-Douglas informe de errores tales como un proceso incompleto o definiciones de elementos incompletos*

```
-----  
                                FACT SHEET  
                                UNDEFINED DICTIONARY ENTRIES  
                                =====  
08/06/86                                13:17:13  
  
ALLOW SETUP  
AUTO  
BIN  
CHANNEL  
CLOCK PULSE  
CURRENT FORMAT  
CURRENT LOGIC STATE  
CURSOR POSITION  
DESIRED TRIGGER WORD  
DISPLAY STATES  
DOWNARROW  
ENABLE/ DISABLE  
ENABLE/DISABLE (LOGIC ACQUISITION PROCESSOR)  
HEX  
LEFT ARROW  
LEVEL  
LIST  
LOGIC ACQUISITION COMPLETE  
LOGIC STATE DISPLAY  
LOGIC STATES  
MANUAL  
MATCH LEVEL  
MENU  
RIGHT ARROW  
SET TIMER  
SETUP PARAMETERS  
START AUTO  
START MANUAL  
STOP  
TIMER EXPIRED  
TRIGGER  
TRIGGER STATE  
TRIGGER WORD CONTEXT  
TRIGGER WORD DISPLAY  
TRIGGER WORD FOUND  
TRIGGER WORD NOT FOUND  
UPARROW  
WINDOW
```

Figura 2.26. El ANALIST/DESIGNER TOOLKIT de Yourdon software Engineering informa sobre las entradas, no definidas, en el depósito

análisis, el diseño, la implantación y la gestión de la información del proyecto. La clave para la alta productividad está en ofrecer la información al programador cuando la necesita y en una forma útil.

El depósito CASE se denomina también diccionario de diseño, base de datos, base de datos orientada al objeto, base del conocimiento y enciclopedia. Sin embargo, independientemente de cómo se le llame, su propó-

sito es el mismo: servir de lugar único donde toda la información del sistema se puede introducir una vez, mantenerse consistente y tenerla disponible para quien la necesite. Es importante observar que el depósito es un concepto mucho más amplio que el de mero diccionario de datos, ya que almacena muchos más tipos de información del sistema, las relaciones entre los diversos componentes y las reglas para utilizar o procesar los componentes.

El depósito CASE realiza las funciones de almacenar, acceder, actualizar, analizar y obtener informes de la información del sistema. Desde la perspectiva de la información, la principal función de las herramientas del **CASE workbench** es gestionar la información del sistema. Capturan la información de especificación del análisis y del diseño; controlan y analizan la información de especificación; transforman la información de especificación en código de programa y en documentación; soportan las funciones de la gestión y del mantenimiento del proyecto. En un **CASE workbench** es el depósito el principal responsable de la gestión de la información del sistema (la gestión técnica y del proyecto). De hecho, mantiene toda la información para crear, modificar, desarrollar y mantener el sistema de **software**. Eso incluye información sobre:

- El problema a resolver.
- El dominio del problema.
- Aportar una solución.
- El proceso del **software** (metodología) que se va a seguir.
- Los recursos del proyecto y su historia.
- El contexto organizacional.

El contenido del depósito

Como se muestra en la figura 2.27, los diagramas estructurados, las definiciones de las pantallas y de los menús, los bosquejos de los informes, las descripciones de los registros, la lógica de los procesos, los modelos de los datos, los modelos de la organización, los modelos de los procesos, el código fuente, las reglas de la actividad del negocio, las formas de la gestión del proyecto, los datos elementales y las relaciones entre los componentes de la información del sistema son ejemplos de algunos tipos de objetos de información que pueden almacenarse en el depósito CASE. No

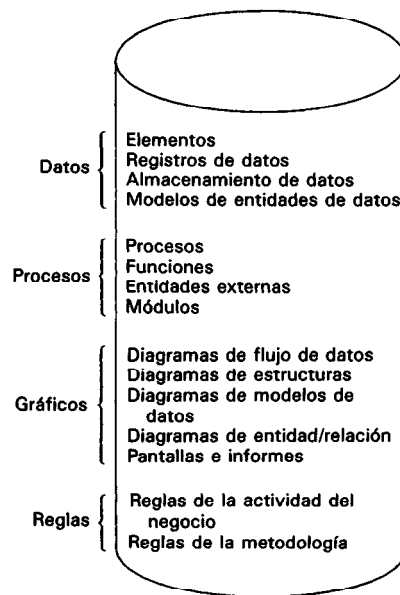


Figura 2.27. El depósito CASE es el corazón del banco de trabajo (workbench) CASE. Contiene toda la información del sistema: técnico, de la gestión del proyecto, y de las relaciones entre los variados componentes del sistema

es nada extraño que un depósito CASE contenga más de cien tipos de objetos y relaciones.

Cada objeto de información almacenado en el depósito CASE se define en función de sus propiedades, a saber:

- La identificación.
- Los nombres de los alias.
- El tipo.
- La descripción narrativa.
- Las partes componentes.
- El fichero donde está almacenado.
- Los rangos de valores.
- Las reglas de edición y de derivación.

También se almacenan en el depósito CASE todas las relaciones de cada objeto con los otros del sistema así como una información de control (**audit**). Por ejemplo, la información de las relaciones incluye todos los niveles de expansión de un objeto, todos los objetos en los que se utiliza este objeto y las referencias cruzadas. La información de control se relaciona con la identificación de la propiedad y las reglas de control de cambio de un objeto. La información de control, por ejemplo, incluiría:

- Quien creó el objeto.
- El proyecto en el cual se creó el objeto.
- Cuándo se creó el objeto.
- Cuándo se actualizó el objeto.
- El número de la versión.
- Si el objeto puede ser actualizado.

Los componentes del depósito CASE van creciendo según se va almacenando la información del sistema. Cuanta más información se almacena, menos información nueva se necesita para crear nuevos sistemas, ya que mucha de la información de la gestión del proyecto, de los modelos de organización, de los modelos de datos, de las entidades de datos y la de algunos modelos de procesos puede ser similar en las aplicaciones de la misma organización o dentro del dominio de una aplicación.

Los informes del depósito

El depósito CASE mantiene y realiza automáticamente el seguimiento de toda la información almacenada. Proporciona un conjunto de informes estandarizados para una visión rápida de su contenido. Se proporcionan informes tanto **on line** como **off line**, además de en formato preestablecido, informes para el usuario y de preguntas **ad hoc**. Por ejemplo, la información acerca de dónde se utiliza cada elemento dentro del sistema, quién lo ha creado, cuándo y cuántas veces se ha revisado la definición del elemento y las referencias cruzadas están en primera línea de los informes de referencia cruzada. Los tipos básicos de informes del depósito CASE incluyen:

- Informes del contenido.
- Informes de dónde se utiliza y de referencias cruzadas.

- Informes de análisis.
- Informes de expansión.

El cuadro 2.7 presenta ejemplos de tipos de informes del depósito. Los informes del **CASE workbench** están disponibles en la estación de trabajo del programador o en el ordenador principal. Las figuras 2.28, 2.29 y 2.30 ofrecen algunos ejemplos de informes del depósito CASE.

Cuadro 2.7. Ejemplos de tipos de informes del depósito CASE

Ejemplos de informes de componentes/listas

- Descripción del flujo de datos, incluyendo sus partes componentes.
- Descripción de todos los **acoplamientos (couples)** en un interfaz en un **diagrama de estructura**.
- Lista de las entradas y salidas de cada proceso en un diagrama de flujo de datos, marcando los datos derivados o actualizados en un proceso.
- Lista de todos los objetos del sistema que han sido cambiados después de cierta fecha.
- Historia de todos los cambios realizados en un objeto.
- Descripción de un módulo, incluyendo su descripción, parámetros y perfil de acceso a la base de datos/fichero.
- Plan de pruebas para un modulo, subfunción o programa en particular.
- Lista de todas las entidades de datos y sus atributos, más las relaciones y reglas de actividad del negocio para un modelo de datos dado.

Ejemplos de dónde se usa y de referencias cruzadas

- Todos los diagramas de flujo de datos en los que aparezca el proceso.
- Todos los módulos invocados por un módulo y todos los módulos que invoquen al módulo.
- Todos los objetos en el depósito sobre los que un programador tiene responsabilidad.

Cuadro 2.7. Ejemplos de tipos de informes de depósito CASE (continuación)

- Todos los diagramas en los se hace referencia una entidad de datos.
- Rastreo del curso de un dato desde el origen al destino en una familia de niveles de diagramas de flujos de datos.
- Para un proceso dado, todas las entidades que soporta y las unidades la organización que lo realizan.

Ejemplos de informes de análisis

- Equilibrio nivel por nivel en una familia de diagramas de flujo de datos.
- Todos los objetos de información indefinidos en un sistema.
- Los diagramas de flujo de datos de cualquier tipo incompletos.
- Análisis de la complejidad del diseño del programa/**diagrama de estructura**.
- Nombre, tipo y descripción de los objetos con inconsistencias contenidos de los diagramas o en el depósito.
- Objetos no utilizados.
- Análisis de consistencia, indicando la ausencia de relación entre dos objetos cuyos antecesores están relacionados.
- Impacto causado al borrar un objeto.
- Balance horizontal entre las salidas de un proceso desde las entradas y las derivaciones.

Ejemplos de los informes de expansión

- Tabla jerárquica de un objeto y de todas sus expansiones de bajo nivel.

La gestión de la información

La gestión automática de toda la información del sistema es la razón principal para almacenarla en el ordenador y uno de los más importantes beneficios que el **CASE workbench** ofrece al profesional del **software**. Es imposible mantener la integridad de la información del sistema con métodos manuales y no formales. Para permitir a los programadores compartir

IEW® Data Flow
Conservation Analysis
(example)

Information Engineering Workbench Report
Data Conservation Analysis
April 29, 1986 2:28:21

Page 1

Activity Package Order
in context of Activity Assemble Customer Orders

Incoming data flows

Data Flow Assembled Order
Data Flow Packaging Materials

Outgoing data flows

Data Flow Finished Goods
Data Flow Decreased Packing Materials

Data Conservation Failure

Outgoing Data Flow Finished Goods involves
Entity Type Customer PO
not involved in Composite Sequential Process Package Order

Data Conservation Failure

Incoming Data Flow Assembled Order involves
Entity Type Customer PO
not involved in Composite Sequential Process Package Order

Data Conservation Failure

Outgoing Data Flow Finished Goods involves
Entity Type Customer PO Line Item
not involved in Composite Sequential Process Package Order

Data Conservation Failure

Incoming Data Flow Assembled Order involves
Entity Type Customer PO Line Item
not involved in Composite Sequential Process Package Order

Data Conservation Failure

Outgoing Data Flow Finished Goods involves
Attribute Type Customer PO Expected Delivery Dt of
Entity Type Customer PO
not involved in Composite Sequential Process Package Order

Data Conservation Failure

Outgoing Data Flow Finished Goods involves
Attribute Type Customer PO Delivery Address of
Entity Type Customer PO
not involved in Composite Sequential Process Package Order

Data Conservation Failure

Outgoing Data Flow Finished Goods involves
Attribute Type Customer PO # of
Entity Type Customer PO
not involved in Composite Sequential Process Package Order

©KnowledgeWare™, Inc.

Figura 2.28. El Data Flow Conservation Analysis Report obtenido del INFORMATION ENGINEERING WORKBENCH de KnowledgeWare proporciona una información detallada sobre el flujo de datos en el diagrama de flujo de datos

Fact Sheet

FACT SHEET
MODIFIED DICTIONARY ENTRIES
=====

08/06/86

13:18:46

Old name: TRIGGER WORD CONTEXT

New name: TRIGGER WORD CONTEXT

Diagram: ANALYZE DEVICE LOGIC (PROCESSOR CONFIGURATION)

Old name: TRIGGER WORD CONTEXT

New name: TRIGGER WORD CONTEXT

Diagram: RECORD DISPLAY SETUP PARAMETER

Old name: TRIGGER WORD CONTEXT

New name: TRIGGER WORD CONTEXT

Diagram: USER INTERFACE MICRO

```

+-----+
+               +
+  KEEP TRACK OF CHANGES IN THE  +
+  DICTIONARY THAT IMPACT DIAGRAMS  +
+               +
+-----+

```

Figura 2.29. El ANALYST/DESIGNER TOOLKIT de Yourdon software Engineering informa sobre las modificaciones realizadas en el contenido del depósito CASE

la información del sistema de una forma ordenada y segura, se hace necesaria una facilidad que gestione la información de manera formal y automatizada.

La prestación de la gestión de la información del depósito CASE proporciona:

- **Controles de seguridad** en forma de clave para limitar en todo, o en parte, el acceso al contenido del depósito CASE.

```

VERSION      :1                      ProKit*WORKBENCH CASE STUDY                      PAGE      1
USER ID      :SYSTEM                 DATA ELEMENT CROSS-REFERENCE REPORT          DATE:11/04/87
WORKSTATION ID:MASTER                COMPLETE CROSS-REFERENCE OF ALL DATA ELEMENTS    TIME:15:11:47
MACRO NAME   :A.EH.DE.XREF
=====
-----DATA ELEMENT NAME-----  SHORT  DATA  LENGTH/  STORAGE
--NAME-- --TYPE--  PRECISION --SIZE-
ADDITIONAL-BENEFIT-CODE          CHARACTE      3      3
-----APPEARS ON STRUCTURE CHARTS (SC)-----
***CONTAINS NO ITEMS***
-----ATTRIBUTE OF DATA ENTITIES (DN)-----
-----ID-----DATA ENTITY NAME-----  ID-----DATA ENTITY NAME-----  ID-----DATA ENTITY NAME-----
DN5      BENEFITS
TOTAL ITEMS :      1
-----BOUNDED BY DOMAIN RANGES-----
***CONTAINS NO ITEMS***
-----BOUNDED BY DOMAIN VALUES-----
-----VALUE-----  -----MEANING-----
DEN                      DENTAL PLAN
HLT                      HEALTH PLAN
HOS                      HOSPITALIZATION PLAN
VAC                      VACATION PLAN
TOTAL ITEMS :      4
-----CONTAINED IN ACCESS PROFILE OF DS/DE (DS,DN)-----
***CONTAINS NO ITEMS***
-----CONTAINED IN DATA STRUCTURES (DX)-----
-----DATA STRUCTURE NAME-----  DATA STRUCTURE NAME-----  DATA STRUCTURE NAME-----  DATA STRUCTURE NAME-----
EMPLOYEE-INFORMATION      JOB-HISTORY-INFORMATION
TOTAL ITEMS :      2
-----CONTAINED IN DATA FLOWS (DF)-----
-----DATA FLOW NAME-----  DATA FLOW NAME-----  DATA FLOW NAME-----
ADDITIONAL BENEFITS        BENEFITS        BENEFITS ADVICE
BENEFITS PACKAGE
TOTAL ITEMS :      4
-----CONTAINED IN DATA STORES (DS)-----
-----ID-----DATA STORE NAME-----  ID-----DATA STORE NAME-----  ID-----DATA STORE NAME-----
DS      BENEFITS
TOTAL ITEMS :      1
****CONTINUED****

```

*Figura 2.30. El PROKIT*WORKBENCH de McDonnell-Douglas proporciona un informe de las referencias cruzadas de todos los elementos almacenados en su depósito CASE*

- **Privilegios de acceso** posibilitando la asignación individual de privilegios de sólo lectura o de actualización completa a los profesionales del desarrollo.

- **Control de la versión** para el seguimiento de la evolución e historia del sistema y permitir versiones múltiples en curso, como producción, prueba y desarrollo del conjunto de trabajos individuales de cada programador.
- **Accesos de multiusuarios** para permitir a múltiples programadores acceder concurrentemente a la misma información del depósito, pero no para actualizar la misma información concurrentemente.
- **Control de los cambios** para realizar automáticamente un seguimiento de los cambios e incorporar éstos al depósito.

Consideraciones en la implantación del CASE

Como se mencionó anteriormente, el depósito CASE es más que un simple diccionario, debido, por una parte, a la necesidad de almacenar muchos tipos de objetos de información y relaciones entre los objetos y, por otra, a la necesidad de una prestación de la gestión de la información. Generalmente, el depósito CASE es una base de datos relacional soportada por un potente sistema gestor (por ejemplo, en arquitecturas DB2, VSAM, DL1, dBASE III).

Esta arquitectura permite la inclusión automática en el depósito CASE de diccionarios y bases de datos externos. Debido a la necesidad de un flujo de información de doble vía entre el depósito CASE y los diccionarios y bases de datos externos, el **CASE workbench** debe proporcionar mecanismos para importar y exportar la información del sistema hacia/desde el depósito CASE.

Prestaciones del depósito

En resumen, el depósito CASE es un concepto central en la tecnología CASE. La información de especificaciones almacenada en el depósito CASE se emplea para generar el código, mantener los sistemas y poder reutilizarlos en el desarrollo de los futuros sistemas. Almacenando la información del sistema en el ordenador, será más fácil de adaptar a los más altos niveles de automatización del **software** según vayan apareciendo en el futuro. El cuadro 2.8 expone algunas de las cuestiones más importantes

que deben preguntarse acerca de las prestaciones del depósito de un **banco de trabajo CASE**.

INTEGRACION

El lenguaje de especificación, las herramientas de diagramación, las herramientas de prototipado, los diccionarios, los sistemas gestores de bases de datos, los compiladores, los generadores, etc., son las herramientas de alta productividad de principios de los años ochenta. Sin embargo, el obstáculo mayor a su facilidad de uso es que suelen ser herramientas individuales capaces de soportar solamente una parte del proceso del **software**. Generalmente, no tienen interfaces estandarizadas entre ellas y, además, son muy dependientes del tipo de ordenador, del sistema operativo y del lenguaje de programación. Como resultado, los profesionales del desarrollo se ven obligados a aprender a utilizar diferentes herramientas según el entorno en el que trabajan. Incluso dentro de un único entorno, no pueden aplicar el conocimiento de una herramienta a otra, porque cada herramienta tiene sus propios formatos de comandos, estructura de ficheros especializada y rango de opciones disponibles.

Muchas herramientas, independientemente de su potencia, han fallado al intentar aumentar la productividad y calidad del **software**, porque no proporcionaban un soporte integrado y continuo a los programadores en su trabajo diario [10]. La falta de integración reduce la productividad.

La integración de las herramientas es la base del concepto de **CASE workbench** para el desarrollo de **software**. Es la clave para que el uso de un **software** potente sea práctico. Las herramientas del **CASE workbench** deben interactuar unas con otras de forma consistente e intuitiva y deben ajustarse un conjunto de estándares. Deben cooperar entre ellas y evitar la duplicación de funciones o de mensajes. Esto reduce las posibilidades de error del usuario y evita la redundancia de datos de entrada cuando la misma información se necesita en diferentes herramientas.

El cuadro 2.9 expone los cinco niveles de integración. El primer nivel, **el interfaz común de usuario**, es lo mínimo que se espera de un **CASE workbench**. Reduce la curva de aprendizaje asociada con el **workbench**, ya que la experiencia adquirida en la utilización de una parte del **workbench** puede aplicarse para aprender otras partes. Por ejemplo, el acceso a la función

Cuadro 2.8. Prestaciones del depósito de la CASE

- ¿Está la información del depósito totalmente integrada o sólo parcialmente conectada al **CASE workbench**? (Un depósito integrado es superior, porque permite un almacenamiento más rápido de la información a través de los diagramas o directamente en el depósito y en un solo paso se actualiza cuando se cambia el contenido del depósito o el diagrama.)
- ¿Es la arquitectura de la información un diccionario o una base de datos? (Se necesita una base de datos con un potente sistema gestor de la información para gestionar toda la información del sistema.)
- ¿Qué tipos de información del sistema pueden almacenarse en el depósito y en qué cantidad? (Todos los tipos de información del sistema —técnica y de gestión del proyecto— además de las relaciones entre los objetos, deben almacenarse en el depósito).
- ¿Cómo se comparte la información del depósito entre los programadores? (Para soportar el equipo del proyecto y el desarrollo y mantenimiento de grandes sistemas, el depósito debe proporcionar acceso y control de multiusuarios.)
- ¿Qué tipos de informes sobre la información del depósito se proporcionan? (Deben proporcionarse informes de formato establecido y de cuestiones *ad hoc* al programador para darle una visión y evaluación fáciles del contenido del depósito.)
- ¿Existe una prestación de la importación/exportación hacia/desde los diccionarios y bases de datos externos al depósito? (La conexión entre los diccionarios y bases de datos externos es necesaria para acceder a la información estándar del sistema que ya ha sido definida para la organización.)
- Cuando se realiza algún cambio en la información del depósito ¿puede analizarse y notificarse el impacto en el sistema producido por el cambio? ¿Se cambian apropiadamente todos los objetos afectados? (El análisis y control automáticos de los cambios son dos de las prestaciones más importantes que debe proporcionar el depósito).

de ayuda (**help**) es de la misma forma en todo el **workbench**. El interfaz común de usuario es el puente que conecta varias herramientas del **workbench**. Cuando se añaden nuevas herramientas de terceros, las diferencias individuales de cada herramienta se cubren bajo el paraguas de un sistema de menús comunes.

Cuadro 2.9. Niveles de integración

- Interfaz común de usuario.
- Transferibilidad de datos entre herramientas.
- Integración de las fases de desarrollo a través de una representación del sistema almacenada en el depósito.
- Transferibilidad de datos y herramientas entre entornos **hardware**.
- Todo accesible en la estación de trabajo.

El segundo nivel de integración, **transferibilidad de datos entre herramientas**, es otro tipo de puente que conecta las herramientas del **workbench**. Cuando es necesario, se suministran las rutinas de selección de datos, las rutinas de conversión y las rutinas de transferencia de ficheros para convertir automáticamente los datos en un formato de entrada apropiado para una herramienta en particular del paquete de **software**. En un **workbench** que trabaja perfectamente es muy fácil para el usuario pasar datos de una herramienta a otra. En este nivel se espera que la salida de cada herramienta sea la entrada de otra [10].

Los dos primeros niveles tratan la integración entre herramientas, un modo de conectar herramientas para utilizar y aprender más fácilmente. El tercer nivel de integración trata de conexión entre las fases del ciclo de vida del **software**, la integración a través de los pasos de los procesos de desarrollo y de mantenimiento de los sistemas de **software**. En este caso el puente entre las varias fases del ciclo de vida es una representación común del sistema (pero que permite múltiples perspectivas o visiones del usuario) almacenada en el depósito CASE y compartida entre todos los componentes del proyecto. Una metodología estructurada es otro tipo de puente que vincula los pasos del desarrollo de **software** en un proceso significativo y manejable. Este nivel de integración une a programadores del equipo

usuarios y gestores, reduciendo los problemas de comunicación al proporcionar una fuente única y fácil de actualizar para toda la información del sistema y un procedimiento de desarrollo común para seguir.

Los dos últimos niveles de integración tratan de la integración a través de entornos de **hardware**, conectando ordenadores principales con micros y, en algunos casos, con miniordenadores también. El objetivo es poder realizar de actividades en el entorno más conveniente. Esto solamente es posible si los textos, códigos, datos y gráficos pueden transferirse fácilmente entre paquetes de **software** y entre entornos de **hardware**. Además, los **CASE workbench** ofrecen la integración entre funciones —gráficos, proceso de textos, ofimática— al proporciona poder todas estas funciones en la estación de trabajo.

RESUMEN

En este capítulo hemos visto el **CASE workbench**, un entorno que soporta todo el proceso de **software**. Las principales prestaciones funcionales del **CASE workbench** se resumen en el cuadro 2.10. Hemos tratado las cuatro primeras prestaciones en este capítulo, las restantes se verán en el capítulo 3.

Cuadro 2.10. El CASE workbench: el nuevo entorno de desarrollo de software

Los **CASE workbenches** proporcionan asistencia por ordenador durante el desarrollo, el mantenimiento y la gestión de los sistemas de **software**. La diferencia con los primitivos entornos de programación es su más amplia cobertura del ciclo de vida del **software**. Un **CASE workbench** completo debe tener las siguientes características:

- Interfaz gráfico para dibujar diagramas estructurados.
- Depósito para almacenar y gestionar toda la información del sistema.

Cuadro 2.10. El CASE workbench: el nuevo entorno de desarrollo de software (continuación)

- Conjunto de herramientas altamente integradas compartidas en un interfaz común de usuario.
- Herramientas para asistir en cada fase del ciclo de vida del **software**.
- Herramientas de prototipo.
- Generación automática de código a partir de las especificaciones de diseño.
- Soporte de la metodología del ciclo de vida del **software** con comprobaciones exhaustivas de lo creado en las herramientas.

BIBLIOGRAFIA

1. S. Gutz, A. Wasserman, y M. Spier, "Personal Development Systems for the Professional Programmer", *Computer*, Vol. 14, N° 4, abril 1981, págs. 45-53.
2. B. Kernigham y J. Mashey. "The Unix Programming Environment", *Computer*, Vol. 14, N° 4, abril 1981, págs. 12-24.
3. W. Teitelman y L. Masinter, "The Interlisp Programming Environment", *Computer*, Vol. 14, N° 4, abril 1981, págs. 25-34.
4. G. Raeder, "A Survey of Current Graphical Programming Techniques", *Computer*, Vol. 18, N° 8, agosto 1985, págs. 11-25.
5. James Martin y Carma McClure. *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs. Nueva Jersey: Prentice-Hall. 1985.

6. James Martin y Carma McClure, *Action Diagrams*. Englewood Cliffs, Nueva Jersey: Prentice-Hall. 1985.
7. Edward Yourdon y Larry Constantine. *Structured Design*. Englewood Cliffs. Nueva Jersey: Yourdon Press. 1978.
8. James Martin y Carma McClure, *Structured Techniques: The Basis for CASE*. Englewood Cliffs. Nueva Jersey: Prentice-Hall. 1988.
9. M. Mariconi y D. F. Hare: "PegaSys: A System for Graphical Explanation of Program Designs. *SIGPLAN Notices*. Vol. 20, N° 7, julio 1985, págs. 148-160.
10. L. Osterweil, "Software Environment Research: Directions for the Next Five Years", *Computer*, Vol. 14, N° 4, abril 1981, págs. 35-44.

CAPITULO 3

EL SOPORTE CASE A LOS PROCESOS DE SOFTWARE

EL SOPORTE DE LAS METODOLOGIAS Y LOS PROCESOS DE SOFTWARE

La función de un CASE **workbench** es soportar cada fase de los procesos de desarrollo y mantenimiento del **software** con un conjunto de herramientas integradas. Las herramientas CASE automatizan la producción de la documentación requerida en cada fase del desarrollo y del mantenimiento.

Por ejemplo, en las primeras fases del ciclo de vida del **software**, las herramientas CASE, como los lenguajes de generación de pantallas y de especificaciones ejecutables, se emplean para construir modelos, prototipos y simulaciones para descubrir los requerimientos del sistema y comprobar el comportamiento del mismo. En la fase de implantación del ciclo de vida del **software**, las herramientas CASE, como los generadores de código y de bases de datos, se utilizan para escribir automáticamente programas documentados a partir de las especificaciones de diseño de alto nivel.

Una parte importante del soporte del ciclo de vida del **software** es apoyar las metodologías que estructuran los pasos del proceso dentro del ciclo de vida. Un CASE **workbench** soporta el empleo de metodologías estructura-

das automatizando laproducción de la documentación requerida por la metodología y guiando al usuario en la correcta utilización de la metodología.

En este capítulo estudiaremos cómo los **CASE workbench** soportan tanto los procesos del ciclo de vida del **software** como las metodologías estructuradas analizando las cuatro prestaciones siguientes del **workbench**:

- La cobertura del ciclo de vida del **software**.
- El soporte de prototipos.
- La generación automática de código.
- El soporte de metodología estructurada.

EL ENFASIS EN LAS PRIMERAS FASES DEL CICLO DE VIDA

Una parte importante de la tecnología CASE se centra en las primeras fases del ciclo de vida del **software**. Este énfasis comienza con el reconocimiento de que las etapas de análisis y de diseño son las más críticas del ciclo de vida.

Los errores en las especificaciones pueden ser muy costosos si no se detectan y corrigen en las primeras fases. La corrección de un error de especificación durante la fase de mantenimiento puede llegar a ser cien veces más caro que si hubiera sido corregido durante la fase de análisis [1]. La integridad y la exactitud de las especificaciones del sistema afectan al éxito del esfuerzo del desarrollo de todo **software**. La especificación es la base de la planificación y asignaciones del proyecto, de la planificación de las pruebas, de la documentación del usuario y del diseño del programa. Una comprensión deficiente del sistema es la causa de los fallos del **software**.

Los errores de diseño a menudo dominan los proyectos de **software** dependiendo de su número y costo de corrección, especialmente cuando no se detectan pronto. En los proyectos grandes, los errores de diseño superan a menudo los de código. En un estudio de TRW sobre errores detectados durante o después de la prueba de aceptación, los errores de diseño fueron del 62% frente al 32% de los errores de código [2]. No solamente ocurrieron más errores de diseño, sino que su corrección más costosa que la de los errores de código.

Cuanto más cuidado se preste al diseño, menos costo y más fiable será el sistema. El diseño del sistema es la guía de su implantación. Si la guía no existe o es incorrecta, el sistema producido probablemente estará mal organizado, documentado y su mantenimiento será una pesadilla.

LOS PROTOTIPOS

Las herramientas de prototipo son una parte muy importante de la automatización de las primeras fases del ciclo de vida. Se utilizan para determinar los requerimientos del sistema y para responder a las cuestiones sobre el comportamiento del nascente sistema. El cuadro 3.1 muestra algunas herramientas CASE utilizadas para realizar los prototipos de varios aspectos del sistema **software**.

Cuadro 3.1. Herramientas CASE para prototipos

- * Generadores de pantallas.
- * Generadores de informes.
- * Constructores de menús.
- * Lenguajes de cuarta generación.
- * Lenguajes de especificaciones ejecutables.

Los generadores de pantallas, los generadores de informes y los constructores de menús se utilizan principalmente para hacer el prototipo del interfaz de usuario como una forma rápida y agradable de clasificar los requerimientos del usuario (ver figuras 3.1, 3.2 y 3.3). Los prototipos proporcionan al usuario un modelo concreto de cómo se verá el sistema desde la perspectiva del usuario. En este nivel de prototipos, el diálogo de las pantallas y la navegación de los datos puede simularse con o sin compilación. También pueden generarse automáticamente el código fuente, las pantallas y las descripciones de los informes. Este es un modo efectivo para identificar y corregir los equívocos sobre las expectativas del usuario del sistema.

Los lenguajes de cuarta generación pueden utilizarse para desarrollar un modelo más completo del sistema. En este caso, el prototipo incluye

```

PANEL DEFINITION MENU ***** END PROCESSING PERFORMED
COMMAND ==> _____

FUNCTION: UP   CR-CREATE   UP-UPDATE   PU-PURGE   SH-SHOW   LI-LIST

ITEM      PI  PI-IMAGE   PD-DEFIN
                FD-FIELD   CE-CONSIG  SL-SEGLOOP
                (UP)       (CR,UP)    (CR,UP,PU)

MEMBER NAME:
  HEADER  TR
  ID      XXAD_ PRE-CLASS ASSIGN, STUDENT NAME
  DESC    _____

ENTER VALUE FOR SPECIFIC ITEM TO BE PROCESSED:
  1. IMAGE      < > + | \ (INPUT OUTPUT OUTIN SELECT LIT-BREAK CHARACTERS)
                24 080 (LINE-COLUMN IMAGE SIZE)
                U      (UPPER/LOWER CASE LITERALS)
  2. DEFIN      Y Y Y Y N (INPUT OUTPUT OUTIN SELECT LITERAL FIELDS LISTED)
  3. FIELD      _____ (NAME OR LINE,COLUMN OR "+PANEL")
  4. CONSIG     _____ (TYPE - "XFEDIT", "SEGEDIT", OR BLANK FOR LIST)
                _____ (NAME - IF TYPE SPECIFIED)
  5. SEGLOOP    _____ (TYPE - "FILE" OR "TABLE")
                _____ (FROM NAME OR LINE,COLUMN)
                _____ (TO NAME OR LINE,COLUMN)

```

Figura 3.1. Ejemplo de una pantalla obtenida con el generador de pantallas de TELON Pansophic

las funciones principales del sistema pero no comprueba las excepciones ni los datos de entrada inválidos, ni se preocupa de los rendimientos de ejecución. Su propósito es proporcionar experiencia con el sistema al usuario al utilizar un modelo bastante completo. En ocasiones, el modelo es adecuado para utilizarlo como sistema real.

Los lenguajes de especificaciones ejecutables son las herramientas de prototipo más sofisticadas. Cambian el desarrollo del sistema en un proceso iterativo donde el sistema se va especificando y las especificaciones se van ejecutando para determinar si el sistema está completo y es correcto. Después, basándose en la experiencia al utilizar el prototipo, las especi-

```
TRANCODE                               EXECUTIVE FILING SYSTEM                                MM-DD-YY HH:MM:SS

                                     UPDATE APPOINTMENT DETAILS
DATE: MM-DD-YY   TIME: XXXXXXXX             LOCATION: XXXXXXXXXXXXXXXX
OBJECTIVE: XXXXXXXXXXXXXXXXXXXXXXXX          CONFIRMATION INDICATOR: X

REMARKS:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

SEL      LAST NAME           FIRST        MI      PHONE NUMBER    COMPANY NAME
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
X  XXXXXXXXXXXXXXXX     XXXXXXXXXX  X   ( 999 ) 9999999  XXXXXXXXXXXXXXXX
<<< PFK <<< PFK <<< PFK <<< PFK <<< PFK >>> PFK >>> PFK >>> PFK >>> PF
<<< ERR <<< ERR <<< ERR <<< ERR <<< ERR >>> ERR >>> ERR >>> ERR >>> EF
```

Figura 3.2. Ejemplo de diseño de pantalla obtenido con el generador de pantallas de INFORMATION ENGINEERING FACILITY™ de Texas Instruments Incorporated

caciones se refinan y vuelven a ejecutar. El proceso iterativo continúa hasta que el sistema es capaz de funcionar de forma que satisfaga totalmente todos los requerimientos del usuario.

REPORT LAYOUT CHART							
1-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80
123456789012345678901234567890123456789012345678901234567890123456789							
<p style="text-align: center;">ABC</p> <p style="text-align: center;">CORPORATION</p> <p style="text-align: center;">PARTS MOVE REPORT</p> <p style="text-align: center;">-----</p>							
PART-ID	MGR	APPR	LOCATION	QTY-STOCK	QTY-REQUEST	STATUS	
-----	-----	-----	-----	-----	-----	-----	
1) 999999	xxx	xx	9,999.99	9,999.99	99/99/99	xxx	
2) 999999	xxx	xx	9,999.99	9,999.99	99/99/99	xxx	
3) 999999	xxx	xx	9,999.99	9,999.99	99/99/99	xxx	
4) 999999	xxx	xx	9,999.99	9,999.99	99/99/99	xxx	

*Figura 3.3. Ejemplo de generador de informes con DESIGNAID de Nas-
tec Corporation*

LA SIMULACION

Ciertos tipos de sistemas de **software** están incorporados en otros sistemas. Tales sistemas, llamados de **tiempo real o embebidos**, deben interactuar con el sistema del entorno mediante señales y controles.

En los sistemas de tiempo real o embebidos es importante simular también como prototipo el sistema en los estadios del análisis y de diseño. En tanto que un prototipo es un modelo del sistema de **software**, la simulación es un modelo del entorno al cual el sistema del **software** debe responder y controlar. La simulación proporciona una visión previa del comportamiento del sistema de **software** en su entorno verdadero de tiempo real [3]. Las entradas verdaderas en tiempo real desde el entorno deben estar provistas de mecanismos para verificar la exactitud e integridad del modelo de diseño del sistema. Con la ayuda de las herramientas CASE de simulación, el programador puede encontrar los errores de diseño incluso antes del comienzo de la fase de codificación. Los sistemas de tiempo real o embebidos se tratan más adelante en este capítulo, en la sección “Sistemas de tiempo real”.

LA GENERACION DE CODIGO

Al otro extremo del prototipo en el ciclo de vida del **software** está la generación de código. Un **CASE workbench** automatiza la fase de implementación del programa partiendo de las especificaciones de diseño del programa. Se genera un esquema o un programa completo.

En el caso de la generación del esquema, se generan automáticamente código para una base de datos, fichero, pantalla y descripción de informes, más una salida comentada de la lógica de procedimiento. La lógica adicional del programa debe codificarse manualmente para completar el programa.

En el caso de la generación completa del programa, desde las especificaciones del diseño, se genera automáticamente el programa completo más su documentación. Lo cual incluye:

- * Código ejecutable.
- * JCL (**Job Control Language**).


```

      (GROUP-OUTPUT-POS)
      COMPUTE LAST-STATEMENT-NUM = 0000000064
*
      MOVE 192 TO LAST-NAME-DYN1 OF LAST-NAME-ATTR OF ATTENDEE OF
        OUTPUT-Z OF GROUP-OUTPUT (GROUP-OUTPUT-POS)
      ADD 32 TO LAST-NAME-DYN1 OF LAST-NAME-ATTR OF ATTENDEE OF
        OUTPUT-Z OF GROUP-OUTPUT (GROUP-OUTPUT-POS)
      MOVE 'B7' TO LAST-NAME-DYN2 OF LAST-NAME-ATTR OF ATTENDEE OF
        OUTPUT-Z OF GROUP-OUTPUT (GROUP-OUTPUT-POS)
      COMPUTE LAST-STATEMENT-NUM = 0000000065
*
      MOVE 192 TO FIRST-NAME-DYN1 OF FIRST-NAME-ATTR OF ATTENDEE OF
        OUTPUT-Z OF GROUP-OUTPUT (GROUP-OUTPUT-POS)
      ADD 32 TO FIRST-NAME-DYN1 OF FIRST-NAME-ATTR OF ATTENDEE OF
        OUTPUT-Z OF GROUP-OUTPUT (GROUP-OUTPUT-POS)
      MOVE 'B7' TO FIRST-NAME-DYN2 OF FIRST-NAME-ATTR OF ATTENDEE
        OF OUTPUT-Z OF GROUP-OUTPUT (GROUP-OUTPUT-POS)

      PARA-0008651125-EXIT.
      EXIT.
*
*
*
      PARA-0008651138.
*
      COMPUTE LAST-STATEMENT-NUM = 0000000070
      MOVE GROUP-REFD TO GROUP-OUTPUT-REF
      MOVE FUNCTION-CODE OF WORK-AREA OF INPUT-Z OF INPUT-GROUP
        (INPUT-GROUP-POS)
        TO FUNCTION-CODE OF WORK-AREA OF OUTPUT-Z OF GROUP-OUTPUT
        (GROUP-OUTPUT-POS)
      COMPUTE LAST-STATEMENT-NUM = 0000000071
      MOVE GROUP-REFD TO GROUP-OUTPUT-REF
      MOVE LAST-NAME OF ATTENDEE OF INPUT-Z OF INPUT-GROUP
        (INPUT-GROUP-POS)
        TO LAST-NAME OF ATTENDEE OF OUTPUT-Z OF GROUP-OUTPUT
        (GROUP-OUTPUT-POS)
      MOVE FIRST-NAME OF ATTENDEE OF INPUT-Z OF INPUT-GROUP
        (INPUT-GROUP-POS)
        TO FIRST-NAME OF ATTENDEE OF OUTPUT-Z OF GROUP-OUTPUT
        (GROUP-OUTPUT-POS)
      MOVE MIDDLE-INITIAL OF ATTENDEE OF INPUT-Z OF INPUT-GROUP
        (INPUT-GROUP-POS)
        TO MIDDLE-INITIAL OF ATTENDEE OF OUTPUT-Z OF GROUP-OUTPUT
        (GROUP-OUTPUT-POS)
      MOVE AREA-CODE OF ATTENDEE OF INPUT-Z OF INPUT-GROUP
        (INPUT-GROUP-POS)
        TO AREA-CODE OF ATTENDEE OF OUTPUT-Z OF GROUP-OUTPUT
        (GROUP-OUTPUT-POS)
      MOVE PHONE-NUMBER OF ATTENDEE OF INPUT-Z OF INPUT-GROUP
        (INPUT-GROUP-POS)
        TO PHONE-NUMBER OF ATTENDEE OF OUTPUT-Z OF GROUP-OUTPUT
        (GROUP-OUTPUT-POS)
      MOVE CORPORATION OF ATTENDEE OF INPUT-Z OF INPUT-GROUP
        (INPUT-GROUP-POS)
        TO CORPORATION OF ATTENDEE OF OUTPUT-Z OF GROUP-OUTPUT
        (GROUP-OUTPUT-POS)
      COMPUTE LAST-STATEMENT-NUM = 0000000072
*
      MOVE 192 TO LAST-NAME-DYN1 OF LAST-NAME-ATTR OF ATTENDEE OF

```

Figura 3.4. Ejemplo de código generado automáticamente en COBOL por el INFORMATION ENGINEERING FACILITY™ de Texas Instruments Incorporated

- * Accesos a las base de datos y a los ficheros (pantallas, direccionamientos, descripciones de las bases de datos).
- * Pantallas de ayuda.
- * Mensajes de error.
- * Documentación del usuario y del programa.
- * Casos de prueba.

Todos estos componentes del programa completo se juntan y se almacenan en el depósito CASE para su fácil acceso, actualización sincronizada y posterior mantenimiento.

El código generado puede ser código objeto o código fuente. Muchos profesionales prefieren el código fuente, porque en caso de necesidad puede ser comprendido y programado por programadores humanos. Además, es más portátil y puede ser más compatible con el sistema de **software** existente. Los lenguajes más extendidos (COBOL, PL/1, C, ADA, Fortran, etc.) y los propios del suministrador son normalmente generados por el **CASE workbench**. Las figuras 3.4 y 3.5 son ejemplos de generación automática de código fuente en COBOL. La figura 3.6 es un ejemplo de generación automática de código en PL/1. La Figura 3.7 es un ejemplo de código fuente en el lenguaje propiedad del suministrador.

El código se genera para operar en entornos como MVS, CICS, IMS/DC, DB2, DC1, VSAM, VM/SQL, DS, TSO, IDMS, ADABAS, DBS y OS/1100. La posibilidad de generar códigos para varios entornos es una parte muy importante de las prestaciones de un **CASE workbench**.

El código objeto ofrece la ventaja de una mayor eficiencia, y probablemente llegue a estar más extendido en el futuro entre los usuarios de los **CASE workbench**, a medida que se convencen de la fiabilidad y eficiencia de la generación automática de código. Además, como la capacidad del **CASE workbench** aumenta el nivel del mantenimiento de los sistemas desde las especificaciones de diseño, el código objeto generado se considerará más práctico que el código fuente.

EL SOPORTE DE LA METODOLOGIA ESTRUCTURADA

El **CASE workbench** soporta metodologías estructuradas como análisis estructurado, diseño estructurado y programación estructurada al proporcionar herramientas para automatizar las metodologías. Hay dos niveles de automatización:

- 1.—La preparación de la documentación.
- 2.—Los pasos del proceso de la metodología.

La preparación automatizada de la documentación significa proporcionar soporte gráfico para el dibujo de diagramas estructurados (de flujo, de entidad/relación, de transición de estado y de acción). También significa la producción automática de especificaciones textuales (miniespecificaciones y pseudocódigo). Las especificaciones textuales se emplean para proporcionar una información más detallada sobre los procedimientos del programa y de la estructura de los datos referenciados en los diagramas estructurados de un nivel más alto.

Como quiera que las diferentes metodologías estructuradas utilizan diagramas distintos para modelar y documentar un sistema de **software**, las metodologías estructuradas soportadas por un **CASE workbench** particular estarán determinadas por los tipos de diagramas estructurados y las convenciones de notación que pueda ofrecer. Por ejemplo, la metodología de diseño estructurado de Yourdon emplea un diagrama de estructura (**structure chart**) derivado de un diagrama de flujo para representar el diseño del programa; la metodología de diseño de Jackson utiliza el diagrama estructurado en árbol y la metodología de desarrollo de sistemas de datos estructurados de Orr emplea los diagramas de Warnier-Orr.

El segundo nivel de soporte de metodología, la **automatización de los pasos del proceso de la metodología**, significa que el **CASE workbench** guía al usuario en el correcto uso de una metodología estructurada. Esto requiere que al menos cierto nivel de comprensión de la metodología esté incorporado en las herramientas del **CASE workbench**. Esto podría ser tan simple como incorporar paneles de ayuda para describir cada paso de la metodología o de listas de comprobación (**check list**) que incluyan las entradas y salidas requeridas en cada paso. Podría incluir un mecanismo de comprobación para asegurar que cada salida requerida por la metodología está presente, correcta y completa antes de permitir al usuario ir al paso siguiente.

```

                                Data Definition
                                (NAME('AAAC2.DSNDBC.DR.T0000174.I0001.A001') -
                                NONINDEXED SHR(3,3) RESUSE -
                                CONTROLINTERVALSIZE(4096) RECORDSIZE(4089,4089)) -
DATA -
                                (NAME('AAAC2.DSNDBD.DR.T0000174.I0001.A001') -
                                CYLS(1 1) VOL()) -
                                CATALOG (AAAC2)

DEFINE CLUSTER -
                                (NAME('AAAC2.DSNDBC.DR.T0000141.I0001.A001') -
                                NONINDEXED SHR(3,3) RESUSE -
                                CONTROLINTERVALSIZE(4096) RECORDSIZE(4089,4089)) -
DATA -
                                (NAME('AAAC2.DSNDBD.DR.T0000141.I0001.A001') -
                                CYLS(1 1) VOL()) -
                                CATALOG (AAAC2)

CREATE DATABASE "DR"
BUFFERPOOL BP0
;

CREATE TABLESPACE "T0000023" IN "DR"
BUFFERPOOL BP0
LOCKSIZE PAGE
CLOSE NO
USING VCAT AAAC2
PCTFREE 48
;

CREATE TABLE EXECUTIVE
(
LAST_NAME CHAR(15) NOT NULL,
FIRST_NAME CHAR(10) NOT NULL,
MIDDLE_INITIAL CHAR(1) NOT NULL,
DEPARTMENT CHAR(25) NOT NULL,
LOCATION CHAR(25) NOT NULL,
JOB_TITLE CHAR(20) NOT NULL,
OPERATOR_ID CHAR(3) NOT NULL,
PHONE_AREA_CODE SMALLINT NOT NULL,
PHONE_NUMBER INTEGER NOT NULL,
)
IN "DR"."T0000023"
;

CREATE TABLESPACE "T0000005" IN "DR"
BUFFERPOOL BP0
LOCKSIZE PAGE
CLOSE NO
USING VCAT AAAC2
PCTFREE 48
;

```

Figura 3.5. Ejemplo de tablas de DB/2 generadas automáticamente desde el diagrama de entidad/relación y el diagrama de estructura de datos por el INFORMATION ENGINEERING FACILITY™ de Texas Instruments Incorporated

```

PL/I OPTIMIZING COMPILER          /* TRPPGDA TELON CICS PROGRAM

STMT

322  E_100_INPUT_EDITS: PROC;                                -- 339
    /******
    *          E _ 1 0 0 _ I N P U T _ E D I T S
    *          *****
    * THIS ROUTINE CONTAINS THE INPUT EDIT LOGIC GENERATED*
    * FROM THE FIELD STATEMENT PARAMETERS. STANDARD
    * EDITS SUCH AS REQ, CONVERT AND VALUES ARE GENERATED *
    * IN THIS SECTION. SPECIAL FLDTYPES ARE LINKED TO
    * WITH CALL STATEMENTS.
    *
    * GENERATED - FIELD EDIT LOGIC
    * COPY CODE - SEGLOOP/ICUSTOM
    * COPY CODE - SCREEN/FLDEDIT
    *          *****
323  SECTION_NAME_TABLE(SEC_INDEX) = TRACE_SECTION_NAME;    -- 353
324  SEC_INDEX = SEC_INDEX + 1;                                -- 354
325  TRACE_SECTION_NAME = 'E_100';                             -- 355
326  TRACE_FIELD_NAME, TRACE_SEGMENT_NAME = ' ';              -- 356
    /*
    * CHECK IF HELP FIELD REQUEST HAS BEEN MADE.
    */
327  CALL M_100_HELP_ANALYZE;                                   -- 359
328  IF CONTROL_INDICATOR = CONTINUE_PROCESS_LIT              -- 360
    THEN GO TO E_100_INPUT_EDITS_RETURN;                      -- 362

    /* ID FIELD */                                           -- 363

329  TRACE_FIELD_NAME = 'ID';                                   -- 364
330  IF TPI_ID = ' '                                           -- 365
    THEN DO;                                                  -- 366
331  CALL IFORMAT (FIELD_EDIT_ERROR,                          -- 367
                  TPI_ID_LTH,                                -- 368
                  TPI_ID,                                     -- 369
                  WORKFLD_ALPHA,                             -- 370
                  FORMAT_ID,                                  -- 371
                  FORMAT_ID_LTH);                             -- 372
332  IF FIELD_EDIT_ERROR = ' '                                 -- 373
    THEN                                                    -- 374
        EMPL_ID = SUBSTR(WORKFLD_ALPHA,1,FORMAT_ID_LTH);    -- 375
    ELSE DO;                                                  -- 376
333  CONTROL_INDICATOR = DO_WRITE_LIT;                         -- 377
334  TPO_ID_ATTR = ERROR_ATTR;                                -- 378
335  END;                                                       -- 379
336  END;                                                       -- 380
337  ELSE                                                    -- 381
    EMPL_ID = ' ';                                           -- 382

```

Figura 3.6. Ejemplo de lenguaje propio del distribuidor, el cual se documenta automáticamente utilizando el formato de un diagrama de acción. Se obtuvo automáticamente con el generador de aplicaciones en CORVISION de Cortex para el entorno DEC VAX.

En este caso, al usuario no solamente se le guía, sino que, además, se le fuerza a seguir los pasos con un orden y método normalizado. El propósito, en definitiva, es normalizar y sistematizar el proceso de desarrollo de software.

```

*
*  VALIDATE.INC
*  -----
*
*  WRITTEN:  07/12/83      JCW
*
*  NARRATIVE:      ROUTINES USED TO VALIDATE ENTRIES
*
*****
*
*  SECTION CHECK_VALUE
*
*****
*
*  INPUT:
*
*      STRING:      VALUE TO BE CHECKED ('T:XXX...XXX')
*                  T=EXTERNAL.TYPE, XXX...XXX=VALUE STRING
*
*  LET ERROR=""
*  IF 1*STRING EQ "C", EXITB
*
*  IF ";" NIN -2*STRING, THEN
*      ##STACK=-2*STRING+" ; ; "
*
*  ELSE
*      ##STACK=""
*
*  IF 1*STRING EQ "D", THEN
*
*      VALIDATE DATE
*
*      IF ##STACK NE "", INPUT DATE$D
*      IF ##STACK EQ "", LET ERROR="INVALID DATE"
*
*  ELSE
*
*      VALIDATE NUMBER
*
*      IF ##STACK NE "", INPUT NUMERIC.DOLLARS
*
*      CONDITIONAL
*      ##STACK EQ "", LET ERROR="INVALID NUMBER"
*
*      -2*STRING LL "...<*.>-...", LET ERROR= ;
*      "NUMBER MAY NOT CONTAIN EMBEDDED MINUS SIGNS"
*
*      (1*STRING EQ "I" AND ;
*      -2*STRING LL "...%...."), LET ERROR= ;
*      "INTEGER VALUE MAY NOT CONTAIN A DECIMAL POINT"
*
*      -2*STRING LL "...%....%....", LET ERROR= ;
*      "NUMBER MAY NOT CONTAIN MORE THAN ONE DECIMAL POINT"
*
*      (1*STRING EQ "S" AND ;
*      -2*STRING LL "...%.***..."), LET ERROR= ;
*      "DOLLAR VALUE MAY NOT HAVE MORE THAN TWO DECIMAL PLACES"
*
*  DISPLAY ANY ERROR MESSAGE
*
*  IF ERROR NE "", THEN
*      PERFORM DISPLAY ERROR ;*SCRSTUFF.INC

```

Figura 3.7. Ejemplo de un lenguaje suministrado por el vendedor que se documenta automáticamente, con formato de diagrama de acción. Se obtuvo automáticamente con el generador de Cortex de CORVISION, para los entornos DEC VAX

Existen dos escuelas de pensamiento sobre si el segundo nivel de automatización de la metodología estructurada debe o no estar incorporado en la herramienta CASE. El argumento a favor es que se introduce un control sobre el proceso de desarrollo del **software** y la normalización impuesto por la herramienta CASE. Además, puede realizarse un control más completo y exhaustivo sobre la calidad e integridad, pues muchas comprobaciones de la calidad son específicas de la metodología. Por ejemplo, las medidas de acoplamiento y cohesión son comprobaciones de diseño que forman parte solamente de la metodología de diseño estructurado de Yourdon. En general, cuanto más pueda definirse específicamente y mayor potencia habrá para la automatización del **software**. La automatización de los pasos de la metodología se trata en el capítulo 12.

Por otra parte, el argumento en contra es que la no incorporación del proceso de la metodología en la herramienta de CASE ofrece al usuario la flexibilidad de elegir la metodología (o la parte de la metodología apropiada para desarrollar un sistema en pantalla).—Esto es un débil argumento, ya que la mayoría de los profesionales del desarrollo apenas conocen bien una metodología y mucho menos están al día de las muchas metodologías para poder elegir la más apropiada para el desarrollo de un proyecto en particular.

Además, incorporar los pasos de la metodología en la herramienta CASE no significa que el usuario no tenga la opción de ignorar la sugerencia de la herramienta en cuanto al siguiente paso apropiado o un aviso sobre la deficiencia en la calidad.

LA CLASIFICACION DE LAS METODOLOGIAS ESTRUCTURADAS

Las metodologías estructuradas pueden clasificarse en pertenecientes a la escuela de *ingeniería de software* y las de la escuela de la *ingeniería de la información*; en orientadas a los procedimientos, a los datos o a la información, y en las que desarrollan sistemas de tiempo real o de información.

La ingeniería del software

La ingeniería de **software** es una fórmula descendente (*top-down*) de la implantación por fases para el desarrollo de programas [4]. Sus premisas

básicas se presentan en el cuadro 3.2. La *ingeniería de software* propone un proceso de desarrollo paso a paso que comienza con la visión funcional más general de lo que puede hacer el programa, descompone esta visión en subfunciones y repite el proceso por cada subfunción hasta que todas las subfunciones sean lo suficientemente pequeñas para ser implantadas en código de programas. El resultado es un programa modular estructurado jerárquicamente. Aunque el diseño de los componentes funcionales y el diseño de la estructura de los datos se realizan en paralelo, el diseño de la estructura de los datos se pospone hasta el diseño de los procedimientos para proteger la independencia y claridad de los componentes funcionales. Los requerimientos de los datos del programa se descubren a través del análisis de los requerimientos funcionales del programa. De esta forma, la *ingeniería de software* se enfoca hacia una visión funcional del programa.

La figura 3.8 muestra los pasos básicos seguidos en un método de *ingeniería de software* de un desarrollo de un programa. Obsérvese que el proceso comienza con una visión dinámica del flujo de datos antes de centrarse en una descomposición funcional. Obsérvese también que las consideraciones del diseño lógico se separan de las consideraciones físicas y que el diseño lógico del programa precede al diseño físico. Los tipos de diagramas estructurados que se utilizan para representar un diseño del programa con la *ingeniería de software* se muestran en el cuadro 3.3.

Cuadro 3.2. Las premisas básicas de la ingeniería de software

- * Un planteamiento descendente (**top-down**) de implantación por fases requiere un procedimiento gradual desde el nivel más alto de control y de definición de datos, descendiendo hasta los módulos funcionales y las estructuras de los datos.
- * Un aspecto muy importante del proceso descendente es la insistencia en la firmeza de los requerimientos.
- * La solución al sistema se define en funciones; los datos se diseñan para preservar la funcionalidad del sistema y la independencia de los módulos del programa. Por tanto, los requerimientos de los datos se descubren a través del análisis de las funciones requeridas por el sistema.

Cuadro 3.3. Los diagramas de la ingeniería de software

Los tipos de diagramas necesarios para soportar un planteamiento en ingeniería de software para el desarrollo del sistema son:

- * **Diagrama de flujo de datos:** muestra los procesos funcionales desde el nivel más alto en un sistema y el flujo de datos desde la adquisición a través del proceso y las eventuales salidas.
- * **Diagramas estructurados en árbol:** muestran las relaciones entre los diseños del programa y muestran la estructura jerárquica de los datos.
- * **Diagrama detallado de la lógica del procedimiento:** muestra detalladamente la lógica del programa: secuencia, selección, iteración.
- * **Diseño de pantallas y de informes:** muestra el diseño del interfaz del usuario.

La ingeniería del **software** representa uno de los primeros intentos para estructurar el proceso de desarrollo del **software**. Desde los años sesenta se ha avanzado en el cambio del desarrollo del **software** desde un arte manual a una disciplina casi mecanizada con el objetivo de introducir un enfoque normalizado al desarrollo del **software**. Como explica Bauer, la ingeniería de **software** es “el establecimiento y el empleo de los principios (métodos) de la ingeniería para obtener económicamente **software** que sea fiable y que trabaje sobre máquinas reales” [5, p.530]. Los conceptos y técnicas involucrados en la ingeniería de **software** incluyen:

- * El diseño descendente (**top-down**).
- * La programación estructurada.
- * La modularidad.
- * El refinamiento progresivo y la descomposición funcional.
- * La métrica de la calidad del **software**.
- * La abstracción de los lenguajes de programación.

- Aproximación descendente orientada-al-programa de Desarrollo de Sistemas, basada en el Análisis del Flujo de los datos.
- Metodología para la Gestión de la Programación

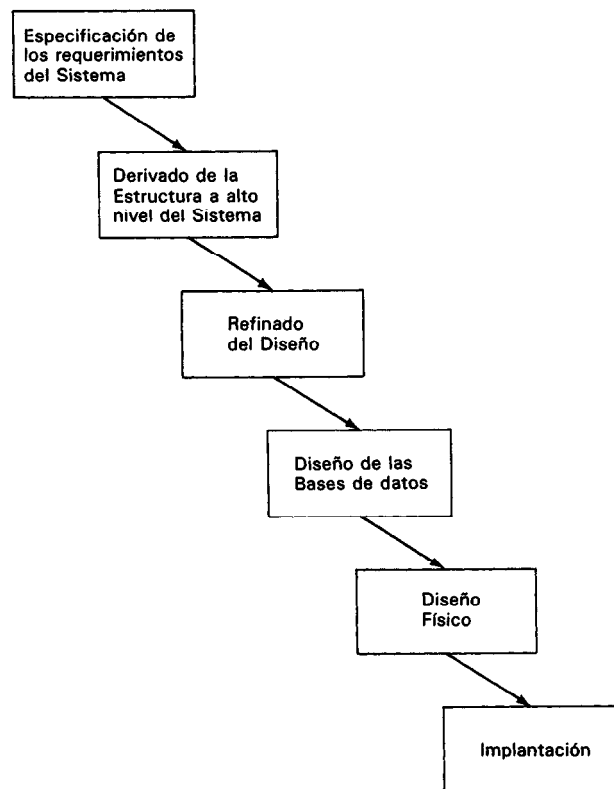


Figura 3.8. Los pasos básicos seguidos en la ingeniería de software durante el desarrollo de un programa

Las metodologías de análisis estructurado de DeMarco y Gane-Searson y la metodología de diseño de Yourdon son ejemplos de metodologías estructuradas populares pertenecientes a la escuela de la ingeniería de **software**.

La ingeniería de **software** es una disciplina general de desarrollo que puede aplicarse al desarrollo de los sistemas de información y a los sistemas de tiempo real. Por ejemplo, las metodologías de diseño para sistemas de tiempo real de Ward-Mellor y de Hatley están basadas en la metodología de diseño estructurado de Yourdon-Constantine empleada para desarrollar sistemas de información.

El desarrollo de software orientado al procedimiento frente al orientado a los datos

La ingeniería de **software** está fundada sobre el modelo básico de entrada/proceso/salida de un sistema:



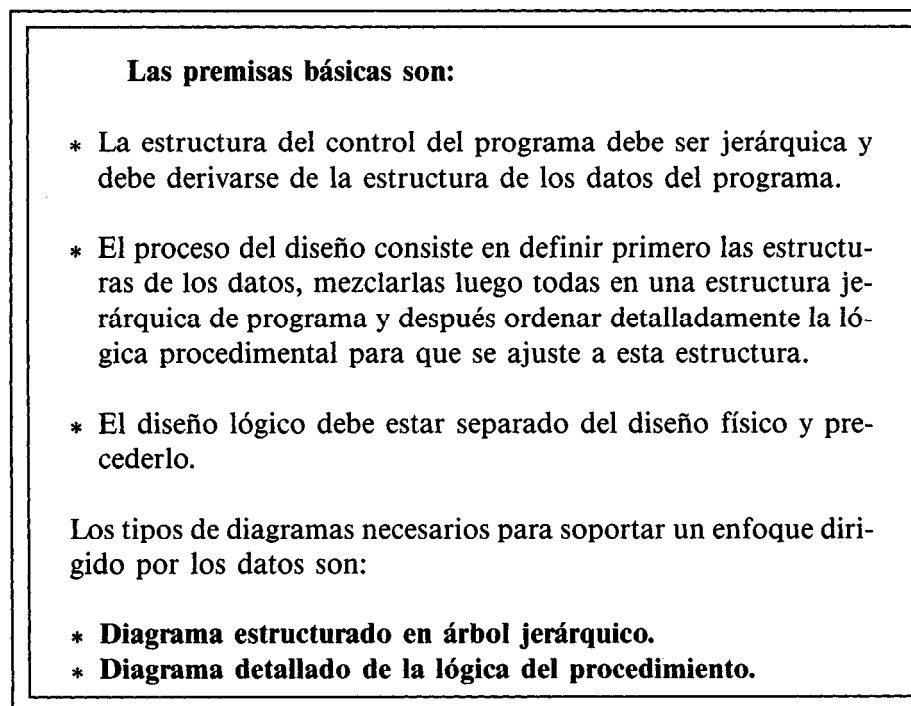
Los datos se introducen en el sistema como entradas, se opera con ellos (se transforman) por medio de un proceso y lo abandonan como salidas [6 págs. 503 - 504].

Este modelo básico se utiliza en todas las metodologías de diseño estructurado. Sin embargo, el orden en que se construye el modelo depende de la metodología particular que se sigue. La ingeniería de **software** tradicional trata la parte del proceso como la fundamental del modelo del sistema y por tanto se describe como un enfoque del desarrollo del **software orientado al procedimiento**. Los datos se derivan de la función.

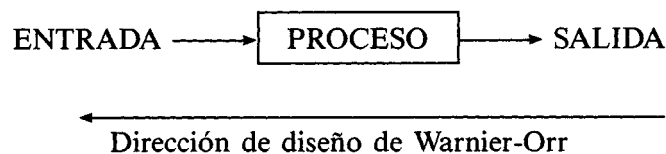
En un enfoque de desarrollo **orientado a los datos**, la parte del modelo correspondiente a las entradas y salidas se considera como la más importante. Por tanto, las estructuras de los datos se definen primero y los componentes procedimentales se derivan de las estructuras de los datos. El cuadro 3.4 muestra las premisas básicas del enfoque de desarrollo dirigido por los datos. Los tipos de diagramas necesarios para soportar un enfoque de desarrollo dirigido por los datos son diagramas de árbol estructurado jerárquicamente para representar las estructuras de los datos y de los programas y el diagrama detallado de la lógica de procedimiento.

La metodología de diseño de Jackson es un ejemplo de metodología orientada a los datos. En el enfoque de Jackson, las estructuras de los datos de entrada y salida se definen primero. Después, se define la estructura del programa mezclando todas las estructuras de los datos del programa.

La metodología de diseño de Warnier-Orr es otro ejemplo de enfoque orientado a los datos. Pero a diferencia del enfoque de Jackson, la formulación de Warnier-Orr se centra en las salidas del programa. Es una meto-

Cuadro 3.4. El enfoque dirigido por los datos para el desarrollo de sistema.

dología **orientada a las salidas**. Utilizando la formulación de Warnier-Orr se comienza por la definición de las salidas del sistema y se trabaja hacia atrás a través del modelo básico del sistema hasta definir las partes del diseño correspondientes a los procesos y a las entradas.



El enfoque del desarrollo centrado en la información

En las metodologías de desarrollo de sistemas de **formulación centrada en la información**, un modelo de dato lógico que represente la informa

ción utilizada a través de una organización es el punto de arranque de todo el desarrollo del sistema. El proceso empieza con un análisis de alto nivel de una corporación, sus objetivos comerciales y las necesidades de que muestra información estratégica. Una visión global de las necesidades de información se representa en un modelo que muestra todas las entidades de datos básicos de la organización y sus relaciones entre sí. Después, basándose en este modelo se construyen individualmente los sistemas de información de la organización. Así los procedimientos se derivan de los datos.

La ingeniería de la información

La **ingeniería de la información** es un ejemplo de una formulación de desarrollo centrada en la información. En el cuadro 3.5 se exponen las premisas básicas de la ingeniería de la información [6 págs. 651-655].

La ingeniería de la información es tanto un enfoque comercial como de ingeniería como una formulación para la construcción de sistemas de **software**. Es una formulación comercial porque comienza con una planificación estratégica de la organización. Es una formulación de ingeniería porque proporciona un procedimiento paso a paso para construir sistemas de información. La figura 3.9 describe los pasos lógicos de la formulación de la ingeniería de la información.

La ingeniería de la información es una disciplina más reciente que la ingeniería de **software**. En cierto sentido tiene un enfoque más amplio que la ingeniería de **software**. Es una construcción del sistema, más que una disciplina de construir programas. Comienza por un nivel más alto (con la planificación estratégica) que la ingeniería de **software**. Sin embargo, en la fase de diseño del programa, su formulación del desarrollo del programa es básicamente la misma que la de la ingeniería de **software**.

En otro sentido, sin embargo, la ingeniería de la información tiene un enfoque más estrecho que la ingeniería de **software**. Se emplea aquélla para construir sistemas de información, mientras que la ingeniería de **software** se emplea para desarrollar toda clase de sistemas (tanto de tiempo real como comerciales).

La ingeniería de la información es un enfoque orientado a la información para el desarrollo de software y por tanto difiere de los enfoques

orientados a los datos, como las de Jackson y de Warnier-Orr. La ingeniería de la información ha sido diseñada para desarrollar sistemas de bases de datos y puede trabajar con estructuras de datos no jerarquizadas. El modelado lógico de datos y la normalización son pasos requeridos en la ingeniería de la información. La ingeniería de la información construye sistemas de información integrados, porque están contruidos sobre el mismo modelo lógico de datos.

- Aproximación descendente y orientada a los datos para el desarrollo de sistema

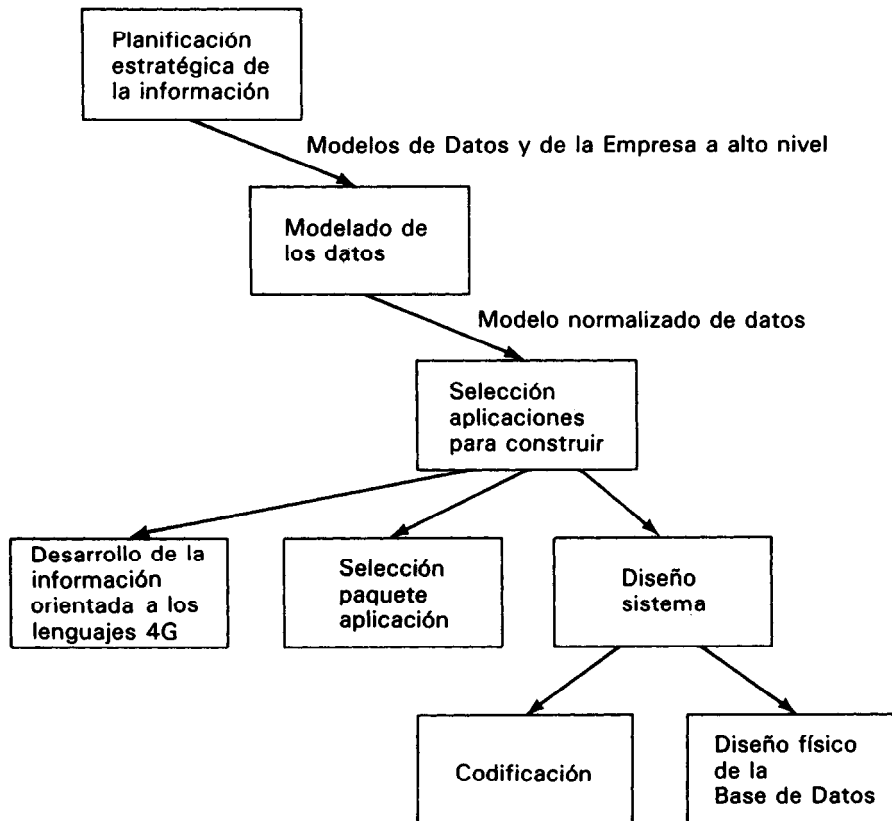


Figura 3.9. Pasos básicos en la ingeniería de la información para desarrollar un sistema

Cuadro 3.5. Las premisas básicas de la ingeniería de la información

- * Se necesita una estrategia general de desarrollo del sistema centrada en una planificación de los objetivos estratégicos del negocio para construir sistemas que satisfagan mejor las necesidades de la corporación.
- * Sistemas de información que puedan integrarse mejor si los datos a compartir se controlan centralmente por ser parte de un mismo modelo lógico de datos.
- * La representación lógica de datos es relativamente estable, en tanto que los procedimientos que utilizan los datos cambian frecuentemente. Por tanto, el modelo lógico de datos, que refleja lo que es la organización, no como trabaja, deberá ser la base del desarrollo del sistema.

Además el énfasis de la ingeniería de la información está en los datos desde el punto de vista de los requerimientos comerciales y no en los requerimientos del sistema de **software**. Todo el enfoque del desarrollo de la ingeniería de la información se centra en la visión lógica de como el sistema utiliza los datos. La ingeniería de la información se trata más adelante en este capítulo, en la sección “Metodología de Ingeniería de la Información de Martin”.

Soporte de los sistemas de tiempo real frente a los sistemas de información

Los diferentes tipos de sistemas requieren diferentes tipos de soporte de desarrollo. Dos ejemplos de tipos de sistemas son los de tiempo real/embebidos y los de información. El cuadro 3.6 muestra las diferencias principales entre ambos [3].

Los sistemas de tiempo real

Los **sistemas de tiempo real** (**real-time systems**) típicamente son sistemas que controlan, y son controlados por, eventos externos. La respuesta a tales eventos (por ejemplo, señal, disparador, disponible/no disponible) de una forma oportuna es una de las principales funciones de un sistema de tiempo real [3]. Cuando el sistema reconoce un evento al recibir una señal de entrada, realiza ciertos cálculos y actividades lógicas para responder al evento produciendo una salida.

Los ejemplos de sistemas de tiempo real incluyen: la navegación aérea, las redes de comunicaciones, los sistemas de **software**, el control de procesos de fabricación y los procesos químicos.

Para especificar los requerimientos de un sistema de tiempo real se incluyen los conceptos para:

- * El manejo de las interrupciones.
- * La comunicación y la sincronización entre tareas.
- * El proceso concurrente.
- * La respuesta oportuna a los eventos externos.
- * Los requerimientos y las restricciones de los sistemas **hardware**.
- * Las interacciones entre el sistema/entorno.
- * Los datos continuos y los discretos.

El diseño de un sistema de tiempo real debe poder representar procesos que puedan ser interrumpidos por eventos externos y que pueden procesarse concurrentemente en distintos ordenadores [7].

Para representar los requerimientos de los sistemas de tiempo real se emplean tipos especiales de diagramas estructurados, entre los que se encuentran los de flujo de control, de transmisión de estado, gráfica de contexto, matrices estado/suceso, y las tablas de decisión. Por ejemplo, el diagrama de control de flujo se emplea para mostrar los procesos del sistema,

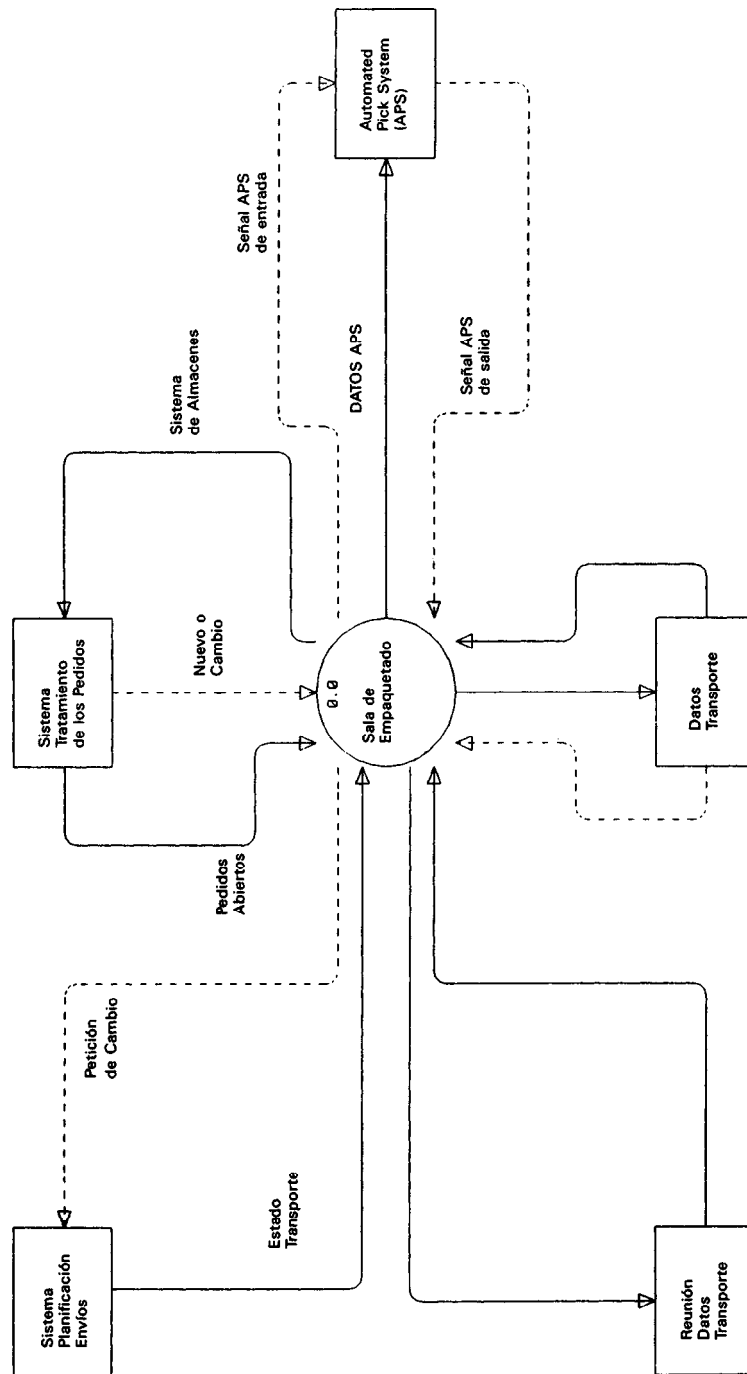


Figura 3.11. El diagrama de contexto es un tipo de diagrama de flujo de control que describe los límites del sistema. Se dibujó con el EXCELERATOR/RTS de Index Technology Corporation

Cuadro 3.7. Las metodologías más utilizadas

Las metodologías de desarrollo más utilizadas en los Estados Unidos	
Diseño estructurado de Yourdon	29.1 %
Análisis estructurado de Gane-Searson	13.8 %
Análisis estructurado de DeMarco	12.3 %
Diseño estructurado de Orr	7.2 %
Diseño estructurado de Jackson	2.4 %
Elección de normalización de la metodología de desarrollo	
Yourdon	24.9 %
Gane-Searson	11.2 %
Ingeniería de la información	11.2 %
DeMarco	9.5 %
Orr	7.1 %
Jackson	3.6 %

El resto del capítulo se dedica a una breve exposición de los pasos del proceso en cada una de estas metodologías. En el cuadro 3.8 se clasifican estas metodologías estructuradas.

EL ANALISIS ESTRUCTURADO

El análisis estructurado propone un enfoque sistemático por pasos para la realización del análisis del sistema y la producción de las especificaciones del sistema [6, págs. 407-416]. Hay dos versiones similares de análisis estructurado:

- Gane-Searson.
- DeMarco.

Ambas representan una disciplina estructurada basada en los siguientes conceptos estructurados:

Cuadro 3.6. Características de los sistemas de tiempo real y de los de in-
formación

Sistemas de información	Sistemas de tiempo real
<ul style="list-style-type: none">* Dirigido por los datos.* Estructuras complejas de los datos.* Grandes volúmenes de datos de entrada.* Operaciones de E/S in- tensivas.* Independiente de la máquina.	<ul style="list-style-type: none">* Dirigido por los eventos.* Estructuras simples de los datos.* Pequeña cantidad de datos de entrada.* Operaciones de calculo in- tensivas.* Dependiente de la máquina.

Por ejemplo, la metodología de Hatley es una extensión de la metodo-
logía de diseño de Yourdon - Constantine. La metodología de diseño de
sistemas de Jakson puede utilizarse tanto para diseñar sistemas de tiempo
real como de información.

**LAS METODOLOGIAS ESTRUCTURADAS DE UTILIZACION MAS
EXTENDIDA**

En el cuadro 3.7 se listan las metodologías estructuradas de desarro-
llo más utilizadas de acuerdo con una investigación realizada entre más de
mil empresas de EE.UU. [8]. En el mismo cuadro también se listan las prin-
cipales metodologías que las organizaciones planean adoptar de acuerdo
con la misma investigación [8]. Obsérvese que la ingeniería de la informa-
ción, aunque no está entre las más utilizadas es una de las que las organi-
zaciones planean adoptar como estándar. Esto probablemente se deba a
que la ingeniería de la información es una metodología más reciente en re-
lación con las otras en el cuadro 3.7. Está ganando aceptación debido a
la incorporación en su metodología del modelado de datos y de la planifi-
cación estratégica.

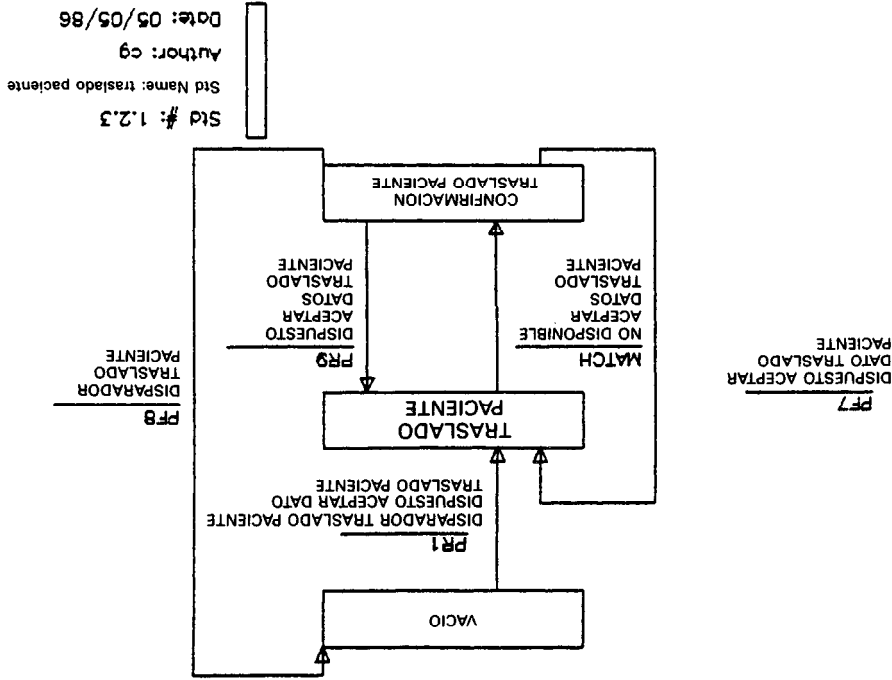


Figura 3.10. Este diagrama de transición de estado, que se utiliza para mostrar las condiciones y acciones que originan el cambio de un estado a otro del sistema, se dibujó con el ANALYST/DESIGNER TOOLKIT de Yourdon Software Engineering Company

el paso de las señales de datos de entrada/salida entre los procesos y las relaciones oportunas entre los procesos. El diagrama de transición de estado mostrado en la figura 3.10 se utiliza para mostrar las condiciones y acciones que ocurren en el sistema al pasar de un estado a otro. La matriz proporciona una información más detallada al describir la lógica condicional empleada en un estado del sistema. El diagrama de contexto, como el mostrado en la figura 3.11, describe los límites del sistema. El diagrama de bloques ilustra el flujo entre los sistemas de software y de hardware. La mayoría de estos diagramas son variaciones de los diagramas estructurados empleados para describir los sistemas de información. De modo similar, las metodologías de análisis y de diseño utilizadas en sistemas de tiempo real son variaciones de las utilizadas en sistemas de información como los de DeMarco, Yourdon y Jackson.

- Organización jerarquizada descendente.
- Herramientas gráficas de comunicación/documentación.

El análisis estructurado utiliza un método descendente de descomposición funcional para definir los requerimientos del sistema. La especificación del sistema producido por el análisis estructurado es un modelo descendente particionado del sistema a construir. El suministro de una descripción de los requerimientos del sistema a construir es la conexión entre el análisis y el diseño.

La unión entre el análisis y el diseño se proporciona con la descripción de los requerimientos del sistema a construir.

El objetivo primario del análisis es producir una especificación del sistema que defina la estructura del problema a resolver según la visión del usuario.

Cuadro 3.8. Clasificación de las metodologías

METODOLOGIA	CLASIFICACION
DeMarco Análisis estructurado	<ul style="list-style-type: none"> • Ingeniería de software. • Orientados al procedimiento. • Sistemas de información. • Informal (extensiones tiempo real).
Gane - Searson Análisis estructurado	<ul style="list-style-type: none"> • Ingeniería de software. • Orientado al procedimiento. • Sistemas de información. • Informal (extensiones tiempo real).
Yourdon Diseño estructurado	<ul style="list-style-type: none"> • Ingeniería de software. • Orientado al procedimiento.

Cuadro 3.8. Clasificación de las metodologías (continuación)

Yourdon Diseño estructurado	<ul style="list-style-type: none"> • Sistemas de información. • Informal (extensiones tiempo real).
Jackson Diseño estructurado	<ul style="list-style-type: none"> • Ingeniería de software. • Orientado a los datos. • Sistemas de información. • Informal (extensiones tiempo real).
Orr Desarrollo de sistemas	<ul style="list-style-type: none"> • Ingeniería de software. • Orientado a los datos. • Orientado a las salidas. • Sistemas de información. • Informal.
Martin Ingeniería de la información	<ul style="list-style-type: none"> • Ingeniería de la información. • Orientado a la información. • Sistemas de información. • Informal.

El propósito principal del análisis es producir una especificación del sistema que defina la estructura del problema a resolver según la visión del usuario. El propósito del diseño es definir la estructura de la solución de forma que sea compatible con la estructura del problema y con los requerimientos del usuario. Los defensores del análisis estructurado sugieren que al utilizar el mismo método de construcción (descomposición funcional descendente, en la especificación y en el diseño, los dos pueden unirse para representar un sistema que satisfaga las necesidades y expectativas del usuario.

La especificación estructurada

La especificación del sistema producida por el análisis estructurado es un modelo lógico, gráfico, particionado, descendente y jerárquico de los procesos del sistema y de los datos utilizados por estos procesos. Se compone de diagramas de flujo de datos, un diccionario de datos y las especificaciones de los procesos.

El diagrama de flujo de datos

Un **diagrama de flujo de datos** es un diagrama estructurado que representa los procesos (funciones o procedimientos) de un sistema y los datos que les conectan. Es la herramienta central del modelado del análisis estructurado y se emplea para fraccionar jerárquicamente el sistema en procesos. En los niveles inferiores, el proceso de nivel más alto se expande para ver si hay otros procesos y los datos más detallados involucrados. La figura 3.12 es un ejemplo de diagrama de flujo de datos.

El diccionario de datos

El **diccionario de datos** es un conjunto de definiciones de todos los datos que aparecen en un diagrama de flujo de datos, en datos almacenados o como flujo de datos. La definición de cada dato consta de los componentes del dato que constituyen el dato y las relaciones entre ellos.

FICHERO DE CLIENTES	=	[registro de clientes]
REGISTRO DE CLIENTES	=	nombre-cliente + dirección-cliente + información-del-pago + órdenes-de-pago + tipo-de-cliente

Las especificaciones de proceso

Una **especificación de proceso** (o miniespecificación) describe lo que ocurre dentro de un cuadro de proceso en un diagrama de flujo de datos. Una miniespecificación tiene generalmente una página de extensión y ex-

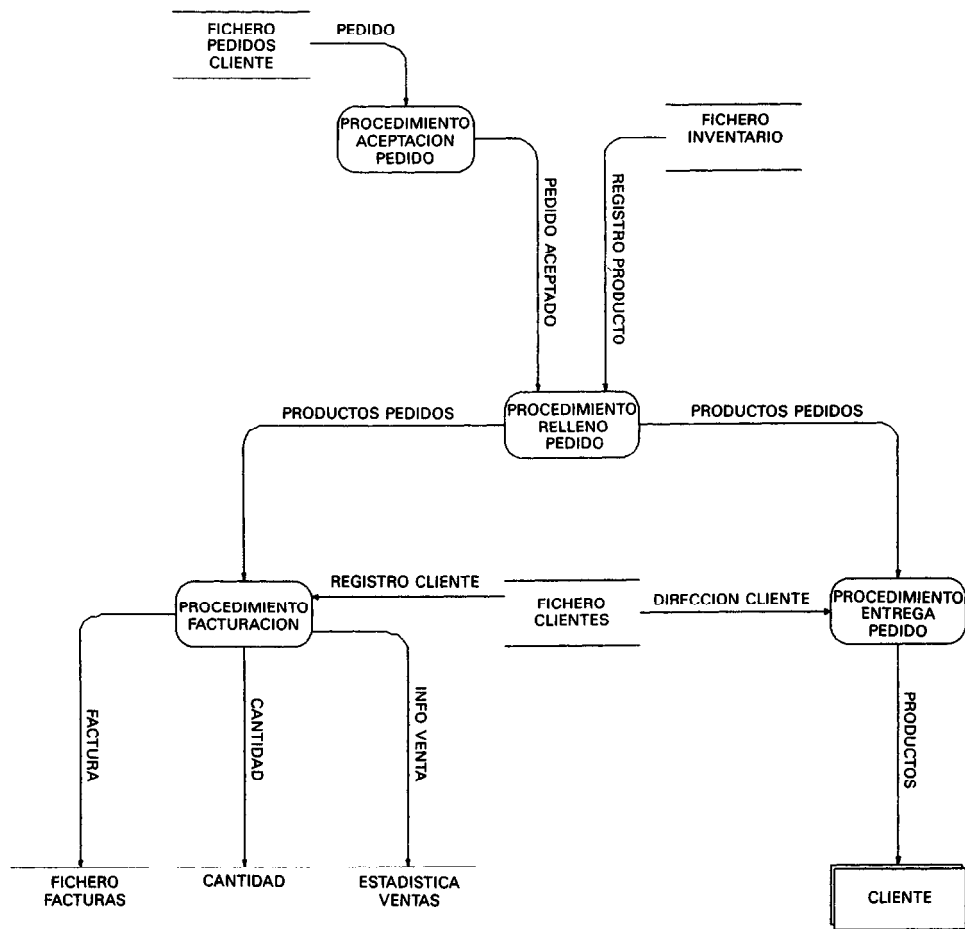


Figura 3.12. Diagrama de flujo de datos mostrando los procesos del sistema de distribución de ventas, dibujado con el INFORMATION ENGINEERING WORKBENCH de KnowledgeWare

plica cómo se transforman los datos de entrada en datos de salida. La miniespecificación se escribe en inglés estructurado (seudocódigo) como una tabla de decisión o como un diagrama de acción. La figura 3.13 es un ejemplo de miniespecificación. Se crea una miniespecificación por cada proceso definido en el diagrama de flujo de bajo nivel en la especificación del sistema.

Los pasos del análisis estructurado

Como se muestra en el cuadro 3.9, DeMarco define el análisis estructurado en siete pasos:

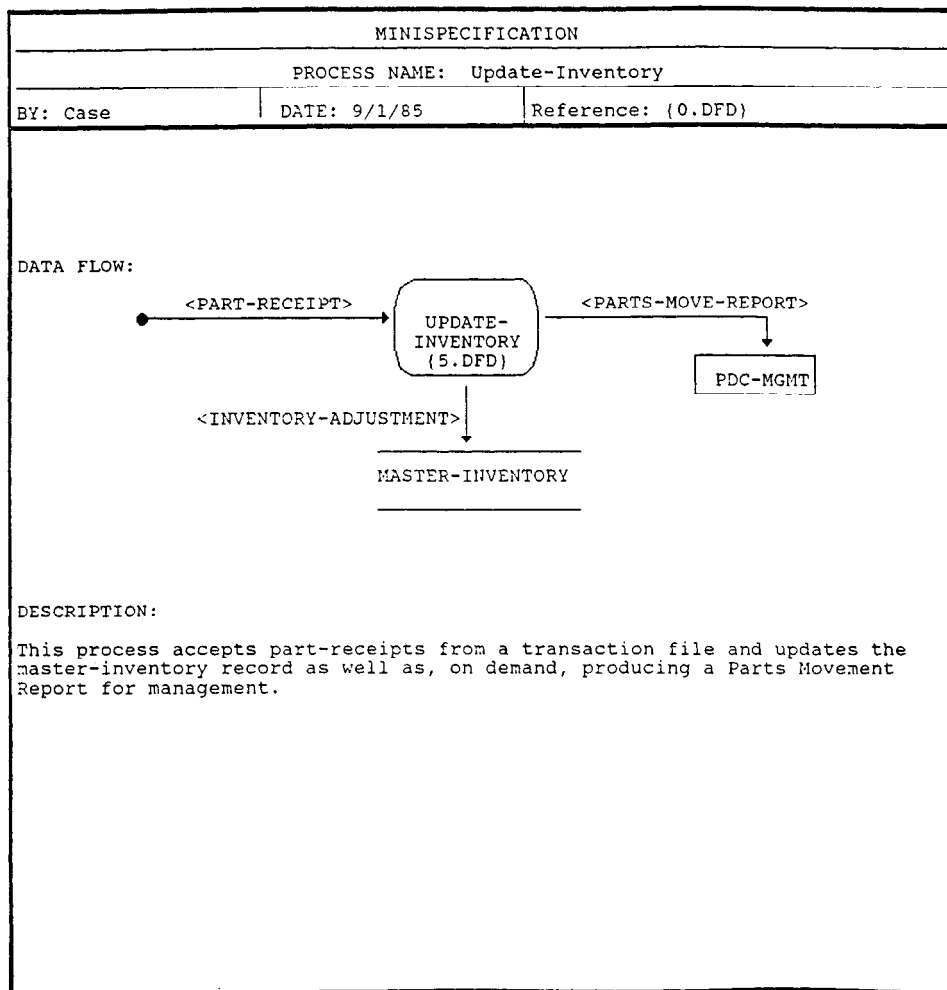


Figura 3.13. Ejemplo de una miniespecificación obtenida con DESIGNAID de Nastec Corporation

Cuadro 3.9. Los pasos en los procesos de análisis estructurado

Los siete pasos del análisis estructurado de DeMarco	Los cinco pasos del análisis estructurado de Gane y Searson
Paso 1: Construir el modelo físico	
Paso 2: Construir el modelo lógico a partir del modelo físico	Paso 1: Construir un modelo lógico en curso.
Paso 3: Construir un modelo lógico del nuevo sistema <ul style="list-style-type: none"> • Construir una especificación estructurada que incluye los diagramas de flujo de datos, un diccionario de datos y las especificaciones de los procesos. 	Paso 2: Construir un modelo lógico del nuevo sistema. <ul style="list-style-type: none"> • Construir una especificación estructurada conteniendo los diagramas de flujo de datos, un diccionario de datos y las especificaciones de proceso. • Construir un modelo lógico de datos que exprese en tercera forma normal el contenido de los datos almacenados.
Paso 4: Crear una familia con los nuevos modelos físicos.	Paso 3: Diseñar la base de datos física.
Paso 5: Estimar los costos y los tiempos para cada modelo.	Paso 4: Crear un nuevo modelo físico del sistema.
Paso 6: Seleccionar un modelo.	Paso 5: Empaquetar la especificación en subsistemas.
Paso 7: Empaquetar la especificación en subsistemas.	

Paso 1

Documenta la forma en que las cosas en curso se realizan en el entorno actual del usuario. La salida de este paso es el diagrama de flujo físico de los datos actuales. Muestra la situación, el personal, los nombres y los

procedimientos manuales y automatizados, y utiliza los términos de los usuarios.

Paso 2

Utiliza el diagrama físico de flujo de datos del paso 1 para crear el modelo lógico del sistema actual. El modelo lógico, igual que el físico se representa como un conjunto de niveles jerarquizados de los diagramas de flujo de datos. La diferencia entre los dos modelos está en que el modelo físico muestra los detalles físicos de cómo un usuario particular hace las cosas, mientras que el modelo lógico muestra lo que se hace en un nivel abstracto, distinguiendo cualquier estado particular de los procedimientos fundamentales del negocio.

Paso 3

Desarrolla el nuevo modelo lógico del sistema a construir, siendo el punto de partida el modelo lógico del paso 2. Se modifica para incorporar los cambios para el nuevo sistema. Indica los procesos manuales y los automatizables y se representa como un conjunto por niveles de diagramas de flujo de datos que describen el sistema en variados niveles de detalle. El paso 3 conlleva la mayor parte del esfuerzo de trabajo del proceso de análisis estructurado.

Paso 4

Crea un nuevo modelo físico del sistema a construir identificando el interfaz hombre/máquina. Puede crearse más de un modelo posible para representar los distintos grados de automatización.

Paso 5

Produce una estimación de los costes y una planificación por cada uno de los modelos físicos alternativos desarrollados en el paso 4.

Paso 6

Basándose en la estimación obtenida en el paso 5, se selecciona un modelo físico para utilizar en la especificación de las funciones y requerimientos del sistema a construir.

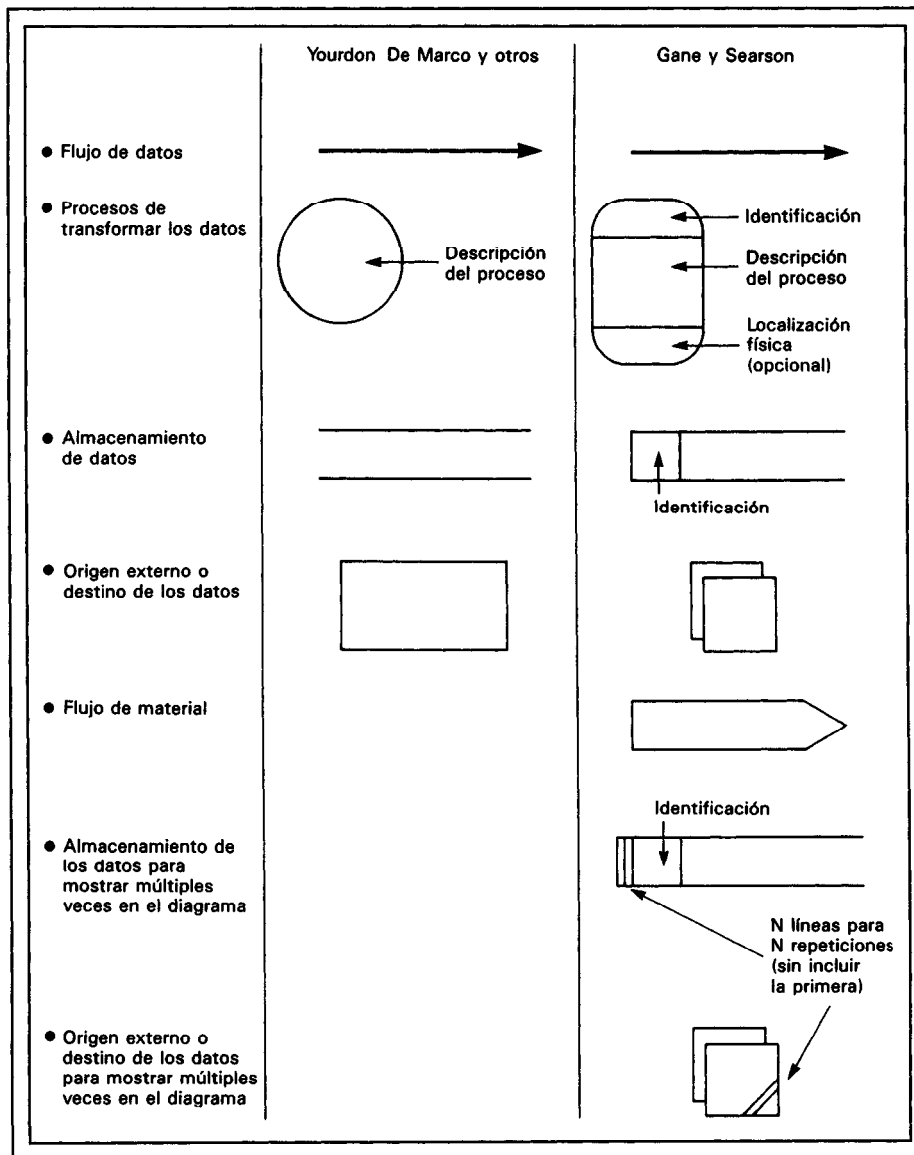
Paso 7

Se empaqueta en su forma final la especificación estructurada, que incluye un conjunto por niveles de diagramas de flujo de datos, el diccionario de datos y las miniespecificaciones.

Gane y Searson definen un proceso similar para el análisis estructurado (ver cuadro 3.9). La principal diferencia es que Gane y Searson incluyen un paso del modelado de datos almacenados mostrados en los diagramas de flujo de datos, posiblemente en tercera forma normal. Gane y Searson incluyen el modelado de datos en el paso 2 de su versión de análisis estructurado. Después, en el paso 3, el modelo lógico de datos se emplea como entrada para el diseño físico de la base de datos. También, como muestra el cuadro 3.10, Gane y Searson utilizan en su versión de diagrama de flujo, un conjunto de símbolos gráficos ligeramente diferentes.

EL DISEÑO ESTRUCTURADO DE YOURDON

La metodología de diseño estructurado de Yourdon proporciona un proceso para el diseño paso a paso de sistemas y programas detallados [6, págs. 425-453]. Unos pasos implican el desarrollo del diseño, otros, la medición y la mejora de la calidad del diseño. Cada paso es soportado por un conjunto de estrategias de diseño, guías y técnicas de documentación. Por ejemplo, las estrategias de diseño, las de análisis de transformación y las de transacción guían al diseñador en las decisiones de diseño. Las medidas de diseño, como las de acoplamiento y cohesión proporcionan al diseñador técnicas para medir la calidad del diseño. El principal producto obtenido por los procesos de diseño estructurado es un diagrama de estructura mostrando los componentes de procedimientos del programa, su ordenación jerárquica y los datos conectados a ellos.

Cuadro 3.10. Los símbolos utilizados en los diagramas de flujo de datos

El diagrama de estructura (structure chart)

El **diagrama de estructura** es un diagrama jerárquico o de árbol que define la arquitectura global de un programa mostrando sus componentes de procedimiento y sus interrelaciones. Los bloques básicos para construir el diagrama de estructura son cajas que representan los componentes de procedimiento y las flechas que los conectan. Los datos que pasan entre los componentes de procedimientos se llaman pares y se escriben a lo largo de las flechas de conexión. La figura 3.19 es un ejemplo de diagrama de estructura.

Los pasos de diseño estructurado de Yourdon

La figura 3.14 muestra los cuatro pasos básicos del proceso de diseño estructurado de Yourdon.

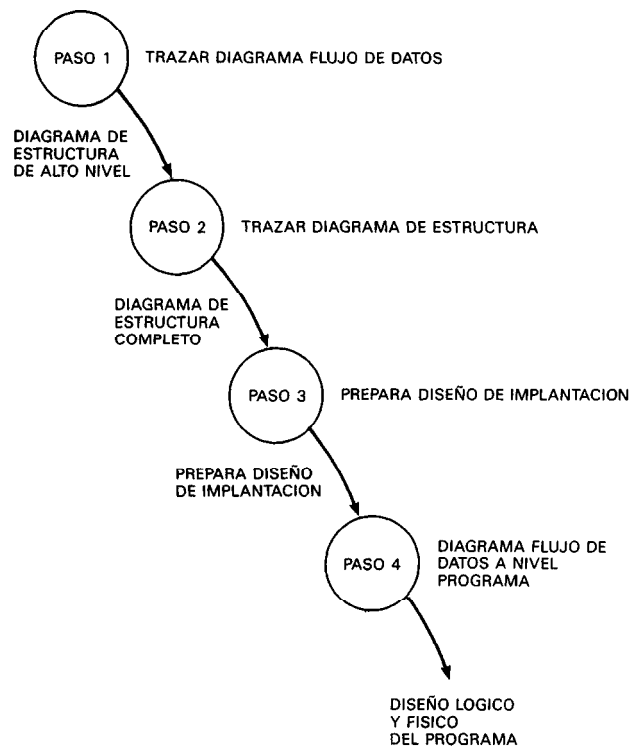


Figura 3.14. El proceso de diseño estructurado consta de cuatro pasos secuenciales en la metodología de diseño estructurado de Yourdon

Paso 1: Trazar el diagrama de flujo de datos

El primer paso en el proceso de diseño estructurado es representar el problema de diseño como el flujo de datos a través de un sistema. El sistema se compone de procesos que operan (transforman) con los datos. Estos procesos y los datos que los enlazan forman las bases para definir los componentes del programa. Se utiliza un conjunto por niveles de diagramas de flujo de datos que los enlazan para representar la primera visión del diseño del programa.

Paso 2: Trazar el diagrama de estructura

El segundo paso es representar el diseño del programa como una jerarquía de componentes de procedimiento. Se emplea un diagrama de estructura para mostrar esta visión del diseño. El diagrama de estructura se deriva del diagrama de flujo de datos obtenido en el paso 1. El diseño estructurado proporciona dos estrategias de diseño para guiar la transformación de uno o varios diagramas de flujo de datos a un diagrama de estructura: los análisis de transformación y de transacción. Estas dos estrategias proporcionan dos modelos de estructuras que pueden utilizarse individual o conjuntamente para dirigir el diseño jerárquico, así como un proceso paso a paso de transformación por cada estrategia.

El **análisis de transformación** es un modelo de flujo de información que divide el diagrama de flujo de datos en tres partes: la entrada que recibe el nombre de rama eferente, el proceso lógico llamado transformación central y la salida denominada rama eferente. Como se muestra en las figuras 3.14, 3.15 y 3.16 el diagrama de estructura se deriva de la identificación de estas tres partes básicas en el diagrama de flujo de datos.

El **análisis de transacción** se emplea cuando se diseñan programas con proceso de transacciones. El diagrama de estructura general para un programa con procesos de transacciones se muestra en la figura 3.17. En la parte superior de los diagramas de estructura está el módulo de la transacción central y debajo hay varios módulos de transacciones. Como se muestra en las figuras 3.18 y 3.19, el **diagrama de estructura** se obtiene definiendo la transformación central y las transacciones.

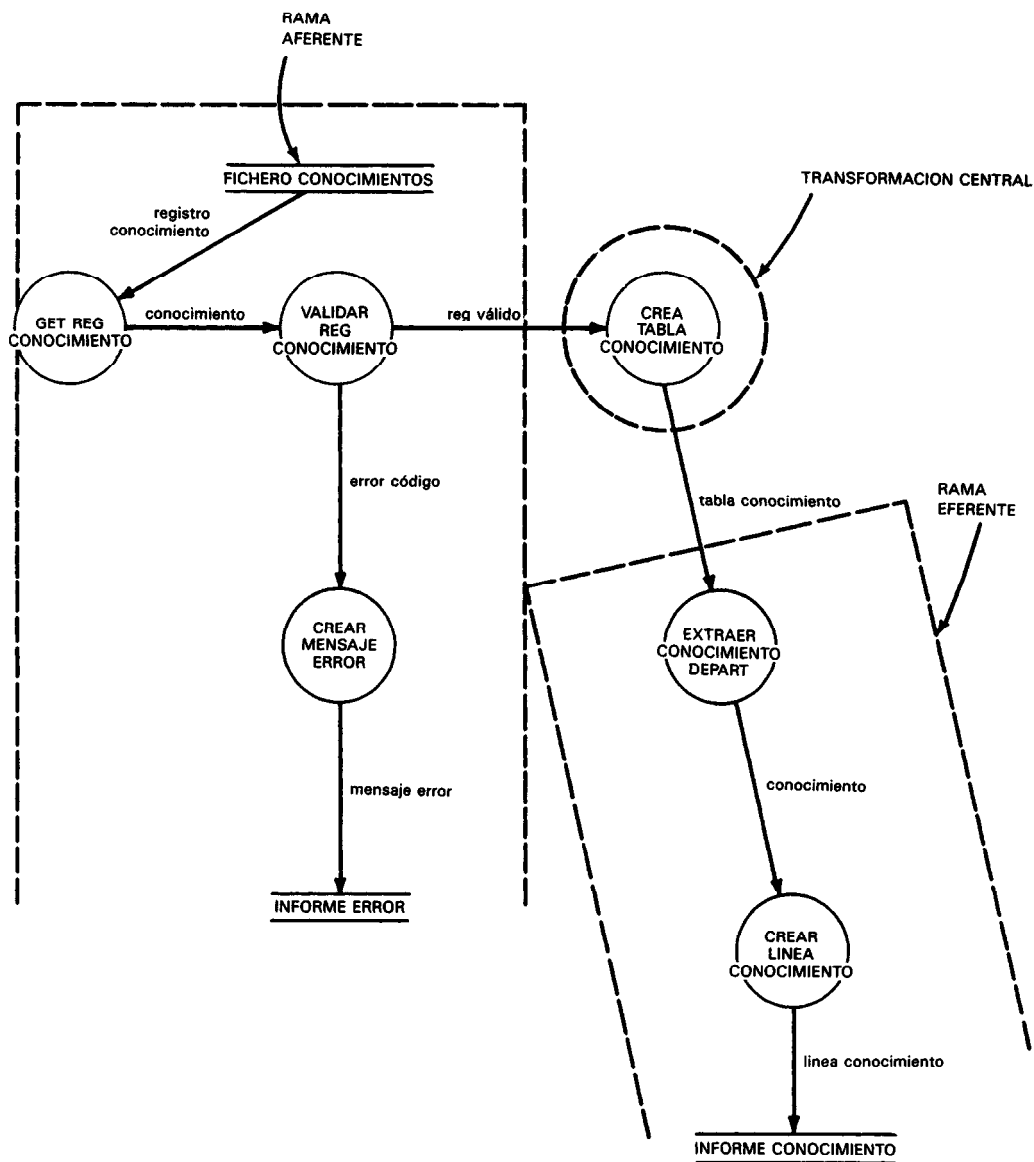


Figura 3.15. Diagrama de flujo de datos del sistema de conocimientos de empleados, indicando las ramas aferentes (entrada), las ramas eferentes (salida) y la transformación central (proceso lógico)

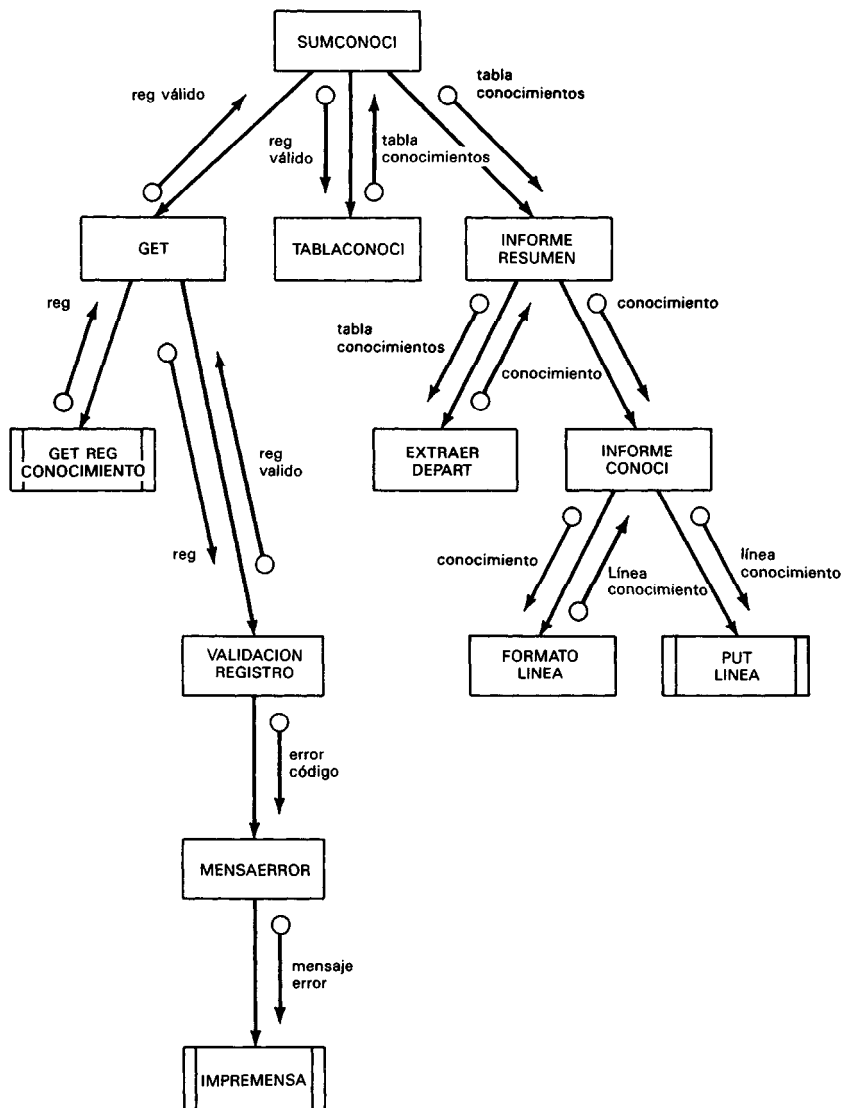


Figura 3.16. Diagrama de estructura del sistema de conocimientos de empleados derivado del diagrama de flujo de datos mostrado en la figura 3.15

Paso 3: Evaluación del diseño

El tercer paso en el diseño estructurado es la medición de la calidad del diseño. El acoplamiento y la cohesión son las dos técnicas de medición del diseño proporcionadas por el diseño estructurado.

El **acoplamiento** mide el grado de independencia entre los componentes del procedimiento (módulos) en el diagrama de estructura. Cuando existe poca interacción entre dos módulos, éstos se describen como **ligeramente acoplados**. Cuando la interacción entre dos módulos es grande, los módulos se describen como **estrechamente acoplados**. Un diseño de alta calidad significa que los módulos están lo más ligeramente acoplados posible. Hay cinco niveles de acoplamiento: **datos, por estampado, de control, común y por contenido**. El acoplamiento por datos es el más ligero y por tanto el mejor tipo. Mientras que el acoplamiento por contenido es el más estrecho y por tanto, el peor tipo.

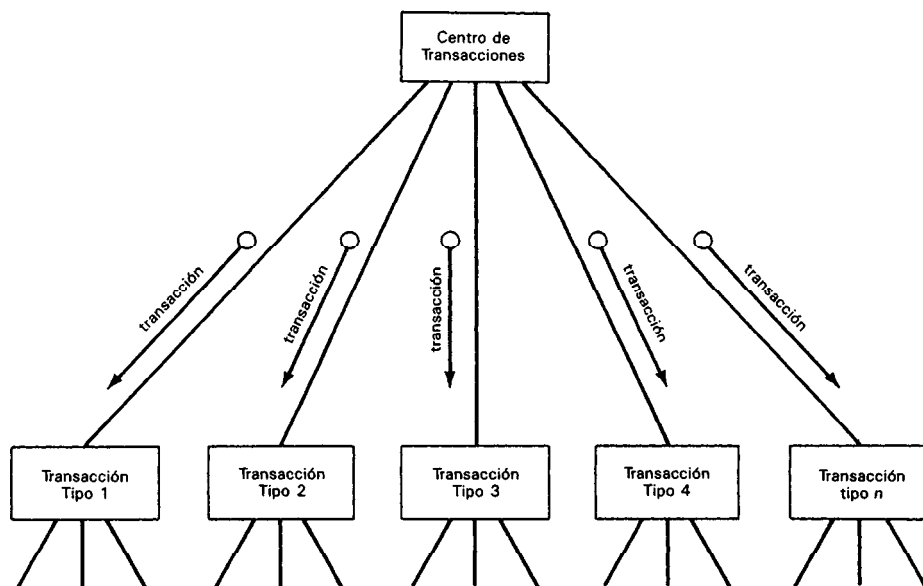


Figura 3.17. El modelo general de diagrama de estructura para un programa con proceso de transacciones, tiene un módulo "centro de transacciones" en lo más alto y por debajo un módulo de transacción para cada tipo distinto de transacción

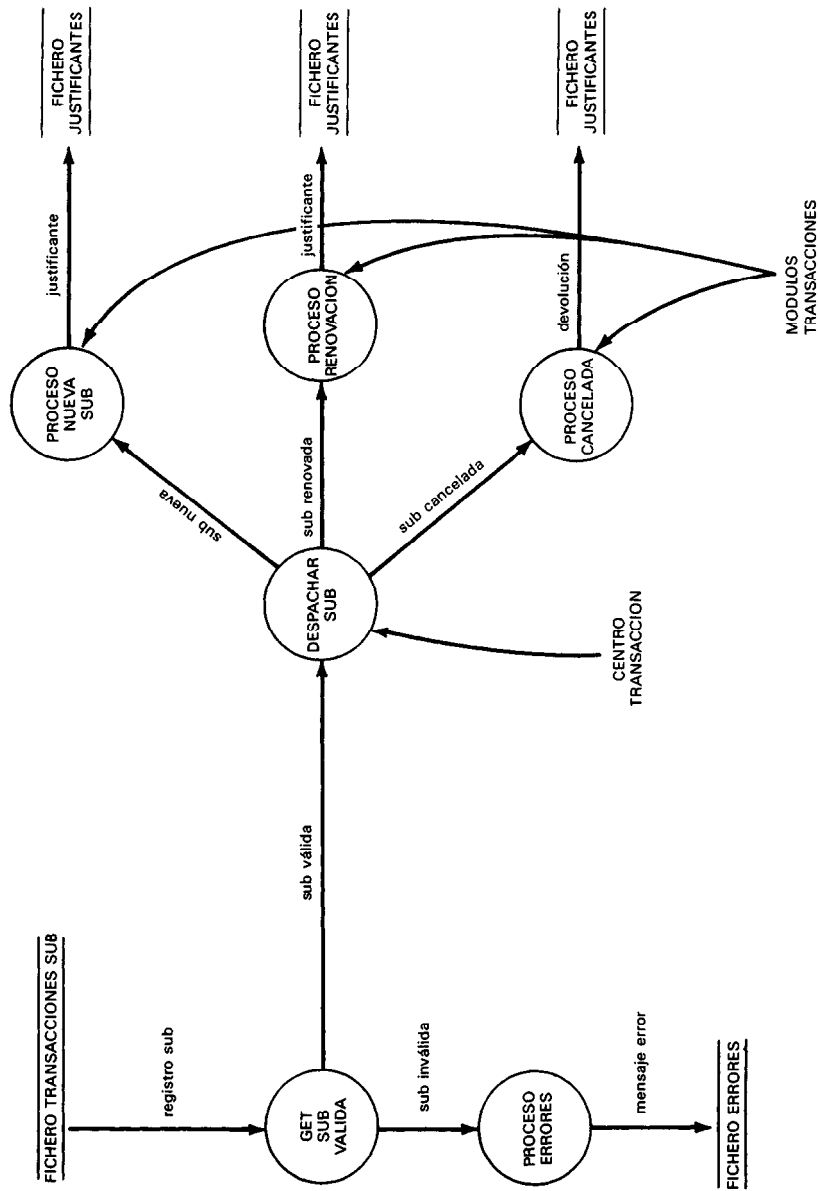


Figura 3.18. Diagrama de flujo de datos del sistema de suscripciones, mostrando el centro de transacción y los módulos de transacciones

La **cohesión** mide la fuerza de las relaciones entre los elementos dentro de un módulo cuanto más fuerte, mejor. Hay siete niveles de cohesión: funcional, secuencial, de comunicación, de procedimiento, temporal, lógica y de coincidencia. La cohesión funcional es la mejor, la de coincidencia, la peor.

Paso 4: La preparación del diseño para la implantación

El último paso de un diseño estructurado es la preparación del diseño para la implantación. Esto se llama **empaquetar** el diseño. Empaquetar es el proceso de dividir el diseño de programa lógico en unidades físicas de implantación llamadas unidades de carga. Es un diseño físico del programa.

Es una descripción de la tradicional metodología de diseño estructurado de Yourdon. Hay variaciones y actualizaciones, algunas de las cuales han extendido las metodologías de diseño de Yourdon (que fue ideada originalmente para el diseño de sistemas de información) para soportar el diseño de sistemas de tiempo real.

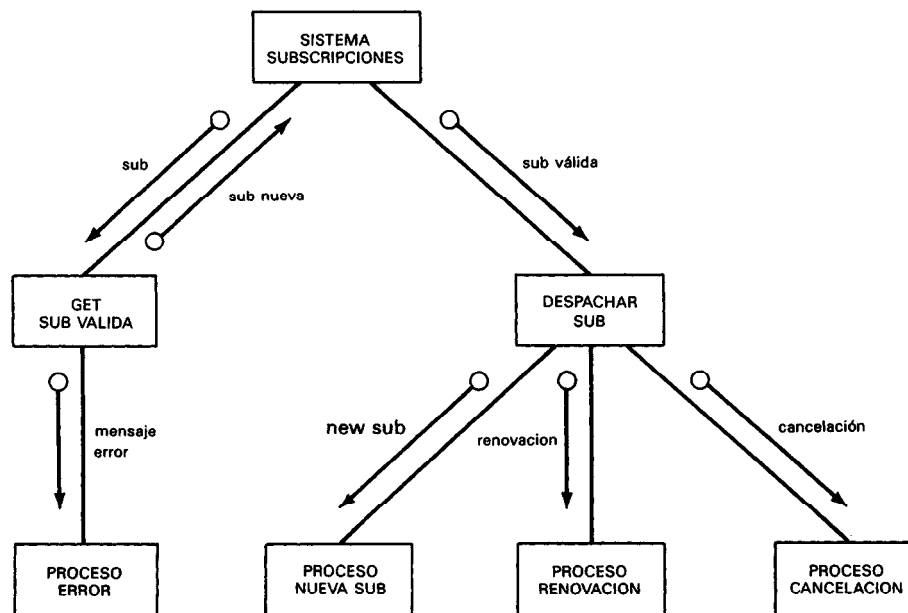


Figura 3.19. Diagrama de estructura a alto nivel derivado del diagrama de flujo de datos utilizando la estrategia de diseño de transacción

LA METODOLOGIA DE DISEÑO DE JACKSON

Como la metodología de diseño de Yourdon, la de Jackson proporciona un proceso descendente por pasos de diseño (6, págs. 455-487). La principal diferencia entre las dos, es que la de Jackson está basada en el análisis de la estructura de los datos, mientras que la de Yourdon se basa en el análisis del flujo de datos. La de Jackson está orientada a los datos y la de Yourdon, al procedimiento. La formulación de Jackson aboga por una visión estática de las estructuras, mientras que la de Yourdon utiliza una visión dinámica del flujo de datos.

La metodología de diseño de programas de Jackson es una técnica dirigida por los datos. Su objetivo es obtener la estructura del programa derivandola de la(s) estructura(s) de los datos, asumiendo que se ha especificado totalmente el problema.

Jackson considera el programa como un proceso secuencial. Tiene entradas y las salidas que le consideran como corrientes secuenciales de registros. El proceso de diseño consiste en definir primero la estructura de la corriente de datos y después ordenar el proceso lógico (operaciones) para ajustar las estructuras de datos. Como se muestra en el cuadro 3.11, el proceso de diseño de Jackson consta de cuatro pasos.

Cuadro 3.11. Pasos básicos en el proceso de diseño de Jackson

- **Paso de datos:** Dibuja un diagrama estructurado por cada entrada y salida de datos en el programa.
- **Paso de programa:** Toma todas las estructuras de datos definidas en el paso de datos y forma con ellas una única estructura de programa.
- **Paso de operación:** Hace una lista con las operaciones necesarias para producir las salidas a partir de las entradas. Después, asigna cada operación de la lista a un componente de la estructura del programa (progresivamente desde la salida hasta la entrada).
- **Paso de texto:** Transcribe la estructura del programa a texto, añadiendo la lógica condicional que rige la ejecución de los bucles y las estructuras de selección.

Diagramas estructurados y pasos de Jackson

La metodología de diseño de Jackson utiliza las técnicas de diagramación estructurada en árbol para describir un diseño de programa:

- **Diagrama de red:** El flujo de corrientes de datos entre programas (ver figura 3.20).
- **Diagrama estructurado en árbol:** Representación jerárquica de estructuras de datos y programas (ver figuras 3.21 y 3.22).
- **Texto:** En forma de pseudocódigo (ver figura 3.25).

Paso 1: Paso de datos

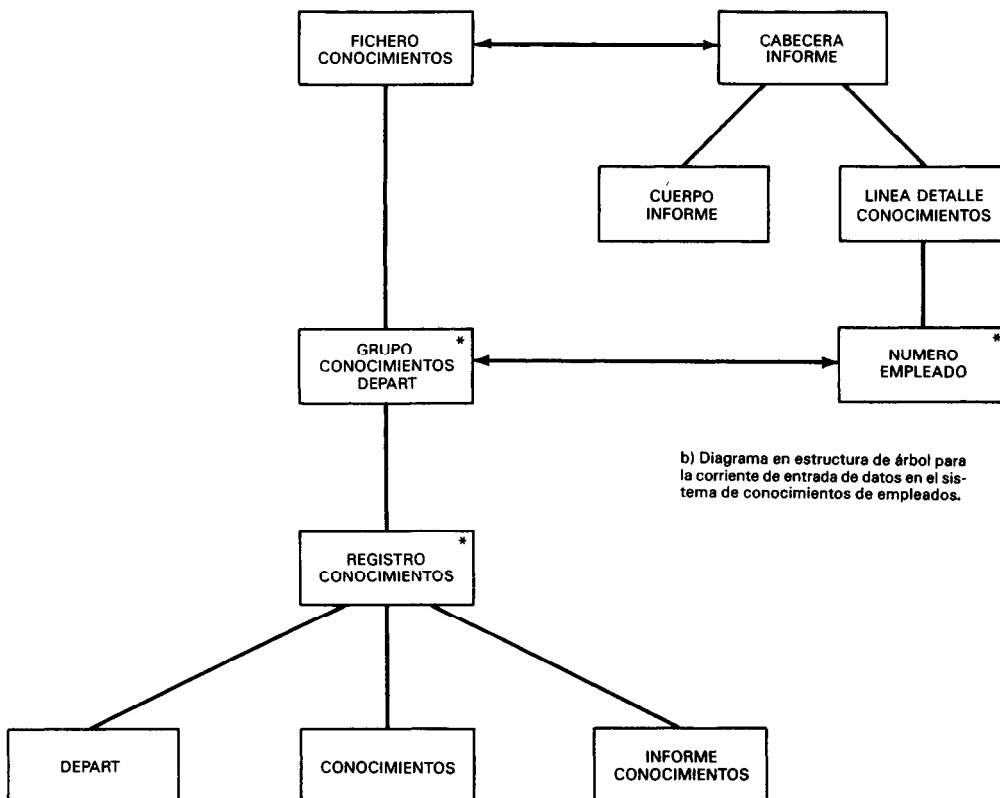
Primero se dibuja un diagrama de red de sistema mostrando todas las corrientes de datos de entrada y de salida (ver figura 3.20). Después, cada corriente de datos se representa como una estructura jerárquica.



Figura 3.20. El diagrama de la red del sistema de conocimientos de los empleados muestra que hay una corriente de entrada, FICHERO CONOCIMIENTOS, una corriente de salida, INFORME CONOCIMIENTOS, y un programa, PROGRAMA CONOCIMIENTOS DE EMPLEADOS. En la terminología de Jackson se diría: El FICHERO CONOCIMIENTOS se consume para producir el INFORME CONOCIMIENTOS

Paso 2: Paso de programa

El segundo paso es combinar las estructuras de datos en una sola estructura de programa. Hay dos partes en el paso de programación. Primera, todas las correspondencias entre los componentes de la estructura de datos se identifican encontrando la relación consumo/producción entre



b) Diagrama en estructura de árbol para la corriente de entrada de datos en el sistema de conocimientos de empleados.

a) Diagrama en estructura de árbol para la corriente de salida de datos en el sistema de conocimientos de empleados

Figura 3.21. El programa de conocimientos de empleados tiene una corriente de entrada y una de salida. Se utiliza el diagrama estructurado en árbol para representar la estructura de los datos. Las flechas sombreadas muestran la relación consumo-producción entre los componentes en la estructura de los datos

los componentes (ver figura 3.21). Se construye la estructura del programa para que se corresponda con esas relaciones (ver figura 3.22).

La segunda parte consiste en verificar la corrección de la estructura de programa. Esto se consigue reduciendo la estructura del programa a sus

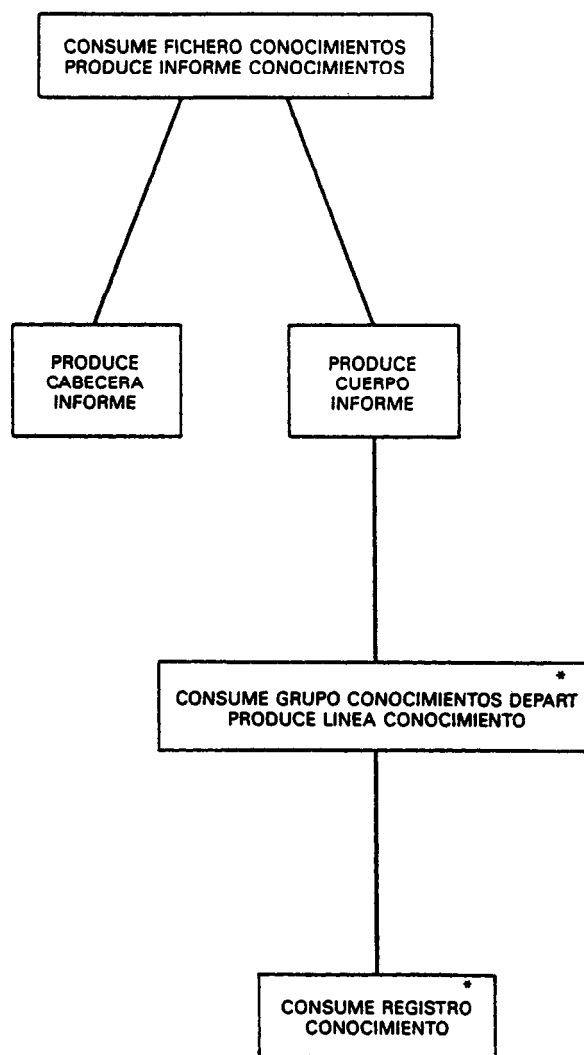


Figura 3.22. La estructura del programa del sistema conocimientos de empleados está compuesto por las estructuras de los datos de entrada y de salida mostrados en la figura 3.21

estructuras de datos individuales. El proceso de verificación se aplica a cada corriente de datos para demostrar que el programa se puede reducir a cada una de sus estructuras de datos (ver figura 3.23). Si esto puede hacerse, se asume que el diseño es correcto.

Paso 3: Paso de operaciones

El paso de operaciones se compone de tres partes:

- 1.— Se listan las operaciones ejecutables requeridas por el programa para convertir su entrada en salida; esto se hace empezando en la salida y yendo hacia atrás hasta la entrada. Las operaciones se determinan por referencia a las estructuras de datos, las cuales, a su vez, se reflejan en la estructura del programa.

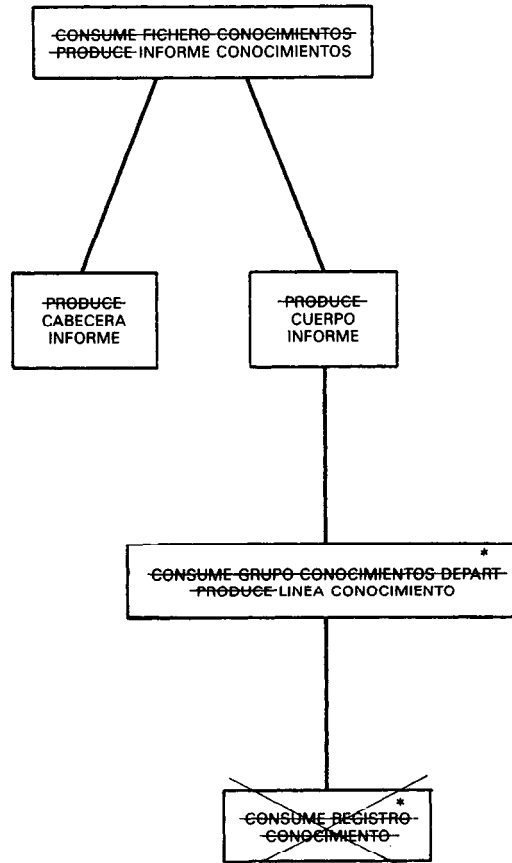
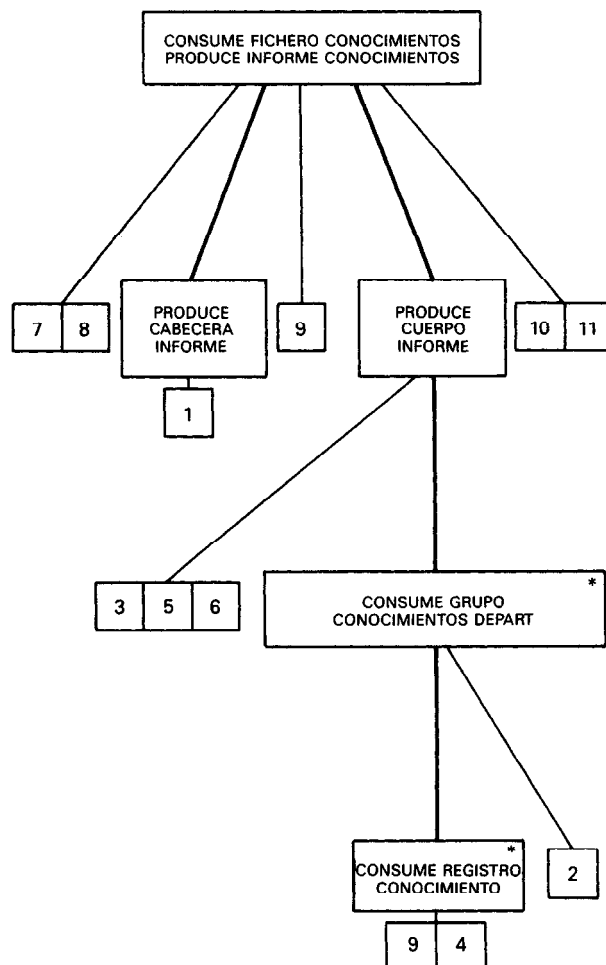


Figura 3.23. La exactitud de la estructura del programa se demuestra reduciéndolo a sus estructuras de datos. En el ejemplo la estructura del programa se reduce a la corriente de datos de salida, INFORME CONOCIMIENTOS. La estructura de los datos del INFORME CONOCIMIENTOS se muestra en la figura 3.21

ESTRUCTURA DEL PROGRAMA



OPERACIONES

1. imprime cabecera-infor
2. imprime línea conocimiento
3. num-de-empleado = 0
4. num-de-empleado = num-de-empleado + 1
5. depart-en-curso = depart
6. conoci-en-curso = conoci
7. abrir fichero informe
8. abrir fichero conocimientos
9. leer registro conocimiento
10. cerrar fichero informe
11. cerrar fichero conocimientos

Figura 3.24. En el paso de operaciones se listan las operaciones ejecutables requeridas por el programa de conocimientos de los empleados

- 2.— Cada operación se asigna al lugar apropiado de la estructura del programa.
- 3.— Se verifica la corrección comprobando si se producen todas las salidas y se consumen todas las entradas. La figura 3.24 muestra la estructura del programa incrementada con sus pasos de operación.

Paso 4: Paso del texto

El último paso en el diseño de Jackson es el de texto. En este paso, el diagrama de estructura del programa, junto con sus correspondientes operaciones, se traduce a un texto de estructura (ver figura 3.25).

El texto de estructura es un tipo formal de pseudocódigo. Tiene reglas estrictas para escribir el control de la secuencia, la selección y la iteración, pero las instrucciones a nivel de programación elemental se escriben informalmente.

```
INFORME-CONOCIMIENTOS seq
  open fichero-conocimientos;
  open fichero-informe;
  read registro-conocimiento;
  CUERPO-INFORME iter while (not end-of-file)
    GRUPO-CONOCIMIENTOS-DEPART seq
      num-de-empleado : = 0;
      depart-en-curso  : = depart;
      conoci-en-curso  : = conoci;
      REGISTRO-CONOCIMIENTO iter while (depart = depart-en-
        curso and conoci = conoci-en-curso)
        num-de-empleado = num-de-empleado + 1;
        read registro-conocimiento
      REGISTRO-CONOCIMIENTO end
      move num-de-empleado to linea-conocimiento;
      write linea-conocimiento;
    GRUPO-CONOCIMIENTOS-DEPART end
  CUERPO-INFORME end;
  close fichero-conocimientos;
  close fichero-informe;
INFORME-CONOCIMIENTOS end
```

LOGICA CONDICIONAL

Figura 3.25. En el proceso de diseño Jackson de un programa el último paso es la transformación de la estructura del programa en una estructura de texto, una versión formal del pseudocódigo. En el ejemplo, la estructura de texto del programa de conocimientos de los empleados

Las condiciones de control que rigen la ejecución de las estructuras selección e iteración se añaden a la estructura de texto en este paso. Al final del paso de texto, el diseño de programa está listo para su implantación. El diagrama de red del sistema, la estructura en árbol de datos, la estructura en árbol del programa y la estructura de texto conforman todo el paquete de diseño.

Procedimiento de diseño básico

El diseño de Jackson descrito antes se denomina **proceso de diseño básico**. Puede utilizarse en el diseño de programas simples, pero no es directamente aplicable a programas complejos. Un programa complejo ha de dividirse antes en una secuencia de programas más simples, cada uno de los cuales se puede diseñar utilizando el procedimiento de diseño básico. La resolución de colisiones y la inversión de programas son técnicas incluidas en el proceso de diseño de Jackson para conducir el diseño de programas complejos. La figura 3.26 ofrece una visión más completa de los pasos del proceso de diseño de Jackson.

El proceso de diseño de Jackson se ha extendido al manejo de diseño de sistema, además de a programas, al diseño de sistemas de tiempo real y a los sistemas de información.

METODOLOGIA DE LA INGENIERIA DE LA INFORMACION DE MARTIN

La metodología de la ingeniería de la información adoptada por James Martin es una formulación descendente por pasos para la construcción de sistemas de información [9]. Los primeros pasos de esta metodología se centran en la información y el modelo de la empresa a alto nivel. El objetivo es gestionar el desarrollo del sistema y la interacción a través del control de los datos. Los datos compartidos en el sistema de información se definen en un modelo lógico y se controlan centralmente. Así, la metodología de Martín revisa todo el sistema en el intento de identificar cómo se utiliza y se comparte la información. Las funciones de alto nivel de la empresa se definen junto con la información del negocio. En los pasos posteriores, se definen con más detalles los datos y las funciones. En el último paso, se definen e implantan los datos y lógica del programa.

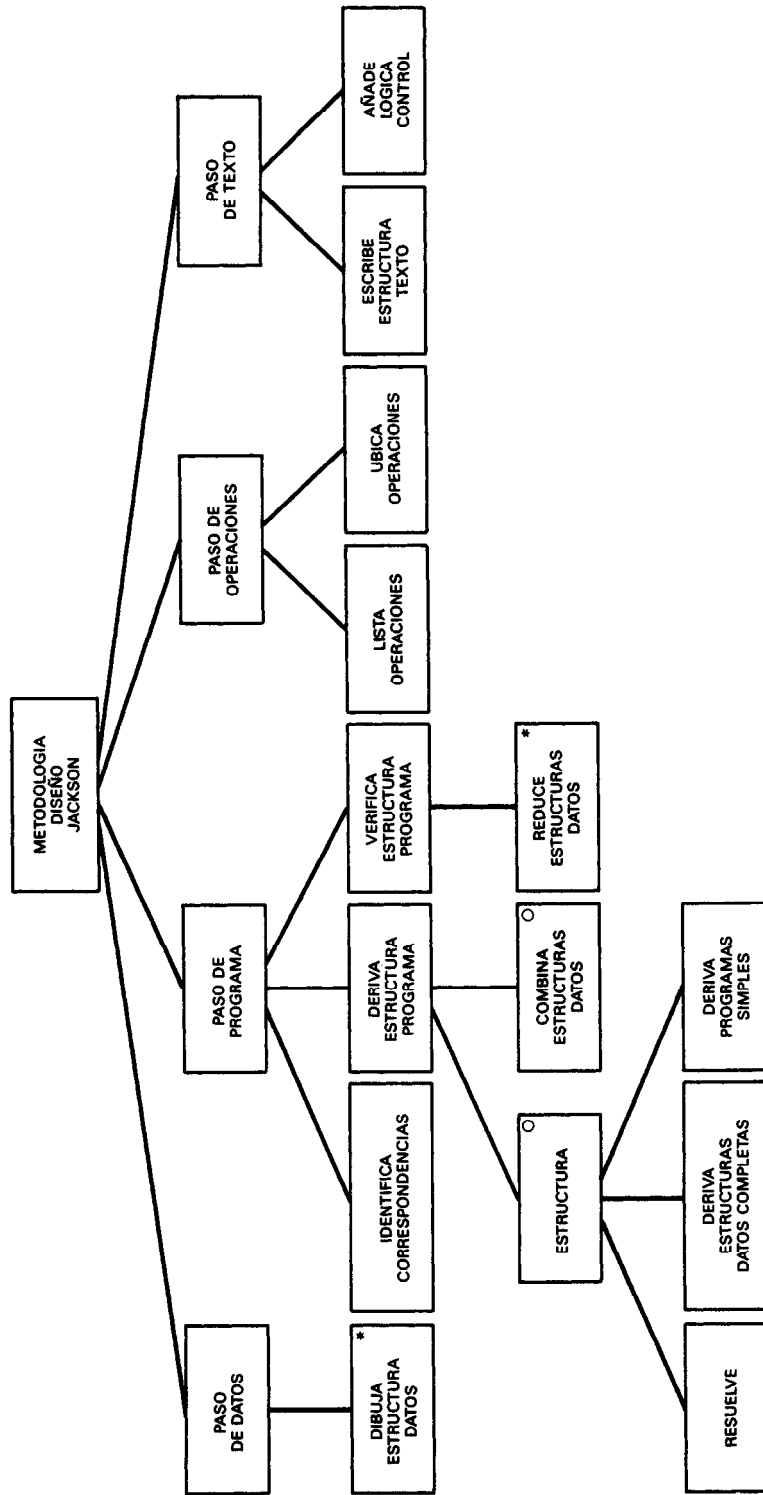


Figura 3.26. Diagrama estructurado en árbol para definir el proceso de Jackson para el diseño de programas

Las estructuras del programa se construyen en el nivel más alto de un modelo de la empresa que establece una infraestructura común de información que aúna los sistemas de información utilizados en toda la organización. Así pues, los procedimientos se derivan de los datos. El diseño lógico está separado del diseño físico y lo precede.

La figura 3.27 muestra los cuatro pasos principales de la metodología de la ingeniería de la información de Martin [10]. El cuadro 3.12 resume los tipos de diagramas estructurados utilizados en la metodología de la ingeniería de la información de Martin.

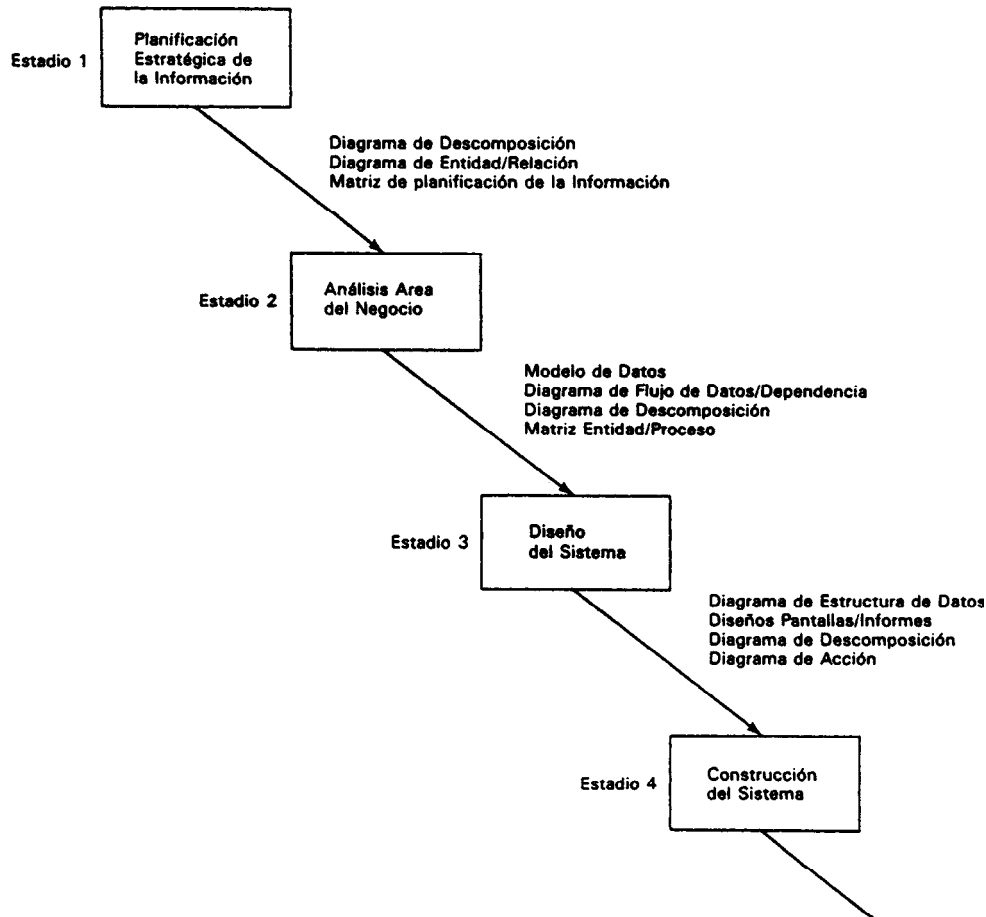


Figura 3.27. La metodología de la ingeniería de la información de Martin, divide el proceso de desarrollo del software en cuatro fases básicas

Cuadro 3.12. Los diagramas de la ingeniería de la información

Los tipos de diagramas necesarios para soportar una formulación de ingeniería de la información para desarrollar un sistema son:

- **Diagrama de entidad/relación:** muestra los tipos de entidades y sus relaciones.
- **Diagrama de la estructura de los datos:** muestra los tipos de entidades, sus atributos y sus relaciones.
- **Diagrama de descomposición jerárquica estructurada en árbol:** muestra el modelo corporativo de alto nivel y la estructura arquitectónica jerárquica de las actividades del negocio, de los procesos, de las aplicaciones, de los procedimientos y de los programas.
- **Diagramas de flujo de datos y de dependencia:** muestran el flujo de los datos entre procesos.
- **Diseños de pantalla y de informes:** muestran el diseño del interfaz del usuario.
- **Diagramas de acción:** muestran detalladamente la lógica de los programas.

Estadio 1: Planificación de la estrategia de la información

Este estadio comienza con la creación de un plan estratégico de los sistemas del negocio en el que se definen los objetivos del negocio para los próximos cinco a diez años. Junto con el plan estratégico se construyen los modelos de alto nivel y datos de la empresa. El modelo de la empresa define las funciones básicas (por ejemplo, VENTAS) y la estructura de la organización. Un diagrama estructurado en árbol jerárquico, llamado **diagrama de descomposición**, es el que se emplea para definir las funciones del negocio y las estructuras de la organización. Para definir los tipos de

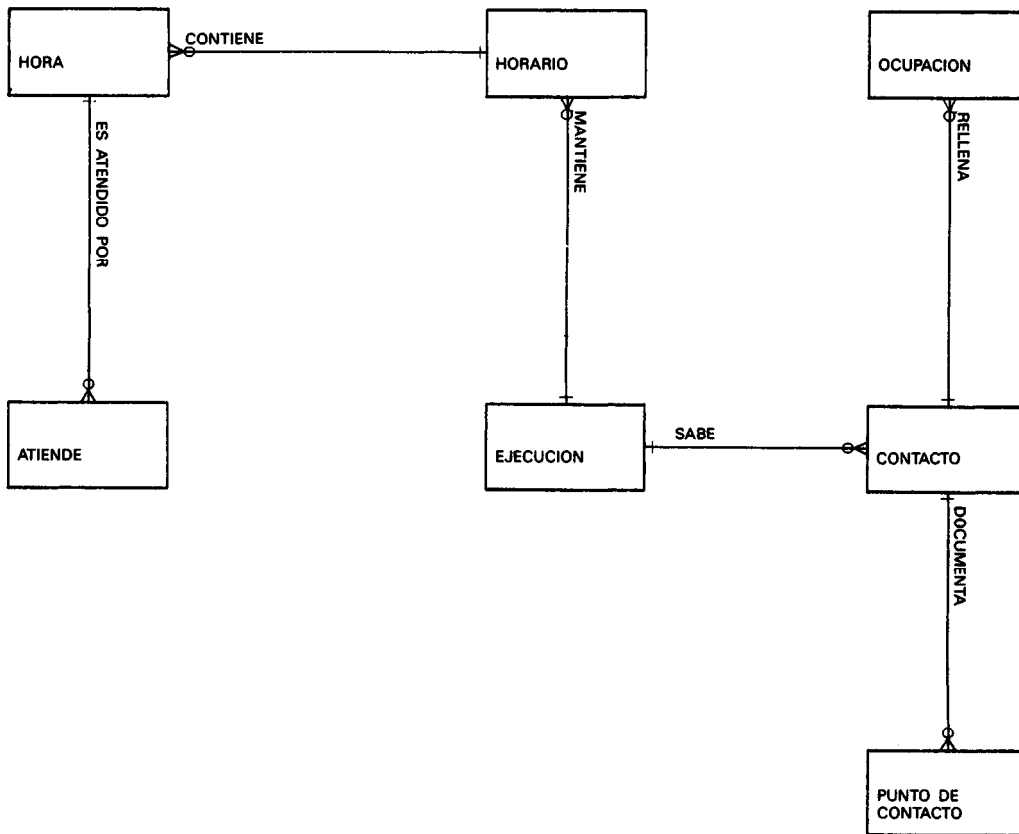


Figura 3.28. Diagrama de entidad/relación obtenido con el INFORMATION ENGINEERING FACILITY de Texas Instruments Incorporated

datos de la organización se emplea un diagrama de entidad/relación (ver figura 3.28).

Una entidad es cualquier información de la empresa que se pueda almacenar (como empleados, departamentos, clientes, facturas, etc.). Las entidades se descubren entrevistando los usuarios del sistema de información. Las visiones de los usuarios se analizan y luego se mezclan y clasifican para crear un modelo compuesto. Además, la ingeniería inversa de los sistemas existentes puede emplearse para revelar el modelo de datos existente, el cual puede utilizarse como punto de partida para crear el modelo necesario para soportar la dirección estratégica de la empresa.

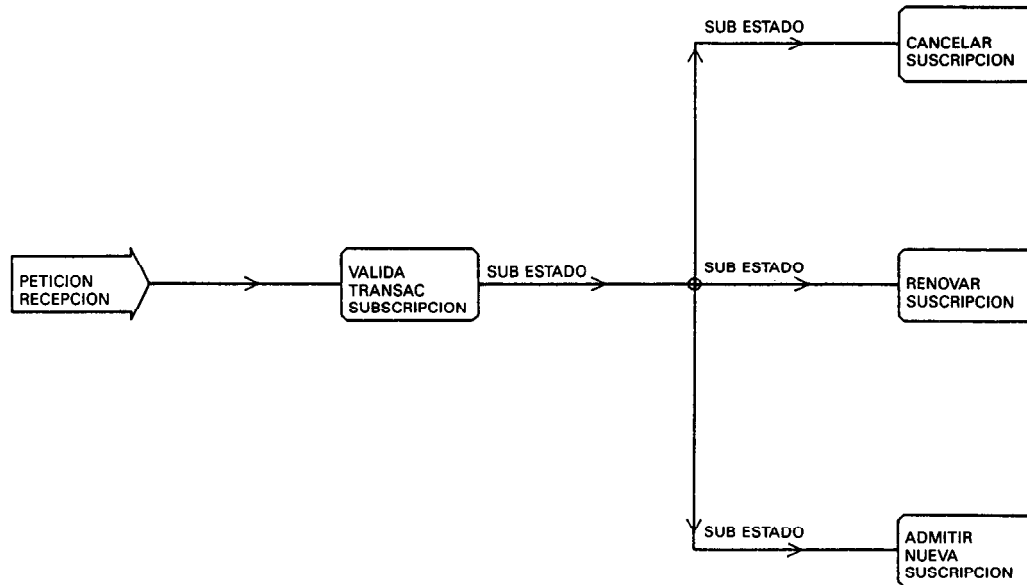


Figura 3.29. Diagrama de dependencia, un tipo especial de diagrama de flujo de datos, dibujado por el INFORMATION ENGINEERING FACILITY™ de Texas Instruments Incorporated

Finalmente, se determina en este estado el orden en el cual se van a desarrollar los sistemas de información que satisfagan los objetivos de la empresa. El usuario tiene una importante función activa en este estadio y en todo el proceso de desarrollo de los sistemas de ingeniería de la información.

Estadio 2: El análisis del área del negocio

Las funciones del negocio definidas en el estadio 1 se usan para dividir la empresa en áreas lógicas del negocio (por ejemplo, COMPRAS). Después se definen y construyen los sistemas de información necesarios para soportar un área del negocio en particular.

Este estadio se concentra en la definición de los datos y procesos necesarios para satisfacer los objetivos de la empresa dentro de un área en particular del negocio. Una parte del diagrama entidad/relación desarrollada

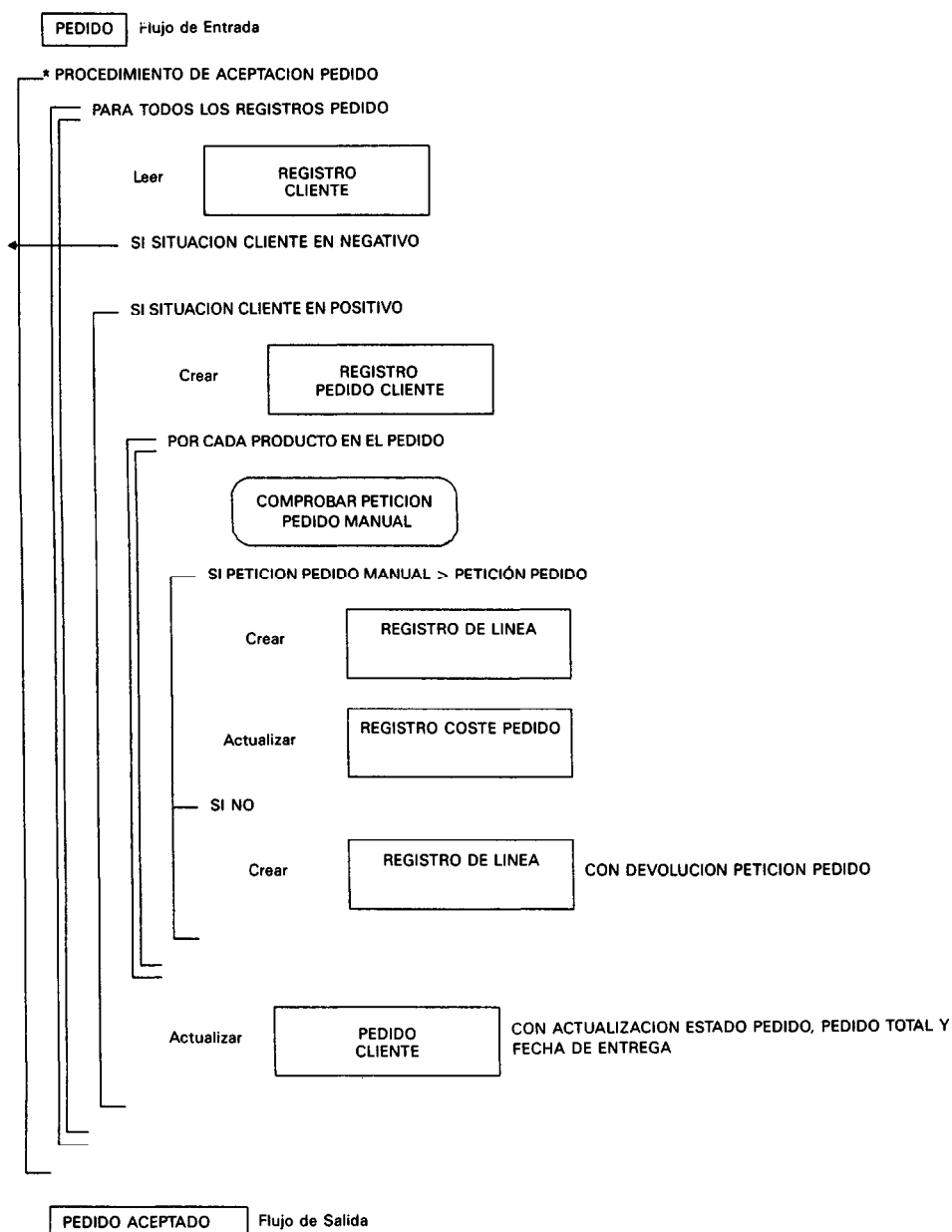


Figura 3.30. Diagrama de acción dibujado con el INFORMATION ENGINEERING WORKBENCH de KnowledgeWare para el procedimiento de aceptación de pedidos en el sistema de distribución de ventas

en el estadio 1 que concierne a un área en particular del negocio se extrae y desarrolla en un modelo de datos completamente **normalizado**. Se definen completamente las entidades, sus atributos y sus relaciones recíprocas. El diagrama de descomposición del estadio 1 que representa las funciones del negocio, se descompone en procesos por cada una de las áreas del negocio. El proceso puede representarse en diagramas de dependencia (un tipo de diagrama de flujo de datos), en diagramas de flujo de datos o en diagramas de descomposición (ver figura 3. 29). Se emplea una matriz de entidad/proceso para relacionar los datos con los procesos en los que intervienen.

Estadio 3: El diseño del sistema

El estadio 3 se ocupa de las consideraciones de diseño del sistema lógico. Los procesos del negocio definidos en el estadio 2 conforman las bases para el desarrollo del sistema en este estadio. Se diseñan los procedimientos necesarios para realizar los procesos y las estructuras lógicas en sus datos. Se utiliza la descomposición funcional descendente para diseñar los procesos. En este estadio, la metodología de la ingeniería de la información se hace similar a la metodología tradicional de la ingeniería de **software** para el desarrollo de programas. El sistema diseñado se representa con diagramas de estructura de datos, con diseños y con diagramas de acción y descomposición. Los diagramas de acción muestran detalladamente la lógica del procedimiento y los accesos a las bases de datos (ver figura 3.30).

El prototipo del interfaz de usuario (en pantallas, diálogo o gráficos) se emplea para descubrir los requerimientos del usuario. Es necesaria la implicación del usuario para ayudar a construir y comprobar los modelos de prototipos.

Estadio 4: La construcción

Mientras que el estadio 3 trataba el diseño del sistema lógico, el estadio 4 se ocupa del diseño del sistema físico y de la implantación del programa. Durante el estadio 4, toda la información del sistema se entrega como una base de datos lógica, una base de datos física y un código documentado de trabajo. Las herramientas CASE, como los lenguajes de cuarta

generación, herramientas de soporte de decisiones y las generaciones de códigos, se emplean en el estadio 4.

LA METODOLOGIA DSSD

La metodología DSSD (**Data Structured Systems Development**), metodología del desarrollo de sistemas estructurados de datos fue desarrollada por Jean Dominique Warnier en el CII Honeywell-Bull de París a finales de los años cincuenta. La metodología de Warnier, llamada LCP (**Logical Construction of Programs, Construcción lógica de programas**), está basada en el desarrollo de programas sobre la estructura de los datos del problema que le va a programar. A finales de los años setenta, Ken Orr modificó el LCP de Warnier creando una nueva metodología que llamó SPD (**Strucutured Program Desing**), diseño estructurado de programas) y que ha evolucionado en el DSSD.

El diagrama de Warnier-Orr

El DSSD utiliza la teoría matemática de conjuntos para describir el diseño del programa. Un **conjunto** es una serie ordenada de objetos que comparten una o varias características comunes (por ejemplo, un conjunto de números enteros o el conjunto de departamentos de una compañía). En matemáticas, un conjunto se describe con la lista de sus miembros encerrada entre llaves. Por ejemplo, los departamentos de una compañía forman el conjunto indicado por:

{contabilidad, mercado, producción, compras}

El DSSD utiliza una notación similar para los conjuntos, excepto que se elimina la llave de la derecha y que los miembros se listan verticalmente:

{
contabilidad
mercado
producción
compras

El **diagrama de Warnier-Orr**, la herramienta central de la metodología DSSD, se compone de conjuntos anidados. Un ejemplo de diagrama de

Warnier-Orr se muestra en la figura 3.21. Los diagramas de Warnier-Orr se emplean para mostrar la estructura jerárquica y el flujo de actividades, procedimientos o datos del proceso.

La formulación orientada a la salida

El desarrollo de sistemas estructurados de datos es una metodología para la construcción de sistemas de información. Es similar a la metodología de diseño de Jackson en el sentido de que ambas son enfoques orientados a los datos. En ambas la estructura del programa se deriva de la estructura de los datos. También en ambas se acentúa que el diseño lógico debe separarse del diseño físico y precederlo.

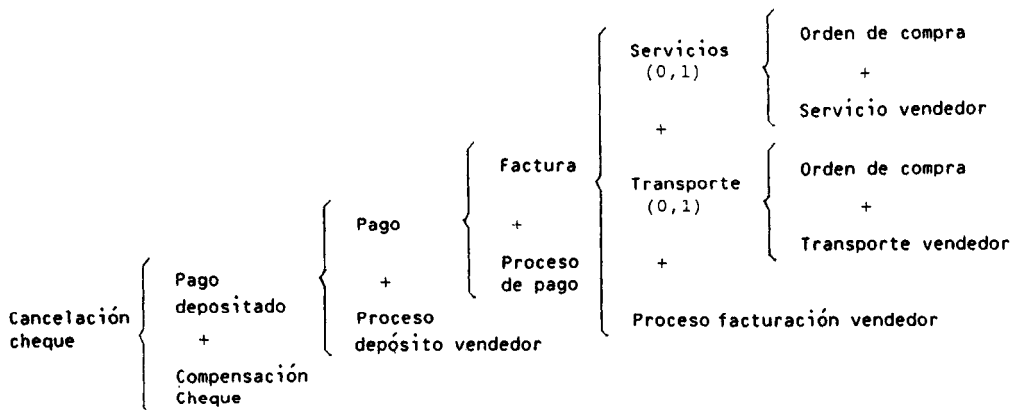


Figura 3.31. El diagrama de Warnier-Orr se utiliza para representar jerárquicamente los planes, los procedimientos y las estructuras de los datos. Este diagrama de Warnier-Orr se dibujó con el DESIGNAID de Nastec Corporation

La principal diferencia entre las dos es que en la formulación de Jackson se mezclan todas las estructuras de datos de entrada y de salida para formar una única estructura de programa, mientras que en la de Warnier-Orr, la estructura del programa y las estructuras de los datos de entrada se derivan de las estructuras de los datos de salida. La idea de Warnier-Orr es que la salida del sistema/programa determina total y absolutamente la estructura de los datos y ésta, a su vez, determina la estructura del programa [11].

Según Ken Orr, “trabajando hacia atrás desde la salida es posible determinar exactamente las bases de datos y las entradas mínimas del sistema” [12].

Los pasos básicos del DSSD

Como se muestra en el diagrama de Warnier-Orr de la figura 3.32, la metodología DSSD consta de nueve fases:

Fase de planificación del proyecto

Durante esta fase se desarrolla el plan del proyecto enlazando todos los objetivos del sistema. Se definen los resultados esperados del negocio que el sistema puede proporcionar y las limitaciones comerciales del sistema.

Fase de definición de los requerimientos

Durante esta fase se definen los requisitos del usuario y organizativos para el sistema. También se define el entorno del ordenador necesario para

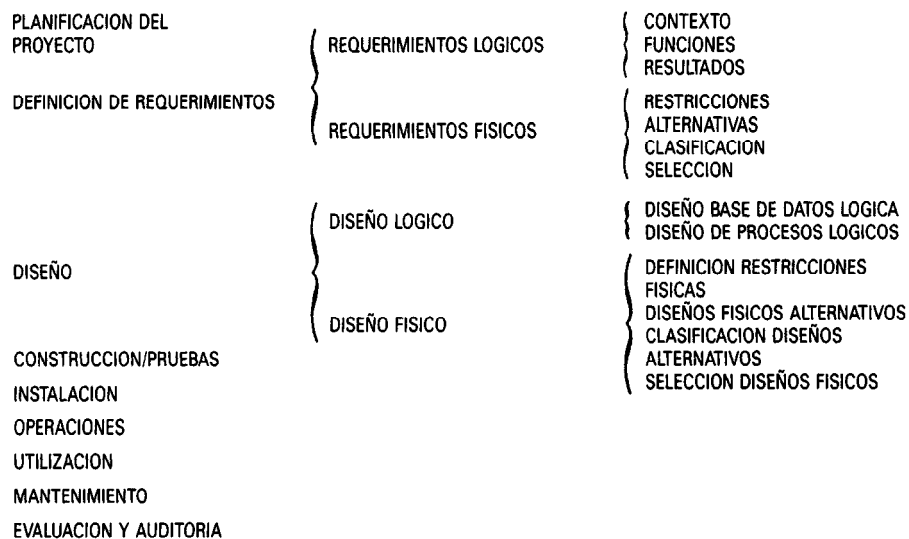


Figura 3.32. Representación de la metodología de desarrollo de sistemas por medio de un diagrama de Warnier-Orr

el sistema. La fase de definición de requerimientos se divide en dos subfases: los requerimientos lógicos y los físicos, y, a su vez, estas subfases se dividen en pasos. El propósito principal de esta fase es definir las salidas principales que va a producir el sistema. Estas salidas son el fundamento del diseño de bases de datos y sistemas durante la fase de diseño del DSSD.

El **diagrama de entidad** (un tipo de diagrama de flujo de datos) se emplea para definir las entidades del sistema y las transacciones que suceden entre ellas. El diagrama de entidad se utiliza para determinar la amplitud del sistema. El **diagrama de línea de montaje** (un tipo de diagrama de Warnier-Orr) se utiliza para definir los procesos funcionales (es decir, la transformación del conjunto de entrada en uno de salida). Las salidas necesarias para soportar los procesos y definir la base de salida lógica se representan en un diagrama de Warnier-Orr y en un diccionario de datos. La figura 3.33 es un ejemplo de una base de salida lógica.

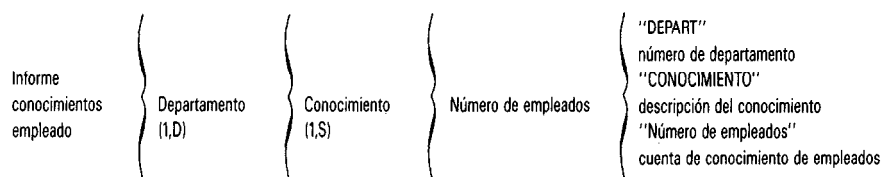


Figura 3.33 Se utiliza el diagrama de Warnier-Orr para representar la base de salida lógica de los datos del sistema de conocimientos de los empleados

Fase de diseño

La metodología DSSD divide la fase de diseño en dos partes: el diseño lógico y el diseño físico.

1. Diseño de la base de datos lógica.
2. Diseño del proceso lógico.

Utilizando la base de salida lógica de la fase previa como entrada, el desarrollo de la base de datos lógica comienza por el desarrollo de la estructura de datos lógica que contienen los datos mínimos requeridos para producir una salida. Seguidamente, se desarrollan los ficheros básicos lógicos y todos sus datos representando una arquitectura de datos normali-

zada. Puede demostrarse que cada sistema salida se produce a partir de los ficheros básicos lógicos y que todos sus datos son necesarios para producir la salida del sistema. Luego se definen la actualización de los datos lógicos, la edición y las transacciones de entrada.

En el paso final del diseño de la base de datos lógica se desarrollan la base de actualización lógica y la base de entrada lógica, que se utilizan más tarde para guiar el diseño de la base de datos física.

Las estructuras de los datos lógica, la base de actualización lógica y la base de entrada lógica se representan en el diagrama de Warnier-Orr. La figura 3.34 es un ejemplo de estructura de datos lógica.

Durante el paso del diseño lógico del proceso se desarrolla el proceso lógico, que también se representa con un diagrama de Warnier-Orr. El objetivo del proceso lógico es definir las transformaciones necesarias para producir las salidas del sistema desde las entradas.

Durante la fase de diseño físico, las restricciones del entorno físico (como las impuestas por el lenguaje de programación) se añaden a la base del proceso lógico.

La figura 3.35 es un diagrama de Warnier-Orr, que representa el diseño completo de un sistema de conocimientos de los empleados.

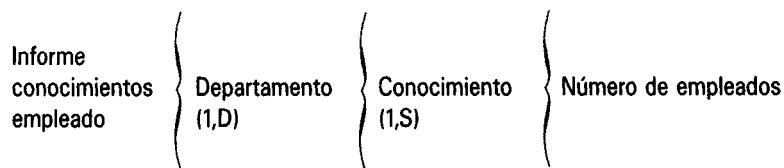


Figura 3.34. Se utiliza el diagrama de Warnier-Orr para representar la estructura lógica de los datos del sistema de conocimientos de los empleados

BIBLIOGRAFIA

1. V. Haase y G. Koch, "Developing the Connection between User and Code", *Computer*, Vol. 15, N.º 5, mayo 1982, págs. 10-11.
2. B. Boehm, R. McClearn y D. Unfrig, "Some Experiences with Automated Aids to the Design of Large-Scale Reliable Software", *IEEE Transactions on Software Engineering*, Vol. 1, N.º 1, marzo 1975, págs. 125-133.
3. S. Houtchens y J. Pollock, "Developing Real-Time Software without Writing Source Code", *Internal Report*. Santa Clara, California: Integrated Systems, Inc., 1987.
4. J. Emos y R. Van Tilburg, "Software Desing", *Computer*, Vol. 14, N.º 2, febrero 1981, págs. 61-83.
5. F. L. Bauer, *Software Engineering*. Amsterdam: North-Holland, 1972.
6. J. Martin y C. McClure, *Structured Techniques: The Basis of CASE*. Englewood Cliffs, Nueva Jersey: Prentice-Hall, 1988.
7. R. Pressman, "Meeting the Challenge of Real-Time Software Engineering", *The CASE Report*. Southfield, Michigan: Nastec Corporation, marzo 1987, págs. 1-3.
8. "CASE 1987 Survey Report", *Software News*. Westborough, Massachusetts: Sentry Publishing Company, 1987.
9. J. Martin, "Information Engineering", *Savant Technical Report*. Carnforth, Lancashire, Inglaterra, 1987.
10. J. Martin y E. A. Hershey, "Information Engineering Whitepaper", *Technical Report*. Atlanta, Georgia: KnowledgeWare, 1986.
11. D. Higgins, *Program Design and Costruction*. Englewood Cliffs, Nueva Jersey: Prentice-Hall, 1979.
12. K. Brown y R. Whinery, editores, *Data Structured Systems Development Methodology*. Topeka, Kansas: Ken Orr & Associates, Inc., 1986.

LAS PLATAFORMAS HARDWARE DE LA CASE

LA REDEFINICION DEL ENTORNO DEL SOPORTE DE SOFTWARE

“Se está convirtiendo en una necesidad ofrecer un entorno totalmente integrado que proporcione un camino rápido y efectivo desde la descripción del problema (software) hasta una solución efectiva” [1]. Sin embargo, la tecnología CASE no es meramente una fórmula para el ensamblaje de un conjunto de herramientas relacionadas con el ciclo de vida del **software**. Es una redefinición de todo el desarrollo del **software** y del entorno de soporte. Como tal, las herramientas, las metodologías y la plataforma **hardware** deben estar dirigidas por la tecnología CASE.

En los capítulos 2, 3 y 5 se analiza cómo la CASE redefine las herramientas y metodologías **software**. En este capítulo se trata la plataforma **hardware** de la CASE.

LAS ALTERNATIVAS DE LAS PLATAFORMAS HARDWARE

Como se muestra en la figura 4.1, hay tres alternativas de las plataformas **hardware** para el entorno de soporte de **software** CASE.

- 1.—Estación de trabajo individual (independiente).

- 2.—Arquitectura a dos niveles, compuesta por un ordenador anfitrión o principal y estaciones de trabajo individuales.
- 3.—Arquitectura a tres niveles, compuesta por un ordenador anfitrión, un ordenador anfitrión intermedio (a nivel de departamentos o de proyecto) y estaciones de trabajo individuales.

La plataforma de la estación de trabajo individual

Las estaciones de trabajo individuales proporcionan una plataforma muy interactiva, de rápida respuesta y dedicada donde realizar las tareas del ciclo de vida del **software**. En particular, sus potentes prestaciones gráficas permiten crear y manipular fácilmente los diagramas estructurados utilizados para especificar y documentar los sistemas de **software**. Además, permiten hacer prototipos rápidos para la creación de los modelos del sistema que ayudan a descubrir y aclarar los requerimientos del usuario. Son las plataformas perfectas para las tareas de análisis y diseño. La estación de trabajo individual proporciona el soporte máximo posible a cada una de las personas que intervienen en el desarrollo y mantenimiento del **software** y en los trabajos de gestión del proyecto.

La arquitectura **hardware** de una estación de trabajo CASE puede ser una máquina de 32 bits (como HP9000, SUN-2, SUN-3, Apollo Series 3000, DEC VAXSTATION II, IBM RT PC) o un ordenador personal (como IBM PC AT/XT, PS/2, Macintosh, TI Professional PC, Wang PC). La mayoría de los sistemas CASE que soportan el desarrollo de sistemas en tiempo real/embebidos se procesan en estaciones de trabajo basadas en máquinas de 32 bits, mientras que otros sistemas CASE que soportan el desarrollo de aplicaciones MIS se procesan en ordenadores personales.

En ambos casos, los aspectos de la elección del **hardware** de las estaciones de trabajo se centran en:

- Las capacidades de memoria y de proceso.
- La calidad de los gráficos.
- Las posibilidades de conexión entre redes.

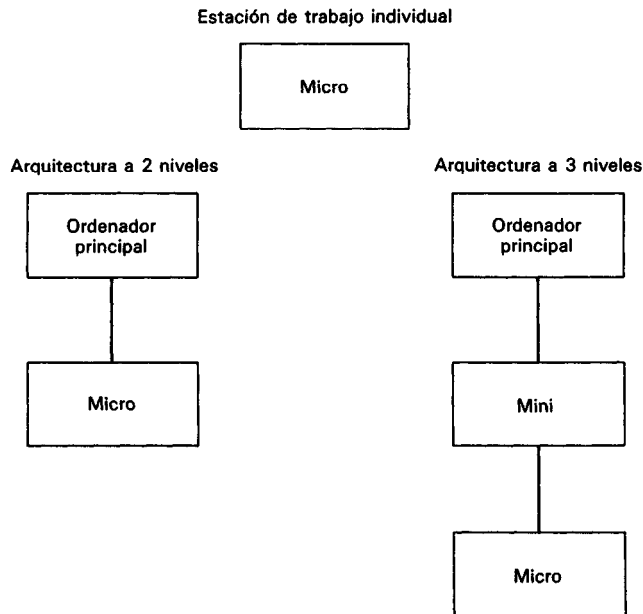


Figura 4.1. Hay tres plataformas alternativas de hardware CASE para soportar el entorno software

- La capacidad de multiusuario/multitarea.
- La conectividad.

Para la mayoría de estas prestaciones, las estaciones técnicas de trabajo de 32 bits, aunque más caras, son superiores a los ordenadores personales, pues la estación técnica de 32 bits es más potente y fiable. Su mayor capacidad de memoria y de proceso permiten al programador trabajar en sistemas grandes y tener un acceso más rápido a más sistemas de información. Como las máquinas de 32 bits pueden ofrecer un verdadero entorno multiusuario/multitarea, el profesional puede trabajar en múltiples tareas simultáneamente. Además, las máquinas de 32 bits pueden ofrecer gráficos de alta calidad con mayor resolución y presentaciones más grandes (por ejemplo, la Apollo Serie 3000 ofrece una resolución de 1280 por 1024 pixels). Finalmente, estas máquinas pueden proporcionar un soporte para la conexión entre redes de comunicaciones. Por ejemplo, la DEC Vaxstation opera con la red DECnet; la SUN, con las redes Ethernet y NSE (Network Software Environment), y Apollo, con Domain.

Los puentes entre las estaciones de trabajo

En la práctica, sin embargo, la estación de trabajo CASE no puede existir aislada. Deben proporcionarse puentes para la conexión de la estación de trabajo del profesional individual a otras estaciones de trabajo y a otros sistemas **hardware**. Esto permite la comunicación y compartir información entre los miembros del equipo del proyecto y el intercambio de datos entre el sistema CASE y los diccionario de datos, bases de datos y herramientas externas (como los lenguajes de la cuarta generación y los generadores de código).

La conectividad entre las estaciones de trabajo se consigue utilizando redes de área local (por ejemplo, IBM Token Ring, PC Net y Novell). El objetivo es compartir la información del sistema almacenada en el depósito CASE. Para compartir los sistemas de información en las estaciones de trabajo, una de las estaciones debe actuar como servidor que se designa como depósito maestro (por ejemplo, la librería del proyecto que mantiene toda la información asociada con la totalidad del proyecto). Los programadores tienen librerías del proyecto en sus propias estaciones de trabajo importando partes seleccionadas a su depósito CASE local. El contenido del depósito CASE maestro se protege con restricciones de los privilegios de actualización a los programadores individuales. Una estructura multiusuario de almacenamiento de datos proporciona un mecanismo de bloqueo de los registros que permite a los múltiples usuarios acceder concurrentemente y sin conflictos al mismo fichero.

LA PLATAFORMA HARDWARE DE NIVELES MULTIPLES

Aunque la conectividad entre las estaciones de trabajo puede soportar muchas tareas del ciclo de vida del **software** y algunas formas de comunicación entre el equipo, no puede soportar el trabajo todo del **software**. Se sigue necesitando un ordenador principal o un miniordenador grande en la plataforma de **hardware** CASE. Algunas tareas, como la generación de código, el análisis del código fuente existente y las pruebas de los programas, requieren la potencia de un ordenador grande. Igualmente, las necesidades de almacenamiento de la información, la consolidación de la información y las tareas de análisis también requieren la potencia de un ordenador grande (ver cuadro 4.1).

Cuadro 4.1 Las funciones de la CASE

Función	Anfitrión	Estación de trabajo
Creación, manipulación y actualización de los diagramas		×
Generador de pantallas y de informes		×
Definición de menús		×
Prototipo/simulación	×	×
Verificación de las especificaciones	×	×
Almacenamiento, gestión y notificación de la información del sistema	×	×
Generación de la documentación	×	×
Generación de código	×	
Pruebas del sistema	×	
Generación de las bases de datos		×
Reestructuración de los datos	×	
Análisis del programa	×	×
Análisis y consolidación de los datos	×	
Síntesis canónica y normalización	×	

La arquitectura **hardware** de soporte para un **CASE workbench** de alto rendimiento (por ejemplo, el INFORMATION ENGINEERING FACILITY de TI o el INFORMATION ENGINEERING WORKBENCH de KnowledgeWare) debe ser de dos o tres niveles. La arquitectura de dos niveles consta de varias estaciones de trabajo (ordenadores personales o máquinas de 32 bits) conectadas a un ordenador anfitrión (un ordenador principal o un miniordenador potente). Los dos niveles **hardware** se conectan mediante transferencia de ficheros y dispositivos de carga/descarga.

Algunos sistemas CASE emplean un nivel de **hardware** intermedio entre el ordenador principal y las estaciones de trabajo con ordenadores personales. El nivel intermedio es un servidor de ficheros que soporta un depar-

tamento o un equipo de proyecto. Por ejemplo, en el sistema Maestro de Softlab, el servidor es un sistema Motorola 5000 que actúa como base de datos en una red de área local para archivar todos los documentos del proyecto [2].

La distribución de las tareas entre los diversos niveles reduce la competencia por la adquisición de los caros recursos del ordenador principal y permite ejecutar tareas en el sistema **hardware** más conveniente.

Las características de una plataforma **hardware** CASE de dos o tres niveles incluyen:

- Conectividad entre niveles y entre estaciones de trabajo.
- El soporte multiusuario.
- Interfaces con diccionarios y bases de datos corporativos.
- Puentes con los de cuarta generación (4GL).
- Conexión de múltiples depósitos de información del sistema.

Los puentes con diccionarios, bases de datos y lenguajes de cuarta generación

La conexión de las estaciones de trabajo CASE con el ordenador anfitrión donde existen diccionarios de información corporativa (como IBM DB/DC Data Dictionary y MSP Data Manager) y sistemas de gestión de bases de datos (como Cullinet IDMS/R) reduce la necesidad de introducir información redundante. Para acceder a definiciones de datos corporativos estándar, se necesita un camino de importación/exportación entre la estación de trabajo CASE y los diccionarios y bases de datos residentes en el anfitrión. Este enlace micro-principal lo proporciona la transferencia de ficheros, la tarjeta de emulación 3278/79, etc. Por ejemplo, con las utilidades de importación un depósito CASE podría generalizarse con los ya existentes IBM IMS Program Specifications Blocks (PSB), la descripción de estructuras de datos COBOL, las pantallas CICS BMS, la descripción de ficheros VSAM y DB2. Mediante las utilidades de exportación desde el depósito de la estación de trabajo se puede pasar información de diseño

de alto nivel al generador de código residente en el ordenador principal. Para simplificar la transferencia de datos, el formato de los ficheros de diccionario y el del depósito CASE podría ser el mismo (por ejemplo, dBASE III).

La conexión entre la estación de trabajo CASE y el ordenador principal (o el miniordenador) también proporciona un puente con los lenguajes de cuarta generación y con los sistemas gestores de base de datos, lo que permite utilizar la tecnología CASE en el desarrollo de aplicaciones y bases de datos de la cuarta generación. El análisis y el diseño de las aplicaciones en lenguajes de cuarta generación pueden realizarse en la estación de trabajo CASE. Después, los ficheros que contienen las especificaciones de la aplicación (por ejemplo, pantallas, informes, descripciones de registros, etc.) pueden transferirse al entorno del ordenador principal, donde llegan como entradas al generador del lenguaje de cuarta generación o al generador de código. En el caso de las bases de datos, la planificación estratégica, el modelado lógico de los datos y el diseño lógico o físico de las bases de datos pueden realizarse en la estación de trabajo. Después, los ficheros que contienen el esquema sintáctico de la base de datos puede transferirse al ordenador principal para la normalización y generación de base de datos.

LOS COMPONENTES DE UN SISTEMA CASE BASICO

Como se muestra en la figura 4.2, podemos dividir un sistema CASE en tres componentes básicos:

- 1.—Front-end
- 2.—Depósito central
- 3.—Back-end

El componente **front-end** corresponde a las fases primarias del ciclo de vida del **software**; es decir, el análisis y el diseño. El **front-end** puede corresponder a la parte del ordenador personal o estación de trabajo de la plataforma de **hardware** CASE. Las herramientas **CASE front-end** proporcionan funciones para soportar las actividades de análisis y de diseño, como diagramación, prototipos y comprobación de especificaciones. Estas actividades

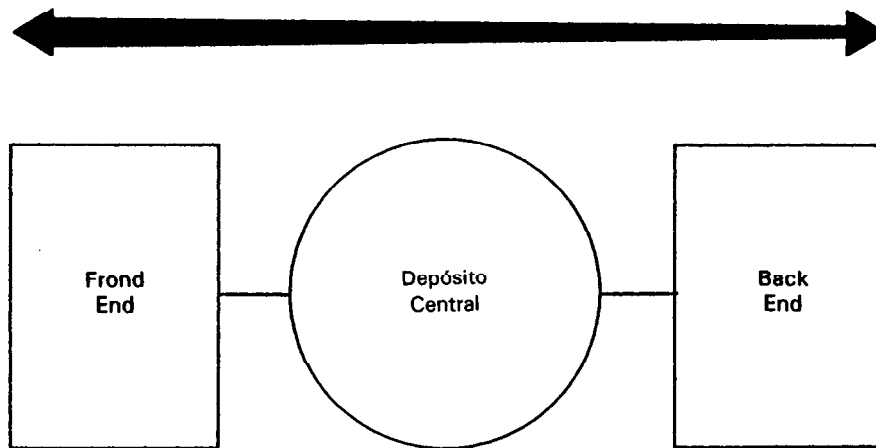


Figura 4.2. Los componentes básicos de un banco de trabajo (workbench) CASE son: el front-end, el depósito y el back-end

se realizan mucho mejor en una máquina personal dedicada con tiempo de respuesta rápida, mapa de bits gráficos de alta resolución y ratón.

El componente **back-end** corresponde a las últimas fases del ciclo de vida del **software**; es decir, la implantación y el mantenimiento del programa. El **back-end** también corresponde a la porción del ordenador principal en la plataforma de **hardware** CASE. Las herramientas del **back-end** CASE automatizan el código, las comprobaciones, la generación de las bases de datos la normalización de los datos, y el análisis del impacto en el sistema existente. Estas tareas requieren la potencia y capacidad de almacenamiento de un ordenador principal o de un miniordenador.

Los depósitos múltiples

El depósito de información es el enlace entre los componentes **front-end** y **back-end** de un sistema CASE. Es el vehículo de la comunicación por el cual toda la información del sistema reunida durante el ciclo de vida del **software** se gestiona y comparte. Es también el vehículo por el cual el trabajo del sistema de los diferentes equipos el puede combinar, analizar y consolidar en una representación del sistema consistente, en progreso y completa. El depósito CASE es el integrador básico del ciclo de vida que

permite a las herramientas CASE utilizadas en una fase del ciclo de vida pasar los datos a la fase siguiente.

Lógicamente, el depósito CASE está dividido en librerías de proyectos y en modelos del sistema. Físicamente, el depósito CASE está dividido en niveles que se corresponden con las plataformas de **hardware** del sistema CASE. A nivel de la estación de trabajo, existe un depósito local para soportar el desarrollo individual. A nivel de ordenador anfitrión, existe un depósito principal para mantener toda la información corporativa. A nivel de departamento o de proyecto hay un depósito intermedio para mantener toda la información del proyecto.

Debe existir un enlace entre los depósitos físicos para que los datos puedan cargarse y descargarse entre los niveles. Por ejemplo, el contenido del depósito anfitrión puede descargarse total o parcialmente en un depósito de proyecto o de estación de trabajo; así mismo, el contenido del depósito a nivel de estación de trabajo puede cargarse y combinarse con los datos del depósito a nivel superior. Los enlaces los proporcionan utilidades de comunicaciones como Ethernet, TCP/IP, NFS, Domain, DECnet, SNA y el protocolo 3270.

Sin embargo, el aspecto real del transporte de información del sistema entre los entornos de **hardware** CASE conlleva no solamente la posibilidad de transferir datos, sino también la facilidad de seleccionar y formatear los datos necesarios. ¿Es fácil extraer los datos necesarios y determinar su disponibilidad e integridad físicas? ¿Pueden los datos comprimirse durante el traslado? ¿Es fácil reestructurar los datos en un formato en el que puedan utilizarse? Así pues, la conectividad entre los sistemas **hardware** debe incluir:

- Selección de los datos a descargar.
- Carga y consolidación de los datos.
- Necesidades de reconstrucción del formato de los datos.
- Control y seguridad de los datos.
- Velocidad.
- Eficiencia.
- Acceso multiusuario a los ficheros.
- Copias de seguridad (**backup**).

Los interfaces transparentes

La figura 4.2 muestra los componentes básicos de un sistema CASE. La figura 4.3 muestra otra visión, quizás más realista, de los componentes básicos de un sistema CASE. Obsérvese que en la figura 4.3 hay múltiples depósitos. Hay varias razones para ello.

- 1.— Se necesita un depósito local para mantener todos los conjuntos de información del trabajo para cada uno de los componentes del equipo. Cada depósito local debe residir físicamente en la plataforma **hardware** de estación de trabajo.
- 2.— Se necesita un depósito anfitrión para mantener toda la información de los sistemas a nivel corporativo. Este depósito debe residir en un ordenador principal o en un miniordenador.
- 3.— Diversos fabricantes pueden suministrar herramientas CASE para soportar las varias fases del ciclo de vida. Por ejemplo, un vendedor suministra una herramienta CASE **back-end** con su propio depósito, y otro vendedor ofrecer su propia versión.

Un elemento importante para integrar las herramientas CASE **front-end** es el interfaz entre los múltiples depósitos. Para crear interfaces lo más transparentes y eficientes posible se requiere la cooperación entre los fabricantes y los estándares.

Los fabricantes pueden instalar puertos propios para los productos de otro fabricante (por ejemplo, APS/PC EXCELERATOR INTEGRATOR para conectar EXCELERATOR a APS) o aplicar una política de arquitectura abierta informando de los requerimientos de un interfaz (por ejemplo, XL/INTERFACE TELON de Pansophic permite a los usuarios de distintas herramientas CASE **front-end** conectarse con el generador de código TELON). En el mundo real del CAD/CAM existe actualmente un método estándar para enlazar las herramientas CAD/CAM. Es el EDIF (**Electronic Document Interchange Form**), que es un protocolo de comunicación de datos entre herramientas. Es posible que el EDIF llegue también a ser un estándar en el mundo CASE (por ejemplo, Cadre's TEAMWORK soporta el estándar EDIF).

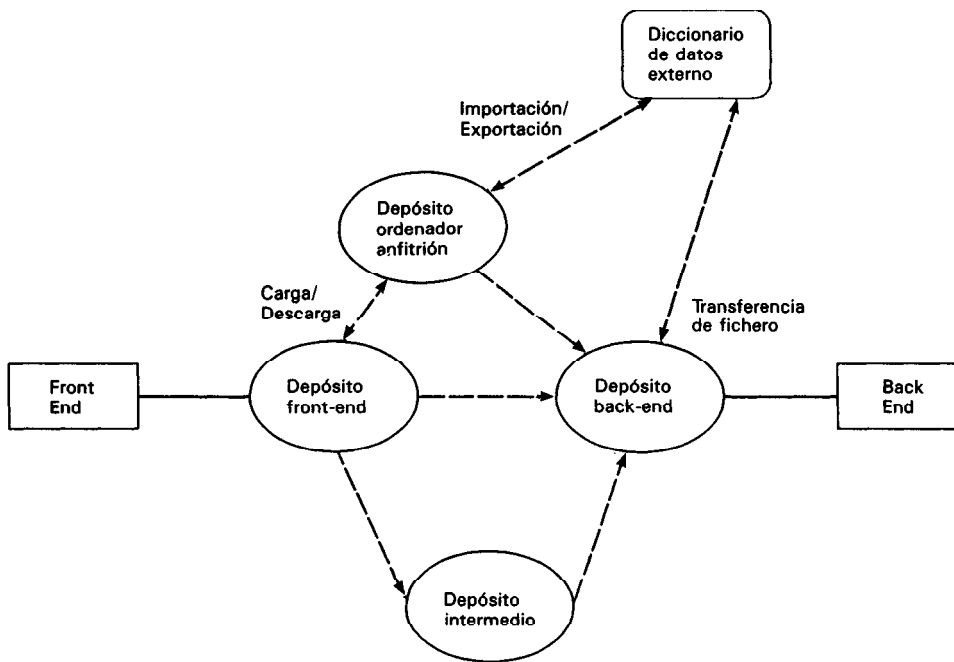


Figura 4.2. Un esquema más completo de los componentes básicos de un banco de trabajo CASE, que muestra la disposición de los distintos depósitos

RESUMEN

Un sistema CASE completo que soporta enteramente el trabajo **software** reside en plataformas de **hardware** múltiples:

1. Estación de trabajo.
2. Miniordenadores.
3. Ordenadores principales.

Compartir información y herramientas a través de las plataformas **hardware** es un aspecto importante para hacer la tecnología CASE práctica y eficiente. La conectividad entre los sistemas **hardware** y los interfaces entre

herramientas son posibilidades importantes que se deben examinar en la evaluación y selección de un sistema CASE. El cuadro 4.2 resume algunas consideraciones sobre la plataforma **hardware** CASE:

Cuadro 4.2. Consideraciones sobre las plataformas hardware CASE

Consideraciones a nivel de estación de trabajo

- La capacidad de memoria y de proceso.
- La calidad de los gráficos.
- La capacidad de la conexión a través de redes de comunicaciones.
- La capacidad de multitarea.
- La conectividad

Consideraciones de la plataforma a dos y tres niveles

- La conectividad entre niveles y entre estaciones de trabajo.
- El soporte multiusuario.
- Los interfaces entre los diccionarios corporativos y las bases de datos.
- Los puentes con los lenguajes de cuarta generación.
- El enlace entre depósitos múltiples.
- La velocidad y la eficiencia.
- La seguridad y el control.
- Las copias de seguridad.
- El acceso concurrente a ficheros y las prestaciones del bloqueo de registros.

BIBLIOGRAFIA

1. William Suydam, "CASE Makes Strides toward Automated Software Development", *Computer Design*, enero 1, 1987, págs. 49-70.
2. "An Overview of Structured Analysis and Design Tools", *CASE Outlook*, Vol. 1, N.º 2, agosto 1987, págs. 1-14.

LAS CATEGORIAS DE HERRAMIENTAS CASE

LAS DISTINTAS HERRAMIENTAS CASE

Las diferentes herramientas CASE se enfocan hacia el soporte de diferentes fases del ciclo de vida del **software** o al desarrollo de diferentes tipos de sistemas **software**. Para distinguir más fácilmente las herramientas CASE, vamos a catalogarlas en este capítulo.

Según se define en el cuadro 5.1, hay tres categorías básicas de herramientas CASE:

1. Juegos de herramientas CASE.
2. Bancos de trabajo CASE.
3. Compañeros de metodologías CASE.

Los **juegos de herramientas (toolkits)** son el tipo más simple de herramientas CASE. Son un conjunto integrado de herramientas que automatizan un tipo de tarea del ciclo de vida del **software**, como el diseño del sistema, el mantenimiento del programa o un tipo de clase de trabajo, como el análisis del sistema.

Cuadro 5.1. Categorías de herramientas CASE

• Juego de herramientas: (Toolkit)	conjuntos de herramientas integradas que soportan un tipo de función del desarrollo del ciclo de vida del software o una clase de trabajo.
• Banco de trabajo: (Workbench)	entornos de propósito general para el desarrollo del software soportando la totalidad del rango del trabajo del software .
• Compañero de metodología: (Methodology Companion)	asistencia por ordenador para una metodología de desarrollo del software en particular.
• Juego de herramientas de compañero de metodología.	
• Banco de trabajo de compañero de metodología.	

Los **CASE workbench** se componen de un conjunto de herramienta integradas que automatizan las tareas a lo largo de todo el ciclo de vida del **software**. Un **compañero de metodología CASE** proporciona asistencia por ordenador para una particular metodología de desarrollo del **software**, como el análisis estructurado de DeMarco, el diseño estructurado de Jackson o la ingeniería de la información de Martin. Las categorías de herramientas CASE se tratan en este capítulo.

LOS JUEGOS DE HERRAMIENTAS CASE

Los juegos de herramientas CASE pueden descubrirse como herramientas a nivel de fase, porque enfocan el soporte de una fase en particular de ciclo de vida del **software** o de un tipo de sistema de **software**. Como se muestra en la figura 5.1 hay herramientas de análisis, de diseño, de programación y de mantenimiento. También hay juegos de herramientas enfoca

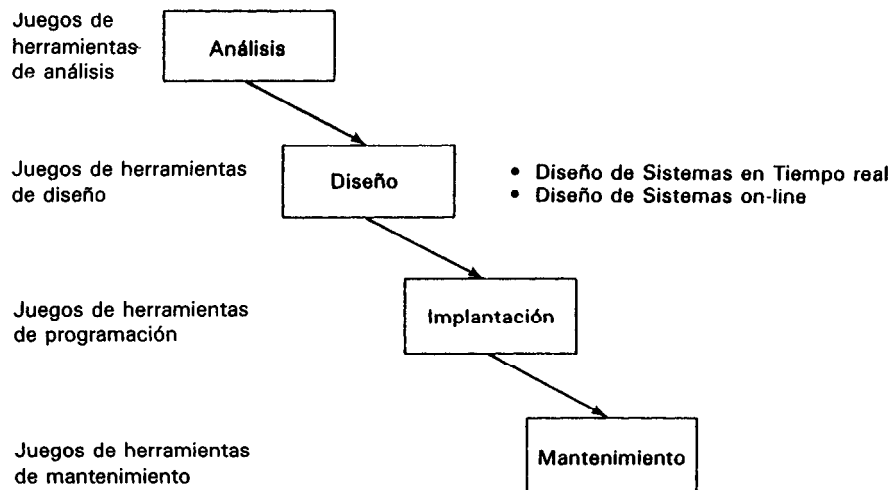


Figura 5.1. Los juegos de herramientas CASE se llaman herramientas CASE a nivel de fase porque se centran en el soporte de una fase en particular del ciclo de vida del software o en el desarrollo de un tipo de sistema

dos al diseño de sistemas de tiempo real, sistemas MIS en línea o de la gestión del proyecto.

Los juegos de herramientas pueden clasificarse, además, según el entorno **hardware** y del sistema operativo en el que se procesan, la línea del producto al que pertenecen, si tienen arquitectura abierta o cerrada y según la metodología que soportan.

Por ejemplo, unos se procesan en ordenadores personales con MS-DOS o PS/2. Otros en estaciones de trabajo de 32 bits, como SUN, Apollo o DEC Vaxstation II. Algunos en Macintosh, Wang, PC y II Professional PC. Finalmente, algunos en ordenadores principales como IBM, Data General o de la serie DEC VAX.

Algunos juegos de herramientas son parte de una familia de productos de **software**. Pueden utilizarse aisladamente o en combinación con otras herramientas dentro de la familia. Por ejemplo, Texas Instruments ofrece Planning, Analysis, Code Generation y Database Generations en su línea de productos de INFORMATION ENGINEERING FACILITY, y Nastec

ofrece DESIGNAID, LIFE CICLE MANAGER y REQUIREMENTS MANAGEMENT SYSTEM en su línea de productos CASE 2000.

Algunos juegos de herramientas son productos de arquitecturas abiertas que pueden utilizarse con las herramientas ofrecidas por varios fabricantes. EXCELERATOR, de Index Technology, es una juego de herramientas de análisis que puede utilizarse con otras herramientas CASE, como TELON de Pansophic o APS de Sage. DESIGNAID de Nastec y TEAMWORK de Cadre también son de arquitectura abierta.

Muchos productos de arquitectura abierta no están sujetos a un **hardware** específico, sistema operativo, lenguaje de programación o metodología de desarrollo. Mientras que los juego de herramientas ANALYST/DESIGNER de Yourdon, Inc soporta solamente la metodología de diseño estructurado de Yourdon, y el juego de herramientas DESIGN MACHINE de Ken Orr & Associates soporta solamente la metodología de desarrollo de sistemas de datos estructurados, otros juego de herramientas, como DEVELOPER de Asyst, PROKIT*ANALYST de McDonnell- Douglas DEFT de Disus, TEKCASE DESIGN de Textronix y PROMOD de Promod, Inc. son juego de herramientas más genéricos que soportan múltiples metodologías estructuradas.

IDMS/ARCHITECH de Cullinet es un juego de herramientas de diseño de bases de datos principalmente de IDMS, pero AUTO-MATE PLUS de Learmont y Burchett y SCHEMAGEN de Chen&Associates son juego de herramientas de diseño de datos más genéricas.

SOFTWARE THROUGH PICTURES de Interactive Development Environments, PROMOD de Promod, Inc., STATEMATE de i-Logix, Inc. y TAGS de Teledyne que soportan el desarrollo en lenguaje ADA.

TEAMWORK/RT de Cadre y EXCELERATOR/RTS de Index Technology son juegos de herramientas de análisis que soportan el desarrollo de sistemas de tiempo real. DESIGNAID de Nastec y ANALYST/DESIGNER TOOLKIT de Yourdon, Inc. son juegos de herramientas que soportan el desarrollo de sistemas comerciales y de sistemas de tiempo real.

El cuadro 5.2 resume la clasificación de los juegos de herramienta CASE.

LOS JUEGOS DE HERRAMIENTAS DE ANALISIS

Los juegos de herramientas de análisis hacen más fácil seguir los principios de un buen análisis y diseño de sistemas al hacerse cargo automáticamente del pesado trabajo burocrático y de los innumerables detalles del sistema. El

Cuadro 5.2. La clasificación de las herramientas

- | | | | |
|---------------------------------------|--|---|---|
| 1. A nivel de fase | <input type="checkbox"/> Análisis | <input type="checkbox"/> Diseño | <input type="checkbox"/> Gestión del proyecto |
| | <input type="checkbox"/> Programación | <input type="checkbox"/> Mantenimiento | |
| 2. Parte de un línea de productos | <input type="checkbox"/> Parte de una familia de productos | | |
| | <input type="checkbox"/> Producto aislado | | |
| | <input type="checkbox"/> Metodología independiente | | |
| 3. Arquitectura abierta | <input type="checkbox"/> Interfaces front-end <input type="checkbox"/> Framework | | |
| | <input type="checkbox"/> Interfaces back-end | | |
| 4. Entorno hardware/Sistema operativo | <input type="checkbox"/> PC/Estación de trabajo <input type="checkbox"/> MS-DOS <input type="checkbox"/> MVS | | |
| | <input type="checkbox"/> Mini | <input type="checkbox"/> Unix | <input type="checkbox"/> VM |
| | <input type="checkbox"/> Ordenador principal | <input type="checkbox"/> Ada | <input type="checkbox"/> VAX/VMS |
| 5. Tipo de aplicación | <input type="checkbox"/> Tiempo real/embebido <input type="checkbox"/> Proceso con alto volumen de transacciones | | |
| | <input type="checkbox"/> En línea | | |
| 6. Tipo de trabajo | <input type="checkbox"/> Analista <input type="checkbox"/> Administrador del proyecto | | |
| | <input type="checkbox"/> Programador de Diseño | <input type="checkbox"/> Programador de mantenimiento | |
| | <input type="checkbox"/> Programador | | |
| 7. Metodología soportada | <input type="checkbox"/> Yourdon/Constantine <input type="checkbox"/> Merise | | |
| | <input type="checkbox"/> DeMarco | <input type="checkbox"/> Martin | |
| | <input type="checkbox"/> Gane-Searson | <input type="checkbox"/> Ward y Mellor | |
| | <input type="checkbox"/> Orr | <input type="checkbox"/> Hatley | |
| | <input type="checkbox"/> Jackson | <input type="checkbox"/> Otros | |
| | <input type="checkbox"/> Bachman | <input type="checkbox"/> DOD Standard 2167 | |
| | <input type="checkbox"/> Chen | | |

objetivo de un juego de herramientas de análisis es automatizar la creación de las especificaciones de un sistema describiendo las necesidades del sistema. La salida producida por un juego de herramientas es una especificación del sistema consistente en lo siguiente:

- Las definiciones de las pantallas e informes.
- Las definiciones estructuradas de los datos.
- Las descripciones de los componentes funcionales.

Un juego de herramientas de análisis tiene cuatro componentes básicos:

- Las herramientas de diagramación estructurada.
- Las herramientas de prototipos.
- El depósito.
- El comprobador de especificaciones.

Herramientas de diagramación

El primer componente, las herramientas de diagramación, es un conjunto de herramientas informatizado para el dibujo, manipulación y almacenamiento de los diagramas estructurados, como los de flujo de datos, de dependencia y de entidad/relación. A menudo, estas herramientas residen en un ordenador personal o en una estación de trabajo que soportan funciones gráficas. Las posibilidades de diagramación de las herramientas CASE se trataron en el capítulo 2, en la sección “Las capacidades gráficas”.

Como mínimo, un juego de herramientas de análisis debe proporcionar herramientas de diagramación que dibujen, actualicen y almacenen al menos una versión de cada uno de estos tipos de diagramas estructurados:

- Diagramas de flujo de datos.
- Diagramas de entidad/relación.

Juegos de herramientas de análisis de sistemas de tiempo real

Para representar las especificaciones de sistemas de tiempo real, este conjunto de diagramas básicos debe ampliarse. Deben incluirse los diagramas que muestran las relaciones de secuencia y de tiempo entre los procesos del sistema y los efectos de los sucesos externos. Los diagramas de flujo de control y los diagramas de transición de estado se suelen utilizar para modelar los sistemas de tiempo real y por tanto deben añadirse al conjunto de diagramas de un juego de herramientas de análisis que soporte tareas, de análisis y diseño de sistemas de tiempo real.

Las herramientas de prototipos

Las herramientas para prototipos son el segundo componente de los juegos de herramientas de análisis. Se emplean para ayudar a determinar las necesidades del sistema y responder a las cuestiones sobre la capacidad de rendimiento del sistema antes de construirlo.

Un juego de herramientas de análisis debe proporcionar dos tipos de herramientas para prototipos:

1. Generador de interfaz de usuario (generadores de pantallas y de informes, constructor de menús).
2. Lenguajes de especificación ejecutables.

La capacidad de hacer prototipos de las herramientas CASE se trata en el capítulo 3, en la sección “El prototipo”.

El depósito

El tercer componente de un juego de herramientas de análisis es el depósito. Toda la información de las especificaciones del sistema se almacena en el depósito. La capacidad de un depósito CASE se discutió en el capítulo 2, en la sección “El depósito CASE de la información”.

El comprobador de especificaciones

La comprobación automática de las especificaciones del sistema comienza tan pronto como el analista empieza a dibujar un diagrama estructurado o a introducir información en el depósito. El diagrama estructurado se comprueba de acuerdo con las reglas sintácticas de la diagramación. También se comprueba su integridad y consistencia con respecto a las especificaciones de otros sistemas. La comprobación automática proporcionada por las herramientas CASE se trató en el capítulo 2, en la sección “La comprobación de errores”.

La ampliación a la fase del diseño

Algunos juegos de herramientas de análisis se amplían para soportar también la fase de diseño del ciclo de vida del **software**. Para este tipo del juego de herramientas, la prestación de la diagramación se amplía para incluir las posibilidades de diagramación y comprobación de:

- Diagramas estructurados jerárquicamente en árbol.
- Diagramas lógicos de procedimiento.

El cuadro 5.3 resume la descripción de los juegos de herramientas de análisis.

Cuadro 5.3. Los juegos de herramientas de análisis

Objetivo: definir los requisitos y las propiedades que el sistema han de tener para satisfacer esos requisitos.

Salida: las especificaciones de los componentes e interfaces del sistema que conectan estos componentes.

- Definiciones de pantallas e informes.
- Estructura de los datos.
- Ficheros y bases de datos.
- Componentes funcionales.

*Cuadro 5.3. Los juegos de herramientas de análisis (continuación)***Utilidades**

- Herramientas de diagramación
 - Diagramas de flujo de datos.
 - Diagramas de entidad/relación.
- Herramientas para prototipos.
 - Generadores de pantallas e informes.
 - Lenguaje de especificaciones ejecutables.
- Depósito de información con informe de capacidad.
- Posibilidad de importación/exportación entre el depósito y los diccionarios externos.
- Analizador y comprobador de las especificaciones.

Ampliaciones en los sistemas de tiempo real

- Diagramas de flujo de control.
- Diagramas de estado-finito.
- Tablas de decisión.

Ampliaciones de diseño de sistema

- Diagrama jerárquico.
- Lógica de procedimiento.

Objetivo de diseño: crear un diseño del sistema de **software** que se va a construir para satisfacer las especificaciones de los requisitos.

Salida del diseño: una estructura de la arquitectura del sistema y un esquema detallado del diseño del programa que incluya los algoritmos y estructuras de los datos del programa.

JUEGOS DE HERRAMIENTAS PARA EL DISEÑO DE LOS DATOS

Los juegos de herramientas CASE para el diseño de los datos soportan el diseño lógico y físico de los ficheros y de las bases de datos. Soportan el modelado lógico de los datos, la conversión automática de los modelos de datos a tercera forma normal, la generación automática de esquemas de bases de datos para un sistema gestor en particular y la generación automática de descripción de ficheros a nivel de programa en código.

Entre los juegos de herramientas de diseño de datos están AUTOMATE PLUS de Learmonth & Burchett, SQL*DESIGN DICTIONARY de Oracle, IDMS/ARCHITECH de Cullinet y el juego de herramientas CHEN de Chen & Associates.

JUEGOS DE HERRAMIENTAS DE PROGRAMACION

El cuadro 5.4 es una lista de herramientas incluidas en un juego de herramientas CASE de programación para soportar la implantación de un programa.

Cuadro 5.4. Juego de herramientas de programación

Propósito: obtener un programa documentado y probado que satisfaga todas las especificaciones de los requisitos del programa.

Herramientas:

- Herramientas de diagramación estructurada de árbol jerárquico con comprobador de sintaxis y de consistencia.
- Diagramación de la lógica del procedimiento y editor en línea.
- Depósito CASE con gestor de información.
- Generador de código.
- Analizador de código.
- Manipulador de código.

Cuadro 5.4. Juego de herramientas de programación (continuación)

- Generador de datos de prueba.
- Analizador de la cobertura de prueba.
- Comparación de fichas.
- Analizador de correlaciones.
- Control de rendimiento.
- Generador y verificador del entorno de ejecución.
- Simulador del entorno de salida/destino.

Muchas de estas herramientas ya son familiares y están muy extendidas entre los programadores. La diferencia está en que las herramientas de los juegos de herramientas están integradas para ser compatibles entre sí. En otras palabras, han sido diseñadas o particularizadas para compartir interfaces comunes y compartidas y ser llamadas, utilizadas y alimentadas recíprocamente. Los entornos de programación UNIX e INTERLISP son dos buenos ejemplos de los primeros juegos de herramientas de programación.

Los generadores de código

Una herramienta muy importante en los juegos de herramientas CASE de programación es el generador de código. Este posibilita la producción automática del código desde el diseño de programación. Los generadores de código ofrecen varias ventajas:

- Incremento de la productividad sobre la escritura manual del programa en código.
- Incrementa la fiabilidad del programa porque el código se produce automáticamente.
- Los prototipos se soportan mediante generadores de pantallas de informes.

- Generación de código compilado y estructurado, como COBOL, PL/1, Fortran, C o Ada.
- Reducción de los costes del mantenimiento, pues el código está mantenido por el generador.

Generadores de código COBOL

El cuadro 5.5 resume los beneficios ofrecidos por generadores de código COBOL tan extendidos como TELON de Pansophic, APS de Sage Software, PACBASE de CGI Systems, TRANSFORM de Transform Logic y GAMMA de KnowledgeWare. Se procesan en entornos IBM DOS y MVS, y soportan importantes métodos de acceso a las bases de datos como IMS, CICS, VSAM y DB/2. Ofrecen generadores de pantalla y de informes, prototipos, manejo lógico personalizado y generación de código, de bases de datos y de documentación. El COBOL Code Generator de Digital Equipment Corporation y el Netron/CAP son ejemplos de generadores de código COBOL que se procesan en el sistema DEC VAX.

En general, el generador de código COBOL soporta el diseño de programa a través de las fases de mantenimiento del ciclo de vida del **software**. Tiene tres posibilidades:

1. Gestión de la especificación del programa y diseño de la información.
2. Generación de código y documentación del programa.
3. Soporte del mantenimiento del programa.

El proceso típico de desarrollo de un programa con la utilización de un generador de código COBOL para la construcción de una aplicación comercial es como sigue. Primero el profesional de desarrollo crea las pantallas y los informes requeridos por la aplicación. El programador almacena en el diccionario del generador la información sobre el formato del informe y de los datos que aparezcan en el informe. De modo similar, el programador almacena en el diccionario la información sobre los literales, los datos y la ordenación de las pantallas. Tal información se utilizará para generar el código fuente y para construir modelos prototipo del programa.

Cuadro 5.5. Beneficios del generador de código en COBOL

- Soporte del diseño y de la programación detallada de las aplicaciones del cliente.
- Método seguro para alcanzar un aumento de la productividad.
- Reemplazo total de las funciones escritas manualmente en COBOL.
- Compatibilidad con los sistemas ya existentes.
- Adaptación a los procedimientos y estándares establecidos.
- Instrucción mínima.
- Código independiente de la herramienta.

Seguidamente, el programador define el contenido de cada pantalla y/o informe con más detalle. Por ejemplo, cada elemento de la pantalla se identifica como entrada o salida. Se especifica la edición, validación y formato interno de cada dato elemental de las pantallas e informes. Después se especifica toda la lógica adicional del programa. La lógica del cliente se añade a través del empleo de subrutinas, macros, dispositivos de alto nivel proporcionados por el generador. Cuando toda la información necesaria relativa al programa está completa, el generador ya puede producir código fuente, definiciones de la base de datos, sentencias de JCL (Job Control Language) y documentación.

JUEGOS DE HERRAMIENTAS DE MANTENIMIENTO

El mantenimiento domina normalmente el ciclo de vida del **software**. Más de treinta mil millones de dólares se gastan anualmente en el mantenimiento del **software** en el mundo [1]. Se ha estimado que hay más de cien mil millones de líneas de código que han de mantenerse. De ellas, más de setenta mil millones son de código COBOL [2]. En muchas organizaciones MIS

(Management System Information) se gasta más tiempo en el mantenimiento que en crear otros sistemas nuevos [3]. En una investigación reciente realizada sobre mil organizaciones MSI se citó más frecuentemente (72%) el mantenimiento del **software** como la causa de mayor consumo de recursos [4].

El mantenimiento también concierne al mundo de los sistemas de tiempo real/embebidos. Por ejemplo, se ha estimado que el coste del desarrollo del **software** del avión de combate F-16 de las Fuerzas Aéreas de EE.UU. es de ochenta y cinco millones de dólares. Sin embargo el coste estimado de mantenimiento durante toda la vida del software se calcula en doscientos cincuenta millones de dólares [5]. Según un ex jefe del Pentágono: "Si algo nos mata antes que los rusos, será nuestro **software** [6].

Las herramientas para analizar, documentar, ingeniería, reingeniería y de conversión de los sistemas existentes son una necesidad imperiosa. El cuadro 5.6 muestra los tipos de herramientas que se incluyen en un juego de herramientas CASE de mantenimiento.

Como ejemplo, en EE.UU. la Administración General de Servicios del Centro de Desarrollo Federal de Software ha construido un juego de herramientas CASE de mantenimiento, llamado **Programmers Workbench** (PWB), para ser utilizado por las agencias federales y contratistas privados utilizando fondos federales.

Cuadro 5.6. Juegos de herramientas de mantenimiento

Propósito: corregir, cambiar, mejorar, analizar, validar, convertir y reespecificar los sistemas existentes.

Utilidades

• **Herramientas para la evaluación del sistema.**

- Utilidad de gestión del programa para mantener la auditoría de seguimiento de todos los documentos del sistema, incluyendo código, especificaciones, JCL, casos de pruebas, etc. del programa.
- Estimador de mantenimiento para predecir los esfuerzos de mantenimiento, cambios y propensión a los errores.

Cuadro 5.6. Juegos de herramientas de mantenimiento (continuación)

- Control de la cobertura de las pruebas para preparar los datos de prueba y determinar en qué grado se ha probado el programa.
- Control de rendimientos.
- Comparación de ficheros para determinar las porciones de los ficheros que no concuerdan.

- **Herramientas para el análisis y la reingeniería**

- Analizadores estáticos del programa para producir esquemas lógicos y la información de dónde se usa y cómo y referencias cruzadas obtenida de la exploración de los códigos fuente existentes, y para valorar el impacto de un cambio en el programa, en la estructura de los datos o en el JCL.

Ejemplos:

- ¿Cuál es el orden secuencial de los procesos y qué enlaces hay entre ellos?
- ¿Cómo recorre un sistema un determinado campo de datos y con que alias?
- ¿Qué campos son compartidos por un registro en un fichero de cliente y un registro en un fichero de la transacción?
- ¿Puede un cambio a un campo *ciudad* en un fichero de transacciones causar un cambio en un campo *código postal* del fichero de clientes?
- ¿Hay algún código inalcanzable?
- Analizadores dinámicos del programa, como compiladores e intérpretes con utilidades interactivas de depuración de errores, validación rápida de la sintaxis, compiladores paso a paso, trazadores hacia adelante y hacia atrás de los caminos/variables/procedimientos del programa, suspensión de la ejecución y reorganización.

Cuadro 5.6. Juegos de herramientas de mantenimiento (continuación)

- Documentador para producir documentación actualizada a partir del código fuente existente.
- Reformador para renombrar y reformar el código fuente de un programa existente.
- Reestructurador para reformar, redocumentar y reemplazar el código fuente de un programa existente por un código bien estructurado.
- Editor de un programa rápido, en línea y completo, incluyendo la función de aumento que excluye las líneas de detalle en el listado de un programa con una estructura característica.
- Generador de código que permita el mantenimiento de las especificaciones del código generado.
- Herramientas de ingeniería de reversión que traduzcan un sistema de vuelta a sus especificaciones para investigación y análisis.
- **Herramientas de migración**
 - Traductor, convertidor de datos y manipulador para migrar a un nuevo entorno de operación y **hardware**; proporcione la capacidad de edición, hoqueo, extracción y reforma de los datos.

La PWB consta de once herramientas integradas en un banco de trabajo residente en un IBM/MVS/TSO para soportar el mantenimiento de programas en COBOL. Las herramientas incluidas en el PWB con:

- Integrador de herramientas, RAND DEVELOPMENT CENTER, de Rand Information Systems, Inc.

- Monitor de cobertura de prueba, ANALYZER, de Aldon Computer Group.
- Sistema de traductor de COBOL, TRANSIT, de UCCEL Corporation.
- Creador de referencias cruzadas COBOL, DCD II, de Marble Computer, Inc.
- Reestructurador COBOL, RETROFIT, de Peat, Marwick, Mitchell & Company.
- Normalizador de nombres de datos, CSA, de Marble Computer, Inc.
- Reformador COBOL, HAWKEYE, de Blackhawk Data Corporation.
- Analizador de la documentación, PATHVU, de Peat, Marwick, Mitchell & Company.
- Comparador de ficheros y fuentes, COMPAREX, de Sterling Software Marketing.
- Analizador de código fuente COBOL, VIA/INSIGHT, de ViaSoft, Inc.

La reestructuración y la ingeniería inversa de los sistemas existentes

Hartford Insurance Company espera reducir sus costos de mantenimiento de **software** de un 25% a un 75% después de reestructurar automáticamente millones de líneas de código COBOL existentes en sus sistemas comerciales actuales [1].

En general, los remedios para los problemas del mantenimiento del **software** son:

- **Reformar:** introduciendo variables y procedimientos del sangrado y red denominación en el código fuente.

- **Reestructurar:** reestructuración jerárquica, para analizar y documentar el código fuente.
- **Reingeniería:** funciones de rediseño, rearquitectura y adición para los programas existentes.
- **Reemplazo:** reemplazo de los programas existentes con tecnologías más recientes o retirar los programas que ya no se utilizan.
- **Ingeniería inversa:** traducir un programa existente de vuelta a sus especificaciones de diseño, incluyendo su estructura de la arquitectura, estructura de los datos y los modelos de datos lógicos.

Afortunadamente muchos de estos remedios pueden aplicarse utilizando herramientas automáticas de mantenimiento. Algunos ejemplos de mantenimiento automático se exponen en el cuadro 5.7. Son algunas de las herramientas más potentes y útiles de los juegos de herramientas de mantenimiento.

Por ejemplo, las herramientas de reestructuración automática de programas pueden abaratar substancialmente la reescritura manual y proporcionar las siguientes ventajas:

- Reducción de los costos del futuro mantenimiento.
- Facilitar la reingeniería del programa.
- Imponen estándares de programación y documentación estructuradas.
- Generan la documentación del programa, incluyendo visiones gráficas de las estructuras de los datos y de los procedimientos de programa.
- Amplían la vida útil del programa.

Las herramientas de reestructuración de programas transforman el código fuente no estructurado en estructurado y documentado. El programa estructurado resultante es de ejecución equivalente al programa preestructurado en el sentido de que ambos realizan la misma secuencia de operaciones con los mismos datos. Por supuesto, los errores lógicos contenido

en el programa primitivo se transmitirán a la versión estructurada. Ahora, no obstante, será más fácil identificarlos y corregirlos.

El proceso de reestructuración consta de cinco pasos. Primero, el programa no estructurado se analiza automáticamente para revisar las deficiencias, códigos en desuso, lenguaje no estándar, etc. Segundo, el programador revisa el análisis realizado en el primer paso y hace las correcciones necesarias para la reestructuración automática. Seguidamente, el código se reestructura y documenta automáticamente. Cuarto, el código fuente es recompilado con la ayuda de un optimizador de compilación para mejorar su eficiencia. Finalmente, se revalida el nuevo programa estructurado.

Cuadro 5.7. Ejemplos de herramientas de mantenimiento

Documentación/Reestructuración

- PATHVU/STRUCTURED RETROFIT de Peat, Marwick, Mitchell & Company.
- SCANCOBOL/SUPERSTRUCTURE de Group Operations, Inc.
- INSPECTOR/RECORDER de Language Technology.
- COBOL/SF de IBM.
- SYSDOC de Syncsort, Inc.
- VERIFY de OnLine Software International.

Herramientas de análisis/Ingeniería inversa de programas

- PM/SS de Adpac.
- MAESTRO de SoftLab.
- DIALOGIC WORKBENCH de Dialogic Systems.
- PSL/PSA de MetaSystems.
- VIA/INSIGHT de ViaSoft.
- FASTBOIL de Tasco.
- COMPAREX de Sterling Software.

JUEGOS DE HERRAMIENTAS DE GESTION DE PROYECTOS

La buena gestión del proyecto es uno de los más importantes factores para incrementar la productividad en el desarrollo y mantenimiento del **software**. Las herramientas para la gestión automática del proyecto ayudan al gestor del proyecto a mejorar el control y la información de los proyectos de **software**. Para ser más efectiva, la herramienta debe formar parte del sistema CASE con acceso al depósito CASE donde está almacenada toda la información del sistema. El depósito CASE es la fuente central actualizada de toda la información técnica del sistema y deberá servir como fuente actualizada para el estado, estimación, presupuesto y seguridad un proyecto para presupuestar, estimar y garantizar la calidad de la información de todo el proyecto.

El cuadro 5.8 lista los tipos de herramientas de gestión de proyectos que deberían incluirse en un juego de herramientas CASE de gestión de proyectos. LIFE CYCLE MANAGER, de Nastec; PROJET WORKBENCH, de Applied Business Technology; MULTI/CAM, de AGS Management Systems; MAESTRO PLUS, de SoftLab, Inc., y LIFE CYCLE PROJET MANAGER, de American Management Systems son algunos de los juegos de herramientas CASE de gestión de proyectos.

Cuadro 5.8. Juegos de herramientas de gestión de proyectos

- Procesador de textos.
- Interfaz con el correo electrónico.
- Hojas de cálculo.
- Formularios de gestión del proyecto.
- Configuración de la gestión para cambios, versiones y control de acceso.
- Planificación de proyectos.
- Calendario y sistema de asignación de tareas.
- Herramientas de horarios y planificación.
- Métricas de control de calidad.
- Depósito CASE con auditoría de seguimiento de toda la información del sistema.

FRAMEWORK

Una **framework** (armazón) es un tipo de juego de herramientas CASE que proporciona una infraestructura para acoplar, adaptar y gestionar herramientas de **software** individuales. Con una **framework**, las herramientas antiguas, nuevas y de diferentes proveedores que de otra manera serían incompatibles, se ensamblan en un entorno en el cual es fácil utilizar unas con otras. Por ejemplo, las herramientas CASE, las tradicionales de implantación de programas y las de gestión de bases de datos pueden integrarse dentro de la infraestructura de la **framework**. Las **frameworks** incrementan la portabilidad de las herramientas, permitiéndoles procesarse en distintas plataformas **hardware**. Las herramientas dentro de la infraestructura de la **framework** se presentan ante el usuario como un interfaz común, comparten interfaces de datos comunes y se procesan en el mismo entorno de **hardware**.

Cuadro 5.9. Características de los juegos de herramientas frameworks CASE

Propósito: crear una infraestructura diseñada para la integración, gestión y particularización de las herramientas.

- Soportan las siguientes áreas de las **toolkits/workbench** de diseño y desarrollo:
 - La estructura de menú.
 - La función de ayuda (help).
 - El formato de la documentación.
 - Los símbolos y reglas sintácticas de la diagramación.
 - Las comunicaciones entre la estación de trabajo y el ordenador principal.
 - La gestión de la configuración del sistema.
 - Los patrones para los formularios.
- La **framework** deberá ser utilizable en una amplia gama de entornos de desarrollo.
- La **framework** debe construirse con una arquitectura abierta para que puedan integrarse las herramientas nuevas y las de diferente origen.

Entre los juego de herramientas **framework CASE** están **SOFTWARE BACKPLANE** de Atherton Technology, **RULE TOOL** de Cadware Group, **RAND DEVELOPMENT CENTER** de Rand Information Systems, **LIFE CYCLE PRODUCTIVITY SYSTEM** de American Management System y **CUSTOMIZER** de Index Technology.

El cuadro 5.9 resume las características de un juego de herramientas **framework CASE**.

LOS BANCOS DE TRABAJO CASE

Un **banco de trabajo CASE (workbench)** es un ensamblaje de herramientas integradas cuya función es automatizar el desarrollo y mantenimiento de los sistemas **software**, además de la gestión de las actividades de un proyecto de **software**. Un banco de trabajo CASE contiene las herramientas que integran las fases básicas del ciclo de vida del **software**, a saber: análisis, diseño e implantación. Las herramientas se integran de forma que la salida de una fase del ciclo de vida pase directa y automáticamente a la fase siguiente. El producto final de un banco de trabajo CASE es un sistema de **software** ejecutable acompañado de su documentación. Las características de un banco de trabajo CASE se tratan detalladamente en los capítulos 2 y 3.

El **INFORMATION ENGINEERING FACILITY (IEF)** de Texas Instruments es un ejemplo de banco de trabajo CASE. El IEF soporta la metodología de la ingeniería de la información. Transforma automáticamente la información representada por diagramas creados en un estadio de la ingeniería de la información en la información y diagramas utilizados en el estadio siguiente.

El **INFORMATION ENGINEERING WORKBENCH (IEW)** de KnowledgeWare es otro banco de trabajo CASE que soporta ingeniería de la información y que automáticamente pasa información de una fase a la siguiente. El IEW transforma automáticamente las actividades del negocio representadas en diagrama de desarrollo en el estadio de planificación estratégica en un conjunto por niveles de diagramas de flujo de datos que se utilizarán en el estadio de análisis del área del negocio de la ingeniería de la información.

Las herramientas TEKCASE de Tektronix soportan el desarrollo de sistemas de tiempo real. TEKCASE DESIGNER transforma el modelo del sistema de tiempo real obtenido con el TEKCASE ANALYST/RT en un diseño estructurado alternativo preservando los datos y la información de control para establecer el seguimiento de los requerimientos desde la fase análisis a la de diseño. Después, la herramienta TEKTRONIX PICKBOSS puede utilizarse para determinar el mejor diseño por medio de la evaluación automática de la complejidad de cada diseño alternativo. PICKBOSS recomienda un diseño que pueda representarse por el diagrama de estructura menos complejo.

PROMOD de Promod, Inc. también se emplea en el desarrollo de sistemas de tiempo real. El PROMOD/MODULAR DESIGN TRANSFORMATION PROCESS convierte el modelo de flujo de datos creado durante la fase de análisis en un control jerárquico de subsistemas, módulos y funciones para empezar la fase de diseño del programa.

Otros ejemplos de **bancos de trabajo CASE** son CORVISION de Cortex, EXSYS de Exsys, Inc. y AUTOCODE de Integrated Systems.

Los diferentes **bancos de trabajo CASE** se procesan en distintos entornos de **hardware** y diferentes sistemas operativos, soportando diferentes metodologías de desarrollo y soportan el desarrollo de distintos tipos de sistemas de salida. El cuadro 5.10 indica cómo pueden clasificarse los **bancos de trabajo CASE**.

Cuadro 5.10. Clasificación de los bancos de trabajo (workbench)

1. Entorno hardware/Sistema operativo		
___ Ordenador principal	___ MVS	___ MS-DOS
___ Mini	___ VM	___ Unix
___ PC/estación de trabajo	___ VAX/VMS	___ Ada
2. Arquitectura abierta		
___ Interfaces alternativas front-end		
___ Interfaces alternativas back-end		

*Cuadro 5.10. Clasificación de los bancos de trabajo (workbench)
(continuación)*

3. Depósito CASE		
<input type="checkbox"/> Centralizado	<input type="checkbox"/> Local	
<input type="checkbox"/> Diccionario	<input type="checkbox"/> SGBD	<input type="checkbox"/> Base de conocimientos
<input type="checkbox"/> Por interfaz	<input type="checkbox"/> Totalmente integrado	
4. Por tipo de aplicación		
<input type="checkbox"/> Tiempo real	<input type="checkbox"/> COBOL	<input type="checkbox"/> PL/I
<input type="checkbox"/> En línea	<input type="checkbox"/> 4GL	<input type="checkbox"/> Ada
<input type="checkbox"/> Gran volumen de transacciones	<input type="checkbox"/> Fortran	<input type="checkbox"/> Otros
5. Metodología soportada		
<input type="checkbox"/> Ingeniería de software		
<input type="checkbox"/> Ingeniería de la información		
<input type="checkbox"/> Metodología independiente		
<input type="checkbox"/> Metodología estructurada específica		
<input type="checkbox"/> Estructura formal		

COMPAÑEROS DE METODOLOGIAS CASE

Un **compañero de metodología CASE** puede ser tanto un juego de herramientas como un **banco de trabajo** que estructura el proceso de desarrollo del **software** de acuerdo con las fases y reglas de una metodología estructurada en particular. La información sobre la metodología se incorpora en el compañero de metodología por medio de los paneles de ayuda, del menú de opciones, de la introducción de funciones y de los controles de calidad. Por ejemplo, las pantallas de ayuda y los menús informan al analista de cuál es el paso siguiente, qué entradas son necesarias para este paso y qué salidas van a producirse. La introducción de funciones no permite al analista ir al paso siguiente en la metodología hasta que el compañero de metodología juzga que la tarea en curso está completa y certifica que las salidas de la tarea son correctas y cumplen con los estándares en curso.

La incorporación de la información de la metodología en las herramientas CASE amplía el uso del sistema de ayuda y de validación. Pero minimiza la información que debe introducir el analista para realizar la tarea, ya que la información puede transformarse automáticamente y pasarse a una tarea posterior en la forma exacta requerida por las tareas basadas en las reglas de la metodología.

Algunos ejemplos de compañeros de metodología se listan en el cuadro 5.11 y su clasificación se relaciona en el cuadro 5.12. El conductor de metodología, que es un sistema experto, se trata en el capítulo 13.

Cuadro 5.11. Ejemplos de compañeros de metodología CASE

Herramienta	Tipo de herramienta	Metodología soportada
DATA RESOURCES LEVERAGE WITH JANUS de D. Appleton Company	Herramienta de diseño	Metodología de desarrollo de prototipo
FOUNDATION de Arthur Andersen	CASE workbench	METHOD-1
DESIGN MACHINE de Ken Orr & Associates	Herramienta de análisis/diseño	Desarrollo estructurado de sistemas
SQL*DESIGN DICTIONARY de Oracle	Herramienta de diseño	Método de desarrollo SQL
AUTO-MATE PLUS de Learmonth & Burchett	Herramienta de diseño	Método de desarrollo estructurado L&B
INFORMATION ENGINEERING FACILITY de Texas Instruments	CASE workbench	Metodología de ingeniería de la información
TAGS de Teledyne Brown	CASE workbench	TAGS/IORL
ANALYST/DESIGNER TOOLKIT de Yourdon, Inc.	Herramienta de análisis/diseño	Metodología de diseño estructurado de Yourdon

Cuadro 5.12. Clasificación de los compañeros de metodología CASE

1. Metodología soportada		
<input type="checkbox"/> Ingeniería de software		
<input type="checkbox"/> Ingeniería de la información		
<input type="checkbox"/> Técnica estructurada	<input type="checkbox"/> Específica	<input type="text"/>
<input type="checkbox"/> Metodología propia	<input type="text"/>	
<input type="checkbox"/> Formalmente estructurada		
2. Entorno hardware/Sistema operativo		
<input type="checkbox"/> Ordenador principal	<input type="checkbox"/> MVS	<input type="checkbox"/> MS-DOS
<input type="checkbox"/> Mini	<input type="checkbox"/> VM	<input type="checkbox"/> Unix
<input type="checkbox"/> PC/estación de trabajo	<input type="checkbox"/> VAX/VMS	<input type="checkbox"/> Ada
3. Nivel de soporte		
<input type="checkbox"/> Producción de la documentación		
<input type="checkbox"/> Guía del sistema		
<input type="checkbox"/> Conductor inteligente de metodología		
4. Tipo de aplicación		
<input type="checkbox"/> Tiempo real	<input type="checkbox"/> COBOL	<input type="checkbox"/> PL/1
<input type="checkbox"/> En línea	<input type="checkbox"/> 4GL	<input type="checkbox"/> Ada
<input type="checkbox"/> Gran volumen de transacciones	<input type="checkbox"/> Fortran	<input type="checkbox"/> Otros

BIBLIOGRAFIA

1. Girish Parikh, "Restructuring Your COBOL Programs", *Computer World Focus*, 19 febrero, 1986, págs. 39-42.
2. *ViaSoft News Release*. Phoenix, Arizona: ViaSoft, 2 diciembre, 1986.
3. Stan Kolodziej, "Gaining Control of Maintenance", *Computer World Focus*, 19 febrero, 1986, págs. 31-36.
4. "CASE 1987 Survey", *Software News*. Westborough, Massachusetts: Sentry Publishing Company, 1987.

5. William Suydam, "CASE Makes Strides toward Automated Software Development", *Computer Design*, 1 enero, 1987, págs. 49-70.
6. *Atherton Technology White Paper*. Sunnyvale, California: Atherton Technology, agosto 1987.

PARTE 3

LA UTILIZACION DE LA CASE

CASOS DE ESTUDIO DE LA CASE

EL INCREMENTO DE LA PRODUCTIVIDAD DEL SOFTWARE

La clave de la productividad es la automatización. Esto se aplica lo mismo al desarrollo de **software** que a cualquier otro tipo de producto fabricado. La ayuda por ordenador a la ingeniería del **software** (CASE) es la automatización de muchas de las tareas del ciclo de vida del **software**. La tecnología CASE es el punto central del incremento de la productividad del **software** desde finales de los años ochenta.

La prueba de que la CASE funciona ya ha llegado para muchas compañías que han conseguido un incremento substancial de la productividad. Aunque son muy diferentes, esas empresas comparten el éxito obtenido con las herramientas CASE en el incremento de la productividad. En cada caso particular, han elegido herramientas que contribuyen a automatizar las metodologías estructuradas.

Las metodologías estructuradas, muchas de las cuales se desarrollaron en los años setenta, proporcionan un método probado para producir sistemas de **software** de alta calidad y fiables. El fallo de la metodología estructurada ha sido, sin embargo, el gran esfuerzo manual y el consumo de tiempo que requieren. Las herramientas automáticas resuelven este problema encargándose de gran parte de los pesados detalles y el papeleo de

la aplicación de estas metodologías. Esto libera al analista para concentrarse en la parte creativa del proceso de desarrollo del **software**. El resultado es un incremento significativo de la capacidad individual de cada profesional del desarrollo para producir más y mejores sistemas de **software**.

La idea básica de la CASE es soportar cada fase del ciclo de vida con un conjunto de herramientas que economicen el trabajo. Algunas herramientas se concentran en el soporte de las primeras fases del ciclo de vida. Ofrecen asistencia automática en forma de diagramas dibujados automáticamente, generadores de pantallas y comprobación de la corrección. Otras se enfocan a las fases de implantación del ciclo de vida, incluyendo los generadores automáticos de código y de casos de prueba.

En algunos casos se utilizan estas herramientas en combinación con lenguajes de tercera y cuarta generación. En otros casos, los reemplazan permitiendo el desarrollo de las especificaciones de alto nivel del programa con las que puede generarse el código.

En este capítulo veremos cómo algunas empresas han incrementado la productividad del **software** con el empleo de tres herramientas CASE diferentes: EXCELERATOR de Index Technology Corporation, INFORMATION ENGINEERING WORKBENCH de KnowledgeWare y APPLICATION FACTORY de Cortex. Las empresas que han utilizado estas herramientas informaron del incremento de la productividad en varias fases del ciclo de vida, desde un factor dos y hasta un factor veinte (ver figura 6.1).

Productividad de la CASE			
	Exclerator	IE workbench	Factory
Tipo	Herramientas de Análisis	workbench CASE de especificación	workbench CASE generador
Utilización principal	Sistemas de análisis y de documentación	Sistemas de análisis y de diseño	Sistemas de desarrollo y de mantenimiento
Incremento de la productividad	Incremento entre 2 y 10 veces	Incremento entre 2 y 4 veces	Incremento entre 2 y 20 veces

Figura 6.1. Incrementos en la productividad registrados por los diversos usuarios de herramientas CASE

EXPERIENCIAS CON EXCELERATOR

EXCELERATOR de Index Technology Corporation es una herramienta dirigida al diseño y la documentación de sistemas de información y de tiempo real. Es una herramienta basada en un ordenador personal para ser utilizada por un analista o diseñador profesional de sistemas.

EXCELERATOR tiene cuatro dispositivos básicos:

1. Una herramienta de diagramación automática para dibujar diagramas estructurados, como flujo de datos, diagrama de estructura, flujo de control y de estado finito.
2. Generadores de pantallas y de informes para la especificación y prototipos del interfaz de usuario en un sistema de información.
3. Un depósito integrado para almacenamiento y referencias cruzadas de la información del análisis y del diseño de todos los sistemas.
4. Una herramienta de análisis automático de comprobación e información de los errores de sintaxis, de la integridad y de la consistencia de los diagramas estructurados.

EXCELERATOR puede describirse mejor como un juego de herramientas **de análisis**, porque proporciona un conjunto integrado de herramientas para automatizar las tareas de análisis y diseño de los sistemas.

EXCELERATOR proporciona a sus usuarios tres beneficios muy importantes. Primero, EXCELERATOR hace que las metodologías estructuradas, como el análisis estructurado de DeMarco y el diseño estructurado de Yourdon, sean útiles en la práctica. El dibujado automático de diagramas estructurados, la captura de la información de diseño en el depósito automatizado, el seguimiento y mantenimiento automatizados de todos los detalles del diseño permiten al analista concentrarse en los problemas del diseño en vez de en el papeleo.

Segundo, EXCELERATOR incrementa la calidad general del diseño del sistema. Manteniendo toda la información del diseño en el ordenador y comprobando automáticamente la consistencia y la integridad se asegura la alta calidad de la información del diseño.

Tercero, EXCELERATOR acelera el proceso de diseño. Con la asistencia automática de EXCELERATOR pueden realizarse modificaciones importantes en el diseño en segundos, el trabajo puede comprobarse al mismo tiempo que se desarrolla el diseño y los diseños pueden recuperarse para reutilizarlos desde el depósito de EXCELERATOR.

En un reciente estudio de doce organizaciones que utilizaban EXCELERATOR en los trabajos de análisis y diseño de sistemas, se informó que la productividad se había incrementado entre dos y diez veces. Las doce organizaciones tenían un entorno de desarrollo similar:

- Ordenador principal IBM utilizando VM y/o MVS.
- Con sistema de gestión de base de datos (IMS y DB2).
- Empleo de herramientas de cuarta generación, como FOCUS, DATA DICTIONARY, DATA MANAGER.
- Empleo del COBOL en las aplicaciones soportadas en grandes ordenadores centrales.

En todas las organizaciones se utilizó EXCELERATOR principalmente en nuevos sistemas de desarrollo como una herramienta para automatizar las metodologías estructuradas utilizadas corrientemente. El principal usuario fue el analista de sistemas y el uso principal fue en la especificación de los requisitos del sistema. Los usuarios de EXCELERATOR declararon una media entre ocho y treinta horas de instrucción para llegar a la utilización competente de la herramienta. Ya conocían los métodos estructurados.

Experiencia del gobierno de EE.UU. con EXCELERATOR

Uno de los doce casos estudiados fue una organización del gobierno de EE.UU., que informó que con la ayuda de EXCELERATOR pudo reducir el tiempo del ciclo de desarrollo a un punto en que el sistema podía ser implantado antes de quedar obsoleto. Los analistas completaron las especificaciones del sistema en cuatro meses, en lugar de los dos años estimados en caso de hacerse manualmente.

Experiencia de ARCO con EXCELERATOR

Otro caso estudiado fue el de ARCO. En ARCO, la mayoría del desarrollo del **software** para aplicaciones en ordenadores centrales se realiza en lenguaje COBOL y con la metodología de desarrollo estructurado SDM. ARCO seleccionó EXCELERATOR porque buscaba una herramienta que automatizara el proceso de modelado de los datos. Se necesitaba un método que incrementara la productividad de los analistas de desarrollo, ya que debido a una reestructuración de la compañía, el equipo de desarrollo había sido reducido considerablemente. ARCO comenzó a utilizar EXCELERATOR en 1984. A mediados del año 1986 ARCO tenía instaladas 23 copias de EXCELERATOR.

ARCO informó que utilizando EXCELERATOR los analistas de sistemas preparaban los modelos de los datos lógicos de los sistemas de información con una ganancia de productividad sobre los métodos manuales de diez a uno.

ARCO mejoró las prestaciones de importación/exportación de EXCELERATOR escribiendo su propio software, que permite la importación/exportación de la información desde el diccionario de EXCELERATOR hacia Data Dictionary y viceversa. Desde el punto de vista de la administración, esto significa que los datos pueden estar mejor controlados. Para ARCO esto último es tan importante como realizar más rápidamente las tareas de **software**.

La experiencia de Touche Ross con EXCELERATOR

Touche Ross, uno de los primeros usuarios de EXCELERATOR, fue otro caso de la investigación. Eligieron EXCELERATOR porque soporta la metodología estructurada de Yourdon. Touche Ross ha utilizado EXCELERATOR para diseñar aplicaciones de muchos tipos diferentes, como entrada de pedidos, control de inventario, derecho penal y sistemas de subastas. El primer usuario de EXCELERATOR en Touche Ross es un analista de sistemas especializado en las técnicas estructuradas de Yourdon. En esta organización hay cuatro ordenadores personales y en cada uno de ellos una copia de EXCELERATOR. Se asigna un ordenador personal a un proyecto donde una persona se responsabiliza de la entrada y actualización de toda la información del sistema en el diccionario de EXCELERATOR.

Informaron que EXCELERATOR había cambiado drásticamente la forma de hacer el análisis de los sistemas. Utilizan el EXCELERATOR desde el nivel de diseño de sistemas. La calidad de la especificación del sistema mejora notablemente, la participación del usuario se hace práctica y con el dibujo automático de diagramas los cambios son fáciles de hacer. Aunque el uso principal de EXCELERATOR es el de las especificaciones de las necesidades del sistema, Touche Ross está utilizando también EXCELERATOR de una forma que nada tiene que ver con el desarrollo de **software**. Han empleado EXCELERATOR para describir toda clase de sistemas (automáticos y manuales) para mejorar la comprensión y racionalización más que con la intención de automatizarlos. Como ejemplo, el presidente del tribunal de justicia del Condado de Milwaukee, Wisconsin, dijo que después de treinta años en el estrado, finalmente había comprendido el sistema de derecho penal, porque Touche Ross, con la ayuda de EXCELERATOR, lo describía con imágenes en lugar de palabras. El sistema se describió con 287 diagramas de flujo de datos después de un esfuerzo de ocho meses y medio en la definición de los requerimientos del sistema.

Touche Ross es consciente de que la mayor ventaja productiva de utilizar EXCELERATOR, es que es fácil la revisión de los requerimientos del sistema porque con EXCELERATOR se actualizan automáticamente los diagramas. Otra ventaja es la consistencia de las salidas que produce.

El consenso entre usuarios de EXCELERATOR se basa en que la clave del éxito en la productividad se debe a una buena gestión, a una comprensión sólida del análisis y diseño estructurados y a una herramienta de automatización potente.

EXPERIENCIAS DE PRODUCTIVIDAD CON APPLICATION FACTORY

APPLICATION FACTORY, de Cortex Corporation, es una herramienta de productividad para la generación de sistemas de información de multiusuarios en línea. El núcleo de APPLICATION FACTORY es su generador automático de código. APPLICATION FACTORY se procesa en los ordenadores DEC VAX.

APPLICATION FACTORY tiene cinco dispositivos básicos:

1. Generadores de pantallas y de informes para la especificación y prototipos del interfaz del usuario.
2. Un depósito para almacenar toda la información sobre el sistema.
3. Comprobación automática de la integridad y consistencia de las especificaciones del sistema.
4. Un generador de código capaz de generar automáticamente el 95% del código a partir de las especificaciones del programa.
5. Un generador automático de la documentación del programa.

Los analistas y programadores utilizan la APPLICATION FACTORY para el desarrollo y mantenimiento de los sistemas informáticos de tipo medio/alto. La APPLICATION FACTORY es apropiada para la construcción de sistemas de tipo medio/alto en un entorno de gran volumen de transacciones con interfaz con los dispositivos de tiempo real.

Los usuarios observaron tres ventajas importantes: Primero, los desarrollos eran más productivos con la APPLICATION FACTORY que con la programación en lenguajes de tercera generación como COBOL o Fortran. Los desarrollos también eran más productivos que programando con los lenguajes de cuarta generación. La APPLICATION FACTORY genera un código eficiente y potente para manejar aplicaciones grandes y pequeñas. A diferencia de los lenguajes de cuarta generación, la APPLICATION FACTORY se ha empleado para crear sistemas con 2.500 puntos de función manejando 15.000 transacciones diarias con 300 usuarios.

Segundo, la APPLICATION FACTORY hace de la construcción rápidas de prototipos iterativos una forma práctica de obtener sistemas que satisfagan las necesidades del usuario. Se construyen unos prototipos del sistema como medio para clasificar mejor los requerimientos del usuario. El sistema final evoluciona a partir de los prototipos. La APPLICATION FACTORY realiza este proceso fácil y rápidamente, con la generación automática de código a partir de las especificaciones del prototipo.

Tercero, la APPLICATION FACTORY simplifica considerablemente el mantenimiento de los programas porque se mantienen las especificaciones del programa y no el código. Cuando es necesario cambiar un sistema se modifica la especificación y el código y la documentación se regeneran

automáticamente a partir de la especificación del sistema. De esta forma, la documentación es un subproducto del desarrollo y el mantenimiento y se mantiene siempre actualizada.

Los casos de estudio de desarrollo de **software** con APPLICATION FACTORY mostraron un incremento promedio de la productividad trece veces superior que con COBOL. Los casos de estudio incluyeron el desarrollo de veintiséis sistemas de aplicaciones en veintidós localizaciones separadas. La medida del incremento de la productividad se realizó con puntos de función, comparando el esfuerzo **estimado** en COBOL con el esfuerzo **real** en FACTORY. El tamaño de los sistemas oscilaba entre 78 y 2418 puntos de función, lo que equivale aproximadamente de 8.300 a 253.000 líneas de programa en COBOL. Además, los casos de estudio revelaron que la productividad de FACTORY con respecto a COBOL aumenta a medida que aumentan el tamaño y la complejidad de la aplicación. La experiencia demuestra que el tamaño del proyecto y la experiencia del programador afectan significativamente a la productividad y que precisamente la experiencia del programador es lo que más influye.

Las experiencias de DuPont con FACTORY

En DuPont Textil Fibers Plant, el grupo de sistemas de información utiliza Fortran y COBOL para desarrollar los sistemas de aplicaciones. En el momento del estudio, se estaba en el proceso de conversión de los entornos DEC PDP-11 y ordenador principal IBM a un entorno DEC VAX. También se había desarrollado una metodología propia llamada RIPP (Rapid Iterative Production Prototyping) para el desarrollo de sistemas (ver figura 6.2). RIPP limita todos los desarrollos de sistemas a un máximo de noventa días, en los que se incluyen la especificación, prototipos e implantación del sistema. Sesenta de los noventa días se consumen en la fase de prototipos. Sus características son:

1. Automatizar el desarrollo del **software** cuanto sea posible.
2. Aumentar la calidad de los sistemas de información.
3. Aumentar la productividad durante todo el ciclo de vida.
4. Integrar la gestión de los datos y el desarrollo del **software**.

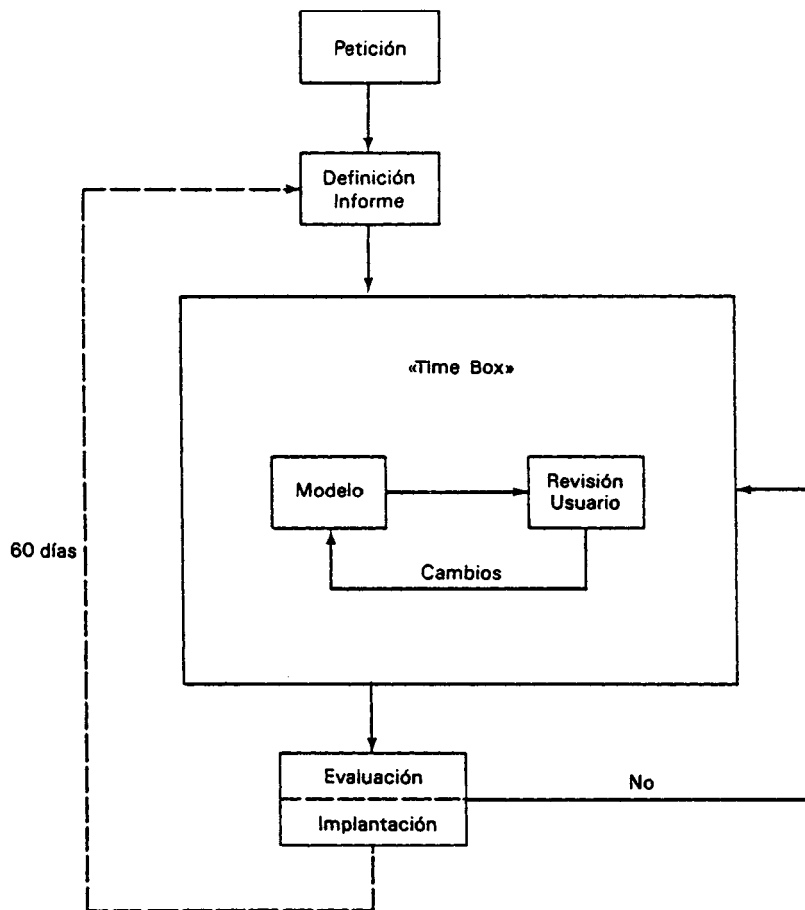


Figura 6.2. Metodología de proyectos a 90 días, RIPP de DuPont

El objetivo de DuPont es aumentar la productividad por diez durante todo el ciclo de vida. DuPont está literalmente inundado de proyectos de sistemas debido a una serie de cambios en la empresa y tiende a una mayor automatización en sus fábricas.

DuPont informó sobre el sorprendente ahorro en los costos cuando se utilizó APPLICATION FACTORY para el desarrollo de sistemas. En nueve proyectos de desarrollo ahorró casi dos millones de dólares al utilizar FACTORY, en lugar de lenguajes de tercera generación como COBOL y Fortran (ver figura 6.3). Con FACTORY, el número de días necesi-

Experiencias DuPont (1985-1986)

Aplicación	Núm. de pantallas	Núm. de informes	Núm. de ficheros	Costo (millones de dólares)		
				Estimado	Real	Ahorro
Fundas BCP	3	5	11	\$ 168.	\$ 30.	\$ 138.
Sugerencias	5	12	3	50.	8.	42.
Instrucción	4	2	2	0.9	0.9	0.
Hilado 1	16	4	12	50.	25.	25.
Hilado 2	11	5	9	29.	14.6	14.4
Reposición Stocks	12	4	11	45.1	7.5	37.6
Transferencias	12	11	12	125.0	20.	105.0
Sistema de información Kevlar	313	119	76	1472.9	420.8	1052.1

Figura 6.3. Informe sobre el incremento de la productividad conseguido en DuPont al utilizar la tecnología CASE

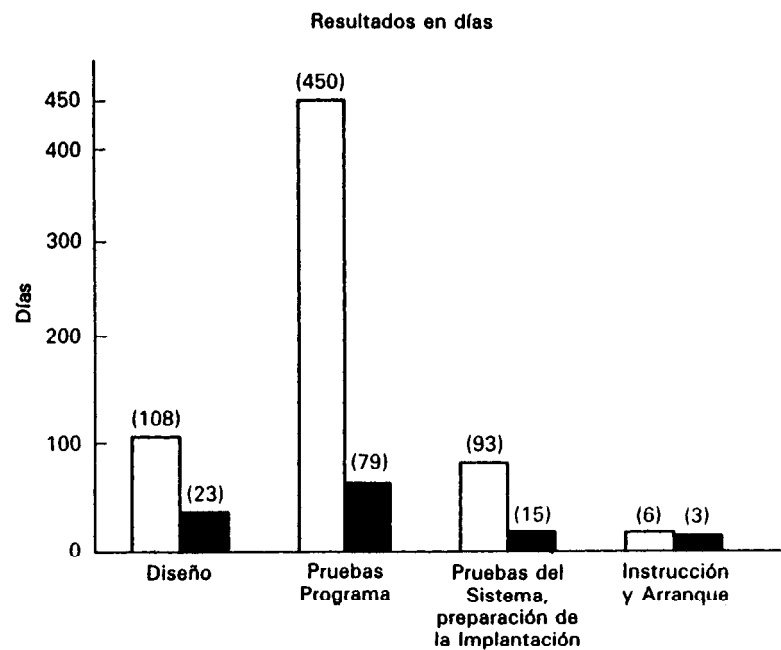


Figura 6.4. El mayor incremento de la productividad en DuPont se produjo durante las fases de programación y de pruebas (resultados en días)
En negro: Tiempo real con Factory, en blanco: tiempo estimado en COBOL

rios para desarrollar los sistemas se redujo considerablemente. El generador automático de código de FACTORY es el que mayor impacto tiene en la reducción del esfuerzo del personal en las fases de programación y pruebas (ver figura 6.4). DuPont afirmó que FACTORY había ayudado a la construcción más rápida de sistemas aún cuando se emplearon programadores menos experimentados. Muchos de estos programadores acababan de salir de la escuela. La curva de aprendizaje con FACTORY estaba en tres meses para llegar al nivel de competencia y en seis meses para alcanzar el nivel de experto.

FACTORY está ayudando a DuPont a alcanzar sus metas de productividad, porque, por término medio, el esfuerzo de desarrollo se reduce a la mitad cuando se emplea FACTORY. Un ejemplo es el proyecto Bulk Continuous Filament, que forma parte del sistema de inspección y embalaje de la planta textil de DuPont en Wagsnboro. Los requerimientos del proyecto eran utilizar técnicas de cuarta generación, técnicas estructuradas, un gestor de base de datos en un ordenador DEC VAX y finalizar la etapa crítica del sistema en 1996. Los análisis preliminares se hicieron utilizando técnicas estructuradas de Yourdon para producir estimaciones del tamaño del proyecto. El empleo de FACTORY con el RIPP permitió a DuPont satisfacer los requerimientos del proyecto experimentados.

La primera versión del sistema se completó, antes del tiempo estimado, con un ahorro de medio millón de dólares. Además, el sistema era más funcional de lo esperado por los usuarios. Aunque el proyecto había estimado, con desarrollo tradicional, entre veintiocho y treinta y nueve meses, quedó completamente acabado en un período real de cinco meses.

DuPont dijo que si el proyecto se hubiera realizado utilizando Fortran, no se podría haber acabado dentro del plazo previsto. Además del empleo de FACTORY, DuPont citó otras dos razones del éxito del proyecto: el RIPP y la colaboración del usuario. El análisis estructurado se siguió utilizando RIPP, cuya metodología permitió desarrollar el sistema por versiones. Los programadores fueron menos reacios al cambio de sistema y los usuarios vieron los resultados más pronto. Además, el proyecto pudo aprovechar al máximo el trabajo de los técnicos más experimentados de la planta. Eso era necesario porque el proyecto necesitaba la conexión con otros sistemas, algunos de los cuales controlaban la producción de la fábrica. El sistema constaba de doce subsistemas, siete informes de FACTORY,

veinte informes particularizados contruidos con CORTEX BUILDER LANGUAGE, sesenta y cinco ficheros de datos y trescientos campos.

En general, los usuarios parecieron satisfechos con el sistema resultante. Dijeron que habían conseguido el sistema mucho más rápido que nunca. Lo que pagaron por esta rapidez de desarrollo fue la particularización de informes y procedimientos. En general, sin embargo, los usuarios estaban dispuestos a aceptar esta limitación mientras que hubiese un servicio más rápido en el desarrollo y mantenimiento (cambios) en el sistema. Los usuarios midieron el éxito en términos de una rápida respuesta a sus requerimientos de cambio; es decir, veinticuatro horas con FACTORY, en lugar de siete días con Fortran.

El incremento medio de la productividad en este proyecto fue de un factor 3 sobre los resultados esperados. DuPont descubrió que las expectativas de los usuarios aumentan con su colaboración en el proyecto y con los buenos resultados, y que la credibilidad de los programadores aumenta considerablemente cuando los proyectos se completaban dentro de los plazos previstos.

DuPont también descubrió que sus usuarios no podían asumir un factor 10 de incremento, porque eso creaba un nuevo cuello de botella, el de la implantación.

Según DuPont, una aplicación perfecta para FACTORY es:

- Un proyecto que un programador individual pueda desarrollar en tres meses (el ciclo de desarrollo de RIPP).
- Un uso mínimo del lenguaje BUILDER, que es el lenguaje de procedimiento específico de FACTORY.

Otros proyectos de aplicaciones en DuPont incluyen el seguimiento de terminales, toxicología, productos químicos peligrosos, ingresos clave y sistemas de control de residuos. DuPont planea formar con la ayuda de FACTORY a doce analistas/programadores más en el manejo del FACTORY y llevar de veinte a treinta de las aplicaciones en IBM COBOL al entorno DEC VAX.

EXPERIENCIAS DE PRODUCTIVIDAD CON INFORMATION ENGINEERING WORKBENCH

El INFORMATION ENGINEERING WORKBENCH (conocido como IEW) de KnowledgeWare es un banco de trabajo CASE para el análisis de sistemas comerciales. El IEW es un banco de trabajo CASE y un compañero de metodología que automatiza una metodología estructurada llamada ingeniería de la información.

El IEW tiene tres utilidades básicas:

1. Una herramienta de diagramación automática para el dibujo de diagramas estructurados.
2. Un depósito CASE, con un sistema inteligente de la gestión de la información llamado **enciclopedia**, para almacenar, analizar y coordinar toda la información del sistema.
3. Una herramienta automatizada de análisis, llamada **knowledge coordinator**, para verificar la integridad y consistencia de toda la información del análisis y del diseño del sistema.

En los casos de estudio que nos ocupan, el IEW se empleó principalmente para proporcionar un soporte automatizado de los pasos de análisis del sistema y de los datos de metodología de la ingeniería de la información.

Los usuarios de IEW comunicaron tres beneficios importantes. Primero, el IEW es un compañero de metodología. Además de automatizar tareas como el dibujo y la revisión de diagramas estructurados, el IEW fuerza y normaliza la utilización de la metodología de la ingeniería de la información. Uno de los usuarios de IEW comentó que la imposición de una metodología estructurada es esencial en la construcción de sistemas de calidad.

Segundo, el IEW aumenta la productividad del desarrollo de software reduciendo el tiempo de los plazos de análisis, diseño y construcción de los sistemas de información. Un usuario de IEW informó que una entrevista con el usuario final que normalmente lleva cuatro horas, con la asistencia de IEW sólo necesita una hora.

Tercero, el IEW mejora la comunicación entre los analistas por medio de la documentación automatizada. El empleo de diagramas estructurados para explicar la especificación del sistema mejora la comunicación entre los usuarios finales, los gestores y los programadores.

Experiencia de Deere & Co. con IEW

En Deere & Co. hay 39 grupos descentralizados para el desarrollo de sistemas y no hay unas normas generales para la compañía. Debido a la depresión en la industria agrícola hubo que congelar la contratación en Deere & Co. desde 1979. Durante este período, la adición de múltiples sistemas de gestión de bases de datos, como (DB2, ORACLE y dBASE III) ha significado un crecimiento de la complejidad en las organizaciones para el desarrollo que previamente estaba soportada únicamente con IMS. Además, durante este período de tiempo Deere & Co. ha desarrollado su propia metodología de desarrollo basada en la metodología de la ingeniería de la información.

Las funciones del departamento de Planificación de Sistemas y Administración de Datos de Deere & Co. son:

1. Realizar un análisis preliminar de los requerimientos de los proyectos que implica compartir datos corporativos.
2. Mejorar la administración de los datos y su integración en el desarrollo de sistemas.
3. Orientar a la organización hacia la mejora del desarrollo de sistemas, soportando la metodología Deere & Co. con herramientas automatizadas.

El principal aspecto productivo de Deere & Co. es conseguir un aumento del 30% en la productividad del desarrollo del **software**. Se desea que menos personal haga más trabajo en menos tiempo. Se pretende realizar un trabajo mejor en la especificación de los requerimientos del sistema. Se pretende poder trasladar más fácilmente a los analistas de un proyecto a otro. Finalmente también se quiere reducir el trabajo de mantenimiento.

Deere & Co. eligió el IEW como ayuda para conseguir los objetivos de la productividad. Desde 1985 se han instalado diez copias del IEW en esta empresa, lo que supone un promedio de un ordenador personal por cada dos analistas. El costo total de cada estación de trabajo CASE (**hardware** y **software**) fue de quince mil dólares. Se necesitaron dos semanas para integrar el uso del IEW en la metodología propia de Deere & Co. El usuario principal de IEW en Deere & Co. es el analista de sistemas y el empleo principal es el diseño de sistemas. El promedio de tiempo para llegar a un nivel eficiente con las herramientas es de dos días, asumiendo que el usuario está familiarizado con las técnicas de análisis estructurados.

Deere & Co. planea tener una estación de trabajo CASE basada en IEW-PC para cada uno de los trescientos analistas y programadores. Actualmente, cinco proyectos de desarrollo de sistemas de tamaño medio utilizan el IEW. La productividad se multiplica por dos en tareas de análisis de los datos y de los requerimientos al utilizar el IEW, en lugar del método manual. Un sistema actualmente en desarrollo, el Track Project Development Cycle System, consta de noventa diagramas de flujo de datos y de cincuenta de entidad/relación para describir cuatrocientas treinta funciones del negocio. Las herramientas automáticas de diagramación logran que la diagramación del análisis sea práctica y ayudan a aumentar la comunicación con los usuarios. Además, Deere & Co. informó que la comprobación en los diagramas estructurados fue el factor más importante del incremento de la productividad. La depuración temprana de los errores en el proceso de desarrollo simplificaba el trabajo de las últimas fases.

La idea y el consejo de Deere es colocar la metodología del desarrollo en lugar preferente. Las herramientas no proporcionan la metodología, sólo automatizan la asistencia a la metodología.

Experiencia de Babcock & Wilcox con IEW

Hay seis divisiones en el Power Generation Group de Babcock & Wilcox. Utilizan lenguajes de cuarta generación y COBOL para desarrollar sistemas por lotes y en línea. Desarrollaron su propia metodología basada en la metodología de sistemas de ordenadores, planificación de los sistemas de información e ingeniería de la información de Arthur Young.

Babcock & Wilcox también eligió el IEW para ayuda a automatizar su versión de la metodología de la ingeniería de la información. Esta-

ban interesados particularmente en las herramientas que automaticen el análisis de sistemas y la fase de diseño del ciclo de vida.

Las funciones del Departamento de Bases de Datos y Normas en Babcock & Wilcox son:

1. Realizar la gestión y la implantación de las bases de datos.
2. Desarrollar estándares y procedimientos para el desarrollo del sistema.
3. Aumentar la productividad de los profesionales del desarrollo de sistemas, especialmente en las fases de análisis y del diseño.

El aumento de la productividad en el análisis y en el diseño eran lo más importante, porque Babcock & Wilcox sentía que no hacía un buen trabajo esas importantes fases del desarrollo del sistema.

Desde enero de 1986 se han utilizado cuatro copias de IEW, que estaban compartidas entre catorce analistas y entre varios proyectos. La enciclopedia IEW se incrementó con los resultados del sistema de planificación con cincuenta y dos funciones del negocio que servían para todos los proyectos del desarrollo. Una de las cuatro estaciones de trabajo IEW se destinó a enciclopedia maestra. El principal empleo de IEW es el diseño de sistemas de aplicación del negocio. El principal usuario es el analista de sistemas, pero los usuarios finales desarrollan los modelos lógicos de datos a través de IEW.

Actualmente, cuatro sistemas de aplicaciones de tamaño medio en Babcock & Wilcox se están diseñando con IEW. Los usuarios finales construyen sus propios diagramas estructurados con las herramientas automatizadas de diagramación de IEW. Babcock & Wilcox espera multiplicar por tres la productividad en las tareas en análisis y de diseño utilizando el IEW, en lugar de hacerlo manualmente.

Mirando al futuro, el desarrollo del **software** en Babcock & Wilcox tienen la vista puesta en el mantenimiento. Con IEW podrán mantener los sistemas conservando la información del diseño en la enciclopedia IEW y generando y regenerando automáticamente el código desde el diseño en lugar de mantener el código.

RESUMEN

Para resumir, todos los usuarios de EXCELERATOR, APPLICATION FACTORY e IEW mencionados en este capítulo informaron que la tecnología CASE tiene un papel importante en el incremento de la productividad. Además, la tecnología CASE tiene un impacto muy positivo en los sistemas de desarrollo del **software** de todos los tipos y tamaños.

Finalmente, también comunicaron que la productividad del **software** era más alta cuando las herramientas automatizan una metodología estructurada aceptada en la organización y cuando las herramientas estén en manos de analistas y programadores expertos.

Es importante hacer hincapié en el hecho de que muchos de los incrementos de la productividad mencionados están confinados a tareas específicas, como el modelado lógico de los datos, en lugar de extenderse al resto de las fases del ciclo de vida del **software**. Esto sugiere que los usuarios deben tener cuidado al seleccionar una herramienta CASE que puede incrementar la productividad donde sea más necesario en su organización en particular. También hay que señalar que la formación en las técnicas de la metodología es un factor crítico en el incremento de la productividad. Aquellas organizaciones que consiguieron un mayor aumento de la productividad ya tenían una formación en metodología. Muchos usuarios entrevistados comentaron que la formación en metodología era más importante y requería más tiempo que el aprendizaje de las herramientas. La elección de la metodología de desarrollo correcta es la base sobre la que se asienta la estrategia de la alta productividad, ya que es la metodología lo que guía el proceso de desarrollo del **software**.

No hay una fórmula mágica para conseguir el éxito en la productividad del **software**, pero si desea un consejo, emplee el mejor personal, las mejores metodologías y las mejores herramientas.

Referencias

El material de este capítulo se obtuvo de las entrevistas a los usuarios de herramientas CASE de las compañías citadas y de los distribuidores de herramientas. Las entrevistas fueron dirigidas por la autora. Una copia de las cuestiones de las entrevistas se incluye en el cuadro 6.1.

Cuadro 6.1. Cuestiones sobre la productividad

1. Información sobre la compañía.
 - Departamento(s) involucrados.
 - Tamaño del departamento de análisis y programación.
 - Tipos de sistemas de **software**.
 - Resultados de la productividad.
2. ¿Cómo se mide la productividad del profesional de **software**?
3. ¿Qué criterios se emplearon en la selección de las herramientas de productividad?
4. ¿Qué otras herramientas y técnicas de metodologías de **software** hay instaladas?
5. ¿Cómo se midió el incremento de la productividad? ¿Qué papel tuvo en el éxito la herramienta? ¿Qué beneficios se obtuvieron?
6. ¿Para qué tipos de tareas/aplicaciones es aplicable la herramienta? ¿Quiénes son los usuarios primarios? ¿Cuánto tiempo se necesita para llegar a un nivel de eficiencia con la herramienta?
7. ¿Cómo se integran las herramientas al entorno instalado?
8. ¿En qué cambió el proceso de desarrollo del **software** con la introducción de las herramientas?
 - ¿Impacto en el mantenimiento?
 - ¿Papel desempeñado por los usuarios finales?
9. Historia del empleo de la herramienta.

CAPITULO 7

CONSIDERACIONES DE LA IMPLANTACION DE LA CASE

PROBLEMAS Y SOLUCIONES DEL SOFTWARE

A pesar de que hay muchas referencias sobre el incremento de la productividad del **software** con la tecnología CASE, se dan casos en los que no supuso un cambio significativo o no mejoró el desarrollo y mantenimiento del **software**. ¿Por qué en algunas organizaciones el empleo de la tecnología CASE pasa desapercibido mientras que en otras se consigue un éxito espectacular?

La experiencia ha demostrado que el empleo de la tecnología CASE no siempre garantiza el éxito [1]. Algunas observaciones frecuentes de usuarios de tecnología CASE, son las siguientes:

- La tecnología CASE no puede incrementar la productividad y la calidad del **software** si no se usa.
- En muchas organizaciones, los programadores no han recibido la formación adecuada y no utilizan la herramienta como metodología de desarrollo.
- La tecnología CASE no es un sustituto de una buena gestión del proyecto.

- La actitud de los profesionales puede afectar beneficiosamente al uso de la tecnología CASE.
- Los proyectos de **software** quedan a menudo fuera de control porque los directivos no imponen el desarrollo de estándares de procedimientos.

Una lista más completa de las causas del fracaso en el empleo de las tecnología CASE se muestra en el cuadro 7.1. Sin embargo, existe una razón básica y muy importante para el fracaso. Las organizaciones en las que fracasó, no supieron reconocer la complejidad de los problemas de su **software** y por tanto fueron incapaces de comprender cómo utilizar la tecnología CASE en la solución de sus problemas.

La crisis del **software** que afecta hoy a muchas organizaciones es un problema con múltiples facetas para el cual no hay una solución única y sencilla. La tecnología CASE puede conducir los aspectos técnicos de la crisis del **software**.

Cuadro 7.1. Causas del fracaso de la CASE

- | |
|---|
| <ul style="list-style-type: none">• Confusión sobre lo que realmente hace cada producto CASE individualmente.• El uso de herramientas CASE en problemas para las que no están diseñadas.• Poner demasiado empeño en las herramientas CASE como solución completa.• Ignorar la importancia de una buena gestión.• No tener desarrollados estándares o metodologías en la organización.• Herramientas CASE pobremente integradas.• Herramientas pobres en documentación y formación.• Sin funcionalidad suficiente en las herramientas CASE.• Falta de claridad sobre que problema del software necesita solución. |
|---|

Cuadro 7.1. Causas del fracaso de las CASE (continuación)

- Falta de métodos para medir el impacto de CASE en el desarrollo y mantenimiento del software.
- Falta de formación en la metodología del desarrollo del **software**.
- Indecisión/falta de predisposición a tomar una decisión sobre cómo utilizar la tecnología CASE.
- Falta de predisposición a cambiar de modo de trabajo en el desarrollo y mantenimiento del **software**.
- Ver la CASE como una tecnología de alto riesgo.
- Inexistencia de una planificación para implantar la tecnología CASE.

La tecnología CASE es una parte muy importante de la solución de la crisis del **software** pero no debe considerarse como la solución total. El simple hecho de comprar unas herramientas CASE no es probable que tenga un efecto importante es la mejora global de la productividad y la calidad del **software**. Una solución total a la crisis del **software** debe verse a través de múltiples facetas: gestión, personal, herramientas y metodologías.

CASE O NO CASE

Cuando se dispone de una tecnología nueva como la CASE, una organización debe tomar una decisión en relación con el uso la nueva tecnología. Las elecciones básicas son simples:

1. No hacer nada.
2. Adoptar la nueva tecnología.

Sin embargo llegar a tomar la decisión puede no resultar tan simple. Las decisiones no deben basarse solamente en una respuesta rápida a las presiones de los usuarios de construir sistemas rápidamente ni a los deseos del personal de desarrollo de probar los últimos avances de la tecnología.

Algunas organizaciones se eternizan en el estudio de los diversos aspectos de la nueva tecnología y nunca llegan a tomar la decisión de aceptarla o no. Por ejemplo, una organización con una facturación de cinco mil millones de dólares y un presupuesto para informática superior a los cien millones de dólares, gastó medio millón poniendo a prueba herramientas CASE, pero la decisión de emplear o no las herramientas CASE nunca llegó a tomarse. La indecisión ante el temor de un posible fracaso fue la causa en gran medida de tomar la decisión equivocada.

En otra organización, después de haber gastado más de un millón de dólares en la formulación de un plan para la instalación de la CASE, todavía no hay tal plan, porque no hay una herramienta CASE que pueda resolver **todos** los diferentes problemas de **software** existentes y, además, porque los profesionales de desarrollo del **software** estaban dudosos respecto a utilizar herramientas desarrolladas fuera de la organización. Las organizaciones que están esperando la solución perfecta y fácil de implantar a sus problemas tendrán que seguir esperando mucho, pero mucho tiempo.

Otras organizaciones actúan demasiado rápidamente y adoptan una tecnología sin considerar la utilidad o el impacto. Muchas herramientas CASE muy potentes se han instalado y no han pasado de ser simples posibilidades sin llegar a utilizarse porque la organización las adquirió sin un plan de integración de la CASE en la cultura del **software** existentes. Muchos profesionales del desarrollo del **software** desean ser tenidos en cuenta durante la selección de las herramientas que van a utilizar.

Una estrategia que puede evitar estos problemas y ayudar a la organización en la toma de una decisión razonable acerca del empleo de la tecnología CASE en un tiempo razonable es:

1. Establecer las necesidades de la organización.
2. Demostrar que la nueva tecnología satisface esas necesidades.
3. Establecer un plan de implantación de la nueva tecnología y aplicarlo.

ESTABLECIMIENTO DE LAS NECESIDADES

La mayoría de las organizaciones se enfrentan ahora a una profunda crisis de **software**. Hay una gran lista de sistemas de **software** en espera de

ser desarrollados. Según una reciente investigación entre más de mil organizaciones, el tiempo medio de espera de las aplicaciones era de 88 meses [2]. Además, las organizaciones no estaban satisfechas con la calidad del **software** que se estaba desarrollando; no pueden cambiar los sistemas con la rapidez suficiente para hacer frente a los cambios de necesidades de los usuarios. Las organizaciones tienen que hacer frente al siempre creciente aumento de las necesidades del mantenimiento de los sistemas existentes. Resumiendo, las organizaciones reconocen la necesidad desesperada de mejorar significativamente los procesos de desarrollo y mantenimiento del **software**. No hacer nada para resolver la crisis del **software** es un grave error que sólo da como resultado la escalada de problemas y de gastos.

Las organizaciones necesitan resolver su crisis de **software** inmediatamente; necesitan desarrollar sistemas de mejor calidad en mucho menos tiempo y con un costo mucho menor. Necesitan reducir la carga de mantenimiento de los sistemas, que es una de las principales causas de los retrasos [2]. Desafortunadamente, no está claro cuál es la mejor forma de afrontar esta necesidad. Las organizaciones sufren las agotadoras consecuencias de anteriores intentos fallidos para aliviar la crisis del **software**, lo que explica en gran medida su rechazo a embarcarse en una nueva tecnología. Una parte muy importante para el éxito en la implantación de la CASE en una organización es superar este rechazo con la definición de lo que debe hacerse y cómo la tecnología CASE puede contribuir.

Cuadro 7.2. La definición de las necesidades de una organización para mejorar la tecnología actual del desarrollo de software

Cuestiones sobre la gestión:

- ¿Cuál es la estructura organizacional de los desarrollos del **software**? ¿Está centralizada o descentralizada?
- ¿Qué formación sobre la herramienta o sobre las metodologías se ha proporcionado al personal de desarrollo?
- ¿Qué estándares y procedimientos de gestión de proyectos están instalados?
- ¿Cuáles son las posibilidades, debilidades y cuellos de botella en el ciclo de vida del desarrollo del **software** actual?
- ¿Cómo se mide la productividad del **software**?
- ¿Cómo se mide la calidad del **software**?

Cuadro 7.2. La definición de las necesidades de una organización para mejorar la tecnología actual del desarrollo de software (continuación)

Cuestiones técnicas:

- ¿Está automatizado el proceso seguido para el desarrollo del software? ¿Y para el mantenimiento de los sistemas?
- ¿Se utiliza una metodología disciplinada para organizar el trabajo de desarrollo del **software**?
- ¿Qué entorno **hardware** se emplea como plataforma para el desarrollo y mantenimiento del **software**?
- ¿Qué herramientas de desarrollo y mantenimiento del **software** hay instaladas (incluyendo los lenguajes de programación)?
- ¿Qué estándares y procedimientos técnicos se siguen para garantizar la calidad del **software**?
- ¿Qué estándares y procedimientos se aplican al mantenimiento del **software**?
- ¿Cuál es la actitud del personal de desarrollo en relación con la automatización del **software**?

La definición detallada de las necesidades

Como cada organización sufre su propia y personal forma de crisis del **software**, no hay una solución universal. La solución práctica debe hacerse a la medida de cada organización, lo que significa que cada una debe empezar por especificar y asignar prioridades detalladamente a sus necesidades. Simplemente decir que se necesita aumentar la calidad y la productividad del **software** no es una especificación adecuada para determinar una solución práctica con una CASE. El cuadro 7.2 es una lista de lo que se debe preguntarse y responderse para determinar las necesidades de una organización. Las cuestiones se dividen en técnicas y de gestión, ya que deben comprenderse ambos aspectos para especificar totalmente las necesidades. La organización debe comprender la tecnología actual del desarrollo y el mantenimiento del **software** y de las áreas más importantes antes de poder determinar una solución CASE práctica a su propia crisis de **software**.

Definición de los objetivos organizacionales

Una vez definidas las necesidades y comprendida la tecnología actual del desarrollo y el mantenimiento del **software** en de la organización, deberán establecerse los objetivos a corto y largo plazo para satisfacer tales necesidades. Se deben establecer prioridades en los objetivos basandose en la urgencia de satisfacción de necesidades concretas, costos, formación y duración del tiempo de implantación.

TOMAR LA DECISION POR UNA CASE

Ahora que la organización tiene una comprensión clara y detallada de sus objetivos, puede tomar la determinación de adquirir la herramienta CASE y de como utilizarla para satisfacer estos objetivos. Deben considerarse tanto las respuestas técnicas como las de gestión. Hay algunas consideraciones importantes en cuanto a la gestión; por ejemplo, ¿es la cultura organizacional abierta al cambio? ¿Está el personal de desarrollo dispuesto a aceptar la CASE, la ve con indiferencia o con rechazo? ¿Cómo pueden venderse más efectivamente los conceptos CASE a la dirección para que esté dispuesta a destinar el tiempo y dinero necesarios para introducir apropiadamente la CASE en la organización y más adelante seguir presando su apoyo?

¿Acepta la organización el concepto de los estándares tanto para el proceso del ciclo de vida del **software** como para el producto y cómo pueden integrarse los estándares? ¿Quiénes toman la decisión de adoptar o no la tecnología CASE? Si se adopta, ¿quién será el responsable de implantar la nueva tecnología?

Además, hay muchas consideraciones técnicas importantes. ¿Qué cambios son necesarios en los actuales estándares y procedimientos para adaptar la CASE? ¿Es la metodología actual compatible con la automatización de la CASE? Si no hay una metodología, ¿cuál se va a elegir? ¿Quién será el usuario principal de la tecnología CASE? ¿Qué formación técnica será necesaria para emplear la tecnología CASE de una forma efectiva? ¿Cómo será la curva de aprendizaje? ¿Quién proporcionará el aprendizaje? ¿Cómo cambiará la CASE el proceso del ciclo de vida del **software**? ¿Dónde están las posibilidades y debilidades del proceso del ciclo de vida actual?

EL PLAN DE IMPLANTACION DE LA CASE

Si se toma la decisión de adoptar la tecnología CASE, debe establecerse un plan de implantación. El plan debe proponer una implantación de la CASE por fases porque las organizaciones que intentan implantar demasiado y demasiado rápidamente aumentan considerablemente los riesgos de fracaso [3]. Por ejemplo, subestimar la necesidad de formación en la metodología CASE y de su curva de aprendizaje puede conducir al fracaso de la adopción de la CASE. El cuadro 7.3 lista las consideraciones técnicas y de gestión que debe cubrir un plan de implantación de la CASE.

Los pasos a seguir en la una implantación de la CASE varían dependiendo de si los estándares y procedimientos del proceso del ciclo de vida existentes en la organización son o no satisfactorios y automatizables. Si lo son, entonces como primer paso se seleccionan las herramientas CASE que soporten la automatización de los estándares y procedimientos para mejorar el entorno actual que soporta el **software**. Si no existen, se empieza creando los estándares y procedimientos donde su ayuda es más necesaria para mejorar los procesos de desarrollo y de mantenimiento del **software**. Después, como siguiente paso se seleccionan las herramientas CASE que soporten estos estándares y procedimientos. Es conveniente tener muy bien definidos los procesos antes de intentar automatizarlos a través de las herramientas CASE. (Las metodologías estructuradas se tratan en el capítulo 3.)

PRIMERO LA METODOLOGIA

Las organizaciones que consiguen mayor beneficio con el uso de la tecnología CASE utilizan un conjunto totalmente integrado de herramientas CASE que automatizan y soportan el proceso integrado del ciclo de vida del **software**. Aunque esto podrá considerarse como el objetivo final, no puede cumplirse en una fase de implantación. Como quiera que una metodología de desarrollo proporciona el marco general para definir y enlazar todos los pasos del proceso del **software**, la selección de la metodología debe ser el primer paso de una implantación CASE.

Una vez definido todo el proceso del **software**, puede implantarse un aproximación CASE aplicandola a los pasos individuales de la metodología, un paso cada vez. El orden en la implantación dependerá de las priori-

dades de las necesidades de la organización. Por ejemplo, si una organización ha determinado que la mejora de los pasos de análisis y diseño del software son prioridad, el primer paso puede ser la introducción de un prototipo y sesiones de aplicación y diseño. Seguidamente, se deben seleccionar las herramientas CASE que soportan estas técnicas. O supongamos que una organización ha determinado que debe mejorar el proceso de revisión del **software** porque los métodos de revisión en curso no revelan los errores del **software** a tiempo.

Cuadro 7.3. El plan de implantación de la CASE

Consideraciones de la gestión

- Aceptación de la adquisición de la herramienta CASE, de la formación y de un apoyo continuado.
- Establecimiento de objetivos y presupuestos.
- Aceptación de la tecnología CASE por la dirección y por los profesionales del desarrollo y del mantenimiento.
- Creación de un grupo de implantación y de soporte continuado de la tecnología CASE.
- Asignación de responsabilidades para la investigación, implantación y evaluación de la herramienta CASE.
- Definición de los controles de la gestión.
- Obligatoriedad de los estándares y procedimientos CASE.
- Creación de los programas de formación sobre la herramienta CASE y sobre las metodologías.
- Creación de una planificación de tiempos para los pasos necesarios del plan de implantación CASE.

*Cuadro 7.3. El plan de implantación de la CASE (continuación)***Consideraciones técnicas**

- Selección de la metodología de desarrollo.
- Definición de los estándares técnicos para interfaces con herramienta, sistemas gestores de bases de datos, lenguajes de cuarta generación, diccionarios y los controles de calidad del **software**.
- Definición de las capacidades de la herramienta CASE.
- Definición de las necesidades del **hardware** y de las redes de área local.
- Selección de las herramientas CASE que satisfagan las metodologías, estándares y procedimientos de la organización.
- Definición de la formación técnica necesaria.
- Definición de las necesidades de soporte técnico para las herramientas y metodologías CASE.
- Definición de los tipos de sistemas a desarrollar y mantener con las herramientas CASE.
- Realización del proyecto piloto de la CASE.
- Gestión de la configuración de la herramienta CASE.

Cuadro 7.4. Los pasos básicos para la implantación de la CASE

- | |
|---|
| <p>Paso 1: Definir el proceso del ciclo de vida del software especificando sus pasos, estándares y procedimientos que controlan el proceso y las comprobaciones que garantizan la calidad del software obtenido.</p> <p>Paso 2: Definir el orden de aplicación de la tecnología CASE al proceso basándose en las necesidades y prioridades de la organización.</p> <p>Paso 3: Seleccionar herramientas CASE para automatizar y soportar el proceso e imponer la utilización de los estándares y procedimientos.</p> <p>Paso 4: Poner a prueba herramientas y metodologías CASE en proyectos piloto.</p> <p>Paso 5: Introducir a gran escala herramientas y metodologías en toda la organización mediante programas de formación y compromisos de gestión y aplicación.</p> <p>Paso 6: Evaluar el impacto de la CASE aplicando medidas de éxito predeterminadas y mantener el compromiso de mejorar continuamente los procesos del ciclo de vida y calidad del software obtenido.</p> |
|---|

Esta organización debe primero implantar las herramientas CASE que realicen una comprobación continuada y automática de los productos de **software** al tiempo que van desarrollándose y proporcionar informes de comprobaciones y análisis.

Los pasos básicos de la implantación de la CASE se exponen en el cuadro 7.4.

EL PROYECTO PILOTO DE LA CASE

Aquellas organizaciones cuya dirección y/o personal de desarrollo son reacias a probar la CASE podrían empezar por establecer un proyecto piloto para comprobar cómo operan las CASE en sus organizaciones. El piloto debe ser un proyecto real no crítico. Los objetivos de cómo utilizar y medir el éxito de la CASE deberán definirse exactamente antes de comenzar la prueba piloto. Las herramientas CASE deberán seleccionarse a partir de su aplicabilidad para satisfacer los objetivos establecidos. Pueden utilizarse herramientas y metodologías alternativas en proyectos paralelos como medio de comparación entre varias aproximaciones CASE. Los participantes en el proyecto piloto deberán ser profesionales del desarrollo deseosos de utilizar la tecnología CASE. Al menos algunos participantes deberán ser expertos en el desarrollo del **software**. Cuando finalice el proyecto piloto, los resultados pueden emplearse para:

1. Sugerir que herramientas y metodologías CASE son más útiles.
2. Convencer a la dirección y a los programadores de los beneficios que pueden derivarse de utilizar las herramientas CASE en su organización.
3. Definir estándares y procedimientos que pueden utilizarse para aplicar la tecnología CASE.

Después, los participantes en el proyecto piloto pueden formar parte de otros proyectos en los cuales se emplee la CASE. Seguidamente se pueden instituir programas de formación y planes de expansión a gran escala de la CASE en la organización.

Incluso las organizaciones que ya tienen una administración y programadores dispuestos a utilizar la CASE pueden encontrar en los proyectos piloto un primer paso útil para la implantación de la CASE en la organización.

IMPLANTADORES DE LA CASE

La tarea de implantar de la CASE se suele asignar a un grupo especial de profesionales del **software**. Algunas organizaciones tienen un centro de

desarrollo, un grupo de estándares y normas, un grupo de tecnología avanzada o un grupo de control de calidad que investiga nuevas tecnologías. Si tal grupo no existe, deberá crearse. Las responsabilidades del grupo consistirían en:

1. La investigación de las nuevas tecnologías, como la CASE.
2. Diseñar un plan para implantar las nuevas tecnologías.
3. Proporcionar un soporte de consultas de las nuevas tecnologías.

Recuérdese que muchas organizaciones presentadas en los casos de estudio del capítulo 6 tenían un grupo de investigación y desarrollo de estándares y normas que era el responsable de implantar la tecnología CASE.

LA SELECCION DE LA HERRAMIENTA CASE

Las herramientas CASE deben seleccionarse en el contexto de:

1. la identificación de necesidades y prioridades de la organización;
2. las herramientas de **software** existentes, y
3. la metodología que estructura el proceso.

En el cuadro 7.5 se listan los aspectos técnicos de la selección de las herramientas CASE. (Las características de un **workbench CASE** completo se discutieron en los capítulos 2, 3 y 4).

VENTA DE LA CASE

Vender los conceptos de la CASE a los directivos y a profesionales de **software** de la organización es una de las más importantes formas de hacer que la CASE funcione en la organización. Una actitud positiva hacia la CASE puede en última instancia marcar la diferencia entre una CASE con éxito o una CASE fracasada.

Cuadro 7.5. Consideraciones técnicas en la selección de las herramientas CASE

- Los costos de **software**, **hardware** y formación.
- Los requerimientos del **hardware**.
- Las herramientas instaladas, incluyendo diccionarios, sistemas gestores de bases de datos y lenguajes de cuarta generación, con los cuales deben integrarse las herramientas CASE.
- Los tipos de sistemas de **software** soportados por las herramientas.
- La calidad de la documentación de la herramienta.
- Las posibilidades de particularizar la herramienta.
- El soporte y mantenimiento de la herramienta.
- La demostración y la experiencia en el manejo de la herramienta.
- Las entrevistas con los usuarios actuales de la herramienta.
- La reputación del suministrador y su línea de productos.
- La dirección futura de la organización y del suministrador.
- La calidad de las capacidades técnicas de la herramienta:
 - Interfaz gráfico.
 - Nivel de integración.
 - Interfaces con herramientas externas.
 - Depósito CASE.
 - Cobertura del soporte del ciclo de vida.
 - Soporte de la metodología estructurada.

Los profesionales del **software** deben estar convencidos de que la tecnología CASE es algo real y que merece la pena, y que es importante aprender la utilización de proyectos de **software** real. Si no se comprende por qué es necesario el cambio y la mejora del actual proceso de **software** a través del empleo de la CASE, no es probable que se acepte.

La participación de eventuales colaboradores es un buen método para vender un nuevo concepto. Permitir a los profesionales del desarrollo que participan en la selección y evaluación de las herramientas CASE ayudará a crear un espíritu corporativo sobre la CASE. Cada profesional debe creer que su participación representa una gran oportunidad para adquirir experiencia, contribuir al establecimiento de estándares y procedimientos corporativos y ser aceptado por la dirección.

La necesidad de vender la CASE no se detiene con sus programas de introducción y formación. Los directivos del proyecto deben seguir vendiendo los méritos de la CASE con cada proyecto del **software** si esperan que la tecnología CASE sea utilizada verdaderamente por los programadores.

La CASE puede venderse en una gran variedad de formas, algunas de las cuales se listan en el cuadro 7.6.

Cuadro 7.6. Formas de vender la tecnología CASE

- Con proyectos pilotos que demuestren que la CASE funciona en esta organización.
- Con medidas que demuestran el impacto positivo de la CASE.
- Con la participación de los usuarios de la CASE en el proceso de selección de la herramienta y en la formulación de los estándares y procedimientos.
- Con la formación en las herramientas y metodologías estructuradas CASE.
- No deteniendo nunca la venta.

LA EVALUACION DE LA CASE

Como la mayoría de los proyectos de **software** se retrasan según lo planificado y como los directivos están ansiosos por empezar a trabajar con el siguiente proyecto en espera de ser desarrollado, no se dedica tiempo a recapacitar. Los profesionales del desarrollo y del mantenimiento raramente se dedican a aprender de los éxitos y fracasos pasados. Las presiones para cubrir las crecientes demandas de más **software** y la permanencia en el mismo nivel técnico, han oscurecido la importancia de evaluar lo bien que funcionan las tecnologías del **software** en la práctica.

La utilización de las herramientas y metodologías de **software** es sólo el primer paso en las mejoras del proceso del **software**. También se debe evaluar su utilidad en la práctica. El mayor problema que bloquea las mejoras del **software** es la falta de información sobre el estudio de casos de proyectos. Es difícil convencer a la dirección del valor de la prueba con la nueva tecnología cuando no hay un soporte de tal evidencia. También es difícil especular si más adelante serán necesarios cambios o mejoras. En el capítulo 6 se presentaron casos de estudio sobre metodologías CASE precisamente para demostrar que la CASE funcionó en organizaciones reales. El estudio de cómo funcionan en la práctica las herramientas y metodologías **software** es el mejor modo de prepararse para unos niveles más altos de automatización del **software**.

Esto no es una idea nueva, ya ha sido proclamada en la literatura sobre el **software**. Por ejemplo, el último de los siete principios básicos de Boehm sobre ingeniería del **software** enunciados en 1977 [4] es:

Mantener el compromiso de mejorar el proceso

Boehm explica que lo fundamental en el proceso de producción de **software** de calidad es el compromiso continuado por parte de los profesionales del desarrollo y del mantenimiento de investigar cómo mejorar el producto **software** y su producción. Deberían probarse nuevas metodologías y herramientas sobre proyectos reales. Deberían definirse métodos para evaluar los efectos de tales herramientas. Deberán recogerse y analizarse datos para descubrir los cuellos de botella, la estimación deficiente de los calendarios, de los costes y de las frecuencias de error.

Aunque proclamado hace más de una década, este principio raramente se practica actualmente. Para poner de manifiesto su importancia, debería considerarse como el paso fundamental en la implantación de una tecnología CASE en una organización. La razón es proporcionar información útil para mejorar los futuros proyectos y productos de **software**:

- ¿Qué funcionó?
- ¿Qué no funcionó?
- ¿Cuál es el próximo paso para intentar un trabajo mejor en el desarrollo y mantenimiento de los sistemas de **software**?

BIBLIOGRAFIA

1. Jim Huling, "Tools of the Trade: Is CASE Really a Cure-All?" *Computer world*, 20 abril, 1987, págs. 73-86.
2. "CASE 1987 Survey Executive Summary", *Software News*. Westborough, Massachusetts: SENTRY Publishing Company, 1987.
3. Roger S. Pressman, "Making Software Engineering Happen", *The CASE Report*. Southfield, Michigan: Nastec Corporation, julio 1987, págs. 1-3.
4. Barry Boehm, "Seven Basic Principles of Software Engineering", en *Infotech State of the Reports; Software Engineering Techniques*. Maidenhead, Inglaterra: Infotech International, 1977, págs. 77-113.

CAMBIOS EN EL CICLO DE VIDA DEL SOFTWARE

EL CICLO DE VIDA DEL SOFTWARE

La utilización de la CASE afecta a todas las fases del ciclo de vida del **software**, desde la definición de los requerimientos, pasando por el análisis, el diseño, la codificación, las pruebas y el mantenimiento. El empleo de la CASE también cambia y perfila el ciclo de vida del **software**.

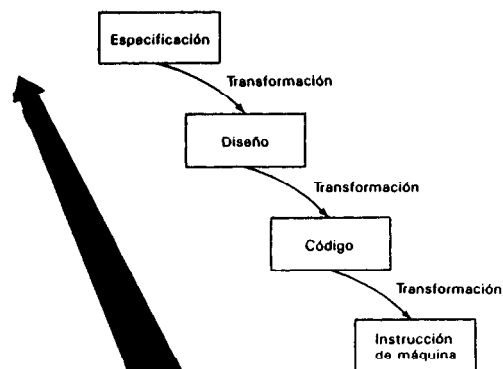


Figura 8.1. La parte de desarrollo dentro del ciclo de vida del software puede verse como una serie de transformaciones donde los objetivos del sistema se transforman en especificaciones de los requerimientos, después en especificaciones de diseño, seguidamente en programas en código y finalmente en instrucciones de máquina

Como se muestra en la figura 8.1, el ciclo de vida puede considerarse como una serie transformaciones. Primero se definen los requerimientos del sistema. Seguidamente los requerimientos se transforman en las especificaciones de diseño para un sistema que cumple esos requerimientos; después el diseño se transforma en programas, y finalmente los programas se transforman en instrucciones comprensibles para la máquina. La parte más creativa del proceso es la definición de los requerimientos del sistema, que es un tipo de solución del problema. Después las transformaciones son relativamente directas y automatizables. La transformación final (las instrucciones a nivel de máquina) es, por supuesto, automatizada por los ensambladores y compiladores. La tecnología CASE propone perfilar el ciclo de vida automatizando también las transformaciones del nivel más alto.

CONCEPTO UNIFICADOR

Un concepto unificador fundamental en la ingeniería del **software** es el tradicional ciclo de vida del **software**, que describe la secuencia de pasos del desarrollo utilizando y mantenimiento los sistemas del **software**. En esencia, es una técnica de gestión para estructurar el desarrollo y mantenimiento del **software**.

El ciclo de vida es un proceso de fases múltiples que comienza con la definición de un problema y continua hasta el envejecimiento, reemplazo o destrucción del sistema (por ejemplo, en caso de guerra). El ciclo de vida típico se compone de cinco fases básicas (ver figura 8.2):

- Análisis.
- Diseño.
- Codificación.
- Pruebas.
- Operación y mantenimiento.

El **análisis** es el proceso de la definición de los requerimientos para una solución del problema. Durante este análisis se examinan las necesidades de los usuarios y se definen las propiedades que debe poseer el sistema para satisfacer esas necesidades. También, se identifican las restricciones y necesidades de funcionamiento. Se deben definir de modo preciso las funciones a realizar, pero no su funcionamiento. El principal resultado de esta fase es la especificación del sistema. Idealmente, la especificación del siste-

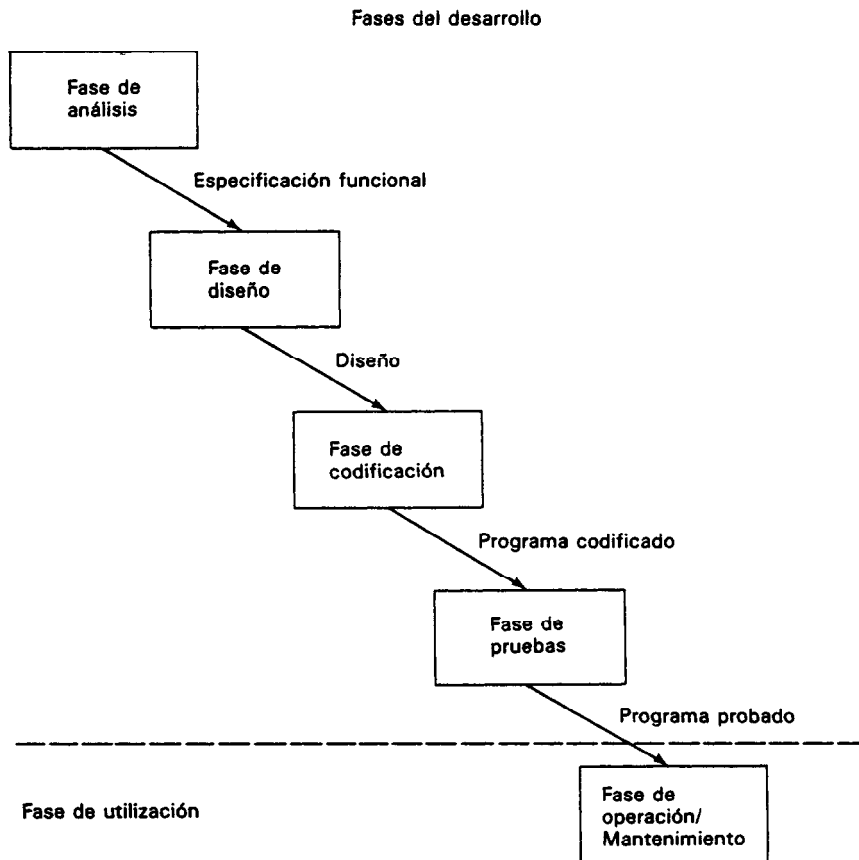


Figura 8.2. El ciclo de vida tradicional del software consta de cinco fases secuenciales: análisis, diseño, codificación, pruebas y operación/mantenimiento

ma establece las propiedades del sistema de forma precisa, comprobable y formal.

El **diseño** es el proceso de planificar cómo va a construirse el sistema; esto es, determinar los componentes de datos y de procedimientos necesarios y cómo esos componentes se ensamblarán para formar la solución. Se desarrollan algoritmos para describir lo que hace cada componente. Las especificaciones del sistema, los requerimientos del problema y las restricciones definidas en la fase de análisis se utilizarán como entrada en la fase de diseño.

La **codificación** es el proceso de transformar el diseño en instrucciones de ordenador. El objetivo de esta fase es producir programas correctos y eficientes. Las pruebas de los módulos codificados se suelen realizar durante esta fase.

Las **pruebas** es el proceso de demostrar que un **software** satisface los requerimientos del problema y funciona correctamente para todos los posibles datos de entrada. El objetivo de las pruebas es la eliminación de todas las condiciones y fallos inesperados del programa y descubrir cualquier implantación incorrecta de los requerimientos del problema. Los módulos de pruebas se integran y prueban en grupos cada vez más grandes hasta que se ha probado todo el sistema.

La **operación y el mantenimiento** son los procesos de la ejecución del sistema **software** en un modo de producción manteniendo un buen orden de operación. Esto implica.

1. Corregir errores y deficiencias que no se descubrieron durante el desarrollo del sistema.
2. Modificar el sistema para satisfacer los cambios de los requerimientos, adaptándolos a los cambios en el entorno y mejorando la eficiencia de operación y calidad general. Todos los recursos y actividades requeridos para asegurar que el **software** continúa satisfaciendo o excede las capacidades de operación requeridas se consideran parte del proceso de mantenimiento.

MODELO DE CICLO DE VIDA TRADICIONAL

En los años setenta, la mayoría de las organizaciones desarrollaron un modelo de ciclo de vida del **software** para describir las acciones y decisiones requeridas para definir, desarrollar, probar, entregar, operar y mantener un sistema de **software**. El modelo se derivaba del modelo **hardware**, que incluía fases para requerimientos, diseño, fabricación, prueba y operación/mantenimiento.

Algunas organizaciones formalizaron su ciclo de vida ampliándolo a una metodología completa. La metodología del ciclo de vida incluye herramienta, técnicas y métodos para proporcionar guías de ordenación y con-

tról de todas las actividades del **software**. Estas actividades se definen en función de:

- Los productos resultantes y disponibles.
- Los recursos necesarios.
- Las metodologías utilizadas.

Un aspecto clave de la metodología del ciclo de vida del **software** es la gestión formal del control. Es el medio principal para evaluar y regular las actividades del ciclo de vida para asegurar que se alcanzan los objetivos y la planificación del ciclo de vida.

Codificar demasiado pronto

La figura 8.3 muestra que en el ciclo de vida tradicional del software, las últimas fases están resaltadas. Como los programadores están ansiosos por empezar a codificar y los directivos del proyecto estaban también ansiosos por mostrar resultados tangibles de progreso a la dirección y a los usuarios, normalmente se dedicaba menos del treinta y cinco por ciento de los recursos de desarrollo a las fases de análisis y del desarrollo [1, págs. 1-44]. Esta distribución de los recursos en las fases de desarrollo del ciclo de vida da como resultado unos sistemas mal especificados y de alto costo.

La definición de los requerimientos es a menudo una tarea difícil, porque es difícil describir con precisión y en términos técnicos las necesidades de los usuarios. Además, los requerimientos no son estables; cambian con las personas y con el tiempo. En la práctica, la fase de análisis se realiza repetidamente a medida que los programadores y usuarios van aprendiendo más sobre el sistema. Los errores en la especificación de los requerimientos se propagan y crecen en las últimas fases del ciclo de vida. “Una especificación que define inadecuadamente los requerimientos del sistema puede dificultar más que ningún otro factor el éxito del proyecto” [2].

El énfasis en el FRONT-END

Cuando se descubrió que los errores más serios y costosos ocurrían en las primeras etapas del desarrollo del **software**, se hizo más hincapié sobre las primeras fases del ciclo de vida que sobre las últimas [3]. Como

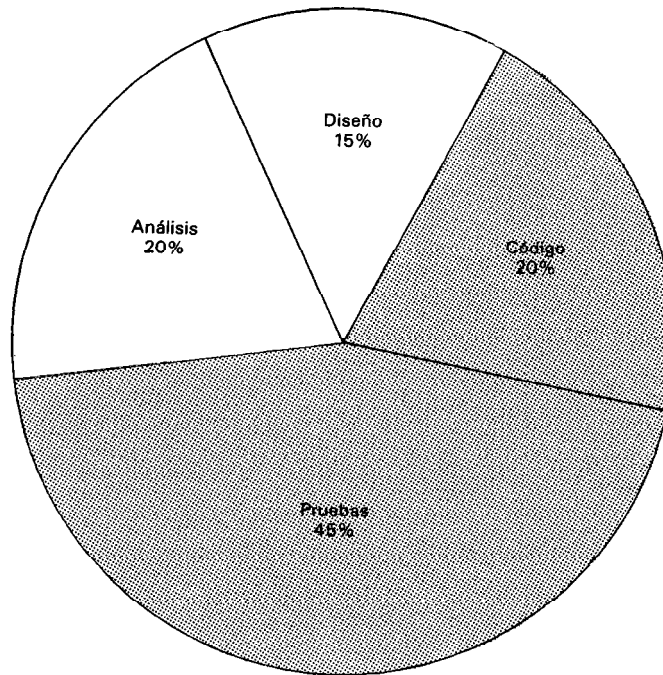


Figura 8.3. El ciclo de vida tradicional del software marca el énfasis en las últimas fases del desarrollo, con un sesenta y cinco por ciento del esfuerzo en las fases de código y de las pruebas [1]

se muestra en la figura 8.4, este énfasis en el **front-end** del ciclo de vida del **software** redistribuye los recursos de forma que más del sesenta por ciento se utiliza en las fases de análisis y diseño. Una parte fundamental de la estrategia de las técnicas estructuradas es dedicar el tiempo necesario a definir cuidadosamente los requerimientos e investigar a fondo las fórmulas de diseño antes de pasar a los detalles de la implantación.

EL CICLO DE VIDA CASE DEL SOFTWARE

Como se muestra en figura 8.5, el énfasis es mayor en las primeras fases del ciclo de vida con la introducción de la tecnología CASE. La automatización de muchas tareas de análisis y diseño, junto con la comprobación automática de la especificación de diseño da como resultado una alta

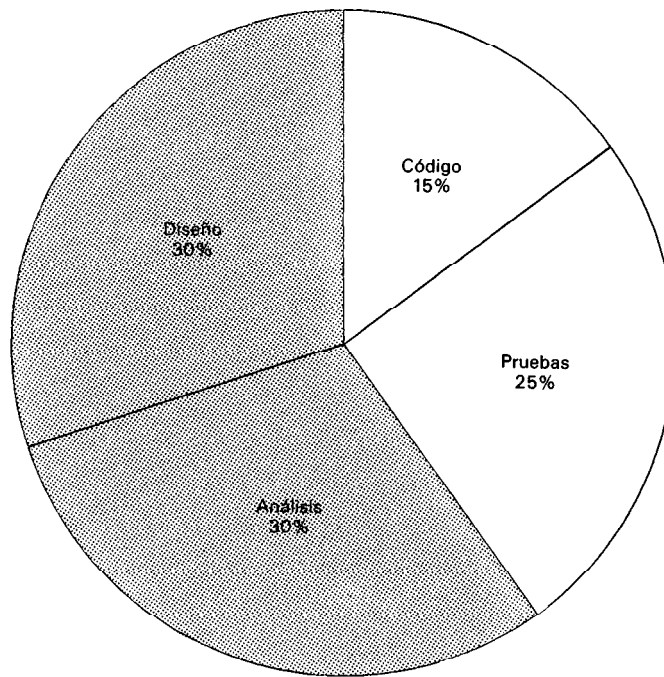


Figura 8.4. Las técnicas estructuradas pusieron el énfasis en las primeras fases del ciclo de vida del software con un sesenta por ciento de esfuerzo en las fases de análisis y de diseño [3]

productividad y menos errores. La comprobación automática permite identificar y corregir errores debidos a especificaciones ambiguas e incompletas mucho antes en el proceso de desarrollo. La generación automática del código elimina virtualmente la fase de codificación del ciclo de vida.

Además, la tecnología CASE racionaliza el ciclo de vida acortando en un cincuenta por ciento la duración del proceso de desarrollo, de acuerdo con la información aportada por muchos usuarios de la tecnología CASE.

Los cambios en el ciclo de vida CASE

La tecnología CASE cambia y mejora el proceso de desarrollo y mantenimiento del **software** de varias formas. El cuadro 8.1 muestra las dife-

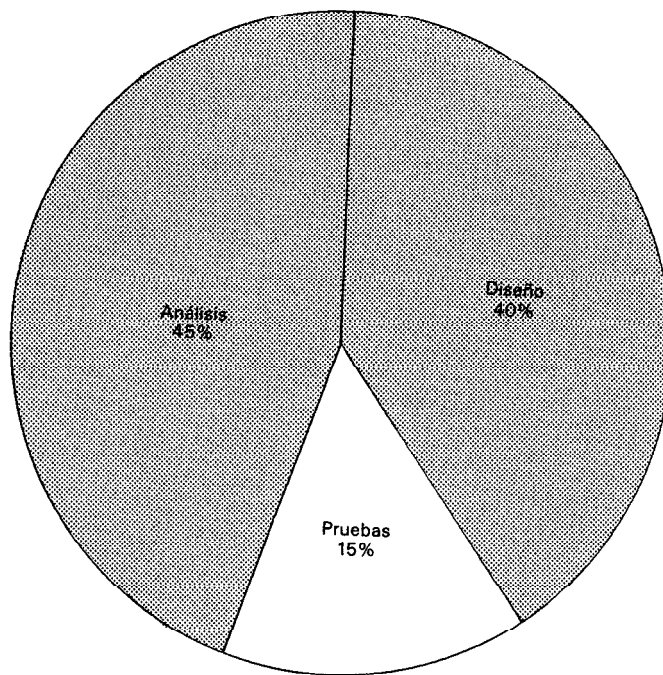


Figura 8.5. En el ciclo de vida CASE del software, la mayor parte del tiempo se emplea en el análisis y la parte de código virtualmente desaparece

rencias entre el desarrollo de **software** tradicional y el de la CASE mediante la automatización.

Primero, con la CASE, los programadores pueden dedicar más tiempo al análisis y el diseño y menos a codificación y a las pruebas. La codificación manual del programa se reemplaza por los generadores de código que automáticamente producen del ochenta al cien por ciento del código del sistema a partir de las especificaciones del diseño. Las especificaciones de diseño adoptan la forma de diagramas estructurados, que representan la estructura de los datos del programa, las entidades y las funciones de los procedimientos. Las especificaciones de diseño se pueden crear interactivamente en las estaciones de trabajo por los analistas de sistemas. Al introducir las especificaciones en las librerías automatizadas de la CASE, su corrección se puede comprobar y se pueden comparar con la información existente de los sistemas.

Segundo, la tecnología CASE cambia los métodos tradicionales de análisis. Los prototipos rápidos e iterativos reemplazan a los métodos manuales en la definición de los requerimientos de los usuarios (ver figura 8.6). Las herramientas CASE, como los generadores de pantallas y de informes y las de lenguaje de especificaciones ejecutables, se utilizan para construir rápidamente modelos prototipo del sistema. El modelo prototipo puede utilizarse para determinar los requerimientos del usuario y servir como especificación “viva” del sistema que elimina la necesidad de largos documentos de especificaciones textuales que debe escribir manualmente el analista de sistemas.

Cuadro 8.1. La CASE cambia el desarrollo de software

Desarrollo tradicional	Automatización del software
<ul style="list-style-type: none"> • Desarrollo tradicional • Énfasis en la codificación y en las pruebas • Codificación manual • Documentación manual • Pruebas de software • Mantenimiento de código 	<ul style="list-style-type: none"> • Énfasis en el análisis y en el diseño • Prototipo rápido e iterativo • Generación automática de código • Generación automática de la documentación • Comprobación automática del diseño • Mantenimiento de las especificaciones del diseño

Tercero, la tecnología CASE reduce considerablemente el esfuerzo de las pruebas del **software**. El análisis y comprobación automáticas de las especificaciones del diseño eliminan muchos de los errores debidos a las especificaciones incompletas o inconsistentes. La generación automática del código elimina los errores que podrían surgir del uso erróneo de los lenguajes de programación. Lo que queda es la prueba de nivel del sistema. El sistema puede ejecutarse eligiendo cuidadosamente los datos de en-

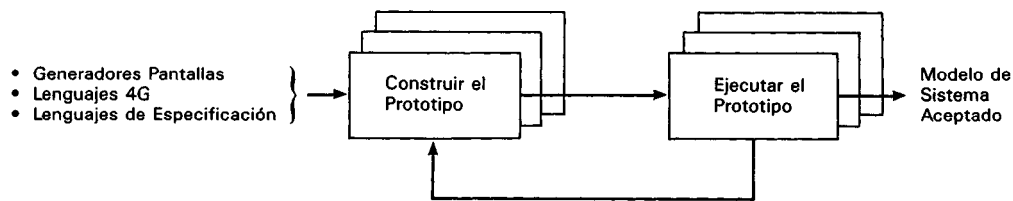


Figura 8.6. El prototipo es un método CASE de especificación que reemplaza los métodos basados-en-el-papel y manuales para definir los requerimientos de usuario

trada para las pruebas. Después, la salida producida por esas pruebas puede comprobarse para asegurar que todo el sistema ha funcionado correctamente. Las herramientas CASE ayudarán en la generación de datos de entrada para las pruebas y en el análisis de los resultados de salida.

Seguidamente, el mantenimiento del **software** se simplifica. Las librerías CASE de diseño y los generadores de código eliminan la necesidad de mantener el código de programa. El mantenimiento se realiza en nivel de diseño. Cuando es necesario un cambio en el programa, se cambia la especificación del diseño y el código se regenera desde la especificación modificada.

Finalmente, toda la formulación del desarrollo de **software** puede basarse en el concepto de reusabilidad. La reusabilidad de los componentes del **software** es la clave de un considerable incremento de la productividad del **software**. La tecnología CASE hace que la reutilización del **software** sea práctica. En lugar de construir cada nuevo sistema de **software** partiendo de cero, los programadores pueden reutilizar los componentes de **software** almacenados en las librerías del **workbench CASE**. Los programadores no sólo reutilizarán los módulos de código fuente, sino que también reutilizarán la planificación del proyecto, los modelos de prototipos, los modelos de datos y las especificaciones de diseño. La reusabilidad se tratará más adelante en el capítulo 13.

Problemas con el modelo CASE de ciclo de vida del software

La figura 8.7 muestra un modelo sencillo del ciclo de vida del **software** CASE. Obsérvese que la fase del prototipo ha reemplazado a la tradicio-

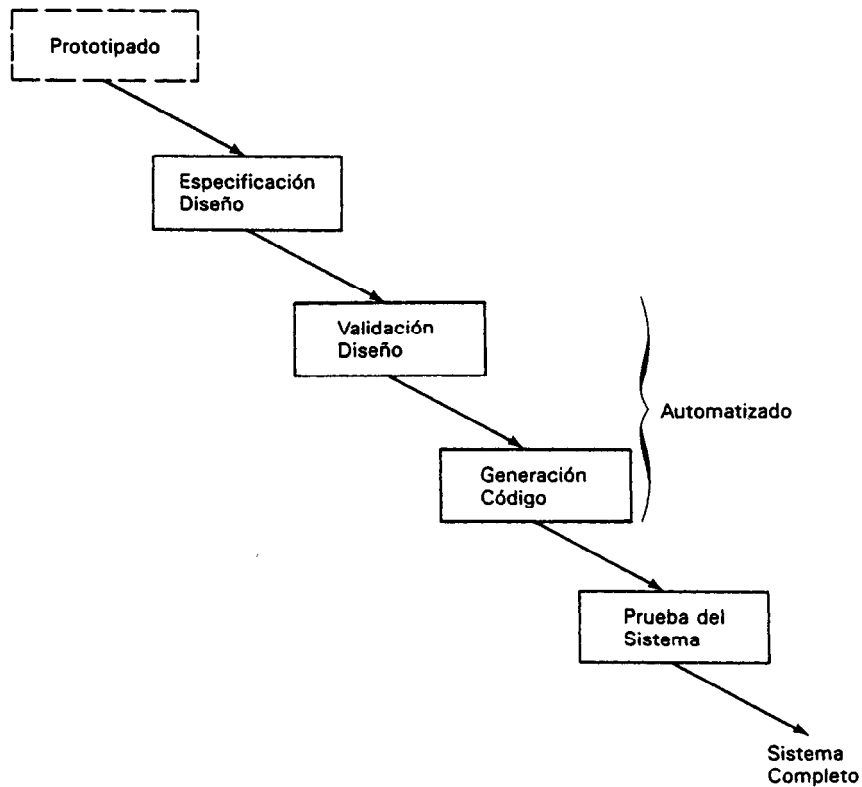


Figura 8.7. Una visión simple del ciclo de vida CASE del software, que muestra el empleo del prototipo y las fases automatizadas de la validación del diseño y de la generación del código

nal fase de análisis del sistema. Obsérvese también que todas las fases están soportadas por herramientas CASE automatizadas, las fases de comprobación del diseño y de generación de código son las fases más automatizadas del ciclo de vida.

Sin embargo hay algunos problemas y limitaciones con este modelo de ciclo de vida. Al igual que con el modelo tradicional mostrado en la figura 8.2, el modelo CASE mostrado en la figura 8.7 desfigura el proceso de desarrollo del **software** en la práctica. Ambos describen el proceso como secuencial de paso único en el cual las fases se siguen unas a otras. Pero en la práctica del desarrollo del **software** raramente es un simple proceso

secuencial. Más frecuentemente es un proceso iterativo donde las fases se repiten y el proyecto entero se recicla una y otra vez durante la vida del sistema. En proyectos menores, las fases se tratan informalmente y se permite mezclar unas con otras. En los proyectos grandes, en interés del tiempo, las fases se realizan en paralelo. Frecuentemente, a causa de los errores y omisiones las fases deben repetirse. En cada fase del ciclo de vida se obtiene nueva información y aclaraciones, haciendo inadecuadas las decisiones previas y obligando a los programadores a retroceder, reconsiderar sus diseños y repetir los pasos.

Afortunadamente, la tecnología CASE realmente reconoce que el desarrollo del **software** es un proceso iterativo en el cual los pasos se repiten y el proceso es reciclado repetidamente. Las herramientas CASE que automáticamente realizan el seguimiento, los cambios, el control y relacionan toda la información soportan los métodos de desarrollo de **software** iterativo.

Un segundo problema con dos de estos modelos simples de ciclo de vida es que ponen quizás demasiado énfasis en las primeras fases del ciclo de vida. Las formulaciones estructurada y CASE para el desarrollo presuponen un análisis y diseño muy detallados para lograr una calidad más alta a un precio más bajo. La experiencia ha demostrado que lo que pasa en las primeras fases del ciclo de vida afecta enormemente a la calidad del **software**. Sin embargo, la experiencia también ha demostrado que no es suficiente con centrarse en el análisis y el diseño. El mantenimiento es la fase más extensa del ciclo de vida y la mayoría de las veces es mucho más costosa que todo el desarrollo. Para controlar los problemas de mantenimiento y los costos, los programadores deben diseñar sistemas pensando en la facilidad de mantenimiento.

El modelo tradicional del ciclo de vida ignora las tareas del mantenimiento hasta llegar a esa fase del mantenimiento, y ya es demasiado tarde para incorporar la posibilidad de mantenimiento. Sin embargo, la tecnología CASE consigue un sistema de mantenimiento fácil porque puede mantenerse.

En el nivel de especificación de diseño, en lugar de en el nivel de código. Además, el cambio con el impacto de las herramientas de análisis CASE hace mucho más fácil y seguro cambiar los sistemas existentes.

Otro defecto importante de estos dos modelos simples de ciclo de vida es que no clasifican el modelo lógico de datos. Para la mayoría de los sistemas

de aplicaciones de negocios, el proceso de desarrollo del sistema deberá empezar con una planificación estratégica de la información de la organización y la construcción de los modelos lógicos de datos, al menos para la función de los negocios, como hace la ingeniería de la información (ver capítulo 3).

Así, un ciclo de vida CASE completo para los sistemas de aplicaciones de los negocios incluye una fase de planificación estratégica de la información seguida de una fase de modelado lógico de datos, los cuales preceden a la fase de prototipos mostrada en la figura 8.7. Obsérvese, sin embargo, que la planificación de información estratégica se hace una vez para todos los sistemas de aplicaciones del negocio a construir.

MODELO Y SOLO UNO

Los estándares asociados al ciclo de vida del software han adquirido fuerza de ley en muchas organizaciones. Solamente se permite un sólo modelo de ciclo de vida, pero, obviamente, se necesitan muchas variantes para soportar el desarrollo de los diferentes tipos de sistemas de **software**.

UNA NUEVA VISION DEL CICLO DE VIDA

Lo que se necesita es una nueva visión del ciclo de vida que admita diversos enfoques para construir los sistemas de **software**. El modelo tradicional automatizado con el soporte de las herramientas CASE resulta apropiado para los sistemas grandes, complejos y científicos cuyos requerimientos pueden definirse completamente en el principio del desarrollo. Para el desarrollo de sistemas de tiempo real puede añadirse un paso simulado como primera fase del ciclo de vida e incluir un paso de prototipos antes de implantar el sistema en su entorno físico.

Para la mayoría de las aplicaciones comerciales, el ciclo de vida podría comenzar con fases de planificación estratégica de la información y de modelado de los datos precediendo a la especificación del sistema. Después, puede seguir un prototipo de ciclo de vida en el cual el sistema evoluciona a través de la construcción de múltiples modelos o un ciclo de vida CASE. Para los programas reducidos puede ser apropiado

un ciclo de vida de cuarta generación en el cual el programa se crea utilizando un generador de aplicaciones.

En el pasado se estaba limitado a un solo modelo de ciclo de vida por falta de experiencia en el desarrollo del **software** y con las herramientas de **software**. Gracias a las herramientas CASE y a la tecnología CASE, ya no existe tal restricción. Limitar a los programadores a un modelo que no maximice el empleo de la automatización del **software** es anticuado y muy caro.

BIBLIOGRAFIA

1. Marvin Zelkowitz, Alan Shaw y John Gannon, *Principles of Software Engineering and Design*. Englewood Cliffs, Nueva Jersey: Prentice-Hall, 1979.
2. William Suydam, "CASE Makes Strides toward Automated Software Development", *Computer Design*, 1 enero, 1987, págs. 49-70.
3. G. Boehm, R. McClearn y D. Unfrig, "Some Experiences with Automated Aids to the Design of Large-Scale Reliable Software", *IEEE Transactions on Software Engineering*, Vol. SE-1, N.º 1, marzo 1975.

RELACION DE LA CASE CON OTRAS TECNOLOGIAS DE SOFTWARE

LA COMBINACION DE TECNOLOGIAS

¿A qué generación pertenece la tecnología CASE? La figura 9.1 muestra las cinco generaciones de tecnologías de **software** que existen actualmente. En el cuadro 9.1 se listan varios ejemplos de herramientas de **software** de las generaciones tercera, cuarta y quinta. La tecnología CASE no puede considerarse realmente como una nueva generación de la tecnología de **software**, más bien es una combinación de técnicas y herramientas de la tercera, cuarta y quinta generaciones. Además, la CASE no solamente combina, sino que integra los componentes de las técnicas y las herramientas.

Cuadro 9.1. Comparación entre herramientas

Herramientas de la tercera generación

- Compiladores.
- Editores de programas y librerías.
- Generadores y comparadores de datos para pruebas.
- Monitores y optimizadores del rendimiento.

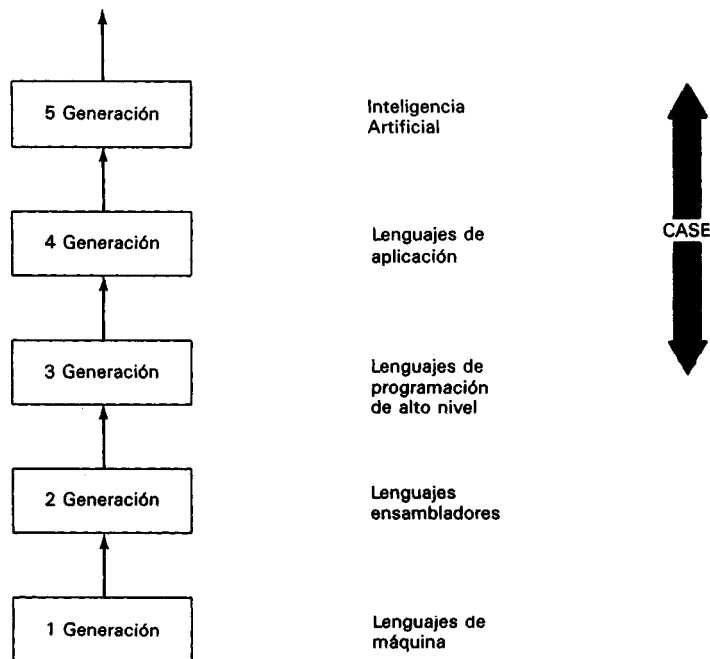


Figura 9.1. Actualmente hay cinco generaciones de tecnologías de software. La tecnología CASE es una combinación de las tecnologías de la tercera, cuarta y quinta generación

Cuadro 9.1. Comparación entre herramientas (continuación)

<p>Herramientas de la cuarta generación</p> <ul style="list-style-type: none"> • Lenguajes de cuarta generación. • Sistemas gestores de bases de datos. • Generadores de aplicaciones y de código. <p>Herramientas de la quinta generación</p> <ul style="list-style-type: none"> • Interruptores en lenguaje natural. • Reconocimiento del habla. • Sintetizadores de la voz. • Procesadores paralelos.

LA TECNOLOGIA DE CUARTA GENERACION

Cuando se oye el término “cuarta generación” generalmente nos viene a la mente la posibilidad de escribir programas en un código no procedimental. Aunque las tecnologías de cuarta generación (4G) empezaron a finales de los años setenta como herramientas generadoras de informes, las posibilidades de la cuarta generación se ampliaron en los años ochenta para incluir utilidades simples, como las de consulta para las bases de datos y lenguajes de programación de muy alto nivel. En el cuadro 9.2 se listan varias categorías de herramientas de la cuarta generación. Según James Martin, la cuarta generación puede describirse mejor como un nivel de capacidad que como una lista de categorías de herramientas de cuarto nivel [1]. Una herramienta 4G debe satisfacer un nivel de capacidad que proporcione:

1. Un incremento de diez a uno de la productividad en la fase de implantación de **software** sobre la utilización de las herramientas tradicionales de la segunda y tercera generación.
2. La regla del aprendizaje en dos días (es decir, un usuario puede aprender un subconjunto **útil** de la herramienta de cuarta generación en sólo dos días).

Cuadro 9.2. Categorías en la cuarta generación

	Usuarios finales	Programadores profesionales
Utilidades de consulta (query) simples		
Lenguajes de consultas complejas y de actualización		
Generadores de informes		
Lenguajes gráficos		
Herramientas de soporte de decisiones		
Generadores de aplicaciones		
Lenguajes de programación de muy alto nivel		

Categorías de las herramientas de cuarta generación

El cuadro 9.2 sugiere que las herramientas de 4G también pueden clasificarse en herramientas para usuario final y herramientas para el progra-

Cuadro 9.3. Cuarta generación frente a CASE

Lenguajes 4G	CASE
<ul style="list-style-type: none"> • Usuario final • Basados en ordenadores principales • Orientados por los SGBD • Propósito especial <ul style="list-style-type: none"> — Consulta (query) — Producción de informes — Proceso de transacciones — Tamaño pequeño • El proceso de desarrollo de aplicaciones asistido por menús • Productividad a través de la línea clásica del proceso de desarrollo (construcción con bloques preprogramados) • Sistemas genéricos • Destinados a la implantación 	<ul style="list-style-type: none"> • Profesional del desarrollo del software • Basado en PC • Orientados por las metodologías estructuradas • Propósito general <ul style="list-style-type: none"> — Software total — De pequeños a muy grandes • El proceso de desarrollo de la aplicación asistida por gráficos • Productividad a través del proceso automático de desarrollo • Sistemas particularizados • Destinados al ciclo de vida completo

sional del desarrollo de **software**. Además, las herramientas de 4G y las CASE difieren en su destino en los sistemas de aplicaciones. Las herramientas de 4G se utilizan principalmente para desarrollar aplicaciones pequeñas o medias (como producción de informes, utilidades de consulta o proceso de transacciones). Las herramientas CASE, sin embargo, son de propósito más general que se emplean en el desarrollo de sistemas de todo

- Los lenguajes de 4G son más fáciles de entender, porque no permiten las construcciones mal estructuradas.
- Muchos lenguajes de 4G están ligados a sistemas de gestión de bases de datos (SGBD) con diccionarios incorporados que ayudan a evitar errores relativos a los datos.
- Muchos lenguajes de 4G son autodocumentables.
- Los usuarios finales pueden escribir y mantener sistemas completos de aplicaciones de cuarta generación.

Por otra parte, las herramientas de 4G tienen algunas limitaciones serias, entre las que están:

- Muchas herramientas son SGBD o sistemas operativos específicos, lo que limita su portabilidad.
- Muchas herramientas de 4G carecen de la posibilidad de proporcionar actualización concurrente de los datos.
- Muchas herramientas de 4G carecen de la capacidad de recuperación/rearranque automática.
- La mayoría de las aplicaciones de cuarta generación son incapaces de manejar eficientemente un gran volumen de transacciones.
- Muchos programas de 4G son incompatibles con sistemas existentes escritos en lenguajes de segunda y tercera generación.
- Los lenguajes de 4G no están estandarizados.
- Las herramientas de 4G producen un código menos eficiente que los lenguajes de generaciones anteriores.
- Los programas de cuarta generación utilizan una cantidad excesiva de recursos de ordenador.

Otra limitación importante es que las herramientas de 4G soportan solamente la fase de implantación del ciclo de vida y no soportan las críticas fases de análisis y diseño, donde comienzan la mayoría de los errores. Además, los programas de 4G son difíciles de mantener si contienen una gran proporción de código procedimental.

La cuarta generación frente a CASE

El cuadro 9.3 compara las diferencias principales entre las herramientas de 4G y las CASE. Primero, mientras que el principal usuario de las herramientas 4G es el usuario final, en las herramientas CASE es el profe-

Cuadro 9.3. Cuarta generación frente a CASE

Lenguajes 4G	CASE
<ul style="list-style-type: none"> • Usuario final • Basados en ordenadores principales • Orientados por los SGBD • Propósito especial <ul style="list-style-type: none"> — Consulta (query) — Producción de informes — Proceso de transacciones — Tamaño pequeño • El proceso de desarrollo de aplicaciones asistido por menús • Productividad a través de la línea clásica del proceso de desarrollo (construcción con bloques preprogramados) • Sistemas genéricos • Destinados a la implantación 	<ul style="list-style-type: none"> • Profesional del desarrollo del software • Basado en PC • Orientados por las metodologías estructuradas • Propósito general <ul style="list-style-type: none"> — Software total — De pequeños a muy grandes • El proceso de desarrollo de la aplicación asistida por gráficos • Productividad a través del proceso automático de desarrollo • Sistemas particularizados • Destinados al ciclo de vida completo

sional del desarrollo de **software**. Además, las herramientas de 4G y las CASE difieren en su destino en los sistemas de aplicaciones. Las herramientas de 4G se utilizan principalmente para desarrollar aplicaciones pequeñas o medias (como producción de informes, utilidades de consulta o proceso de transacciones). Las herramientas CASE, sin embargo, son de propósito más general que se emplean en el desarrollo de sistemas de todo tipo y tamaño. Además, mientras que una herramienta de 4G da como resultado una solución genérica, el empleo de las herramientas CASE da como resultado una solución particularizada al problema del usuario.

Tanto las herramientas de 4G como las CASE son de alta productividad. Sin embargo, las herramientas de 4G consiguen el incremento de la

productividad mediante la estructuración del proceso de desarrollo con funciones comunes preprogramadas, mientras que las herramientas CASE logran el incremento de la productividad automatizando muchas tareas de desarrollo de **software**.

Finalmente, las herramientas de 4G y las CASE difieren en que las primeras se utilizan principalmente durante las fases de implantación del ciclo de vida del **software**, mientras que muchas herramientas CASE se utilizan en todo el ciclo de vida, incluyendo el mantenimiento y la gestión del proyecto.

LA TECNOLOGIA DE QUINTA GENERACION

Cuando se oye el término tecnología de quinta generación (5G) se piensa en sistemas inteligentes; esto es, sistemas de **software** capaces de realizar tareas normalmente asociadas a la inteligencia humana. Algunos ejemplos de la tecnología de 5G son la robótica, el procesamiento en lenguaje natural, la demostración automática de teoremas, la ingeniería del conocimiento y los sistemas expertos.

Los sistemas expertos

Según Feigenbaum, un **sistema experto** es un “programa inteligente de ordenador que emplea procedimientos de conocimiento e inferencias para resolver problemas que requieren una considerable experiencia humana para su solución” [2]. Las características de un sistema experto se exponen en el cuadro 9.4. Obsérvese que una característica predominante de un sistema experto es la captura de experiencia en un dominio específico y limitado. No es una herramienta de resolución de problemas de propósito general.

Como se observa en la Figura 9.2, un sistema experto tiene tres componentes:

1. Un interfaz de usuario en lenguaje natural.
2. Una base de conocimientos (o hechos) y heurísticas (reglas empíricas informales) formalizados en un dominio específico de resolución de un problema que capturan el razonamiento empleado por el experto humano para resolver un problema o realizar una tarea.

Cuadro 9.4. Las características de un sistema experto

- Resuelve problemas en un área especializada simulando al experto humano.
- Explica los resultados y el razonamiento seguido.
- Trabaja con datos incompletos, imprecisos y cualitativos.
- Tiene la capacidad de razonar, esto es, realizar deducciones e inferencias lógicas empleando las reglas de la lógica formal y/o las reglas de producción.
- Tiene la capacidad de añadir fácilmente nuevas reglas y modificar el razonamiento.
- Emplea métodos de búsqueda heurísticos en lugar de algoritmos para enfocar la solución del problema.
- Sabe cuando detenerse en una regla.
- Puede considerar simultáneamente varias hipótesis competitivas.

3. Un motor de inferencias que emplea el conocimiento para resolver problemas o realizar tareas, controlando la alimentación de reglas desde la base de conocimiento y la creación de nuevas reglas.

Algunos sistemas expertos reemplazan a los expertos, otros ayudan a los expertos en la resolución más efectiva de los problemas al ampliar su base de conocimientos y experiencia.

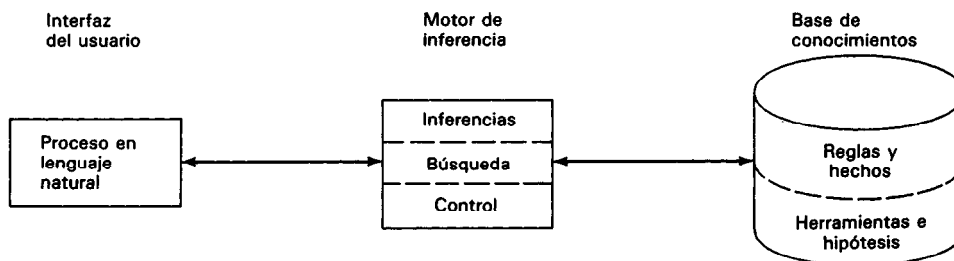


Figura 9.2. Un sistema experto tiene tres componentes: el interfaz en lenguaje natural, el motor de inferencia y la base de conocimientos

La quinta generación y la CASE

Las herramientas CASE basadas en los sistemas expertos pertenecen a las tecnologías de la 5G y en su mayor parte pertenecen al futuro, no al presente. Un tipo de herramientas CASE basada en los sistemas expertos que se emplearán en el futuro para desarrollar los sistemas de **software** es el **conductor de metodología**, que tiene conocimientos y experiencia en el desarrollo y el mantenimiento de los sistemas de **software**. Tiene inteligencia suficiente para tomar decisiones sobre como realizar el proceso del ciclo de vida del **software**. El conductor de metodología se trata más ampliamente en el capítulo 12.

LA COBERTURA DEL CICLO DE VIDA

La figura 9.3 muestra cuales son las fases del ciclo de vida soportadas por los diversos tipos de herramientas de **software**. Así, las herramientas de 4G soportan algunas tareas de diseño, los prototipos y principalmente las fases de implantación. Mientras algunos juegos de herramientas CASE soportan principalmente las primeras fases del ciclo de vida, otras soportan las últimas fases y la gestión del proyecto. Por otra parte, los generadores de código CASE soportan principalmente el diseño, los prototipos y la implantación, pero se han ampliado para soportar más plenamente la fase

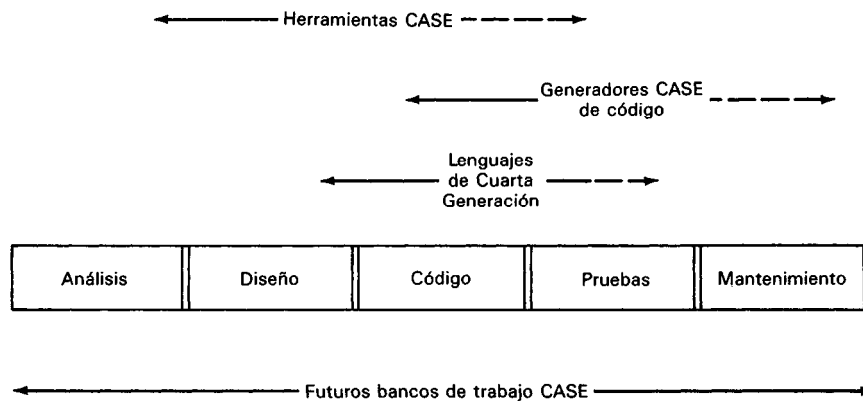


Figura 9.3. Ya que las herramientas CASE ofrecen una cobertura total del ciclo de vida del software, han reemplazado a las herramientas de Cuarta Generación

de diseño. La forma en que muchos juegos de herramientas CASE de análisis/diseño y los generadores de código CASE se han ampliado para proporcionar un soporte completo al ciclo de vida ha sido por la combinación de estos dos tipos de herramientas CASE en un banco de trabajo CASE integrado. Otra forma de conectar estos dos tipos de herramientas es mediante los distintos tipos de interfaces.

La figura 9.3 también muestra cómo los diferentes tipos de herramientas de **software** se solapan, cubriendo las mismas fases del ciclo de vida. Cuando sucede esto las herramientas se hacen competitivas y el usuario debe elegir la herramienta más apropiada para el trabajo. Como los juegos de herramientas y generadores de código CASE se integran en un banco de trabajo de herramientas compatibles diseñado para trabajar al unísono, las herramientas CASE son muy competitivas con las herramientas de 4G por el liderazgo de la alta productividad.

VENTAJAS DE LA CASE SOBRE LA CUARTA GENERACION

La mayor ventaja de la CASE sobre la cuarta generación es que el nivel de integración que une las herramientas CASE hará que éstas sean más fáciles de utilizar y de aprender a utilizar que las herramientas de 4G. Además el interfaz gráfico de usuario empleado por la mayoría de las herramientas CASE es más amigable para el usuario que los menús orientados a las herramientas de la cuarta generación.

Otra ventaja de CASE sobre la cuarta generación es el depósito CASE en el que se almacena y gestiona toda la información del sistema durante todo el ciclo de vida, desde la fase del análisis hasta el mantenimiento. El depósito CASE promueve la información compartida, reduce la necesidad de reintroducir información y protege la integridad de la información.

Otra ventaja más es que los generadores CASE de código producen código fuente y código objeto, que a menudo es más eficiente, más portable y más compatible con los sistemas existentes que las herramientas de cuarta generación generadas por el código.

Finalmente, las herramientas CASE superan a las herramientas de 4G porque son más de propósito general en relación con las herramientas de 4G. Las herramientas CASE se pueden emplear en el desarrollo de sistemas pe-

queños, medianos, grandes y muy grandes, mientras que las herramientas de 4G son más útiles en el desarrollo de sistemas pequeños o medianos.

Todas estas ventajas ofrecidas por las herramientas CASE las hacen no solamente competitivas, sino, además, muy amenazadoras para el futuro de las herramientas de 4G. Las prestaciones que hacían a las herramientas de 4G superiores a las de tercera generación, como los generadores de pantallas y de informes, los prototipos y la generación de bases de datos ya han sido incorporadas a los bancos de trabajo CASE. Además, muchas de las capacidades de las herramientas de la tercera generación se han incorporado a las herramientas CASE. A menos que las herramientas de 4G se mejoren con la inclusión de capacidades para soportar el análisis y el diseño, serán reemplazadas en favor de unas herramientas CASE más potentes, más amigables y mejor integradas. La excepción pueden ser las herramientas de 4G, para usuario final utilizadas sobre todo por los centros de información.

RESUMEN

La tecnología CASE es una combinación de técnicas y prestaciones de las tecnologías de la tercera, cuarta y quinta generación. El cuadro 9.5 predice lo que sucederá a las tecnologías de la tercera, cuarta y quinta generación cuando en el futuro el empleo de la CASE esté totalmente extendido.

Cuadro 9.5. ¿Qué sucederá con las tecnologías de la tercera, cuarta y quinta generación

¿Qué sucederá con la tecnología de la tercera generación?

Los sistemas escritos en lenguajes de tercera generación (especialmente COBOL) no dejarán de escribirse en un corto plazo. Sin embargo, la carga de mantener estos sistemas se aliviará con la ayuda de las tecnologías CASE. Además su migración a nuevas tecnologías y a otros entornos operativos será asistido por la tecnología CASE.

Cuadro 9.5. ¿Qué sucederá con las tecnologías de la tercera, cuarta y quinta generación? (continuación)

Los sistemas de tercera generación serán reestructurados y mantenidos con la asistencia de los juegos de herramientas CASE de mantenimiento. Las herramientas del análisis de código, de documentación, de reestructuración y de conversión serán añadidas al sistema CASE.

¿Qué sucederá con la tecnología de cuarta generación?

En los años noventa, la tecnología CASE reemplazará a las tecnologías de 4G como la aplicación más avanzada de la metodología de desarrollo. La CASE producirá un código más eficiente y más portable y proporcionará una mayor reutilización de las aplicaciones producidas.

Las funciones de cuarta generación, como los generadores de pantallas e informes y las tecnologías de sistemas gestores de bases de datos serán absorbidas en los sistemas CASE para promocionar al máximo la distribución de la información. Un depósito CASE central será el mecanismo para compartir, controlar y gestionar los recursos del ordenador.

Los usuarios finales continuarán accediendo a las bases de datos, generando informes, realizando análisis y modelos y produciendo hojas de cálculo con la ayuda de las herramientas de 4G dirigidas al usuario final. Sin embargo, la utilización de los recursos del ordenador estará controlada y sujeta a las normas corporativas mediante la coparticipación de los modelos de datos, las definiciones de los datos, los modelos de procesos y los programas almacenados en el depósito CASE.

¿Qué sucederá con la tecnología de la quinta generación?

En los años noventa, las tecnologías de 5G se añadirán a la tecnología CASE. El proceso del lenguaje natural, el reconocimiento del habla y los sintetizadores de voz se agregarán al interfaz del usuario del sistema CASE. Los sistemas expertos, como los sistemas tutoriales, los sistemas de diagnóstico y los diseñadores inteligentes de **software** se incorporarán a los sistemas CASE.

BIBLIOGRAFIA

1. James Martin, *Fourth-Generation Languages*, Vol. I. Englewood Cliffs, Nueva Jersey: Prentice-Hall, 1985.
2. I. Zualkerman, W. T. Tsai y D. Volovik, "Expert Systems and Software Engineering: Ready for Marriage?" *IEEE Expert*, Vol. 1. N.º 4, invierno de 1986, págs. 24-31.

PARTE 4

**EL DESARROLLO DEL
SOFTWARE EN LOS AÑOS NOVENTA**

LAS CARACTERISTICAS DE LA AUTOMATIZACION DEL SOFTWARE

MAS AUTOMATIZACION DEL SOFTWARE

¿Cómo cambiará el desarrollo del **software** en los años noventa? Ciertamente, que la automatización tendrá un papel más importante que en el pasado. La disponibilidad de potencia de cálculo barata acentuará la necesidad de más **software**. La única forma realista de satisfacer esta necesidad es simplificar y automatizar los procesos de desarrollo del **software**.

Es de conocimiento general que no se puede soportar una gran carga de sistemas en espera de desarrollo. Una forma de reducir esta carga de espera es aumentar el número de personas que desarrollan sistemas. Enseñando a más personas el desarrollo de sistemas, haciendo los sistemas de ordenador más fáciles de utilizar e implicando a los usuarios en el proceso de desarrollo del **software**, se podrán crear más sistemas. Esta es la estrategia del usuario final. Aunque es una solución meritoria, no funciona demasiado bien. La implicación del usuario final y las herramientas de cuarta generación se introdujeron a principios de los años ochenta, pero todavía no han tenido un impacto significativo en la reducción de la carga de **software**, porque no pueden incrementar significativamente la productividad del **software** y porque trabajan solamente en una clase limitada de tipos de sistemas.

Otra solución al problema de la sobrecarga es enseñar al ordenador a construir sistemas de **software**; esto es, la automatización del desarrollo del **software**. Muchas tareas del desarrollo que son demasiado laboriosas o detalladas para los humanos, podrían ser buenas candidatas a la automatización. Algunos ejemplos son recordar las reglas sintácticas de un lenguaje de programación en particular o la estructura de los comandos de un sistema operativo, y la depuración de errores del sistema. También las tareas del **software** que requieren el conocimiento y la experiencia de los expertos serían buenos candidatos a la automatización.

Probablemente, en los años noventa el ordenador no trabajará exclusivamente por sí solo para desarrollar **software**, sino que se usará la fór-

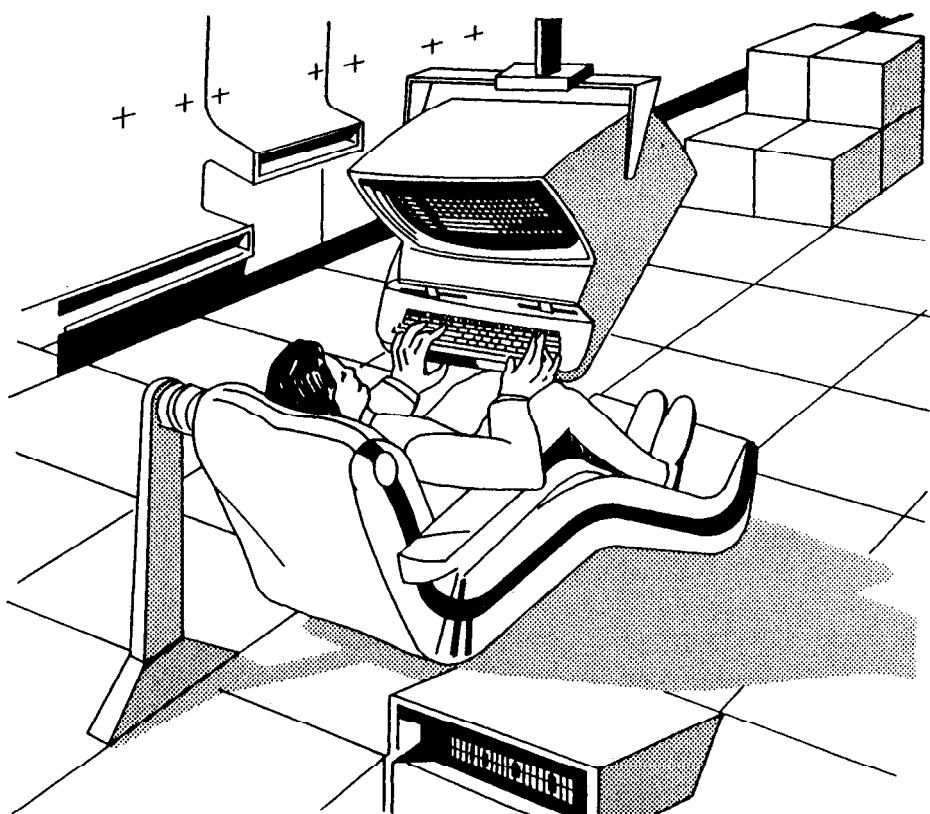


Figura 10.1. El entorno de desarrollo de software en los años noventa será un ordenador personal potente e inteligente

mula de un equipo mixto. Algunos miembros del equipo serán programadores humanos, y otros serán herramientas inteligentes cuyo trabajo sería asistir al programador. Más que reemplazar a los profesionales, las herramientas inteligentes aumentarán su capacidad para desarrollar sistemas. Será como si cada programador se reúne con un grupo de expertos —en ingeniería de **software**, en gestión de proyectos e incluso en el área de las aplicaciones— que lo aconsejan, toman decisiones, realizan tareas complicadas y guían al programador en el proceso de construir y mantener el sistema.

IMAGINE DE NUEVO

Imagine de nuevo que está desarrollando **software** en su propia estación de trabajo dedicada (ver figura 10.1). Pero esta vez vamos a trasladarnos al futuro, a los años noventa. Ahora deje volar su imaginación e imagine el entorno ideal de soporte del **software** y que está dedicado a satisfacer las necesidades de desarrollo y mantenimiento del **software**, sus caprichos y deseos tan pronto se acomode frente a la poderosa e inteligente “máquina pensante”.

Su entorno de soporte de **software** reside en un potente ordenador personal conectado a un ordenador principal, donde las bases del conocimiento le enlazan con el resto de la organización y con el mundo exterior. Usted se comunica con su estación de trabajo señalando objetos en la pantalla o simplemente hablándole. Se usa el lenguaje natural y la estación de trabajo responderá también en lenguaje natural, a veces con un sintetizador de la voz.

Así pues, en los años noventa, el profesional del desarrollo y la herramienta inteligente formarán un equipo para obtener sistemas de **software** de forma que superen con creces la productividad y calidad conseguidas en los años ochenta. Cada paso en la producción del **software** estará asistido por el ordenador, con una serie de herramientas de **software** disponibles en la estación de trabajo individual. Tales herramientas asistirán a los programadores en todos los aspectos de su trabajo y realizarán completamente ciertas tareas. Le aconsejarán sobre lo que tiene que hacer a continuación, le recordarán lo que ha olvidado, descubrirán lo que es

erróneo, explicando el motivo y realizarán automáticamente muchas tareas rutinarias sin preocupar al programador humano.

LOS CAMBIOS EN EL PROCESO DEL SOFTWARE

La estación de trabajo del futuro estará equipada con herramientas potentes y sofisticadas técnicamente mucho más avanzadas que las posibilidades de las herramientas de **software** actuales. Con tales adelantos se producirá un gran cambio en el proceso de desarrollo del **software**, que será un proceso muy automatizado. Las características de la automatización del software son (ver cuadro 10.1):

- Un entorno muy interactivo y sensible dedicado al profesional individual.
- Una especificación densa del proceso de desarrollo caracterizado por la creación rápida de prototipos para reemplazar las especificaciones sobre el papel.
- Comprobación automática y continua de la consistencia, integridad y corrección realizada según se va desarrollando el sistema.

Cuadro 10.1. Las características de la automatización del software

- | |
|--|
| <ul style="list-style-type: none">• Entorno muy interactivo y sensible.• Procesos de especificación densa con prototipos rápidos.• Comprobación automática y continua.• Generación automática de código y de documentación a partir de las especificaciones de diseño.• Sistemas expertos automáticos para guiar el proceso del software. |
|--|

- Generación automática de código y de la documentación a partir de diagramas gráficos que representan el diseño del sistema.
- Librerías automáticas de módulos reutilizables de **software** que aceleren el proceso de construcción con el uso de componentes reutilizables de **software**.
- Herramientas inteligentes que utilicen la base de conocimientos e interfaces en lenguaje natural.
- Sistemas expertos automáticos incorporados al entorno de desarrollo para guiar el proceso del **software**.

LOS CAMBIOS EN LAS HERRAMIENTAS DE SOFTWARE

Muchas herramientas utilizadas actualmente, como los lenguajes de especificación, herramientas de diagramación, herramientas de prototipos, generadores, diccionarios y sistemas gestores de bases de datos forman el núcleo del entorno de soporte de **software** de los años noventa. Sin embargo, estas herramientas mejorarán de dos aspectos muy importantes:

1. La integración.
2. La inteligencia.

Primero, las herramientas del entorno que soporta el **software** no serán solamente una colección de herramientas (ver figura 10.2), sino que serán un conjunto de herramientas **muy integradas**. Estarán unidas por un interfaz de usuario común con un lenguaje común —probablemente un subconjunto de lenguajes naturales— para simplificar su uso. Estarán diseñadas no solamente para coexistir, sino para saber unas de otras y llamarse y alimentarse recíproca y automáticamente, a fin de cumplir con las tareas que se les han sido encomendadas.

Segundo, en torno a estas herramientas se añadirá un **shell** (entorno) inteligente que aisle al usuario de todo lo relacionado con las diferencias entre las herramientas individuales y de los detalles técnicos (ver figura 10.3). El **shell** inteligente proporcionará:



Figura 10.2. El entorno de soporte de software de los años noventa será un conjunto de potentes herramientas altamente integradas y diseñadas para llamarse y alimentarse automáticamente entre ellas

- Comunicaciones en lenguaje natural.
- Bases de conocimientos que contienen conocimiento general de la ingeniería de **software** e información del dominio específico de la aplicación.
- Un conjunto de expertos automáticos de desarrollo del **software** (por ejemplo, expertos en el área de aplicación, expertos en el diseño de **software**, expertos en la detención y corrección de errores).

Las herramientas de **software** actuales son muy potentes pero “ton-tas”. Las herramientas de los años noventa serán potentes e inteligentes. Esta inteligencia se deberá a la aplicación de tecnologías avanzadas como el procesamiento de lenguajes naturales, reconocimiento del lenguaje natural, reconocimiento de la voz, sintetización de la voz, ingeniería del conocimiento, procesamiento en paralelo y animación de los programas.

Por ejemplo, la animación de programas es una técnica avanzada para el análisis de los programas. Además de proporcionar una visión estática del programa y de la estructura de los datos (diagramas estructurados), las herramientas de análisis del programa también proporcionarán una visión dinámica de la ejecución del programa para una mejor comprensión del

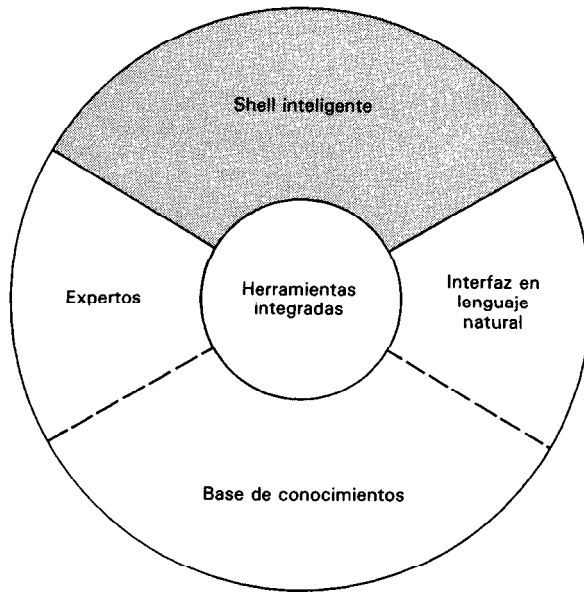


Figura 10.3. Las herramientas de software de los años noventa rodeadas de un Shell inteligente que liberarán al usuario de las preocupaciones técnicas

comportamiento del **software**. Los animadores de programas se emplearán para el seguimiento de los cambios de valor de los datos durante la ejecución del programa y para mostrar el flujo de control visualizando las pilas de ejecución del programa en curso.

EL SHELL INTELIGENTE

Como se muestra en la figura 10.4 el **shell** inteligente consta de tres componentes:

1. El **entorno habitable** que es un interfaz inteligente de usuario [1].
2. El **conductor de metodología**, que es un sistema experto que proporciona unas guías metodológicas durante el ciclo de vida del **software**.

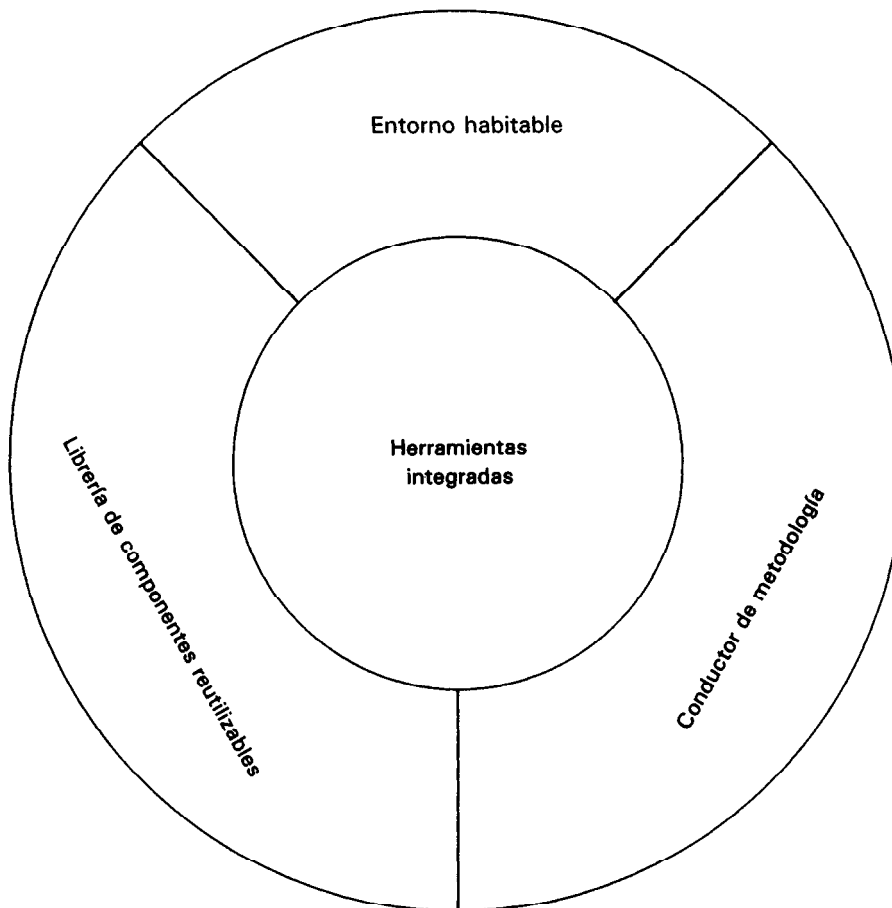


Figura 10.4. El Shell inteligente rodeando a las herramientas de software, consta de tres partes: (1) el entorno habitable, (2) el conductor de metodología y (3) la librería de componentes reutilizables

3. La **librería** de partes reutilizables y un sistema inteligente gestor de la librería, que eleva la **reutilización** al estado de estrategia dominante del desarrollo del **software**.

En los capítulos siguientes se estudiará cada uno de estos componentes con más detalle. En los años noventa llegará una nueva ola de avances tecnológicos, que traerá nuevas herramientas y técnicas, así como una mejora substancial de las técnicas y herramientas existentes en la actualidad.

BIBLIOGRAFIA

1. D. Gleeman y J. S. Brown, editores, *Intelligent Tutoring Systems*.
Londres: Academic Press, 1982.

EL ENTORNO HABITABLE

EL INTERFAZ DEL USUARIO

En la figura 11.1 se divide un **banco de trabajo CASE** en tres componentes básicos. El componente **front-end** pone el énfasis en las herramientas gráficas y en las de prototipos rápido para automatizar el análisis y el diseño. El depósito de información, junto con su sistema de gestión de la información, captura todos los productos de **software** y la información del proyecto de software. Finalmente, el componente **back-end** proporciona el programa en código, la generación de la base de datos y las utilidades de pruebas. Los tres componentes deben estar presentes y completamente integrados en el **banco de trabajo CASE** en espera de lograr el prometido incremento de la productividad.

Además, hay un cuarto componente que es al menos tan importante como los otros tres para el incremento de la productividad del profesional del desarrollo. Este cuarto componente es el **interfaz del usuario**, que enlaza al profesional del desarrollo con las herramientas del **banco de trabajo**. Aunque muy a menudo se descuida este componente, su impacto en la productividad no debería ser subestimado, ya que en última instancia es el que determina si una herramienta se utiliza o no en la práctica. Según Smith [1, págs. 297-313]:

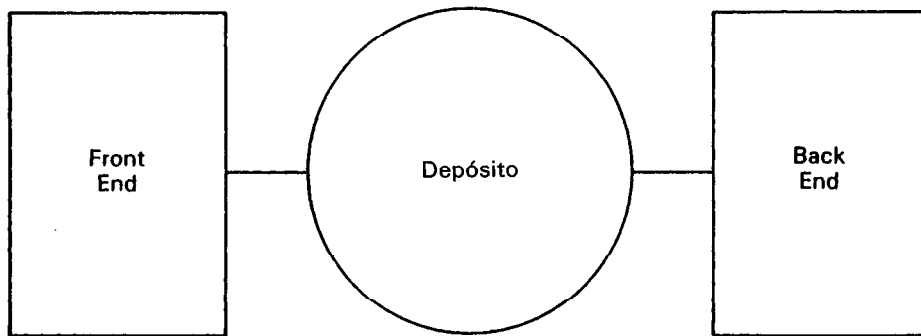


Figura 11.1. El banco de trabajo (workbench) consta de tres componentes básicos: (1) el front-end para soportar el análisis y el diseño; (2) el depósito para almacenar toda la información del sistema; y (3) el back-end para soportar la implantación del programa

“Los factores más importantes que afectan a lo extendido que estará el uso de un ordenador, serán un coste reducido, el incremento de la funcionalidad, el aumento de la disponibilidad y del servicio, y quizás el más importante de todos, el progreso en el diseño del interfaz del usuario. Los tres primeros son necesarios, pero no suficientes, para un empleo extendido. Reducir los costes permitirá comprar ordenadores, pero mejorar los interfaces de usuarios permitirá a las personas utilizar los ordenadores”.

El entorno habitable reconoce la importancia del interfaz del usuario en las herramientas de **software**, al enfatizar características tan importantes como el añadir la comprensión humana, la facilidad de uso y la facilidad de aprendizaje.

MAS ALLA DE LOS ENTORNOS CASE

Los entornos habitables tienen unas capacidades que van mucho más allá de las ofrecidas por los entornos **banco de trabajo** CASE de finales de los años ochenta. Pueden realizar muchas tareas de **software** por sí mismos sin molestar al usuario. Pueden enseñar al usuario y aprender de él. Pueden incrementar su habilidad para construir sistemas al crecer y cambiar según aumenta su experiencia en la construcción de sistemas. Pueden

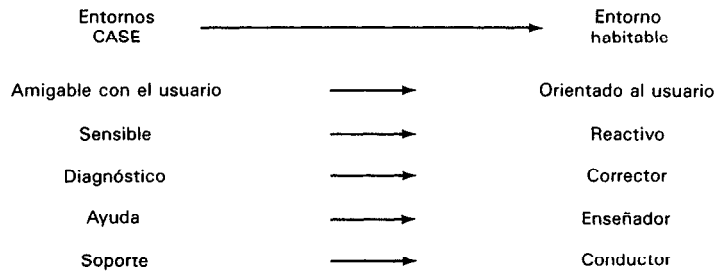


Figura 11.2. El entorno habitable va más allá de amigable con el usuario transformando el interfaz de usuario en un entorno inteligente con características tales como orientado al usuario, reactivo, corrector, instructor y conductor

explicar su comportamiento mediante conceptos fundamentales de software. Finalmente, pueden ofrecer consejos sobre metodología de **software**.

Los entornos CASE pueden caracterizarse como amigables, sensibles, diagnosticadores, consejeros y sustentadores. Los entornos habitables van más allá de estas cualidades, al añadir las características de centralizados, reaccionarios, correctores, instructores y directores (ver figura 11.2). Los entornos habitables transforman el interfaz de usuario en un compañero amigable e inteligente con quien el usuario puede compartir su experiencia en el desarrollo y mantenimiento del **software**.

AMIGABLE CON EL USUARIO

La característica de amigable con el usuario está, por supuesto, en la cima de las características de un interfaz con los humanos bien diseñado. Se considera el avance clave de las herramientas de 4G sobre las de tercera generación.

Un estilo “amigable con el usuario” tiene las siguientes características:

- Fácil de usar.
- Robusto.

- Colaborador.
- Detector y evitador de errores.
- Ampliable.
- Fácil de aprender.
- Adaptable.

Si un sistema es o no amigable depende de su estilo, y el estilo se considera normalmente una cuestión de preferencia personal y una marca de individualidad. Los diferentes sistemas (productos) se exponen con diferentes estilos. Por ejemplo, el Lotus 123 tiene su estilo particular, UNIX tiene un estilo diferente e Interlisp tiene otro estilo. Pero cualesquiera que sean las particularidades, siempre hay elementos comunes a todo buen estilo. Un buen estilo es simple, consistente y cumple con los estándares convenidos. Sus reglas no son tan complicadas como para empantanarse en la comprensión de la forma y no llegar nunca a alcanzar el contenido. Por encima de todo, clarifica, no oscurece, lo que se está comunicando. Si el estilo es bueno, desaparecerá gran parte de la dificultad de comprender el contenido.

Fácil de usar

Un sistema que es **fácil de usar**, es **evidente** al tiempo que **transparente**. Es evidente, porque sus usuarios pueden contar con la intuición para saber como se espera que trabaje el sistema. Como explica Stahl: “El **software** intuitivo le sienta muy bien al usuario que tan sólo debe acomodarse en su silla, realizar unas cuantas sugerencias acertadas y marcharse” [2].

Los sistemas evidentes tienen comandos con nombres significativos, indicadores (**prompts**) autoexplicativos, mensajes de error descriptivos y rutinas de ayuda para proporcionar información más detallada. Los sistemas con **manipulación directa** son un buen ejemplo de sistemas evidentes [3]. Todo se representa visualmente en la pantalla.

Los sistemas transparentes no se interponen en la forma de realizar las tareas [4]. En un sistema transparente, los aspectos subyacentes del ordenador son invisibles para el usuario, hay un procedimiento de introducción fácil y el diálogo hombre/máquina se conduce con un mínimo retraso y un máximo de flexibilidad.

Robusto

Un sistema **robusto** es a prueba de usuario, lo que significa que a efectos prácticos es imposible que el usuario pueda deteriorar el sistema. No hay virtualmente nada que pueda hacer el usuario para causar una terminación anormal del sistema o un bloqueo que rechace más comandos, incluidos EXIT o QUIT (salida) [5]. Un sistema robusto responde a todas las entradas de usuario, independientemente de si son o no correctas. Esto requiere una comprobación exhaustiva de las entradas y lógica del usuario para tratar tanto las entradas correctas como las incorrectas.

Fácil de aprender

Zloof cita Query-By-Example, un lenguaje de consulta a las bases de datos, como ejemplo de sistema **fácil de aprender** a utilizar [6]. Query-By-Example permite al usuario acceder a una base de datos con muy escasos conocimientos acerca de los registros o campos. Al usuario se le presenta en la pantalla del ordenador un formato de tabla (ver figura 11.3). Para formar una consulta, el usuario simplemente rellena el nombre de la tabla y los nombres de la tabla y de los campos como cabeceras de la tabla. Después, el sistema responde a la consulta rellenando las columnas de la tabla con la información solicitada por el usuario. En menos de tres horas de instrucción cualquier usuario debería poder crear una consulta relativamente complicada.

Query-By-Example es fácil de aprender a utilizar, porque sólo son necesarios unos pocos símbolos, unas pocas reglas de sintaxis y unos pocos conceptos básicos para que cualquier usuario pueda empezar a formular consultas.

Adaptable

Un sistema adaptable está diseñado para ser manejado por diferentes tipos de usuarios. La figura 11.4 muestra cinco tipos de usuarios clasificados por su nivel de experiencia [7, págs. 130- 138]. La primera clase, evidentemente, el aprendiz, apenas tiene experiencia; la quinta clase, muy experto, es la de mayor experiencia. Evidentemente, las diferentes clases de usuarios mantienen una relación interactiva muy diferente con el sistema. El aprendiz

# Empleado	Nombre	Sueldo	Departamento

Figura 11.3. El Query-By-Example es un lenguaje de consulta (query) muy fácil en el cual las consultas (query) se forman rellenando los campos de una tabla

necesita guía e instrucción constantes, probablemente cometa muchos errores e intente solamente tareas simples utilizando un reducido subconjunto de los comandos disponibles. El experto, por otra parte, necesita pocas instrucciones y explicaciones del sistema, investiga la forma más corta y más eficiente para realizar una tarea y en ocasiones necesita cambiar o ampliar las funciones del sistema.

Además, debe ser evidente que el sistema tendrá que interactuar de forma distinta con las diferentes clases de usuarios. Para los usuarios de las dos primeras clases, el sistema deberá llevar la iniciativa y ser explícito. Deberá proporcionar al usuario guías y orientación de trabajo para responder a preguntas como:

- ¿Dónde estoy?
- ¿De dónde vengo?
- ¿A dónde puedo ir desde aquí?
- ¿Qué puedo hacer ahora?
- ¿Cómo puedo conseguir ayuda?
- ¿Cómo puedo salir?

El mejor tipo de asistencia es normalmente un ejemplo que el usuario pueda seguir. Debe comunicarse al usuario el éxito de cada comando (por ejemplo, el FICHERO DE EMPLEADOS HA SIDO GUARDADO). Cuando se comete un error, el sistema debe responder con una explicación de lo que está equivocado y de cómo corregirlo (por ejemplo, EL NOMBRE DE FICHERO SOLO PUEDE CONTENER LETRAS Y DIGITOS).

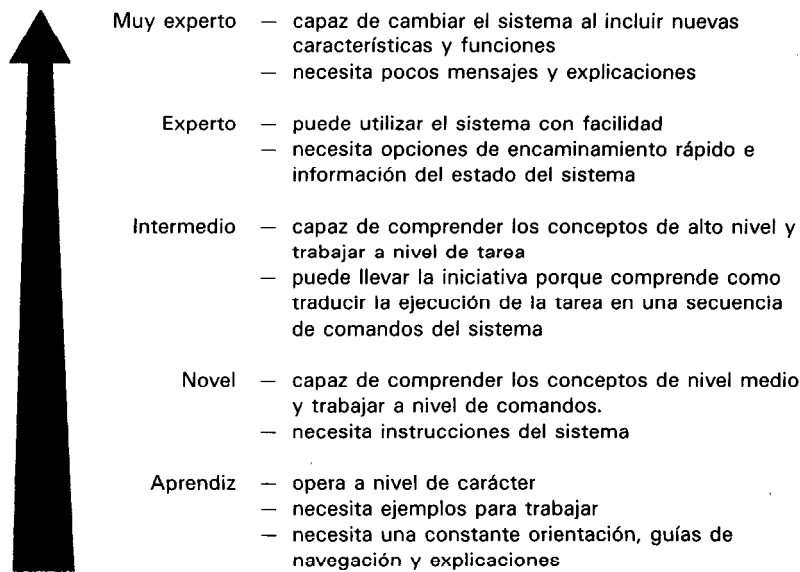


Figura 11.4. Las distintas clases de usuarios interactúan diferentemente con el sistema y requieren diferentes tipos de soporte y de información procedente del sistema

Una forma de que el sistema puede distinguir automáticamente al novicio del experto, es medir el nivel de experiencia del usuario en función de:

- El número de preguntas realizadas (peticiones de ayuda).
- El número de entradas erróneas.
- El número de diferentes tipos de comandos que ha pedido el usuario.

Los usuarios de clases intermedias y altas son los que llevan la iniciativa, no el sistema. Se asume que el usuario de este nivel comprende la tarea a realizar y es capaz de traducir la realización de la tarea a una secuencia de comandos que se deben ejecutar. La información del estado del sistema, los recursos utilizados y los que restan, el tiempo de espera, las opciones disponibles, etc.) y la asistencia en la búsqueda de la vía más rápida para realizar las tareas son útiles.

Cuadro 11.1. Las características de un interfaz amigable con el usuario

- **Fácil de utilizar**
 - Un número mínimo de conceptos a aprender necesarios para poder empezar.
 - Un número mínimo de reglas de sintaxis.
 - Un lenguaje unificado para todo el sistema.
 - Los aspectos básicos del ordenador están ocultos para el usuario.
- **Robusto**
 - Es virtualmente imposible para el usuario causar una terminación anormal del sistema.
- **Colaborador**
 - En cualquier punto de utilización del sistema están disponibles comandos de AYUDA e información de orientación al usuario.
- **Detector y avisador de errores**
 - Cuando se comete un error aparecen mensajes de error significativos, explicando lo que está equivocado y cómo corregirlo.
 - El sistema se ha diseñado para evitar los errores más comunes de usuario.
- **Ampliable**
 - El sistema es fácil de cambiar y las nuevas funciones pueden incluirse fácilmente.
- **Fácil de aprender**
 - Las características fundamentales del sistema pueden aprenderse en unas pocas horas.
- **Adaptable**
 - El sistema se ajusta al usuario experto de forma totalmente diferente a como lo hace con un usuario novel.

ORIENTADO AL USUARIO

En un entorno habitable, el concepto **orientado al usuario** reemplaza al concepto **amigable con el usuario** al ampliar el interfaz de usuario. La partícula ‘final’ desaparece del término “usuario final” para poner más énfasis en la importancia del usuario [1, págs. 297-313]. Pensar primero en el usuario, no al final. El concepto de orientado al usuario se materializa mejor en la estación de trabajo de desarrollo de software, que actúa como asistente personal dedicado a las necesidades de su usuario. Esto significa que el interfaz de usuario se considera como el objetivo central al diseñar la estación de trabajo de desarrollo de software. Lo correcto es que se diseñe el interfaz de usuario al principio, en lugar de agregarlo al final del diseño.

Así el interfaz orientado al usuario es amigable para el usuario y mucho más. El cuadro 11.1 resume las características de un interfaz amigable para el usuario. Además de tener todas características de interfaz amigable con el usuario, un interfaz orientado al usuario también es:

- Simple.
- Consistente.
- Flexible.
- Controlado por el usuario.
- En lenguaje natural.

El cuadro 11.2 resume las características de un interfaz orientado al usuario.

Cuadro 11.2. Las características de un interfaz orientado al usuario

- **Simple.**
 - Un número mínimo de símbolos y de reglas de sintaxis.
 - Un número mínimo de excepciones a las reglas.
 - Cada símbolo tiene un solo significado y un solo objetivo en todo el sistema.
 - Un número mínimo de comandos solapados.

Cuadro 11.2. Las características de un interfaz orientado al usuario (continuación)

- **Consistente.**
 - Un modelo único de cómo trabaja el sistema para todo el sistema.
 - Un lenguaje unificado utilizado en todo el sistema.
- **Flexible.**
 - El sistema permite cierta libertad en las entradas de usuario, permitiendo abreviaturas, errores ortográficos y consultas semánticamente significativas pero sintácticamente incorrectas.
- **Controlado por el usuario.**
 - Que siempre parezca que es el usuario, no el sistema, quien tiene el control.
- **Lenguaje natural.**
 - El usuario puede comunicarse con el sistema utilizando un reducido vocabulario en lenguaje natural y un conjunto reducido de reglas de construcción sintáctica.

Simple

En un interfaz humanizado, la sencillez es la verdadera elegancia. Lo simple es lo opuesto a lo complejo. Mientras que un sistema complejo complica el trabajo del usuario, un sistema simple intenta hacer cada tarea lo más sencilla posible.

Un sistema **simple** emplea el mínimo número de símbolos sintácticos, el mínimo número de reglas sintácticas y el número mínimo de excepciones, según Zloof [6]. Cada símbolo tiene un solo significado y un solo propósito en todo el sistema.

Un buen ejemplo de sistema simple es el Xerox Star, cuyo diseño de interfaz de usuario introdujo las ideas de iconos, ventanas y sobremesa (**desk-**

top). El desarrollo de Xerox Star influyó fuertemente en el diseño de sistemas posteriores, como LISA y Macintosh. Durante el desarrollo de Star, Xerox clasificó ciertos de conceptos como difíciles y otros como fáciles de comprender para el usuario (ver figura 11.5). Xerox mantuvo el Star simple incorporando sólo aquellos conceptos fáciles de comprender para el usuario.

Difícil	Fácil
Abstracto	Concreto
Invisible	Visible
Crear desde la nada	Copiar y Modificar
Rellenar lo vacío	Seleccionar de una lista
Generalizar	Reconocer
Programar	Editar
"Batch"	Interactivo

Figura 11.5. Algunos conceptos son difíciles de comprender para la mente humana; otros más fáciles. Al diseñar un interfaz, deben implantarse los conceptos más fáciles de comprender

Lamentablemente, son más los ejemplos de sistemas complejos que los simples. Un ejemplo muy común es el empleo de los **modos** en los sistemas. Un modo permite que un símbolo, un comando o una tecla tengan dos o más propósitos. Un modo identifica cierto estado del sistema y el significado de un símbolo cuando el sistema está en ese estado en particular. La interpretación de cómo se comporta el sistema depende del modo. Los modos violan el principio de simplicidad porque un símbolo tiene más de un significado o propósito. Xerox cita un curioso, pero lamentable, ejemplo de las molestias que pueden causar los modos cuando el usuario no presta la debida atención al modo en que está el sistema.

Bravo es un editor de textos desarrollado por Xerox [1, págs. 297-313]. Tiene dos modos: Insert y Command. En la historia de Xerox, un usuario perdió la totalidad de un documento porque equivocadamente pensó que estaba en modo Insert cuando el sistema estaba realmente en modo Command. El error fatal fue teclear la palabra "edit" en el documento. En modo Command, Bravo interpretó "edit" como la siguiente secuencia de comandos [1, págs. 297-313):

E seleccionar algo en el documento.

D borrarlo.

I introducir el modo Insert.

T teclear una “t”.

El resultado fue que se perdió todo el documento, quedando únicamente la letra “t”.

Otro ejemplo de violación del principio de simplicidad se muestra en el código FOCUS de la figura 11.6. En la segunda línea del código la barra “/”, representa el operador aritmético de la división y significa dividir CANTIDAD entre UNIDADES. Sin embargo, en la tercera línea del código, la “/” tiene un significado diferente. Se emplea para separar el nombre de un dato de la descripción de formato del campo. En este ejemplo, el dato AJUSTE se define como un número decimal con ocho dígitos de longitud, incluyendo dos posiciones a la derecha del número decimal.

Es un ejemplo de permitir que un símbolo tenga múltiples significados. Para determinar lo que significa el símbolo, el usuario debe entender antes el contexto en el cual se utiliza. Evidentemente, esto complica la comprensión del sistema.

Un sistema también es más difícil de entender cuando varios nombres de comandos diferentes tienen un significado similar o cuando varios comandos diferentes realizan la misma función. El primer caso es más confuso para el usuario. Por ejemplo, en IBM/VM los sinónimos DELETE, PURGE, ERASE y DISCARD (eliminar, purgar, borrar, quitar) representan comandos que realizan diferentes funciones. No es realista esperar que el usuario típico recuerde, por ejemplo, que DELETE significa borrar líneas en un fichero que se está editando, mientras que ERASE significa borrar fichero de un disco [8]. Casualmente, en el DEC System 10 ERASE y DELETE tienen justamente el significado opuesto que en IBM/VM.

Consistencia

La integridad del diseño se manifiesta en la característica de la **consistencia**. Existe un modelo básico de cómo funcionan las cosas y de cómo

```

DEFINE FILE PROD
PRICE = AMOUNT/UNITS;
ADJUST/D8.2 =
  IF FACTOR EQ 0 THEN PRICE
  ELSE FACTOR *.5* PRICE

```

Figura 11.6. En este ejemplo de FOCUS, el símbolo “/” tiene dos significados. En la línea segunda, es el operador aritmético de la división; en cambio en la línea tercera, es el símbolo separador entre el nombre del campo y la descripción del formato del campo

parece que han sido hechas durante todo el sistema. La integridad no se ha visto comprometida con el tiempo por la adición de nuevas características ni por los programadores que han trabajado con el modelo. Es como si el sistema fuera la creación ininterrumpida de un arquitecto de sistemas. Se utiliza un lenguaje común en todo el sistema. Las reglas sintácticas son siempre las mismas, al igual que el significado de los símbolos y las reglas. Los menús y pantallas de entrada/salida tienen el mismo formato.

La consistencia se traduce en una uniformidad predecible que capacita al usuario a aplicar lo que ha aprendido en una parte del sistema en otras partes. Esto acorta el tiempo necesario para aprender el sistema, reduce el número de errores que puede cometer el usuario y, en general, incrementa la productividad del usuario.

La consistencia se explica mejor con ejemplos. En un sistema consistente, los procesos de ejecución de un comando, emisión del siguiente comando y cancelación de un comando son siempre los mismos. Los argumentos de los comandos aparecen siempre en el mismo orden. La información de la orientación y del estado del sistema está siempre localizada en el mismo sitio de la pantalla. Por ejemplo, el botón de la izquierda del ratón siempre se utiliza para seleccionar un objeto y después para seleccionar una operación a realizar sobre dicho objeto. El botón de la derecha siempre se utiliza para volver al menú principal. El comando EXIT siempre se utiliza para abandonar el sistema y siempre se pregunta al usuario si desea guardar el fichero en curso antes de que se ejecute el comando EXIT.

Como la consistencia, al igual que la simplicidad, es una característica muy difícil de conseguir, hay muchos ejemplos de inconsistencia en los sistemas. Por ejemplo, considérense los sistemas operativos IBM VM/CMS. Como muchos sistemas, el IBM VM/CMS intenta acomodar al usuario permitiéndole acortar los nombres de los comandos. Para desconectar una máquina virtual, el usuario puede emitir el comando CP DISK. Pero si el usuario intenta preguntar al sistema por medio de HELP para aprender sobre el comando de desconexión mediante HELP DISK, el comando falla porque el sistema operativo no permite que se acorten los comandos cuando se pregunta a través de HELP. Este tipo de inconsistencia es muy frustrante y da como resultado el desánimo, en lugar de animar al usuario en la utilización de la consulta HELP al sistema.

La ilusión del usuario

La simplicidad y la consistencia llevan el sistema un buen trecho hacia la orientación al usuario. Los sistemas que son simples y consistentes se dice que se adhieren al **principio del asombro mínimo** [9]. Tales sistemas nunca se comportan de forma que puedan sorprender al usuario (ver figura 11.7).

Como el usuario interactúa con el sistema, crea un modelo conceptual que explica el comportamiento del sistema. Este modelo conceptual se denomina **ilusión del usuario** [9]. Un sistema inconsistente y complejo hace añicos la ilusión del usuario y viola el principio del asombro mínimo, porque el usuario no puede confiar en su intuición y teme que sus expectativas se vean decepcionadas. Cuando el usuario no tiene un modelo exacto de los límites del sistema estará pidiendo continuamente capacidades que el sistema no tiene y nunca utiliza otras que el sistema puede proporcionarle.

Flexible

Un sistema **flexible** ofrece al usuario cierta libertad en la introducción de datos al sistema. Las entradas de usuario van desde las abreviaturas de los comandos hasta frases completas. Para tener la certeza de que el sistema ha interpretado correctamente el comando del usuario, deberá parafrasear la entrada del usuario en su formato sintáctico y después preguntar al usuario si aprueba su interpretación antes de ejecutar el comando.

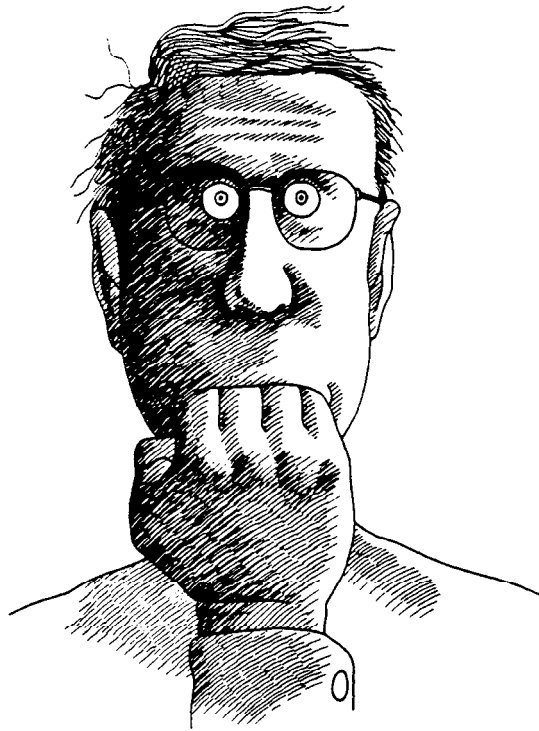


Figura 11.7. Los sistemas simples y consistentes se adhieren al principio del asombro mínimo. Nunca sorprenden al usuario con un comportamiento inesperado

Muchos errores de usuario están causados por la incapacidad del sistema para aceptar las posibles variaciones de la entrada de usuario. Como se muestra en la figura 11.8, Hebditch muestra un ejemplo donde el usuario debe teclear una fecha tres veces porque el sistema sólo permite una forma sintáctica estricta (DD/MM/AA) para la introducción de la fecha [4]. Hebditch afirma que el cincuenta por ciento de los mensajes de error se deben a una irrazonable inflexibilidad de los sistemas. En un estudio realizado por Senay y Stabler sobre las consultas HELP en el sistema operativo IBM 4341 VM/CMS, descubrieron que una de cada cinco consultas fallaba por la incapacidad del sistema para comprender la pregunta del usuario [8].

PLEASE ENTER DATE: 20.8.80
ERROR-DATE MUST BE DD/MM/YY. TRY AGAIN
PLEASE ENTER DATE: 80/8/80
ERROR-DATE MUST BE DD/MM/YY. TRY AGAIN
PLEASE ENTER DATE: 20/08/80

Figura 11.8. El cincuenta por ciento de los mensajes de error se deben a la inflexibilidad del sistema al no permitir las variaciones en las entradas al sistema [4]

Un sistema flexible acepta abreviaturas, consultas del usuario con sinónimos, falta de palabras, palabras juntas, palabras en orden incorrecto, errores ortográficos y con significado semántico pero sintácticamente incorrectas.

Controlado por el usuario

Mientras que los usuarios inexpertos pueden preferir que el sistema lleve el control, los usuarios más experimentados tienden a rechazar la apariencia de que el sistema es quien manda y ellos tienen que obedecerlo. Cuando parece que el sistema lleva el control, existe el riesgo de que el usuario se convierta en un observador pasivo y no interesado de las operaciones del sistema. La ilusión de llevar el control puede incrementar la motivación del usuario y conducir al incremento de la productividad [10].

En los sistemas orientados al usuario, éste siempre lleva el control del sistema o al menos lo aparenta. Esta sensación de control se establece de varias formas:

1. El sistema siempre responde inmediatamente al usuario.
2. El sistema siempre informa al usuario de su estado correcto.
3. Los mensajes de error son amables y significativos, explicando lo que está equivocado y cómo corregirlo.
4. Al usuario se le notifica y se le solicita la confirmación en situaciones especiales, como borrar un fichero o la recuperación de un gran número de registros como respuesta a una consulta.

Lenguaje natural

Aunque las personas preferimos el lenguaje natural, los ordenadores no. Los ordenadores siempre han tenido dificultades con el análisis del lenguaje natural, debido a sus inherentes ambigüedades y su falta de estructura. Por otro lado, las personas a menudo introducimos errores al traducir el lenguaje natural al requerido por el ordenador. En un entorno habitable, se reconoce esta preferencia del usuario. El usuario es capaz de comunicarse en lenguaje natural, pero probablemente utilizando sólo un subconjunto del lenguaje natural con un vocabulario entre 300 y 1.500 palabras y un restringido subconjunto de construcciones sintácticas [7, págs. 72-82]. Cuando el vocabulario se restringe de esta manera, el procesamiento del lenguaje natural puede ser realizado eficientemente por el ordenador.

Este nivel de restricción no es realmente un impedimento para alguien que use las herramientas de **software** en tareas específicas o en el desarrollo de ciertos tipos de sistemas de aplicaciones. Cuando el contexto de interpretación está claramente definido, muchas de las ambigüedades posibles desaparecen, pues ciertos significados no tiene sentido en este contexto específico.

En un experimento realizado por Shneiderman, se demostró que la restricción del vocabulario al usuario no necesariamente era un obstáculo para éste. En este experimento se dijo a uno de los dos grupos que limitara el número de palabras utilizadas en la resolución de un problema. El resultado del experimento fue que el grupo con el vocabulario reducido a un quinto resolvió el problema en menos tiempo que el grupo sin restricción [7, págs. 72-82]. En otro experimento, a los usuarios se les pidió que escribieran tantas formas como fueran posibles para hacer una determinada pregunta. Escribían fácilmente una fórmula pero tenían dificultades con la segunda y siguientes [1, págs. 240-244]. Parece ser que la limitación de la construcción sintáctica no es una restricción grave, especialmente cuando el usuario está trabajando en el dominio de una aplicación específica.

Un problema con los interfaces en lenguaje natural es que el usuario puede no tener un conocimiento exacto del alcance de la comprensión del sistema. Por ejemplo, el usuario puede pensar que el sistema comprende más de lo que realmente hace. Este problema se denomina **exceso semántico** [7, págs. 130-138]. Es difícil que el sistema detecte y muestre errores de exceso semántico que no sea enviar un mensaje diciendo que no entiende

una o varias palabras o limitando al usuario a expresar un solo concepto por sentencia [11, págs. 204- 244].

DE SENSIBLE A REACTIVO

Se espera de un sistema amigable para el usuario que sea sensible, pero un entorno habitable es más que sensible, es reactivo (ver figura 11.9). Un sistema **sensible** responde a cada entrada de usuario, ya que una falta de respuesta del sistema puede ir en detrimento de la comprensión del sistema por el usuario. De acuerdo con la **teoría del refuerzo** de Skinner, la inmediata respuesta favorece el comportamiento mientras que la ausencia de éstas lo debilita [12, págs. 10-32]. Si la entrada del usuario es incorrecta, un sistema sensible informa y explica lo que está equivocado e indica cómo corregirlo. Si la entrada de usuario es una petición que puede tener mayores consecuencias, como borrar un fichero o recuperar un número excesivo de registros, un sistema sensible informa al usuario y pide confirmación antes de procesar la petición.

Un sistema **reactivo** es sensible y **adaptable**. No solamente responde a las peticiones del usuario sino que también estudia y aprende cómo está utilizando el sistema el usuario. Captura la información sobre cuales son las funciones que el usuario utiliza más frecuentemente, menos frecuentemente o que no utiliza, que errores son más comunes y las entradas que el sistema no comprende. Basándose en esta información, un sistema reactivo puede cambiarse fácilmente para satisfacer mejor las necesidades del usuario.

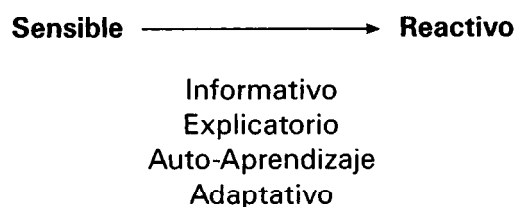


Figura 11.9. Un sistema reactivo no solo responde inmediatamente a cada una de las peticiones del usuario de una forma comprensible, sino que también aprende del usuario como utilizar el sistema

La posición que toma el sistema reactivo es ayudar evitando los errores en primer lugar, ya que los errores pueden llevar a problemas de motivación y productividad. Las funciones y construcciones que con frecuencia se utilizan incorrectamente, se identifican y cambian. Las ayudas en el aprendizaje de nuevas funciones se ofrecen al usuario antes de que éste las aprenda incorrectamente. De acuerdo con Skinner, el usuario aprende más eficazmente diciéndole lo que es correcto que diciéndole que se ha equivocado después de cometer un error y luego tener que aprender la forma correcta [12, págs. 10-32].

DIAGNOSTICO

¿Cómo debe responder un entorno habitable cuando el usuario comete un error? Hay varias posibilidades. Primero, el sistema puede responder con un mensaje de error. Segundo, el sistema no solamente envía el mensaje de error, sino además puede impedir que el usuario prosiga hasta que se corrija el error. Tercero, el propio sistema puede intentar corregir el error. Finalmente, el sistema puede no hacer nada, ignorando la entrada errónea.

Esta última posibilidad, no hacer nada, es inaceptable, porque la ausencia de respuesta del sistema no proporciona ninguna asistencia al usuario para corregir el error. En esta situación, el usuario intenta repetidamente introducir la misma entrada errónea, con lo que cada vez se queda más y más frustrado sin conseguir arreglar el problema.

Como mínimo, cuando se produce un error, el sistema debe informar al usuario con un mensaje de error. Aunque esto es una práctica común, muchos mensajes de error no son muy informativos. Por ejemplo, del compilador de COBOL CDC 6600 se ha dicho que el ochenta por ciento de los errores se diagnosticaron incorrectamente [13]. Algunos ejemplos, divertidos, pero igualmente frustrantes, de mensajes de error en sistemas actuales se listan en la figura 11.10. Mensajes así no dan ninguna pista de donde está el error ni de cómo corregirlo. Sólo sirven para frustrar, exasperar y desmoralizar al usuario.

En algunos sistemas, los mensajes de error van acompañados del sonido del timbre que desconcierta todavía más al usuario que está intentando realizar un trabajo y no ve la gracia en eso.

?
 NO EXISTE FICHERO
 DATO INVALIDO
 ERROR 7036D
 TRABAJO CANCELADO
 ERROR CATASTROFICO
 RESPUESTA PENDIENTE LINEA 100
 ERROR FATAL EN PASO A CERO

Figura 11.10. Los mensajes de error en la mayoría de los sistemas no son muy informativos y no ofrecen al usuario una pista real sobre como corregir el error

Diagnóstico —————> Corrector
 Cortés
 Insensible
 Constructor del consejo
 Autocorrectivo
 Evita el error

Figura 11.11. Un entorno habitable no solamente diagnostica al error del usuario, sino que le ayuda a corregirlo con cortesía, con mensajes de error significativos y en ocasiones autocorrector de error del usuario

El entorno habitable proporciona al usuario que ha cometido un error un consejo constructivo y en algunas ocasiones intentará corregirlo por él (ver figura 11.11). Es a un tiempo **cortés** e **insensible**. Un sistema cortés nunca reprende al usuario ni le critica, ni implica que es estúpido, ni le reclama nada. No hay pruebas de que la amenaza, la ansiedad, el castigo sean motivadores eficientes [12 págs. 25-30]. En su lugar, un sistema habitable informa cortésmente, diciendo específicamente lo que está equivocado y como corregirlo.

Así no
 DATO INVALIDO

Así sí
 SALARIO DEBE ESTAR
 ENTRE 10.000
 Y 100.000 DOLARES

Los mensajes de error desconcertantes e inexactos conducen a una productividad pobre y a la insatisfacción de los usuarios. Según un experimento de Shneiderman se consiguió una mejora de casi el 50% en la productividad del usuario al mejorar los mensajes de error. De acuerdo con Bader y Shneiderman [7, pág. 55]:

Los indicadores (**prompts**) normales, los mensajes de aviso y las respuestas del sistema a los comandos influyen en la percepción del usuario, pero los textos de los mensajes de error o de aviso de diagnóstico son críticos. Como los errores ocurren debido a la falta de conocimientos, a comprensión incorrecta o a un desliz inadvertido, el usuario se puede sentir confuso, incómodo y ansioso”.

CORRECTOR

Un sistema que es sensible puede corregir por sí mismo muchos tipos de deslices y errores sintácticos. Los sistemas autocorrectores se denominan **DWIM (Do-What-I-Mean)** [13, págs. 411-432]. Analizando la entrada, el sistema sospecha lo que el usuario realmente quería y cambia la entrada en consecuencia. Por supuesto, en ocasiones la hipótesis del sistema esta equivocada y por tanto el cambio es también incorrecto. Para proteger al usuario de un cambio incorrecto en el sistema, el autocorrector siempre debe informar al usuario de la corrección. Además, el autocorrector debe incluir el comando **DESHACER (UNDO)** para permitir al usuario deshacer cualquier cambio realizado por el sistema.

AYUDA

La función de ayuda (**HELP**) existe en la mayoría de los sistemas en línea. Se proporciona de diversas formas:

1. A nivel de comando.
2. Como indicador (**prompt**) del sistema.
3. En los mensajes de error.
4. En la documentación en línea.
5. En los manuales de referencia fuera de línea.
6. En el tutor en línea.

La ayuda a nivel de comando

La ayuda a nivel de comando es la más comúnmente empleada. El usuario teclea **HELP** (ayuda) seguido del nombre del comando. El sistema responde con una explicación de cómo funciona el comando y en ocasiones se incluye un ejemplo. Para utilizar esta forma de **HELP**, el usuario debe conocer el nombre del comando o una abreviatura aceptable. Algunos sistemas proporcionan un comando **INDEX** que lista todos los comandos del sistema para ayudar al usuario a elegir el que más le interesa. Cuando los nombres de los comandos son significativos, el usuario tiene menos necesidad de recurrir a **HELP** y tiende a recordar mejor los nombres de los comandos.

Los sistemas más sofisticados ofrecen el **HELP sensible al contexto**. En este caso, el sistema debe saber lo que está haciendo el usuario en el momento de solicitar el **HELP**. Utilizando esta entrada del usuario, el sistema ayuda a construir el comando que se intenta utilizar. Esta es la forma de aprender haciendo, una forma de **HELP activo**. Es mucho más efectivo que leer algo sobre el comando. De acuerdo con Skinner, la lectura pasiva sin manipulación de los conceptos en juego es poco instructivo [12, pág. 14].

Además, el **HELP** a nivel de comando, la mayoría de los sistemas en línea proporcionan una forma de introductorio o de orientación para explicar en general como utilizar sistema. Un usuario primerizo necesita saber qué hace el sistema, cómo se estructuran los menús y los comandos, cómo empezar, cómo salir y cómo conseguir ayuda.

Los indicadores

Los indicadores (**prompts**) también pueden utilizarse como ayuda al usuario. Cuando un indicador del sistema solicita una entrada de usuario, puede describir qué tipo de entrada se necesita. Por ejemplo, el sistema podría solicitar: **INTRODUCIR 4 DIGITOS PARA CODIGO ESTADO CLIENTE**. Si el usuario necesita más información, puede solicitar ayuda (**HELP**). El sistema entonces responde con una lista de los códigos válidos y de sus definiciones. Esta es una forma de **HELP escalonado o estratificado**, en la cual la información se ofrece al usuario por pasos. La idea es no sobrecargar al usuario con información innecesaria.

Los mensajes de error significativos ofrecen otra forma de HELP. Sin embargo, pueden utilizarse los indicadores del sistema en lugar de los mensajes de error. Cuando el sistema detecte una entrada de usuario incorrecta o incompleta, responde al usuario por medio del indicador con información sobre el formato de la entrada. En este caso, el sistema, no el usuario, ha reconocido la necesidad de ayuda y ha iniciado el proceso de HELP.

La documentación en línea

Muchos sistemas en línea han almacenado en línea el manual de referencia del sistema. Esto no es muy útil, a menos que el usuario disponga de un mecanismo para encontrar convenientemente los conceptos y los comandos del manual. Esta forma de ayuda puede inundar en demasiada información al usuario, dificultándole el paso hasta el punto adecuado. La mayoría de los usuarios a los que se les presentan páginas de documentación sobre un comando en particular, prefieren experimentar directamente con el comando o preguntar a otro usuario antes que leer la documentación. Hay evidencia de que ni los usuarios experimentados ni los noveles leen los manuales, independientemente de si están en línea o no, porque los manuales no proporcionan una información lo suficientemente específica o lo bastante completa para ser de ayuda [13, págs. 399-410]. Es más, el HELP en línea es más útil para los usuarios experimentados, porque los noveles están demasiado confundidos o les falta la confianza necesaria para solicitar el HELP.

Así pues, la tendencia es evitar el desarrollo de manuales completos y amigables e incorporar indirectamente la ayuda en el sistema. Los sistemas que se diseñan como entornos habitables reducen la necesidad del usuario de recurrir al HELP. El cuadro 11.3 resume algunos métodos directos e indirectos para introducir el HELP en el sistema.

En ocasiones, los usuarios necesitan saber cómo opera un determinado comando del sistema, pero con más frecuencia necesitan algo más que ayuda a nivel de comando, incluso más que ayuda a nivel de comandos sensible al contexto. Por ejemplo, se necesita conocer cual es la secuencia de comandos necesaria para realizar una tarea, qué diferencia existe entre dos comandos que aparentemente realizan el mismo cometido, una forma mejor de utilizar el sistema o por qué (a nivel conceptual) algo está equivocado.

Cuadro 11.3. Métodos para integrar la ayuda en el sistema

- Minimizar la necesidad de la ayuda (HELP).
 - Menús explícitos.
 - Nombres de comandos significativos.
 - Mensajes de error significativos.
 - Información orientativa.
 - Corrección automática de errores y la posibilidad de “des-hacer” (comando UNDO).
 - Índice de comandos.
 - Glosarios de términos.
- Recordar que el HELP en línea es más útil para los usuarios experimentados.
- Información a nivel de HELP para evitar sobrecargar con datos innecesarios al usuario.
- Proporcionar el HELP estratificado y sensible al contexto como una combinación poderosa.
- Concentrar el HELP en la búsqueda de errores en los ejemplos y la asistencia.

Tipos de ayuda (HELP)

Como los usuarios pueden no conocer cuáles son las preguntas HELP correctas, necesitan poder dialogar con el sistema. Como los usuarios pueden no saber cuando necesitan ayuda o les falta confianza para pedirla, el sistema debe poder reconocer cuando un usuario tiene dudas e iniciar el proceso de ayuda. El cuadro 11.4 muestra una tabla de dos tipos fundamentales de ayuda que puede proporcionar el sistema. A la derecha de la tabla se muestran los tipos de ayuda que normalmente suelen ofrecer los sistemas amigables. Este tipo de ayuda es a nivel de comando, lo inicia el usuario y describe cómo trabaja un comando. Es una forma **pasiva** de ayuda que puede incluir una **lectura activa** cuando se incluye ayuda sensible al contexto.

La parte izquierda de la tabla muestra otro tipo de ayuda que también se debe ofrecer. Es a nivel de tarea, puede iniciarla el sistema y proporcio-

Cuadro 11.4. Tipos de ayuda (Help)

Entorno habitable	Amable con el usuario
Tarea de procedimiento	Comando
Iniciado por el sistema	Iniciado por el usuario
Explicación de ayuda	Descriptivo

na asesoramiento y explicaciones al usuario. Es una ayuda de alto nivel, una forma de ayuda **interactiva** y, en algunos casos, **inteligente**. Los ejemplos de Rich mostrados en los cuadros 11.12 y 11.13 ilustran las diferencias entre estos tipos de ayuda [5].

En la primera petición de ayuda (HELP) mostrada en la figura 11.12, HELP FTP, se requiere que el usuario conozca el comando sobre el que se busca información y conocer también la abreviatura apropiada del nombre del comando. La segunda petición 'HELP FILE COPY' no requiere que el usuario conozca el nombre del comando específico, ni que el usuario componga la petición HELP de una forma particular. Utilizando palabras del lenguaje natural, el usuario simplemente pregunta al HELP cómo copiar un fichero. La tercera petición HELP es una cuestión en lenguaje natural que pregunta cómo se puede realizar una tarea específica. La cuarta petición es en forma de diálogo entre el sistema y el usuario. Es una petición al sistema para que explique por qué no funcionó el comando DELETE (borrar) y cuál funcionará.

Las dos primeras peticiones son a nivel de comando y cualquier sistema amigable debe proporcionarlas. Las dos últimas peticiones requieren un sistema inteligente que comprenda el lenguaje natural y tenga algunos conocimientos sobre los errores de usuario asociados con el borrado de un fichero. En un entorno habitable se espera que proporcione ayuda inteligente con la cual el sistema pueda ofrecer asistencia y explicaciones.

En el ejemplo de la figura 11.13, el usuario está intentando hacer el formato de un impreso con el comando SCRIBE PAPER.MSS, cuando el sistema queda sin espacio y responde con el mensaje QUOTA EXCEEDED

HELP FTP

HELP FILE COPY

HOW CAN I COPY A FILE FROM ONE MACHINE TO ANOTHER ON
THE NETWORK?

USER: I NEED TO DELETE 2 FILES

SYSTEM: USE DELETE COMMAND

USER: I TRIED THAT

SYSTEM: HAVE YOU CHECKED THE PROTECTION ON THE FILES

Figura 11.12. Los sistemas amigables ofrecen el HELP a nivel de comando, como se ilustra en las dos primeras consultas HELP de la figura. Los entornos habitables van más allá y ofrecen asesoramiento y explicaciones al usuario, como se observa en las dos últimas consultas de la figura [15] (Copyright © 1984 IEEE)

@ SCRIBE PAPER.MSS

QUOTA EXCEEDED AT 676647

@ DELETE *.TMP

@ CONTINUE

ADVICE:

DELETE MARKS FILES FOR DELETION. EXPUNGE REMOVES FILES
FROM DIRECTORY

Figura 11.13. Un sistema con prestaciones de HELP inteligente puede anticiparse cuando el usuario tiene problemas y ofrecerle un HELP explicativo antes de ser consultado por el usuario [15] (Copyright © 1984 IEEE)

AT 676647. El usuario experto quizá sepa que el mensaje se refiere a que es necesario más espacio; en cambio, para un usuario inexperto resultará bastante misterioso. En este ejemplo el usuario responde con los comandos DELETE *.TMP y CONTINUE. Sin embargo, ésta no es una respuesta correcta, pues debe usarse también el comando EXPUNGE (suprimir) para liberar el espacio de los ficheros temporales. La mayoría de los sistemas responderían con mensajes similares a QUOTA EXCEEDED 676647

y no dan al usuario ninguna pista de lo que debe hacerse o, todavía peor, dando la impresión al usuario de que debe eliminar otros ficheros para dejar espacio libre y poder continuar. Un sistema con una ayuda inteligente supera estas barreras al anticipar la necesidad del HELP. Sin que el usuario pregunte, el sistema observa que el usuario necesita más información y el sistema ofrece asesoramiento explicando que para liberar espacio son necesarios los comandos DELETE y EXPUNGE. Ahora, el usuario ya tiene la información necesaria para continuar.

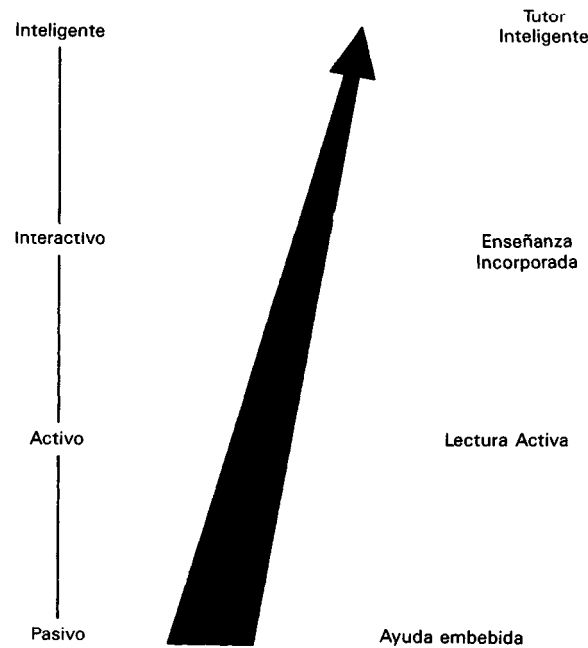


Figura 11.14. Un entorno habitable proporciona el mayor nivel de ayuda inteligente. No solamente ayuda al usuario, sino que además le enseña

La figura 11.14 muestra cuatro niveles de ayuda, que van desde el nivel más bajo, ayuda pasiva, hasta el más alto, ayuda inteligente. La ayuda a nivel de comando y la ayuda sensible al contexto son ejemplos de ayuda pasiva y ayuda activa, respectivamente. En el nivel interactivo, el sistema cruza la frontera entre ayudar e instruir al usuario. En este nivel el sistema puede enseñar los conceptos básicos al usuario y ayudar a corregir un error

o decidir qué comando ejecutar a continuación. En el nivel superior, el sistema tiene capacidades inteligentes, es sensible al contexto, pronostica errores y tiene capacidades explicativas y de procesamiento en lenguaje natural.

EL TUTOR INTELIGENTE

Un obstáculo importante al uso de nuevas herramientas de **software**, independientemente de lo potentes que puedan ser, es la gran cantidad de formación del usuario que se necesita. Para muchas organizaciones, la formación es el mayor problema que retrasa los avances tecnológicos y es un elemento costoso. Según T. Capers Jones [16]:

“El número de personas dedicadas a la educación y a la formación en el conjunto de las compañías de Fortune 500 en EE.UU. es el mismo que el personal docente en todas las universidades de EE.UU. En horas de clase, es posible que los programas internos de formación en las grandes empresas privadas se aproximen a las horas lectivas de las instituciones académicas dedicadas a la informática en EE.UU. Para muchas organizaciones, la formación técnica es el mayor compromiso y los programas de diez a veinte días al año de cursos de formación para directivos no son raros.

Los tutores inteligentes ayudan a resolver el problema de la formación incorporando la instrucción en la propia herramienta. Los tutores inteligentes proporcionan una enseñanza mejor que el aula tradicional con un instructor humano. El cuadro 11.5 resume las características de un buen tutor (humano o artificial) según Skinner [12, pág. 12] y el cuadro 11.6 resume las capacidades de un tutor inteligente.

Con el tutor inteligente se vuelve al entorno de aprendizaje particularizado a las necesidades individuales del usuario (alumno). Como el tutor tiene sólo un alumno con el que tratar, puede centrar toda su atención en él. El tutor puede tener al alumno en continua actividad, lo que proporciona un entorno de aprendizaje óptimo. Hay un intercambio constante entre el tutor y el alumno, capaz de absorber más información en menor tiempo. El alumno puede aprender según su propio ritmo, realizar preguntas y revisar sus conocimientos cuando lo considerar apropiado, lo que da una sensación de control sobre su aprendizaje. El tutor proporciona información inmediata y puede detectar los errores tan pron-

to se producen, lo que evita al alumno de seguir con errores de concepto que más adelante serían difíciles de corregir.

Cuadro 11.5. Características de un buen tutor

- Intercambio constante entre el tutor y el alumno.
- Mantiene al alumno en constante actividad.
- Insiste en la completa comprensión de cada punto antes de pasar a otro.
- Presenta sólo el material que pueda asimilar el alumno.
- Ayudar al alumno a salir adelante correctamente.
- Estimula cada respuesta acertada.

Cuadro 11.6. El tutor inteligente

- Transmite sus propios heurísticos de resolución de problemas al usuario.
- Selecciona los ejemplos apropiados para el usuario.
- Maneja los problemas propuestos por el usuario.
- Compara las habilidades del usuario con el tutor.
- Aprende del usuario heurísticos mejores de resolución de problemas.

El tutor emplea un método de aprendizaje progresivo con el cual el alumno va aprendiendo sólo lo que está preparado para aprender. Trabajando en un entorno de aprendizaje experimental, el alumno va construyendo nuevos conocimientos a partir de los que ya posee. El nuevo conocimiento desarrolla, a través de ejemplos, la generalización, el refinamiento y la modificación. Sin embargo, el tutor no permite al alumno pasar a conceptos más complejos hasta que haya comprendido completamente los conceptos básicos. La enseñanza de los conceptos fundamentales es importante, porque el aprendizaje real y duradero no tiene lugar hasta comprender los conceptos básicos.

Los tutores inteligentes son superiores a los métodos no inteligentes, más simples, como la enseñanza asistida por ordenador (CAI, **Computer Aided Instruction**), porque son menos rígidos. En un CAI no inteligente, el método de aprendizaje está predefinido y no puede ajustarse a los requerimientos de cada persona. Con los tutores inteligentes, el alumno puede explorar el sistema a su propio gusto, comunicarse en lenguaje natural y esperar a que el tutor le aconseje.

Para proporcionar este entorno de aprendizaje superior, el tutor inteligente necesita cuatro tipos de conocimiento [7]:

1. Área a enseñar.
2. Información del alumno.
3. Cómo enseñar.
4. Cómo comunicarse.

El tutor debe ser a un profesor experto y un usuario experto. Tiene un nivel de conocimiento experto del tema en el cual va a utilizarse la herramienta (por ejemplo, el diseño de **software**, la depuración de programas, etc.). El tutor es un sistema experto que puede identificar por el alumno el próximo paso o la tarea a realizar, puede explicar en términos básicos por qué algo es correcto o incorrecto a partir de los conceptos básicos y puede sugerir enfoques que el alumno no tuvo en cuenta. El tutor utiliza el lenguaje natural para comunicarse.

Si el tutor no tiene más conocimientos que el alumno, será de poca ayuda. Por tanto, una cualidad importante del tutor es que puede aprender, además, de enseñar. Un tutor inteligente deberá tener una estrategia de automejora para estudiar cómo el alumno emplea la herramienta en la realización de una tarea. Si el método del alumno es mejor que el del tutor (por ejemplo, realiza la tarea en menos pasos), éste deberá sustituir su método por el del alumno. O si el alumno realiza una tarea que el autor no había tratado antes, la tarea deberá agregarse a la base de conocimientos del tutor sobre el uso de la herramienta.

Un tutor inteligente debe dirigir y adaptarse a distintos aspectos culturales del alumno. Modifica su estrategia tutelar dependiendo de los conocimientos previos, su rendimiento y de los errores del alumno. El tutor almacena la información en un modelo del alumno. El dominio o tema a enseñar por el tutor inteligente consta de un conjunto de hechos y reglas. Los heurísticos se añaden a las reglas para dar al tutor más flexibilidad

en la interpretación de las cuestiones del alumno y explicar su propio razonamiento. El conocimiento del alumno se modela como un subconjunto y variaciones simples del conocimiento del tutor. Así, la tutoría es un intento de incrementar el subconjunto ampliando reglas y hechos del estudiante sobre el dominio en particular [11, pág. 51]. Para cumplir con su labor de enseñanza, el tutor requiere no solamente el conocimiento sobre el alumno, sino también el conocimiento sobre los errores que más frecuentemente comete éste, las manifestaciones de tales errores y cómo se construyen. Los errores más comunes incluyen subpasos que faltan, subpasos extra o seguir correctamente un procedimiento incorrecto o ineficaz para realizar una tarea.

Ejemplos de tutores inteligentes

Las figuras 11.15 y 11.16 son ejemplo de diálogos entre el usuario y el tutor en dos sistemas tutores inteligente. La figura 11.15 es un tutor de cálculo desarrollado en Stanford (en LISP y se utiliza para la enseñanza de técnicas de integración en cálculo [11, págs. 239-304]. La figura 11.16 es un tutor llamado SPADE-O, desarrollado en los laboratorios de inteligencia artificial del MIT y se emplea para enseñar métodos de diseño de programas [11, págs 123-140].

En ambos ejemplos, el tutor y el usuario se comunican en lenguaje natural. El usuario a menudo responde con una sola palabra, pero el tutor emplea frases completas para asegurar que se le entiende fácil y claramente. En ambos casos el tutor es un experto en resolver problemas de cálculo por sí mismo (en el caso del tutor de cálculo) o de diseñar programas (en el caso de SPADE-O).

Obsérvese que el tutor conoce al usuario y los comandos del sistema que éste ha utilizado hasta el momento. Tanto el proceso de integración como en el del diseño del programa, en cada paso el tutor es capaz de seleccionar el paso mejor siguiente, pero lo dejan a la preferencia del usuario.

Por ejemplo, el tutor de cálculo permite a su usuario, Ruth, elegir la sustitución como método para resolver la integral de $x \log(x) dx$, pero no critica severamente a Ruth cuando se evidencia que la sustitución no ayudará a resolver la integral. En ese momento, Ruth pide al tutor que le muestre cómo resolver la integral, lo cual el tutor hace con acierto. Al final de la sesión, el tutor compara su solución con la de Ruth. La del tutor es mejor.

(INTEGRATE)

HELLO! WHAT IS YOUR NAME?

RUTH

HAVE YOU EVER STUDIED PARTIAL FRACTIONAL EXPANSION?

NO

OK, DO YOU HAVE A PROBLEM?

NO

ALL RIGHT, I'LL SELECT ONE FOR YOU. THINKING FOR A MOMENT.

OK, RUTH

OUR INTEGRAL IS $\int x \times \log(x) dx$

WHAT SHALL WE DO TO SOLVE IT?

SUB

CAN YOU THINK OF A SUBSTITUTION?

(YES, HELP, OR EXIT)

YES. LET $U = \log(X)$

OK, RUTH. WE WILL LET $U = \log(X)$

OUR NEW INTEGRAL IS $\int Ue^U dU$

PLEASE TYPE GOOD, BAD, OR GIVE UP

BAD

CAN YOU THINK OF A SUBSTITUTION?

(YES, HELP, OR EXIT)

HELP

I CAN'T FIND A SUITABLE SUBSTITUTION!

WE'LL EXIT . . .

OK, RUTH

OUR INTEGRAL IS $\int x \times \log(x) dx$

WHAT SHALL WE DO TO SOLVE IT?

HELP

I SUGGEST YOU TRY INTEGRATION BY PARTS

WHAT SHALL WE DO TO SOLVE IT?

PARTS

WE WILL USE THE NOTATION

$\int U dV = UV - \int V dU$

CAN YOU THINK OF A CHOICE FOR U?

(YES, HELP, OR EXIT)

HELP

HOW ABOUT LETTING $U = \log(X)$

THUS $dV = X dX$

HMMMM . . .

AN INTRIGUING CHOICE!

THUS $V = X^2/2$ and $dU = 1/X$

.
.
.

YOU DID THAT PROBLEM IN 3 STEPS COMPARED TO 2 FOR ME!

MY SOLUTION

1. INTEGRATION BY PARTS
2. KNOWN INTEGRAL ROUTINE

YOUR SOLUTION

1. SUBSTITUTION
2. INTEGRATION BY PARTS
3. KNOWN INTEGRAL ROUTINE

Figura 11.15. Un fragmento del Calculus Tutor [11]

WHAT IS YOUR NAME?
> **JOHN SMITH**
HELLO JOHN. GOOD TO SEE YOU AGAIN.
DO YOU WANT ME TO GET YOUR WORK FROM THE FILE (JS
WORK 35)?
> **NO**
OK. WHAT NAME FOR PROCEDURE YOU WISH TO PLAN TODAY?
> **WW**
WHAT TYPE OF PLAN WOULD YOU LIKE TO USE?
A. IDENTIFY AS PREVIOUSLY SOLVED PROBLEM
B. DECOMPOSE INTO SUBPROBLEMS
C. REFORMULATE INTO ALTERNATIVE PROBLEM
> **SEQUENTIAL**
OK. I AM ASSUMING
PLAN → DECOMPOSE → SEQUENTIAL
> **WELL POLE ROOF**
DO YOU WANT TO INCLUDE A SETUP STEP?
> **LATER**
OK. JOHN. I AM POSITIONING THE [SETUP] OF WW UNTIL LATER.
WHAT NOW?
> **GO TO A SECOND [INTERFACE] OF SEQUENTIAL PLAN.**
WHAT NOW?
> **RUN WW**
{RUNNING WW . . . DONE} WHAT NOW?
> **DEBUG**
DEBUGGING WW. THE TOP LEVEL OF WW CONTAINS 7 DESIGN
DECISIONS WHICH COULD BE FAULTY. THERE ARE WARNINGS
ATTACHED TO SEVERAL NODES IN THE PLAN FOR WW.

DO YOU WANT TO TRY TO DIAGNOSE BUGS IN WW?
> **YES**
WHAT NOW?
> **WARNINGS**
I AM ASSUMING DIAGNOSE → PLAN → WARNINGS
THE WW PLAN CONTAINS THE FOLLOWING WARNINGS
THE [SETUP] GOAL OF THE SEQUENTIAL PLAN
HAS NOT BEEN COMPLETED.
THE FIRST [INTERFACE] HAS NOT BEEN COMPLETED.
WHAT NOW?
> **REPAIR THE [INTERFACE] BETWEEN POLE AND WELL.**
I SEE JOHN. YOU HAVE CONCLUDED THAT THE [INTERFACE]
BETWEEN POLE AND WELL HAS A SEMANTIC BUG; SPECIFICALLY,
THE MISSING STEP, WHICH IS SYNTACTICALLY OPTIONAL, IS
NEEDED FOR THIS PROBLEM. COMPLETING . . . THE RULE IS
DEBUG → [DIAGNOSE] + [REPAIR].

Figura 11.16. Un fragmento del SPADE-O

> **YES**
DIAGNOSING MEANS FIGURING OUT WHICH NODE IN THE PLAN
REPRESENTS AN INCORRECT CHOICE. THERE ARE FOUR
SOURCES OF EVIDENCE
.
.
.
OK, THE WW PROBLEM IS WELL-DEFINED. WHAT NOW?
ROOF POLE AND WELL.
THESE ARE PARTS IN PROBLEM SPECIFICATION ARE YOU
PURSUING A SEQUENTIAL DECOMPOSITION PLAN?
> **YES**
OK. WHAT NOW?
> **ADVICE**
WELL IS DESCRIBED AS A SQUARE IN THE WW SPECIFICATION.
THE ANSWER LIBRARY CONTAINS A PROCEDURE, RECT, WHICH
DRAWS RECTANGLES. A SQUARE IS A KIND OF RECTANGLE.
HENCE, I SUGGEST TRYING AN IDENTIFICATION INVOLVING RECT
AS THE PLAN FOR WELL.

Figura 11.16. Un fragmento del SPADE-O (continuación)

Sin embargo, si la solución de Ruth hubiera sido la mejor de las dos, el tutor habría cambiado la suya por la de Ruth. Durante las seis primeras semanas de experimentación con el tutor de cálculo, el veinticinco por ciento de sus soluciones fueron sustituidas por las de los usuarios, porque empleaban menos pasos.

En el ejemplo de SPADE-O, el usuario, John Smith, está diseñando un programa cuyo propósito es dibujar un pozo con brocal, el cual consta de tres partes, (1) el pozo, (2) los postes y (3) el tejado. El tutor denomina "plan" al programa de diseño de John. Cuando se inicia la sesión, el tutor sugiere a John que hay tres tipos de planes que puede seguir para diseñar un programa:

- A Identificarlo con un problema resuelto anteriormente.**
- B Descomponerlo en subproblemas.**
- C Reformularlo en problemas alternativos.**

Sin embargo, John no selecciona ninguna de esas opiniones, sino que prefiere una formulación secuencial. El tutor es bastante flexible para ad-

mitir un método que no está incluido en su repertorio. John continúa especificando que el plan del programa constará de tres componentes, correspondientes a las tres partes del pozo. John no solamente diseña su programa, sino que cuando lo ha diseñado por completo pide al tutor que genere el código, que lo ejecute y que le ayude a depurarlo. Como el tutor reconoce que John nunca ha utilizado la facilidad de la depuración, se ofrece a explicársela a John para evitar que la utilice incorrectamente. Finalmente, cuando John pide consejo, el tutor explica que una fórmula de diseño del programa del pozo es reutilizar un procedimiento llamado RECT, que ya existe en la librería del sistema, en vez de construir un procedimiento nuevo partiendo de cero.

Cuadro 11.7. Las características de un entorno habitable

- **Amigable para el usuario:** El sistema es fácil de usar, fácil de aprender, colaborar y adaptable a diferentes usuarios.
- **Sensible:** El sistema responde rápida y significativamente a cada petición del usuario.
- **Diagnóstico:** Cuando se comete un error, el sistema informa al usuario de lo que está equivocado, por qué y cómo corregirlo.
- **Colaborador:** Un sistema colaborador proporciona una descripción de alto nivel de lo que el sistema hace y una descripción a nivel de comandos y de cómo trabaja cada comando del sistema.
- **Apoyo:** El sistema asiste al usuario en la realización de las tareas, pero el usuario, no el sistema, debe siempre tomar la iniciativa y dirigir al sistema seleccionando los comandos a utilizar.
- **Orientado al usuario:** El usuario se considera el centro del sistema y se le considera en el primer, y principal, durante el diseño del sistema.
- **Reactivo:** El sistema estudia cómo se le está utilizando y los errores que se cometen más frecuentemente, y sugiere cómo podría ser modificado para satisfacer mejor las necesidades del usuario.
- **Educativo:** El sistema puede enseñar, así como ayudar, al usuario a utilizar el sistema proporcionando una tutoría inteligente.
- **Directivo:** El sistema puede tomar la iniciativa seleccionando los comandos a ejecutar y en qué orden para realizar una tarea determinada.

RESUMEN

Los entornos habitables proporcionan un interfaz inteligente de usuario que es fácil de utilizar, fácil de aprender y promueve la comprensión humana. El entorno habitable tiene capacidades muy superiores a las proporcionadas por los entornos amigables CASE de finales de los años ochenta. Las características de los entornos habitables se resumen en el cuadro 11.7.

BIBLIOGRAFIA

1. David C. Smith, Charles Irby, Ralph Kimball, Bill Verplank y Eric Harslem, "Designing the Star User Interface", in *Integrated Interactive Computing Systems*. Amsterdam: North-Holland, 1983.
2. Bob Stahl, "Friendly Mainframe Software Guides User toward Productivity", *Computerworld*, febrero 3, 1986, págs. 53-66.
3. Ben Shneiderman, "Direct Manipulation: A Step beyond Programming Languages", *Computer*, Vol. 16, N.º 8, agosto 1983, págs. 57-59.
4. D. Hebditch, "Dialogue Design for User-Friendly Systems", User Friendly Systems Report, *Infotech State of the Art Report*, Vol. 9, N.º 4, 1981, págs. 289-294.
5. Anthony Wasserman, "User Software Engineering and the Design of Interactive Systems", *Infotech State of the Art Report*, Vol. 9, N.º 4, 1981, págs. 375-387.
6. Moshe Zloof, "Query-By-Example", *AFIPS Conference Proceedings*, Vol. 44, 1975.
7. A. Badre y B. Shneiderman, editores, *Directions in Human/Computer Interaction*, Norwood, Nueva Jersey: ALEX Publishing, 1982.
8. Hikmet Senay y Edward Stabler, "Diambiguating HELP Queries for a User Friendly Interface", *COMPSAC Proceedings*, 1985, págs. 399-405.

9. R. E. Sweet, "The Mesa Programming Project", *SIGPLAN Notices*, Vol. 20, N.º 7, julio 1985, págs. 216-239.
10. J. M. Carroll, "The Adventure of Getting to Know a Computer", *IBM Computer*, Vol. 15, N.º 11, noviembre 1982, págs. 49-58.
11. D. Sleeman y J. S. Brown, editores, *Intelligent Tutoring Systems*. Londres: Academic Press, 1982.
12. W. A. Deterline, *An Introduction to Programmed Instruction*. Englewood Cliffs, Nueva Jersey: Prentice-Hall, 1962.
13. Donald Norman y Stephen Draper, editores, *User-Centered System Design*. Hillsdale, Nueva Jersey: Lawrence Erlbaum, 1986.
14. Raymond C. Houghton, Jr. "Online Help Systems: A Conspectus", *CACM*, Vol. 27, N.º 2, febrero 1984, págs. 126-132.
15. Elaine Rich, "The Gradual Expansion of Artificial Intelligence", *Computer*, Vol. 17, mayo 1984. N.º 5, págs. 4-12.
16. T. Capers Jones, "How Not to Measure Productivity", *Computer-World*, 13 enero, 1986, págs. 65-76.
17. B. Woolf y D. McDonald, "Building a Computer Tutor: Design Issues", *Computer*, Vol. 17, N.º 9, septiembre 1984, págs. 61-73.

EL CONDUCTOR DE METODOLOGIA

CONOCIMIENTO Y EXPERIENCIA EN LA METODOLOGIA DEL SOFTWARE AUTOMATICO

El segundo componente de un **shell** inteligente es el **conductor de metodología**. Su propósito es introducir el conocimiento sobre las metodologías de desarrollo de **software** en las herramientas de **software**. Los profesionales expertos del desarrollo suelen ser mucho más productivos que los novicios; porque utilizan su experiencia en desarrollos previos similares para guiarse a través de los complicados procesos del desarrollo de **software**. Capturando este conocimiento en las herramientas de **software** se pone la experiencia de los mejores constructores de sistemas a disposición de cualquier profesional del desarrollo. Esto libera a los programadores para concentrarse en la estrategia en lugar de en los aspectos del proceso.

Hay tres niveles de conocimiento que pueden ser soportados por la automatización del **software**:

1. Documentación.
2. Tutela.
3. Proceso.

El conocimiento a nivel de documentación

Las herramientas de **software** que automatizan la producción de documentación requerida por una metodología de desarrollo de **software** residen en el primer nivel. Los ejemplos incluyen las herramientas CASE de diagramación y documentación que soportan los símbolos gráficos y los convenios de notación requeridos por las distintas metodologías estructuradas (por ejemplo, el diagrama de flujo de datos de DeMarco utilizado para representar la especificación de un sistema en la metodología estructurada de análisis).

El conocimiento a nivel de tutela

A nivel de **tutela**, las herramientas de **software** guían al programador en el uso de una metodología de desarrollo de **software** (ver cuadro 12.1). Técnicas tales como los paneles de ayuda, las listas de control, los menús dependientes del contexto y los mecanismos de comprobación se emplean para indicar al usuario la utilización correcta de la metodología (ver figura 12.1). Estas herramientas proporcionan información dependiente del contexto sobre cómo empezar, qué hacer a continuación, qué entradas y salidas se requieren en cada paso de la metodología y cómo comprobar la calidad del sistema que se está desarrollando.

El conocimiento a nivel de proceso

Las metodologías de automatización del **software** son el componente más fundamental de la automatización del **software**, porque es la metodología de **software** la que guía el proceso de desarrollo. La experiencia ha demostrado que la clave del éxito de un proyecto de **software** es la elección correcta de la metodología (que pueda conducir al programador a desarrollar un **buen sistema de software**). La elección de la metodología adecuada es más importante que utilizar las mejores y más potentes herramientas.

Las herramientas que automatizan las disponibilidades requeridas por la metodología de **software** son el primer paso hacia la automatización del **software**. Automatizan solamente la producción de las disponibilidades requeridas por la metodología, no el proceso de producción de esas disponi-

bilidades. Las herramientas que automatizan los pasos de los procesos de la metodología son las bases de la verdadera automatización del **software**.

Cuadro 12.1. La tutela automática del sistema

- El profesional de desarrollo debe seguir los pasos de la metodología de un modo ordenado.
- La guía se implantará vía menús que limiten las opciones válidas basadas en las reglas de la metodología y en las comprobaciones de la calidad (por ejemplo, el programador no puede ir al paso siguiente de la metodología hasta que la tarea en curso esté completa y se certifique que las salidas de esa tarea están completas, correctas y cumplen con los estándares en curso).
- Con la utilización de pantallas de ayuda y menús, se informará al programador del próximo paso a dar, la entrada necesaria para realizar ese paso y la salida que se producirá.
- Cuando se realice un cambio, se informará al programador de los pasos previos que se pueden ver afectados, qué salidas se verán afectadas (y se ha cambiado automáticamente) y que pasos deben revisarse antes de permitir que el programador continúe.

Las herramientas que proporcionan el primer nivel de conocimiento de la metodología, el nivel de documentación, pueden ser genéricas; es decir, que soportan los requerimientos de documentación de varias metodologías de **software**. Sin embargo, por su propia naturaleza, las herramientas que procesan los niveles de guía y proceso deben corresponder a metodologías específicas. Como tales herramientas animan activamente la utilización de metodologías específicas, al programador debe comprometerse con una metodología específica.

Para automatizar los niveles más altos de la metodología del conocimiento, debe integrarse en las herramientas un modelo del proceso de la metodología de **software**. Esto presupone (1) una comprensión de cómo se lleva el proceso a la práctica; (2) qué modelos pueden garantizar un alto grado de productividad, control de gestión y de éxito que culminen en un sistema de **software** de alta calidad, y (3) una forma de codificar el modelo en las herramientas [1].

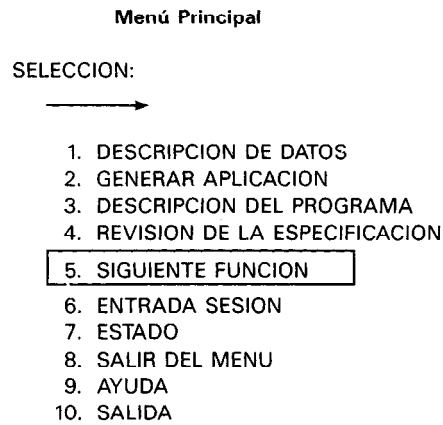


Figura 12.1. Los menús dependientes del contexto guían a los usuarios en el uso correcto de la metodología al seleccionar el siguiente paso en el proceso

Las herramientas a nivel de guía se quedan cortas en la automatización del **software**, porque no incorporan un conocimiento real de la metodología que pretenden imponer. No pueden llevar adelante el proceso de la metodología por sí mismas, independientes de la guía del experto humano. Sin embargo, las del tercer nivel, **el nivel de proceso**, el conductor de metodología tiene esta comprensión y esta capacidad. Es un verdadero sistema experto. Puede aconsejar como empezar, sugerir qué hacer a continuación, notificar cuándo se ha acabado, informar de lo que está equivocado y explicar por qué, sugerir un método mejor y, en muchos casos, realizar el proceso de la metodología por sí mismo. En otras palabras, puede imitar el comportamiento de un experto humano en metodología del **software**.

LOS COMPONENTES DE UN CONDUCTOR DE METODOLOGIA

La figura 12.2 presenta al conductor de metodología como una pirámide formada por cuatro componentes. En la cúspide hay un conjunto de herramientas de **software** potentes e integradas, como las herramientas de diagramación, lenguajes de especificación, herramientas de prototipos y generadores de código. La utilización de estas herramientas durante el proceso de construcción del sistema está dirigida por una base inteligente constituida por los tres niveles inferiores de la pirámide:



Figura 12.2. El conductor de metodología es un sistema experto que comprende y puede llevar a cabo los pasos de una metodología de software. Consta de cuatro componentes básicos. Las estrategias de software, las reglas y los hechos, y los principios básicos forman la base de conocimientos que guían la utilización del conjunto de herramientas integrado y potente

1. Estrategias y heurísticos del desarrollo de **software**.
2. Reglas y hechos de la tecnología de **software**.
3. Principios básicos de la tecnología de **software**.

Los principios básicos

En la base de la pirámide se encuentran los **principios básicos**, que son el fundamento del razonamiento de todos los sistemas expertos. Son los principios básicos de la tecnología de **software** que se han ido descubriendo a lo largo de la historia de la programación [2]. Los cuadros 12.2 y 12.3 exponen algunos de estos principios básicos pertenecientes a las disciplinas de las ingenierías de **software** y de la información, además de ser las bases de la filosofía estructurada. Empezaron en el mundo de la programación y se extendieron al diseño, al análisis y a la planificación de la información. Aunque las técnicas de creación de **software** cambiarán, estos principios básicos permanecerán constantes. Son principios importantes para controlar la complejidad y tratan de la introducción de la normalización y de la disciplina en los procesos y productos de **software**.

Divide y vencerás es uno de los principios básicos. Es el principio de la resolución de problemas difíciles, dividiendo el problema en otros más pequeños e independientes más fáciles de comprender y resolver. Es una técnica esencial y potente para tratar la complejidad. Permite al profesional del desarrollo del **software** trabajar confiadamente en una parte de un sistema grande sin preocuparse por el enorme número de detalles que cubren el sistema. Cuando se aplica a un sistema de **software** significa seccionar un programa en partes independientes que son manejables y comprobables separadamente. Cuando se aplica al proceso de desarrollo del **software** significa seccionar el proceso en pasos manejables.

El principio de la **ordenación jerárquica** es otro principio básico. Está muy relacionado con el principio de divide y vencerás. Además de descubrir que es más fácil entender un problema (y un programa) si se divide en partes, la ordenación de las piezas es igualmente importante para su comprensión. Organizar siempre las partes de una estructura jerárquica en árbol mejora la comprensión. El programa puede entenderse y construirse nivel a nivel, añadiendo más detalles a cada nivel.

Cuadro 12.2. Los principios de la ingeniería de software

Principio de abstracción

Para resolver un problema, separar todos aquellos aspectos que estén relacionados con una realidad en particular, a fin de representar el problema de una forma general y simplificada.

Principio de formalización

Seguir un procedimiento metódico y riguroso para resolver un problema.

Divide y vencerás

Para resolver un problema complejo, dividirlo en varios problemas independientes que sean más fáciles de comprender y resolver.

*Cuadro 12.2. Los principios básicos de la ingeniería de software
(continuación)*

La ordenación jerárquica

Organizar los componentes de una solución en una estructura jerárquica en árbol.

Principio de ocultación

Ocultar la información innecesaria, permitiendo que un módulo conozca sólo la información que necesita.

Principio de colocación

Colocar físicamente próximos los datos relacionados lógicamente.

Principio de integridad conceptual

Seguir una filosofía y una arquitectura de desarrollo consistentes.

Principio de totalidad

Comprobar que no falta nada.

Principio de independencia lógica

En el análisis y el diseño, concentrarse en funciones lógicas que hay que realizar, independientemente de la implantación física.

*Cuadro 12.3. Principios de la ingeniería de la información***Principio de análisis de los datos**

El análisis de los datos para identificar formalmente la estructura inherente a los datos, deberá hacerse **antes** de diseñar la lógica del proceso.

Principio de independencia de los datos

Los modelos de datos que representan la estructura lógica inherente a los datos deberá diseñarse formalmente, con independencia de cómo vayan a utilizarse los datos e independientemente de la estructura y distribución físicas de los datos.

Principio de planificación estratégica de los datos

Es una empresa, los datos requieren planificación, definición y estructuración, a fin de poderlos intercambiar entre procesos y que la gerencia pueda obtener una información más importante para el funcionamiento de la empresa.

Principio de acceso para el usuario final

Deberán proporcionarse herramientas al usuario final para poder acceder a bases de datos que pueda utilizar directamente sin programación.

Reglas y hechos

Por encima de los principios básicos (ver figura 12.2) están las **reglas** y los **hechos** para la implantación de los principios. Por ejemplo, la modularización de los programas es una aplicación del principio de divide y vencerás. Las reglas para los módulos de los programas se dan en el cuadro 12.4. El diseño orientado a objetos es otra aplicación del principio de divide y vencerás.

La estructura de control jerárquico de un programa, que es la estructura arquitectónica requerida por todos los programas estructurados, es una aplicación del principio de ordenación jerárquica (ver cuadro 12.5). Una de las diferencias esenciales entre los programas estructurados y los no estructurados, es que los primeros están ordenados jerárquicamente, mientras que los no estructurados están ordenados sólo secuencialmente. Una estructura jerárquica permite quitar los niveles superiores y tener todavía utilizables los niveles inferiores, que se pueden convertir en las piezas para construir una variante del sistema o ser la base de un nuevo sistema. Este puede ser un mecanismo potente para la creación de programas que sean fáciles de modificar y de introducir la reutilidad como técnica de desarrollo. La característica hereditaria de los lenguajes de programación orientados al objeto es otra aplicación del principio de ordenación jerárquica.

Cuadro 12.4. Las reglas para los módulos de programas

- Cada módulo representa una tarea lógica e independiente.
- Los módulos son simples.
- Los módulos son cerrados.
- Los módulos son discretos y visibles.
- Los módulos pueden probarse por separado.
- Cada módulo se implanta con una función del programa simple e independiente.
- Cada módulo tiene un solo punto de entrada y un solo punto de salida.
- Cada módulo sale a un punto de retorno estándar del módulo en el cual se ha ejecutado.
- Los módulos pueden combinarse en módulos más grandes sin el conocimiento de la construcción interna de los módulos.
- Los módulos tienen interfaces bien definidos. Los módulos tienen el control de las conexiones a través de sus puntos de entrada y salida; los módulos tienen conexiones de datos a través de parámetros y de datos compartidos; los módulos tienen conexiones funcionales a través de los servicios que se realizan recíprocamente.

Cuadro 12.5. Las propiedades de un programa estructurado

- El programa se divide en un conjunto de módulos ordenados en una jerarquía que define sus relaciones lógicas y de tiempo de ejecución.
- El flujo de ejecución de un módulo a otro se restringe a un esquema simple en el cual el control se devuelve al módulo que invoca.
- La forma del módulo es estándar y las construcciones legales de control están restringidas a la secuencia, la selección y la iteración.
- Se requiere documentación para explicar lo que hace el programa, pero no cómo lo hace.

Cuando se aplican a los procesos del **software**, las reglas del principio de divide y vencerás toman la forma de una metodología que especifica la secuencia de los pasos seguidos para realizar una tarea. Por ejemplo, la figura 12.3 muestra la versión tradicional en cascada del ciclo de vida del **software** como una secuencia de cinco fases. Hay varias metodologías estructuradas de uso común que dividen cada una de estas fases en una secuencia de pasos. En cada paso se describe la función a realizar, las entradas necesarias y las salidas producidas. El propósito es normalizar el desarrollo del **software** y asegurar la calidad del sistema de **software** obtenido.

Muchas metodologías se enfocan hacia una fase del ciclo de vida, como el análisis o el diseño. Algunas metodologías han sido diseñadas para acoplarse y alimentarse unas a otras. Por ejemplo, la metodología de análisis estructurado de DeMarco (ver figura 12.4) produce las entradas necesarias para la metodología de diseño estructurado de Yourdon (ver figura 12.5) [3,4]. La metodología de la ingeniería de la información de Martin se divide en un conjunto de estadios integrados que cubren todo el proceso de desarrollo de **software**, empezando con la planificación estratégica de los datos (ver figura 12.6) [5].

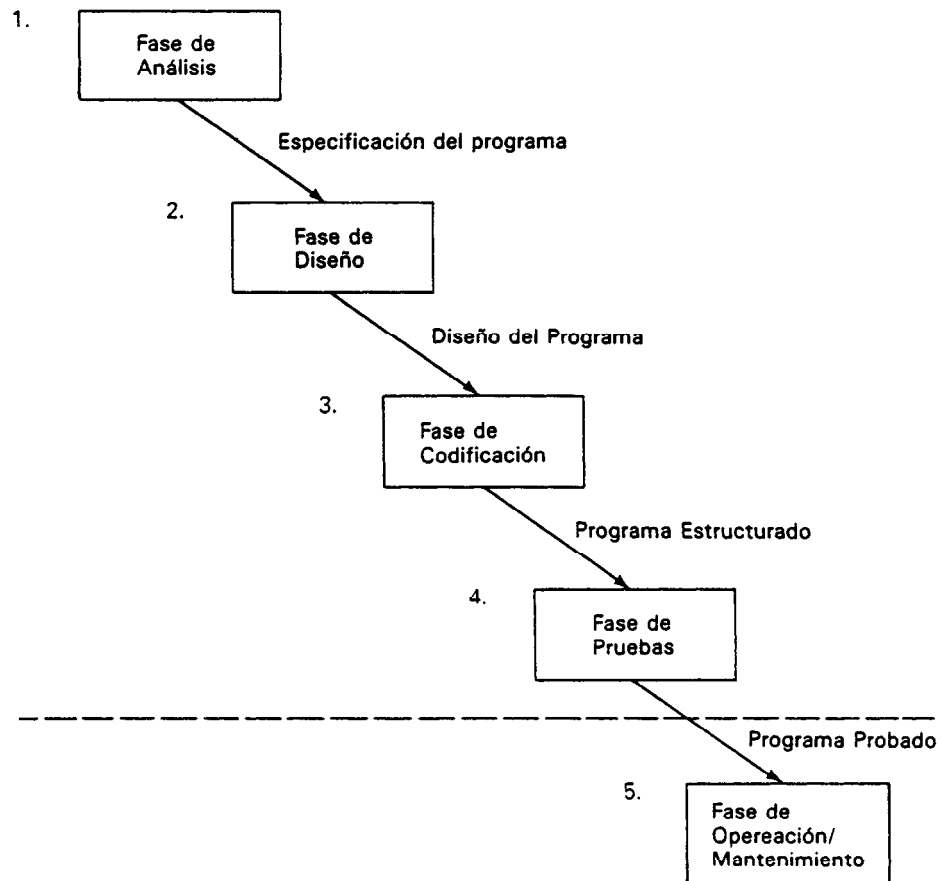


Figura 12.3. El ciclo de vida del software es un proceso que consta de cinco fases básicas secuenciales: análisis, diseño, codificación, pruebas y mantenimiento

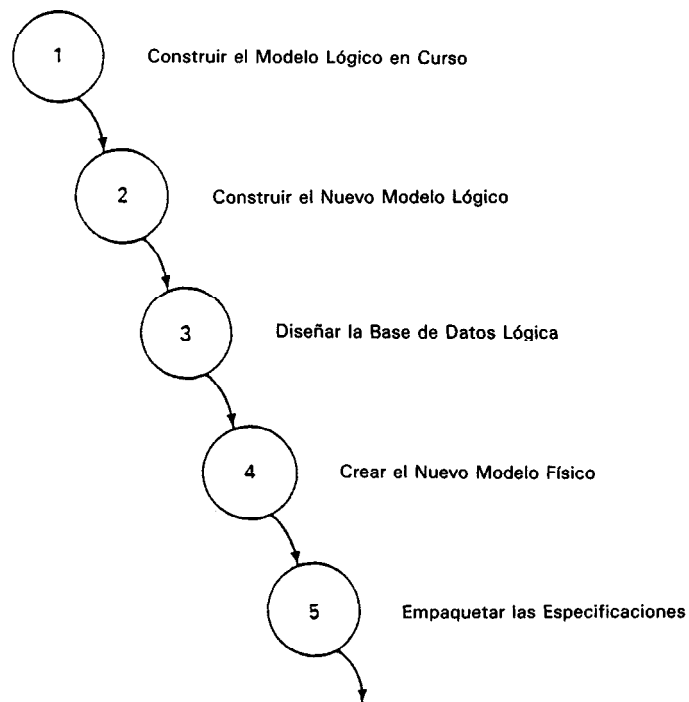


Figura 12.4. La versión del análisis estructurado de Gane-Searson consta de cinco pasos

Estrategias y heurísticos

Las **estrategias** para desarrollar el **software** están un nivel por encima de las reglas y los hechos. Incluyen estrategias tales como:

- Desarrollo descendente con descomposición funcional (ver cuadro 12.6).
- Desarrollo ascendente con reutilización de los componentes del **software** desarrollados previamente.
- Combinación de las aproximaciones descendente y ascendente.
- Diseño orientado al objeto con descomposición de objetos y herencia de las clases.

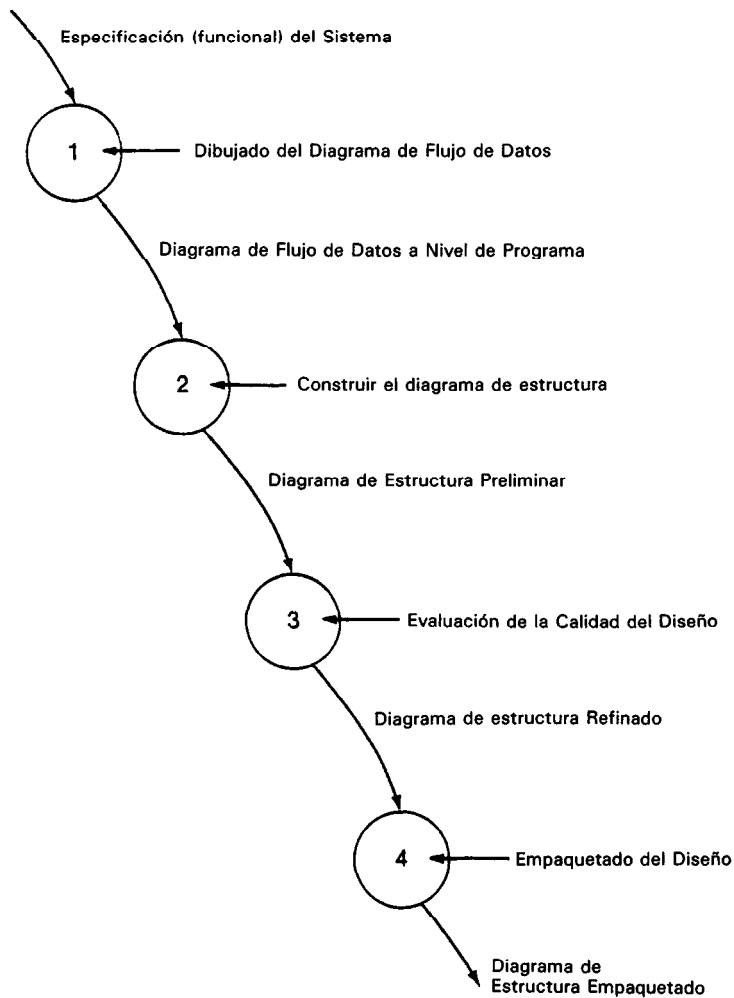


Figura 12.5. El proceso de diseño estructurado de Yourdon-Constantine se compone de cuatro pasos secuenciales

El conductor de metodología ayuda al programador a aplicar correctamente esas estrategias mediante:

- Imposición de las reglas y principios básicos de las tecnologías de software.
- Comprobando la plenitud y casos especiales.

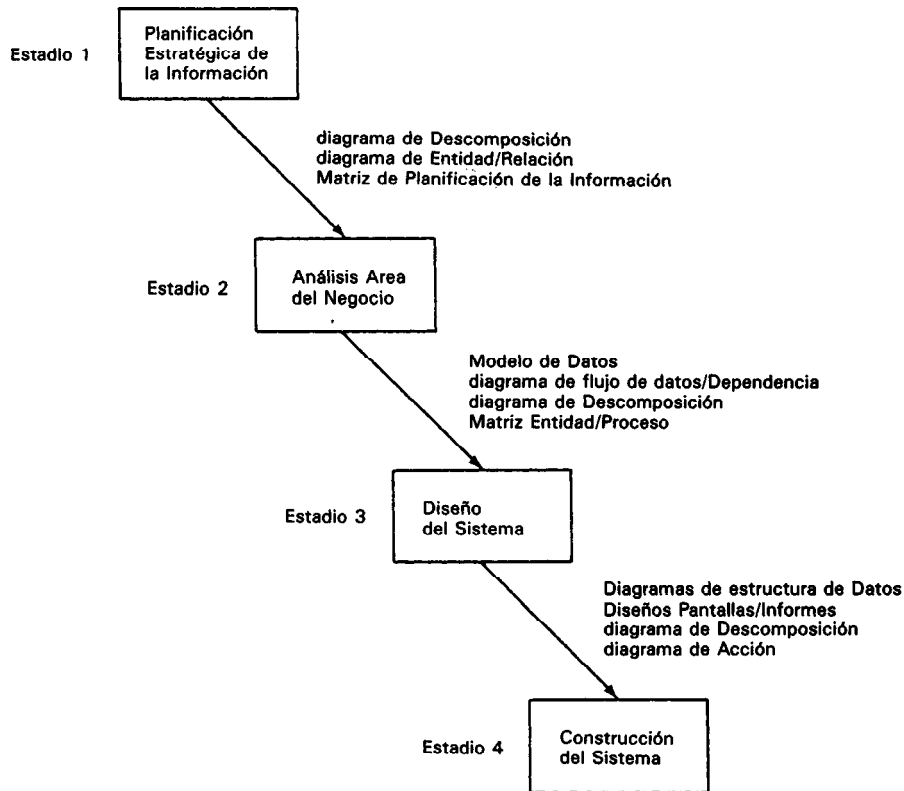


Figura 12.6. La ingeniería de la información es un proceso descendente y orientado hacia los datos para el desarrollo de los sistemas. Arranca con la Planificación Estratégica de la Información

- Informando de cómo aplicar prácticamente la estrategia.
- Seleccionando la tarea próxima a realizar

Por ejemplo, el conductor de metodología puede guiar el proceso de diseño del **software** utilizando el método de descomposición funcional y respondiendo a cuestiones como:

- ¿Cuándo debo detener la descomposición de un componente?
- ¿Qué esquema de modularización es el mejor? o ¿Es este esquema aceptable por lo menos?

Cuadro 12.6. Estrategias para guiar la descomposición funcional

- Descomponer en partes independientes.
- Posponer las decisiones sobre los detalles de implantación.
- Mantener interfaces simples entre los módulos.
- Ocultar información no esencial a los módulos.
- Mantener el uso de datos lo más local posible.
- Tomar primero las decisiones de diseño más fáciles.

- ¿Dónde pongo las decisiones en la estructura del programa? Esto es, ¿he colocado cada decisión en el mejor sitio para minimizar su impacto en el sistema cuando necesito hacer algún cambio.
- ¿Cuál será el próximo paso?
- ¿Existe algún componente que pueda utilizar?

Heurísticos

Los heurísticos son una parte importante de las estrategias. Los **heurísticos** son unas reglas empíricas no escritas que comprenden mucha de la experiencia práctica que separan al constructor experto del novel. Los heurísticos incluyen conocimiento sobre atajos y cómo limitar o ignorar las reglas para conseguir que el trabajo sea mejor y más rápido.

Por ejemplo, el proceso de diseño estructurado se guía por una de las dos estrategias de diseño siguientes:

1. Aproximación descendente.
2. Aproximación ascendente.

La aproximación descendente emplea la descomposición funcional para crear el diseño dividiendo las funciones del sistema en subfunciones. En los pasos simples de diseño, los componentes de los procedimientos y de las estructuras de datos del sistema se definen con detalle cada vez mayor.

La aproximación ascendente sigue la estrategia opuesta. Crea el diseño del sistema combinando los componentes simples para ir formando componentes más complejos. Se prefiere la aproximación descendente cuando el programador está seguro de que es posible construir componentes de niveles bajos (por ejemplo, sistemas de interfaces de los sistemas operativos) que cumplan las necesidades del sistema. También se prefiere cuando las decisiones sobre las estructuras de datos (por ejemplo, los métodos de acceso a los ficheros) deben posponerse. La aproximación ascendente se prefiere cuando el sistema es una generalización o extensión de un sistema existente. En este caso la reusabilidad del componente de **software** acorta el tiempo de desarrollo.

Sin embargo, un programador experimentado sabe que seguir estrictamente una estrategia tanto descendente como ascendente puede causar problemas y errores de diseño. Por ejemplo, cuando se sigue una aproximación descendente, las funciones comunes pueden no ser reconocidas o un diseño descendente puede requerir mucho tiempo para el desarrollo de sistemas grandes. Por otro lado, siguiendo una aproximación ascendente pura puede dar como resultado interfaces de módulos de diseño pobre, haciendo totalmente imposible la prueba minuciosa. En la práctica, el proceso de diseño generalmente es una combinación de procesos descendentes y ascendentes.

EL NUEVO PROCESO DE DESARROLLO DEL SOFTWARE

El empleo de un conductor de metodología inteligente cambia el proceso de desarrollo del **software** y la división del trabajo entre el programador y sus ayudantes automáticos. Desde la perspectiva de los programadores, el proceso resulta menos formal y con más especificaciones. La mayor parte del trabajo de desarrollo del sistema que ha de realizar el programador está relacionada con la especificación del problema a resolver (las necesidades del sistema a construir).

La forma de esta especificación cambia de una descripción narrativa a una representación gráfica y, en muchos casos, a un modelo de prototipo ejecutable. Las primeras metodologías de desarrollo estructurado requerían que se produjese el documento completo y correcto de especificaciones antes de poder desarrollar el sistema. Lo que significaba que los programadores gastaban una gran cantidad de esfuerzo para crear, revisar y

corregir los documentos de especificaciones hasta que juzgaban que estaba completo y correcto. Sin embargo, esta era una tarea casi imposible, porque las necesidades cambiaban frecuentemente. Es muy difícil acertar a un blanco móvil.

El conductor de metodología inteligente emplea técnicas de inteligencia artificial, como lógica difusa, redundancia, razonamiento probable y revisión de la idea, que pueden trabajar con datos incompletos e incorrectos y cambios en los datos de las necesidades. El papel del conductor de metodología en el proceso de desarrollo es recoger la especificación parcial e informal descrita por el programador y transformarla en un sistema completo.

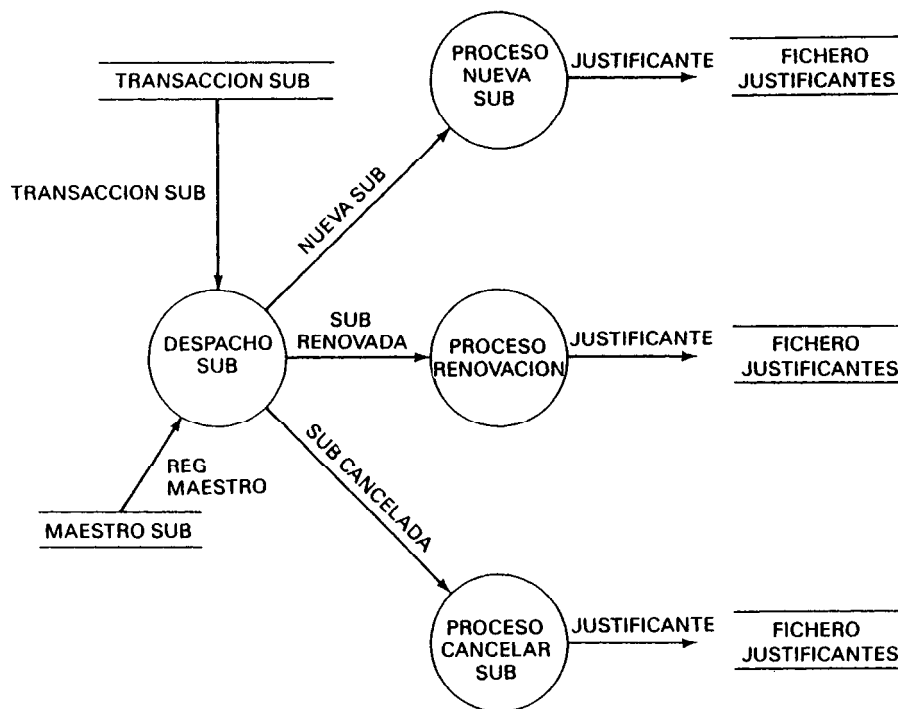


Figura 12.7. Diagrama de flujo de datos de alto nivel describiendo el sistema de suscripciones

Supongamos, por ejemplo, que los programadores empiezan describiendo un sistema de suscripciones que desean construir. El propósito del sis-

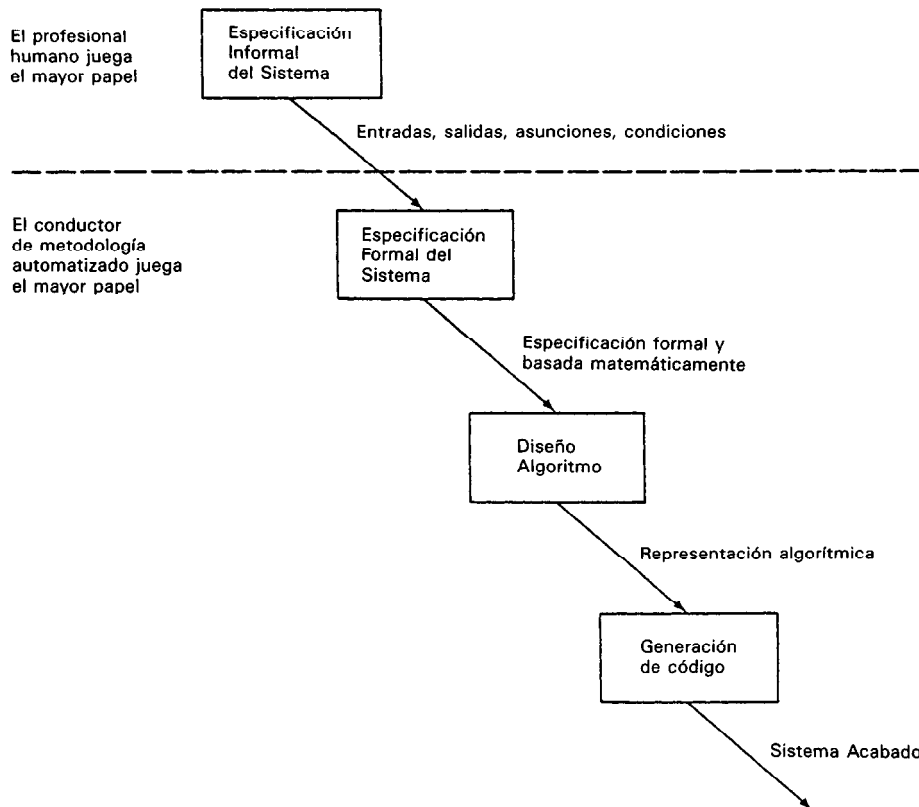


Figura 12.8. El empleo del conductor de metodología inteligente cambia el proceso de desarrollo de software y la distribución del trabajo entre el humano y el ayudante automatizado. El papel del profesional es definir los requerimientos del sistema. El papel del conductor de metodología es transformar una especificación informal en un sistema completo

tema de suscripciones es procesar tres tipos de transacciones (nuevas suscripciones, renovaciones y cancelaciones) basándose en un fichero maestro de suscripciones para emitir facturas y devoluciones. Esta descripción podría expresarse en un diagrama de flujo de datos de alto nivel como el mostrado en la figura 12.7. Describe las entradas, salidas, relaciones y suposiciones básicas para el proceso del sistema de una forma específica para la aplicación.

El programador comienza el proceso de desarrollo del sistema dibujando el diagrama de flujo de datos en la pantalla de la estación de trabajo. A partir de este punto, el conductor de metodología asiste al programador en la construcción del sistema de suscripciones. El primer paso hacia una descripción formal de este sistema es refinar el diagrama de flujo de datos para mostrar más detalles, como las entradas condicionales y los errores de proceso. El conductor de metodología asiste aquí sugiriendo que este sistema debe ser modelado según las especificaciones de un sistema existente almacenado en su base de conocimiento. Le presenta al programador una, o varias, especificación candidata, después de haberla buscado y comparado con el contenido de su base de conocimiento. También amplía la especificación para incluir componentes de proceso normalizados y poder manejar funciones como la validación de los datos, el acceso a las bases de datos, la actualización de ficheros y el proceso de errores.

La construcción de la especificación es un proceso interactivo donde el programador y el conductor de metodología intercambian información y se hacen mutuamente preguntas. El conductor de metodología requerirá, por medio del indicador (prompt), al programador la información que le falte. Sugerirá el posible paso siguiente, pero dejará al programador la decisión, por si éste elige otro. Mantendrá una memoria de lo que queda por hacer y recordará al programador los pasos que se ha saltado. Identificará lo que se ha ido añadiendo a la especificación y toda inconsistencia que haya que resolver.

Aunque al programador pueda parecerle que se utiliza un método informal, como la diagramación del flujo de datos, para definir la especificación del sistema, lo que realmente utiliza el conductor de metodología es un lenguaje de especificaciones formal con base matemática. Cada objeto de la especificación que se ha definido tiene un significado formal asociado. Un conjunto de reglas formales gobiernan como puede relacionarse con otros objetos y qué operaciones puede realizarse. Por medio del conductor de metodología, una especificación informal se transforma en una especificación formal cuya exactitud puede comprobarse.

Una vez que la especificación normal está completa, puede diseñarse el sistema y después el diseño puede transformarse en código ejecutable. Como se muestra en la figura 12.8, el conductor de metodología tiene un papel importante en los pasos restantes del desarrollo, que son una serie de transformaciones que culminan en el sistema final [6].

De nuevo, el conductor de metodología utiliza su base de conocimiento, que contiene diseños de un sistema de aplicación similar para encontrar posibles algoritmos de diseño para el sistema de suscripciones. Con la ayuda de los programadores, los algoritmos se adaptan al sistema refinando los tipos de datos, el flujo de datos y el flujo de control. La última transformación del diseño en código es bastante directa y se realiza por medio de un generador de código.

RESUMEN

El conductor de metodología es un sistema experto que incorpora conocimientos sobre metodologías de desarrollo de **software** en las herramientas de software. Su propósito es guiar y dirigir al usuario en el uso correcto de las metodologías de desarrollo de **software**. El conductor de metodología comprende los pasos del proceso de una metodología en particular y es la base de la verdadera automatización del **software**. El cuadro 12.7 resume las características de un conductor de metodología.

Cuadro 12.7. El conductor de metodología

- Incluye herramientas que imponen activamente el empleo de metodologías de **software**.
- Proporciona listas de control dependientes del contexto para tareas específicas.
- Automatiza el conocimiento sobre los mejores planes y prácticas.
- Ofrece sugerencias, consejos y explicaciones.
- Emplea un sistema experto que modela los procesos de desarrollo de **software**: imita el comportamiento de un experto en ingeniería de **software**.

BIBLIOGRAFIA

1. William E. Riddle y Lloyd G. Williams, "Software Environments Workshop Report", *ACM Software Engineering Notes*, Vol. 11, N.º 1, enero 1986, págs. 73-102.

2. James Martin y Carma McClure, *Structured Techniques: The Basis for CASE*, Edición revisada. Englewood Cliffs, Nueva Jersey: Prentice Hall, 1988.
3. Gane y T. Sarson, *Structured Systems Analysis: Tools and Techniques*. Nueva York: IST Databooks, 1977.
4. M. Page-Jones, *The Practical Guide to Structured Systems Design*. Englewood Cliffs, Nueva Jersey: Yourdon Press, 1980.
5. James Martin, *Information Engineering*. Englewood Cliff, Nueva Jersey: Prentice Hall, 1989.
6. David R. Barstow, "A Perspective on Automatic Programming", *Schlumberger-Doll Research Notes*, 14 noviembre, 1983.

CAPITULO 13

LA REUSABILIDAD DEL SOFTWARE

LA REUSABILIDAD CON LOS SISTEMAS DE CAD/CAM

A menudo se establece una analogía entre los sistemas CASE para desarrollar **software** y los sistemas CAD/CAM, que son muy utilizados en los procesos de fabricación. Los sistemas CAD/CAM han introducido un alto grado de automatización a los procesos de diseño de muchos tipos de productos manufacturados, desde aviones hasta lentes.

Para crear el diseño de un nuevo producto, el diseñador se instala en una estación de trabajo CAD/CAM y comienza por acceder a una librería automática en busca de un producto similar sobre el que basar el diseño del nuevo producto. Una vez encontrado el diseño apropiado, busca entre los componentes, accesorios e inventario de materiales disponibles, y después modifica el diseño para cumplir con las especificaciones del nuevo producto. Este método reduce substancialmente los riesgos, el costo y el tiempo de diseño y de fabricación de los productos. El diseñador no está creando a partir de cero, sino de partes que ya existen. Este proceso de diseño se basa en la reusabilidad y se ha hecho práctico gracias a los sistemas CAD/CAM.

CAMBIO REVOLUCIONARIO EN EL DESARROLLO DE SOFTWARE

La analogía entre los sistemas CAD/CAM y los CASE no es realmente apropiada hasta que la reusabilidad sea el enfoque principal en la construcción de **software**. Cuando esto suceda, veremos un verdadero cambio revolucionario en la forma de construir el **software**. Cuando utilicemos la reusabilidad como metodología de desarrollo, nunca más se volverá a crear un sistema de **software** a partir de cero, sino que iremos a la librería automática de partes del **software** reusables, tomaremos las partes, las modificaremos y las juntaremos para formar el nuevo sistema.

Usar de nuevo los componentes de **software** es la clave para un sorprendente aumento de la productividad del **software**. La reusabilidad puede acortar los programas de desarrollo en meses e incluso años. El depósito CASE, en el cual están almacenados los componentes del **software** reusable, hace práctica la reusabilidad. La CASE no solamente permite la reusabilidad y compartir código fuente, sino también la reusabilidad de la planificación de los proyectos, de los modelos prototipo, los modelos de los datos y las especificaciones de diseño.

Reutilizando los componentes de **software** y la experiencia en el desarrollo de **software**, la reusabilidad puede en gran medida:

1. Perfilar y simplificar el desarrollo del **software**.
2. Mejorar la fiabilidad del **software**.
3. Reducir el costo del **software**.

Pero es necesaria una palabra de precaución aquí. “Revolución” es un término usado hasta la saciedad en el campo del **software**. Respecto a eso, muchos directores de informática y programadores no se sienten demasiado excitados ante la sugerencia de otra promesa de revolución en el **software**, porque hasta el momento no ha habido ninguna revolución que cambiara o acelerara significativamente los procesos de desarrollo del **software**.

Básicamente, hay dos formas de acelerar el desarrollo del **software**:

1. El empleo de técnicas automatizadas de generación del código de los programas.

2. El empleo de componentes reutilizables de **software** y de la experiencia para reducir los esfuerzos de desarrollo.

Las técnicas automáticas reducen principalmente el tiempo de implantación, mientras que la reusabilidad reduce el tiempo de diseño. Parece obvio que ambas son necesarias para incrementar sustancialmente la productividad del **software**. La tecnología CASE, tratada anteriormente en este libro, acelera los procesos de desarrollo con la automatización de muchas tareas que antes se realizaban manualmente.

La experiencia de la reusabilidad en el desarrollo del **software** es hoy el medio principal de reutilizar el software. En el futuro, esta experiencia se capturará en sistemas basados en el conocimiento, llamados conductores de metodología tratados en el capítulo 12. El conductor de metodología es un sistema experto que recoge la experiencia de los programadores expertos y permite a todos los programadores el acceso a este conocimiento. En este capítulo se verá cómo al combinar el concepto de reusabilidad con la tecnología CASE se cambia y mejora el desarrollo del **software**.

LIBRERIAS DE SOFTWARE

El concepto de reusabilidad del software no es realmente nuevo. Las librerías de **software** existen desde hace años en la mayoría de las organizaciones. Están repletas de rutinas que se encuentran en muchos tipos de sistemas de **software** (sistemas de conversión de datos, rutinas matemáticas, librerías de estadísticas, paquetes de programación lineal, rutinas de procesos de entrada/salida). Las subrutinas fueron las primeras técnicas inventadas por/para ahorrar trabajo. En el año 1944 se escribió una subrutina para calcular el seno de x para el calculador Mark I [1, págs. 128-188].

Muchos programadores expertos tienen sus propias librerías personales de componentes reusables de **software**. Se ha observado que algunos de los programadores más productivos han rendido hasta cinco veces más que la media. Gran parte de su productividad puede atribuirse a un empleo exhaustivo de componentes reusables de **software**, formatos normalizados de programas y métodos estándar de implantación.

Las ventajas del **software** reusable son claras. Como se muestra en el cuadro 13.1, la reusabilidad reduce el riesgo de que falle el proyecto, acorta

el tiempo de desarrollo e incrementa considerablemente la productividad del programador individual. Según T. Capers Jones [2].

Dependiendo de la cantidad de código reusable, las tasas de productividad de más de veinticinco mil líneas de código por persona/año no es rara, con picos de más de diez mil líneas de código. Ahora es técnicamente posible desarrollar nuevas especificaciones que constan enteramente de código reutilizado, sin ningún módulo realizado manualmente.

Cuadro 13.1. Las ventajas de la reusabilidad

- Reduce el tiempo y el costo del desarrollo.
- Mejora la calidad del **software**.
- Incrementa la productividad.
- Distribuye el conocimiento sobre el sistema y sobre cómo construir los sistemas.
- Facilita el aprendizaje sobre la arquitectura del sistema y de cómo construir buenos sistemas.
- Distribuye la especificación, el diseño y el código del sistema, y de otros documentos de proyectos realizados por otros equipos.

FORMAS DE LA REUSABILIDAD DEL SOFTWARE

La aplicación del concepto de **software** reusable puede adoptar muchas formas diferentes, como:

- Prototipos reusables (la reutilización del **software** puede realizar prototipos realmente rápidos).
- Datos reutilizables.
- Arquitecturas reutilizables de sistemas y programas.
- Diseños reutilizables de programas y estructuras de datos.
- Modelos reutilizables de datos.
- Programas en códigos reutilizables.
- Paquetes de **software** reutilizables.

ESTUDIOS DE REUSABILIDAD Y PROYECTOS

Los estudios realizados sobre programas de **software** demuestran que la mayoría de las funciones realizadas por un programa son comunes a muchos tipos de programas. Por ejemplo, un estudio de aplicaciones comerciales programados en COBOL de Raytheon Company, reveló que muchas funciones comunes a nivel de organización, planificación y de aplicaciones eran reutilizables [3]. Basándose en examen cinco mil programas en COBOL, Raytheon desarrolló una librería de programas con tres mil doscientos módulos reutilizables que cubrían las siguientes clases de programas:

- Descripción de ficheros.
- Descripción de registros.
- Rutinas de edición.
- Rutinas de entrada/salida en las bases de datos.
- Módulos de interfaz de las bases de datos.
- Argumentos de búsqueda en las bases de datos.
- Llamadas a las bases de datos.
- Rutinas de funciones de aplicaciones.

Además, la librería contiene seis tipos de estructuras lógicas de programas:

- Estructura del programa de actualización de ficheros secuenciales.
- Estructura de programas de informes.
- Estructura de programas de edición de ficheros.
- Programas de compendio.
- Programas puente.
- Programas de datos fijos.

Estas estructuras lógicas básicas cubren las seis principales funciones realizadas por los programas de aplicación comercial: la clasificación, edición, manipulación, combinación, ampliación, actualización y obtención de informes sobre los datos.

Utilizando la librería de Raytheon, los programadores de software han producido programas con un 60% de código reutilizable y un intervalo de código reutilizable entre el 15% y el 85%. Raytheon cree que la reusabilidad puede suponer un incremento del 50% en la productividad del desarrollo del **software**. Además, los programadores de Raytheon opinan que

la reusabilidad introduce un nuevo nivel de estandarización que hace más fácil entender y mantener los programas. Esto puede significar hasta un 60% de reducción en los esfuerzos de mantenimiento.

El Programming Environment Project de la Universidad de California-Irvine, informó que un 62% en la construcción de los prototipos ayudó a incrementar la productividad en un veinte por ciento [4]. En las fábricas japonesas de **software**, un nivel del 85% de **software** reutilizable ayudó a incrementar en ocho veces la productividad del **software** [4].

El Departamento de Defensa de EE.UU. invirtió a mediados de los años ochenta diez millones de dólares anuales en **software** de ordenador para misiones críticas. La introducción de sistemas de defensa más complicados prometía un incremento de esta inversión. Como el Departamento de Defensa cree que el **software** reutilizable es una parte crítica en el control del creciente costo del **software**, lanzó el proyecto STARS (Software Technology for Adaptable, Reliable Systems), una iniciativa de trescientos millones de dólares para la investigación del **software** reutilizable [5]. CAMP (Common Ada Missile Packages) es el programa del STARS que está desarrollando una librería de partes de software para misiles reusable y un sistema de composición de partes. CAMP tiene definidos tres tipos de partes de **software**:

1. **Partes simples:** partes atómicas que no pueden ser modificadas.
2. **Partes genéricas:** plantillas para crear partes específicas.
3. **Partes esquemáticas:** borradores por una familia de partes y reglas de construcción para ensamblar las partes.

LOS PROBLEMAS DE LA REUSABILIDAD DEL SOFTWARE

Aunque se reconoce como una gran idea de productividad, el **software** reutilizable solo ha tenido un éxito limitado en la práctica. ¿Por qué? La primera razón es que la reusabilidad requiere una planificación de la reutilidad cuando se define e implanta originalmente un componente. La segunda razón es la dificultad de representar un componente **software** de forma que sea fácil de clasificar, describir, comprender y reutilizar sin causar efectos secundarios inesperados. En el cuadro 13.2 se listan los principales obstáculos a la reutilidad del **software**.

La solución a la practicabilidad de la reutilidad del **software** es un problema a tres bandas:

1. Elegir un formalismo apropiado para representar el componente de **software** reutilizable.
2. Proporcionar herramientas que soporten el desarrollo reutilizable.
3. Proporcionar una librería con partes reutilizables.

Cuadro 13.2. Los problemas de la implantación práctica de lo que parece una buena idea

- La cuestión de lo que es una parte reutilizable.
- La falta de normalización de los programas.
- La dependencia del lenguaje de programación.
- La decisión de qué partes van a almacenarse en la librería.
- La comprensión de los requerimientos del interfaz y de las interioridades de un componente de **software**.
- La falta de compatibilidad de los conectores.
- La comprensión de los efectos secundarios del cambio.

Cuadro 13.3. ¿Qué es un componente reutilizable de software?

- **Programa en código**
 - Fuente
 - Objeto
 - Programa completo
 - Subrutina/macro
 - Módulo/procedimiento
 - Estructura de datos
- **Diseño de programa**
 - Documentos de especificación
 - Diagramas
 - Boceto de informes/diseño de pantalla
 - Plantillas/esquemas de programas

Cuadro 13.3. ¿Qué es un componente reutilizable de software? (continuación)

- **Formas de la gestión del proyecto**
 - Hojas de cálculo
 - Horarios
 - Asignaciones
 - Formatos de informes
 - Planificación del proyecto

LA REPRESENTACION DE UN COMPONENTE REUTILIZABLE

¿Qué tipos de componentes de **software** son candidatos a la reutilización? El cuadro 13.3 sugiere algunas posibilidades.

PAQUETES

Lo más frecuente, al hablar de la reutilización del **software** es pensar en la reutilidad del código de programa, tanto completo como algunas de sus partes. Los paquetes de **software** son un ejemplo de reutilidad a nivel de programa. En el caso de los paquetes, el usuario toma el programa completo en su forma genérica, ajusta el programa por medio de los parámetros o cambia más profundamente parte del programa para particularizarlo. Los paquetes son muy populares en áreas como nóminas, banca, seguros, contabilidad, redes de comunicaciones, control de inventario y sistemas de **software**. En el cuadro 13.4 se listan los ejemplos típicos de paquetes de aplicaciones disponibles.

Quizá el mayor peligro imprevisto con los paquetes de aplicaciones es la dificultad de mantenimiento. Muchas aplicaciones comerciales cambian mucho con el tiempo y se hace necesario modificar los paquetes. Si las modificaciones se han de hacer en lenguajes como COBOL, Fortran o PL/1, requieren mucho trabajo. Para facilitar el mantenimiento, se requiere una excelente documentación y un diseño claro del paquete. Estas características deben examinarse al adquirir el paquete. En el cuadro 13.4 se relacionan los principales fallos de los paquetes.

Cuadro 13.4. Ejemplos de paquetes de aplicaciones

Administración de hospitales	Gestión de recursos
Administración Pública Autonómica	Gestión de stocks
Administración Pública Estatal	Gráficos
Administración Pública Local	
Almacenamiento de la información	Industria petrolífera
Aplicaciones científicas	Ingeniería aeroespacial
Arquitectura	Ingeniería mecánica
Ayuda a la gestión	Ingeniería eléctrica
Ayuda a las conversiones	
Ayudas en el desarrollo de aplicaciones	Lenguajes de consulta (query)
	Lenguajes de ordenador
Bibliotecas	Lenguajes de programación
Ciencias gerenciales	Matemáticas
Contabilidad de clínicas	Medidas del rendimiento del ordenador
Contabilidad de empresas	Medidas del rendimiento de los sistemas
Contabilidad de la construcción	Modelado
Contabilidad de trabajos (en el ordenador)	Nóminas
Contabilidad financiera	
Contabilidad fiscal	Planificación de proyectos
Contabilidad general	Planificadores de rutas
Contabilidad hotelera	Preprocesadores
Control de depósitos	Procesadores de documentos
Control de facturación	Procesadores de textos
Control de ingresos	
Control de inventarios	Query (lenguajes)
Control de la correspondencia	
Control de la producción	Recuperación de la información
Control de los trabajos en curso (en ordenador)	Remote job entry (RJE)
Control de pagos	Reservas de hoteles
Control de presupuestos	Reservas en líneas aéreas
Control de proyectos	
	Seguridad
Diseño asistido por ordenador	Seguros
Distribución de trabajos (en el ordenador)	Sistemas de ahorro
Distribución y ventas	Sistema de correo (mailing)
DBMS (Sistemas gestores de bases de datos)	Sistemas de distribución
	Sistemas de teletratamiento
Educación	Sistemas financieros
Enseñanza asistida por ordenador	Sistemas gestores de datos
Entrada de pedidos	Sistemas personales
Estadística	Sistemas Gestores de Bases de Datos (SGBD) o DBMS
	SGBD (Sistemas gestores de bases de datos)
Fichero de información del cliente	
	Utilidades generales
Generadores de informes	
Gestión de herencias	Verificación de los procesos

Cuadro 13.5. Fallos de los paquetes de aplicaciones

- El paquete no está lo suficientemente parametrizado y no se adapta completamente al cambio de los requerimientos.
- El proceso de datos modifica el paquete durante su instalación y el consiguiente mantenimiento llega a ser casi tan caro como los programas de aplicaciones internos de la organización.
- Se hace necesario más adelante un mantenimiento caro cuando cambia el **hardware**, el sistema operativo, los terminales, las redes o cuando cambian los requerimientos de usuario.
- El paquete se hace difícil de mantener porque la documentación es muy pobre, por falta de “ganchos” para el código creado por el usuario, por el diseño mal construido, por la ausencia de código fuente, por la excesiva complejidad, por el bajo nivel de los lenguajes, por una codificación de baja calidad.
- El paquete ha sido difícil de mantener porque ha sido mal manipulado y se han hecho modificaciones mal documentadas y difíciles de entender.
- El paquete no cumple con la implantación y estrategia de la base de datos corporativa.
- La casa suministradora del paquete interrumpe sus actividades.

Para evitar tales fallos se requiere un análisis cuidadoso de la aplicación en consideración antes de tomar la decisión de si se van a desarrollar los programas o si se va a adquirir un paquete:

- Considerar las características de la aplicación. ¿Cómo es de compleja? ¿Cuál es la prioridad de implantación? ¿Cuál es el tiempo previsto de desarrollo?
- Un largo retraso en el desarrollo puede ser una buena razón para pensar en un paquete. En la propia instalación el desarrollo de los

proyectos puede necesitar años de desarrollo y afectar seriamente al presupuesto. Un paquete de aplicación puede implantarse en uno o dos meses y a un precio fijo.

- A menudo, algunos aspectos del desarrollo interno de la instalación nunca llegan a codificarse por varias razones, mientras que el contenido de un paquete está bien definido y es conocido.
- ¿Tienen los datos de la aplicación conexión íntima con los de otras aplicaciones; por ejemplo, los entornos de las bases de datos? ¿Puede el administrador de la base de datos acomodar el paquete por medio de algún puente entre éste y los sistemas de bases de datos?
- En ocasiones, la dirección se resiste a la idea de comprar aplicaciones de origen externo. Una forma de rodear este problema es presentar claramente las ventajas económicas y resaltar que el equipo de programación quedará libre para concentrarse en áreas más importantes en el desarrollo de aplicaciones.
- La documentación, que a menudo se descuida en el desarrollo en la propia empresa, puede ser un prerrequisito necesario para adquirir un paquete de **software**. Es también una buena indicación de la calidad del producto.

La selección de un paquete de aplicación debe realizarse a través de un proceso sistemático, formal y lógico. El cuadro 13.6 ofrece una lista de sugerencias en la adquisición de un paquete.

PARTES

La reutilización de partes de un programa (es decir, reutilización del código) implica tomar código de un programa y reutilizarlo en otro programa. Normalmente, las partes reutilizables se toman de un programa que realiza una función similar y se utilizan a discreción del programador. La mayoría de los programadores tienden a reutilizar código de programas que ellos o su equipo han escrito previamente, porque conocen su existencia y comprenden íntimamente como trabaja.

Cuadro 13.6. Los pasos en la adquisición de un paquete de aplicaciones

- Enumerar en detalle los requerimientos presentes y futuros de la aplicación.
- Estudiar todos los paquetes disponibles para la aplicación.
- Examinar su documentación.
- Comprobar si está lo suficientemente parametrizado.
- Comprobar si tiene las ayudas adecuadas para su mantenimiento.
- Plantear una lista corta de paquetes apropiados.
- Examinar al proveedor ¿Proporciona un servicio adecuado?
- Hablar con otros usuarios del paquete.
- ¿Puede el paquete conectarse con las bases de datos corporativas?
- Establecer marcas si el rendimiento del paquete es crítico.
- Permite al usuario final emplear paquetes de forma temporal si el interfaz de usuario final es crítica
- Firmar un contrato apropiado.

Otra forma de reutilizar partes son las librerías compartidas de módulos de código. Estos módulos realizan tanto funciones específicas de la aplicación y funciones comunes de programas, como ayudas, entradas/salidas, clasificación, invocación, control y terminación de llamadas de programas.

Se puede conseguir un grado mayor de reutilidad cuando las partes se diseñaron originalmente con la idea de la reutilidad. Por ejemplo, en un componente de procedimiento reutilizable todas las descripciones de los datos, literales, constantes y control de entrada/salida son externas al componente. En el cuadro 13.7 se relacionan las propiedades de los componentes reutilizables ideales.

LAS LIMITACIONES DEL CODIGO REUTILIZABLE

A nivel de código, el **software** reutilizable es muy limitado. El código reutilizable es dependiente del lenguaje, del sistema operativo, de la aplica-

Cuadro 13.7. Las propiedades de un componente reutilizable

- **Expresividad:** capaz de expresar muchos tipos distintos de componentes.
- **Agregación:** capaz de combinar los componentes con un mínimo de efectos secundarios y sin interacción destructiva.
- **Base matemática formal:** permite establecer las condiciones de corrección y el proceso de combinación para preservar las propiedades clave de los componentes.
- **Forma representable en la máquina:** manipulable y computable en la máquina.
- **Independiente:** cada componente incorpora una sola idea.
- **Fácil de describir:** fácil de comprender.
- **Independiente del lenguaje de programación:** no innecesariamente específico sobre detalles superficiales del lenguaje.
- **Capaz de representar los componentes de datos y los componentes fragmentarios del procedimiento:** ofrece una fina granularidad.
- **Verificable:** fácil de probar.
- **Interfaz simple:** número mínimo de parámetros pasados y parámetros pasados explícitamente.
- **Fácil de cambiar:** fácil de modificar con un mínimo de efectos secundarios.

ción y de la discreción del programador individual. También están los problemas de (1) encontrar el código a reutilizar, (2) que la mayor parte del código no se escribió pensando en su reutilización (es decir, no está bien documentado, no utiliza el paso explícito de parámetros, no tiene un interfaz simple y fácil de entender, no realiza una función lógica). Finalmente, no se proporciona una metodología de desarrollo para el nivel de código reutilizable entre aplicaciones disimilares, ni para clasificar ni seleccionar los módulos a reutilizar.

Sin embargo el mayor problema con el código reutilizable es que a menudo hay que cambiar el código antes de poderlo reutilizar. Con los cambios existe el riesgo de introducir errores y efectos secundarios imprevistos. La dificultad de cambiar el código ha sido la principal objeción para considerar la reusabilidad del **software** como técnica de desarrollo adecuada.

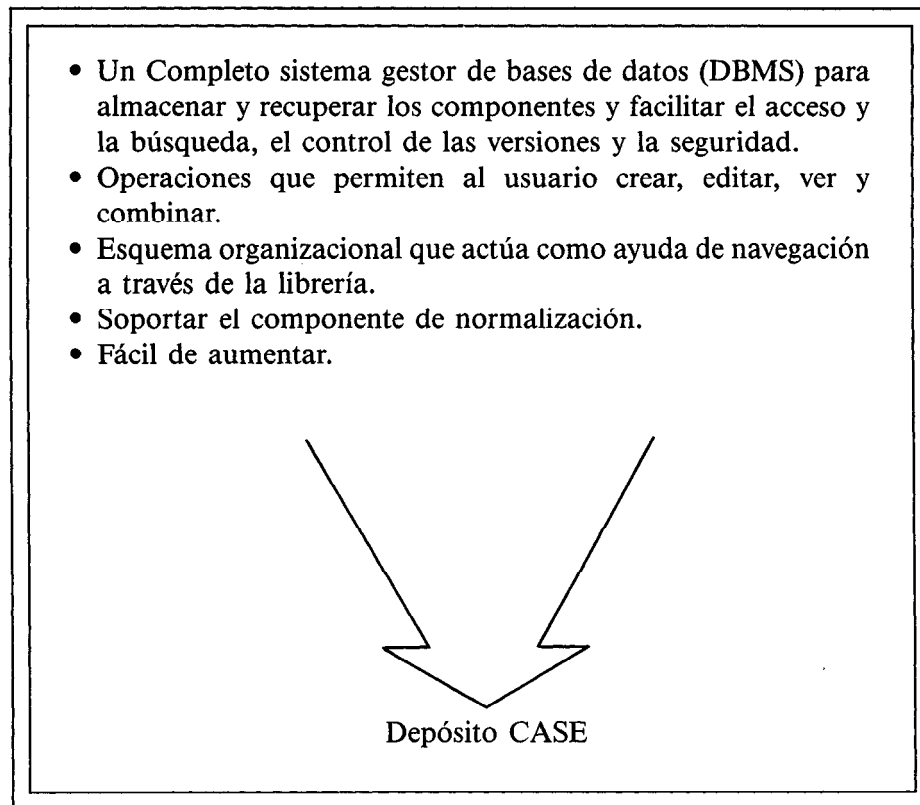
LA CASE Y EL CODIGO REUTILIZABLE

La tecnología CASE puede resolver muchos de los problemas asociados con el código reutilizable. En primer lugar, el código reutilizable no es realmente práctico sin una librería de componentes reutilizable y sin un potente sistema de gestión del sistema. Los componentes almacenados en la librería deberán tener un formato estándar y estar bien documentados y bien estructurados. Los módulos de procedimiento deben estar modularizados, orientados a tablas y parametrizados para mantener un interfaz simple. Deberán ser de alta calidad (es decir, baja complejidad, acoplamiento bajo y cohesión alta). Deberán separar los procesos de entrada/salida de los procesos algorítmicos, no debiendo limitar los valores y volúmenes de datos y deberán evitar las operaciones no normalizadas y dependientes de la máquina. Los componentes de datos deberán tener un formato del intercambio de datos estándar.

El sistema de gestión de la librería es necesario para organizar efectiva y eficientemente la indización, las referencias cruzadas y el mantenimiento de los componentes del **software**. Debe instrumentarse un esquema para la clasificación y selección de los componentes. Una posibilidad puede ser un esquema organizativo jerárquico en la cual los componentes se organizan por el tipo de aplicación y por la función dentro del tipo de aplicación. Las funciones podrían dividirse en tipos específicos de la aplicación y tipos de funciones comunes (como el acceso a las bases de datos, el tratamiento de los errores o la conversión de datos). Probablemente, la mayoría de las librerías de código reusable serán de la aplicación o de la industria (por ejemplo, una librería de componentes de software reusable para una empresa de seguros). Puede conseguirse un alto grado de reutilidad cuando se construyen aplicaciones como seguros, banca, control de inventario y reservas. Para estos tipos de aplicaciones, las versiones particularizadas del sistema pueden crearse principalmente a partir de la librería de partes reutilizables.

El depósito CASE

En el cuadro 13.8 se relacionan las propiedades de una librería de componentes reutilizables. El depósito CASE es el mecanismo que proporciona la librería de partes reutilizables y el sistema de gestión de la librería. El depósito CASE proporciona las técnicas para describir y especificar las

Cuadro 13.8. Las propiedades de la librería de componentes reutilizables

partes para que sean fáciles de identificar y recuperar para su reutilización. También contiene el conocimiento compartido entre el programador y las herramientas CASE que automatizan las distintas tareas del desarrollo de software.

La CASE cambia el impacto de las herramientas de análisis

Segundo, el código reutilizable no es práctico a menos que el código pueda modificarse fácilmente y con seguridad. Se necesitan herramientas de análisis automático del cambio y para implantar el cambio. Solamente en raras ocasiones puede reutilizarse el código sin modificaciones. Cuando se reutiliza una parte tiene que dividirse en partes menores para el nuevo

programa y modificarse para cumplir con los nuevos requerimientos (como tipos de datos, condiciones de los límites, condiciones de error).

Las herramientas CASE de análisis que ofrecen información sobre el uso (dónde y cuándo) determinan el impacto del cambio. Con una herramienta CASE de análisis el programador de desarrollo puede encontrar automáticamente la respuesta a preguntas como:

- ¿Dónde están todos los lugares en el código en los que una variable en particular se modifica o referencia?
- ¿A qué módulos llama este módulo?
- ¿Qué parámetros se pasan a un módulo?
- ¿Hay algún código inalcanzable?
- ¿Qué casos de prueba darán una mayor cobertura al cambio de código?

Mientras otras herramientas CASE de análisis sólo proporcionan una visión estática de la estructura de los datos y de los procedimientos, los animadores CASE proporcionan una visión dinámica de la ejecución del programa para ofrecer a los programadores una comprensión exacta del comportamiento del programa en tiempo-de-ejecución para evaluar su eficiencia. Los animadores ayudan al programador a detectar los problemas de eficiencia relacionados con el código reutilizable.

Finalmente, el código reutilizable no es práctico sin las herramientas que ensamblan los componentes reutilizables de **software**. Las herramientas CASE como conductores de sintaxis del programa, generadores, sistemas de verificación, calculadores de tiempos y comprobadores de la portabilidad guían al programador para combinar eficaz y correctamente los componentes para crear el nuevo sistema. Por ejemplo, el editor inteligente del Programmer Apprentice System tiene conocimiento sobre la sintaxis del lenguaje de programación y un comprobador de efectos secundarios. Tiene un conjunto especial de comandos (DEFINE, FILL, SHARE, REPLACE, REMOVE y COMMENT) diseñados para facilitar la reutilidad de los componentes de **software** y las modificaciones de programación [6]. Antes de ejecutar un comando, el editor del programa comprueba automáticamente los efectos secundarios del cambio en el programa e informa al programador.

EL NIVEL MAS ALTO DEL SOFTWARE REUTILIZABLE

Se necesita un nivel más alto de reutilización del **software** para superar la limitación del código reutilizable. Lo más cerca se está del nivel de código, menos factible es la reutilización del software. Por tanto, debemos pensar en la reutilización de las especificaciones del programa, en vez de en el código de programa. A nivel de especificación, los problemas originados por la dependencia del lenguaje de programación, la dependencia del sistema operativo y la ineficiencia del código reutilizable desaparecen. La tecnología CASE y el depósito CASE hacen práctica la reutilización de las especificaciones del programa.

LA REUTILIZACION DE LAS ESPECIFICACIONES

La reutilización de las especificaciones con el soporte de CASE funciona como sigue. El programador selecciona en el depósito CASE las especificaciones de diseño de un programa similar al que va a construir. Localiza al candidato buscando en el directorio del depósito CASE los sistemas que realizan una función igual o parecida a la que va a diseñar. El diseño de especificación está en forma de una familia de diagramas estructurados relacionados que representan las estructuras arquitectónicas del programa, los componentes de procedimiento, la estructura de datos, las entradas y los atributos. El depósito CASE reúne automáticamente todos los diagramas de sistema relacionados. La herramienta CASE generadora de código se emplea para generar automáticamente el código de programa y las definiciones de la base de datos a partir de la especificación de alto nivel.

Lo más probable es que el diseño no pueda reutilizarse tal como está. Puede ser demasiado general o, por el contrario, dirigirse sólo a un subconjunto de funciones. Por tanto, debe modificarse el diseño existente para poder satisfacer todos los requerimientos del nuevo programa. Es importante tener en cuenta que todos estos cambios se realizan a nivel de especificación de diseño y no a nivel de código. Como el depósito CASE almacena toda la información del sistema y las relaciones entre los componentes del sistema, el depósito automáticamente registra y controla el seguimiento de todos los cambios, con lo que se consigue que la reutilización del diseño de programas sea práctica. Con el uso de las herramientas CASE de análisis, el programador puede ajustar el diseño a los requerimientos del nuevo sistema.

Al ajustar un diseño, en lugar de diseñar uno totalmente nuevo, se reducirá sustancialmente el esfuerzo de análisis y diseño. Al generar el código a partir del diseño adaptado, en lugar de codificarlo manualmente, se reduce considerablemente el tiempo de implantación.

Las especificaciones formales

La reutilización a nivel de diseño requiere una mayor formalidad en la definición. Cada objeto de la especificación requiere estar formalmente definido. En la especificación formal del sistema, la composición de las especificaciones de diseño de los objetos se controla por un conjunto de estrictas reglas formales. Las relaciones entre los objetos también se definen formalmente y están limitadas por un conjunto de reglas. Este nivel de formalidad tiene base matemática. Las especificaciones formales son de la forma de cálculos de predicados, algebraicas, relacionales o declarativas, lo que permite la comprobación de la corrección sintáctica, alguna comprobación de la corrección semántica y el control de la redundancia necesarios para la transformación automática en un programa ejecutable y eficiente donde está garantizada la equivalencia computacional con la forma de las especificaciones.

Las especificaciones formales, para tener un uso práctico, deben ser lo suficientemente concisas y fáciles de entender, requiriendo menos esfuerzo para crearlas que a nivel de código de programa. Además, las especificaciones formales deben ser suficientes y fáciles de depurar a nivel de especificación, no teniendo que recurrir el programador a descender al nivel de código para la implantación final y el mantenimiento.

LA CONSTRUCCION DEL PROGRAMA CON COMPONENTES REUTILIZABLES

Como se muestra en la figura 13.1, hay dos aproximaciones básicas para la construcción de programas a partir de componentes reutilizables:

1. Aproximación por partes.
2. Aproximación en conjunto.

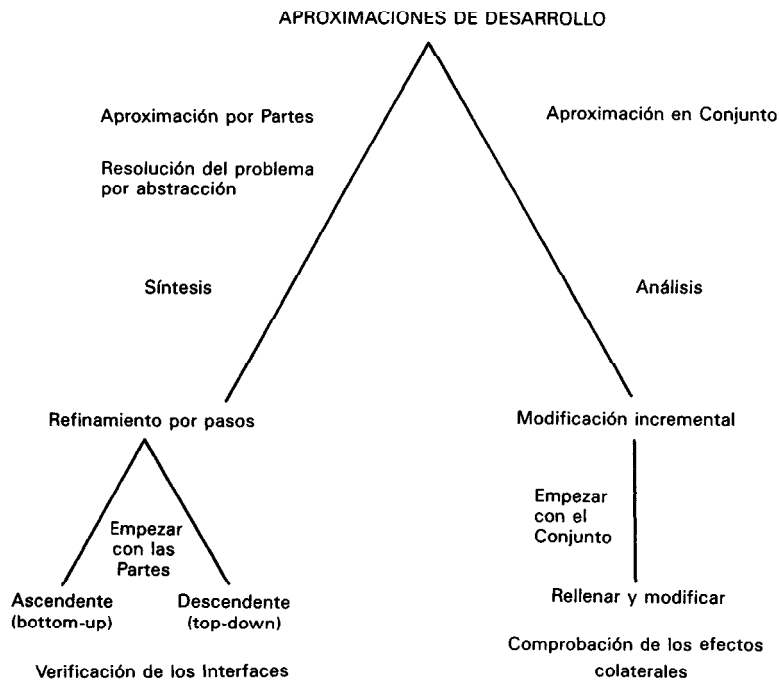


Figura 13.1. Hay dos enfoques básicos en la construcción de los programas a partir de los componentes reutilizables de software: aproximación por partes y aproximación en conjunto

La **aproximación por partes** es un **método de composición** que comienza por las partes atómicas y las ensambla para crear el nuevo programa. Se emplean para la construcción del programa tanto las estrategias de refinamientos descendentes como ascendentes. Es la solución de un problema por abstracción (es decir, utilizando una visión simplificada del problema). Los métodos de construcción en el enfoque por partes trata principalmente de la organización y combinación de las partes para formar nuevos programas de acuerdo con unas reglas de composición bien definidas. Un ejemplo de reglas de composición es el **Unix pipeline**, que es un método de composición de programas a partir de otros más simples, conectando la salida de un programa con la entrada de otro [7].

Las partes se diseñan de forma independiente y representan solamente un dato simple o un componente de procedimiento. Antes de utilizarse en un bloque, debe probarse si cada parte es totalmente correcta. El pro-

grama va creciendo parte a parte. El esfuerzo de desarrollo se centra en la integración de las partes. La comprobación de los interfaces entre las partes es crítica, porque es donde está la máxima complejidad del problema y es donde suelen ocurrir la mayoría de los errores de desarrollo.

Por otro lado, la **aproximación en conjunto** es un **método de análisis** que empieza con la arquitectura total de un programa y la modifica en pasos para satisfacer los requerimientos del nuevo programa. Se sigue una estrategia de modificación progresiva con la cual un programa existente se va ajustando hasta llegar al nuevo programa [6]. Entre los posibles ajustes están:

- El relleno de parámetros o adaptadores en plantillas.
- La unión de dos componentes en un interfaz común (por ejemplo UNIX Pipeline, objetos Smalltalk).
- La mejora de un componente ya existente con nuevas características.
- La eliminación de un componente existente para eliminar la funcionalidad no deseada.
- Cambiar los tipos de datos.

Los ajustes del programa se hacen uno de cada vez. Es una resolución del problema por reconocimientos y edición; es decir, encuentra una solución casi satisfactoria y después la modifica gradualmente hasta llegar a la solución satisfactoria. Por cada modificación del programa pueden surgir efectos secundarios imprevistos. Por tanto, gran parte del esfuerzo se centra en la comprobación de posibles efectos secundarios del cambio.

El cuadro 13.9 resume los diferentes conceptos asociados con la aproximación por partes frente a la aproximación en conjunto. Representan dos teorías distintas de la ingeniería de la resolución de problemas. De las dos, la de conjunto es la más simple. Para la mente humana es más fácil reconocer que crear algo de la nada, editar que ensamblar, empastar que refinar, siempre y cuando se disponga de potentes herramientas de apoyo.

Las herramientas CASE ayudan al programador a encontrar el diseño de programa reutilizable apropiado, a analizar el impacto de los cambios en el diseño, a comprobar la consistencia y la exactitud del diseño y a generar un programa equivalente computable a partir del diseño del programa.

De las dos, la aproximación en conjunto, tiene mayor potencial para incrementar la productividad. Puede alcanzarse un nivel muy alto de pro-

ductividad al reutilizar los componentes grandes y de alto nivel. La aproximación en conjunto permite al programador reutilizar los programas y las especificaciones de diseño en su totalidad, mientras que la aproximación por partes se centra en la reutilización de los componentes de **software**, como los módulos funcionales y las macros.

Cuadro 13.9. Los conceptos de aproximación por partes frente a la aproximación en conjunto

Conceptos de Aproximación por partes	Conceptos de Aproximación en conjunto
Abstracción	Generalización
Síntesis	Análisis
Creación	Reconocimiento
Ensamblaje	Edición
Refinamiento	Empaste
Construcción por bloques	Plantilla

LA REUSABILIDAD INCREMENTA LA PRODUCTIVIDAD

La aproximación en conjunto no sólo cambia drásticamente el desarrollo del **software**, sino que, además, alcanza la fase de mantenimiento del ciclo de vida. Cuando se emplea la aproximación en conjunto, las técnicas necesarias llegan a ser similares a las de mantenimiento de programas (es decir, técnicas para realizar cambios y comprobar el impacto de los cambios). La línea que siempre ha existido entre el desarrollo y el mantenimiento del **software** desaparece. Esta es la razón por la que la reutilidad tiene el potencial necesario para mejorar sustancialmente la productividad general del **software**. Es también la razón por la que la reutilidad revoluciona los procesos del desarrollo del **software**.

RESUMEN

La reutilidad del **software** es una respuesta importante al problema de la productividad del **software**. La reutilidad del **software** abarca mucho más que el simple código reutilizable. Las posibilidades de reutilización del **software** incluyen el código reutilizable, el diseño, las especificaciones formales, los paquetes de **software**, los generadores de prototipos y de aplicaciones y los sistemas de transformación. Los niveles altos de reutilidad (es decir, por encima del nivel de código) son muy prometedores pues evitan los problemas de la dependencia del lenguaje y las sutilezas de los efectos secundarios ocasionados por las modificaciones al código existente. Sin embargo, los componentes reutilizables de alto nivel (diseño reutilizable) serán en su mayor parte específicos del dominio o la aplicación. Es probable que se creen librerías de componentes reutilizables para diferentes dominios (aplicaciones). Las industrias tales como las de seguros o banca y los sistemas como los de defensa y de comunicaciones se prestan particularmente a la reutilidad.

Con la reutilidad debemos reconsiderar las formas fundamentales de desarrollo del **software**. Cuando la fórmula central del desarrollo es la reutilidad, la modificación progresiva reemplaza al refinamiento paso a paso en la construcción de los programas. El programa se crea modificando un programa existente que realiza una función similar para adaptarse a los requerimientos del nuevo programa.

Las herramientas CASE tienen un papel importante en hacer viable y operativa la reutilidad del **software** en la estrategia de la construcción de sistemas. El depósito CASE (y su sistema gestor de la información) es la piedra angular de las herramientas CASE necesarias para la reutilidad. Otras herramientas CASE, necesarias incluyen las herramientas de análisis, los editores orientados a la sintaxis, los generadores, los sistemas de verificación y los comprobadores de la portabilidad.

BIBLIOGRAFIA

1. Donald Knuth, *The Art of Computer Programming*, Vol. I. Reading, Massachusetts: Addison-Wesley, 1973.

2. T. Capers Jones, "Reusability in Programming", *IEEE Transactions on Software Engineering*, Vol. SE-10, N.º 5, septiembre 1984, págs. 488-493.
3. R. G. Lanergan y C. A. Grasso, "Software Engineering with Reusable Design and Code", *IEEE Transactions on Software Engineering*, Vol. SE-10, N.º 5, septiembre 1984, págs. 498-501.
4. T. A. Standish, "An Essay on Software Reuse", *IEEE Transactions on Software Engineering*, Vol. SE-10, N.º 5, septiembre 1984, págs. 494-497.
5. Christine M. Anderson, "Reusable Software—A Mission Critical CASE Study", *IEEE Compsac 85 Proceeding*, Chicago, 9-11 octubre, 1985, pág. 205.
6. R. C. Waters, "Programmer Apprentice: A Session with KBEmacs", *IEEE Transactions on Software Engineering*, Vol. SE-11, N.º 11, noviembre 1985, págs. 1296-1320.
7. T. J. Biggerstaff y A. J. Perlis, "Foreword: Special Issue on Software Reusability", *IEEE Transactions on Software Engineering*, Vol. SE-11, N.º 11, pág. 474.

PARTE 5

EPILOGO

LA CASE, CASO CERRADO

ULTIMAS CONSIDERACIONES

En el momento de escribir estas líneas hay entre 600.000 y 700.000 analistas y programadores que pueden verse beneficiados con el uso de la tecnología CASE para incrementar su productividad significativamente y preparar el camino para los altos niveles de la automatización del **software** que seguramente se conseguirán.

Actualmente, hay casi cien distribuidores en todo el mundo que ofrecen herramientas CASE. Además la CASE es una de las áreas de mayor crecimiento en la industria de los ordenadores. En 1985 solamente se habían vendido unos miles de copias de herramientas CASE en EE.UU. y fuera no había prácticamente mercado. El mercado CASE estaba dominado por un puñado de distribuidores.

En 1986, el número de copias CASE llegó a diez mil, el mercado alcanzó los 50 millones de dólares y se extendió a todo el mundo. Entraron nuevos distribuidores en el mercado, algunos de los cuales capturaron una parte sustancial del mismo.

En 1987, el mercado total de herramientas **front-end** CASE fue de 100.000 de dólares, y el de los generadores, de 50 millones de dólares. Las

compañías compraron varias copias de las herramientas CASE para su departamento de **software**. Por ejemplo, en 1987, Nastec tenía una media de dieciocho copias de DESIGNAID instaladas por cliente e INDEX TECHNOLOGY tenía más de veinte clientes con cien copias también en 1987. Muchas compañías han implantado la tecnología CASE y la han impuesto como base de su estrategia de productividad del **software**. En el futuro, las tasas de crecimiento del mercado de herramientas CASE se cree que serán de al menos un cien por cien anual.

Aunque es probable que la tecnología CASE evolucione en el futuro hacia una tecnología mucho más potente y completa, en la actualidad ya tiene más capacidades muy potentes. Tres capacidades particularmente potentes presentes hoy en las herramientas CASE son:

1. La comprobación automática.
2. El depósito CASE.
3. La generación automática de código.

Si el lector pertenece a una organización que se caracteriza por alguno de los puntos siguientes, está en una organización que puede beneficiarse de la tecnología CASE ahora:

- Una organización con una gran sobrecarga de sistemas de **software** en espera de ser desarrollados.
- Una organización que emplea la mayoría de sus recursos de personal en el mantenimiento de los sistemas existentes.
- Una organización que necesita una disciplina y un control mejores en los procesos de desarrollo del **software**.
- Una organización que desea incrementar substancialmente la productividad del **software**.
- Una organización que desea estar al día en el empleo de las tecnologías del **software**.

En los años noventa, las herramientas CASE y las estaciones de trabajo para el desarrollo serán tan comunes para el desarrollo de **software**

como lo han sido durante tres décadas los lenguajes de programación y los compiladores. La ingeniería de **software** asistida por ordenador tendrá una posición dominante entre las tecnologías de **software**.

INDICE ALFABETICO DE TERMINOS

A

Abstracción, principio de, 323
acceso del usuario final, principio del, 214
acción, diagramas de, 53, 57, 58, 64, 74, 112, 156, 157
acoplamiento, 140
acoplamiento de contenido, 140
acoplamiento de datos, 140
activa, AYUDA, 306
activo, HELP, 306
ADS/ONLINE, 262
aférentes, ramas, 137, 138
almacén de datos, 73
análisis de la complejidad, 71-72
análisis de la conservación, 73
análisis de los datos, principio, 330
análisis, ciclo de vida del software, 246
análisis, herramientas de, 183-187
análisis, metodología en la construcción de sistema, 362-365
ANALYST DESIGNER, 50, 55, 70, 84, 91, 123, 182
animación, 280
animadores CASE, 360
aplicación, paquetes de a. disponibles, 353
aplicación, paquetes de a., fallos (aplicación, paquetes de a., fallos de los), 354

aplicación, paquetes de a., pasos en la adquisición, 356
APPLICATION FACTORY, experiencias de productividad, 214-220
APS, 182
ARCO, experiencias en la utilización de EXCELERATOR, 213
ascendente (bottom-up), desarrollo, 334, 337
asombro mínimo, principio, 298
Asyst, 182
automatización del software, 21-22
automatización del software, características, 275-283
AUTO-MATE PLUS
ayuda inteligente, 308-312

B

Babcock&Wilcox, experiencias con INFORMATION ENGINEERING WORKBENCH, 223-224
back-end, componente, 171, 172, 285, 286
banco de trabajo (workbench) CASE, 36, 42-44, 59, 64, 180, 200-202, 285-286
bottom-up (ascendente), desarrollo, 334, 337
Bravo, 295
BUILDER LANGUAGE, 58

C

- Cadre, 52, 63, 81, 182
- CAD/CAM, sistemas, 345
- Calculus Tutor, 315-318
- CAMP, 350
- capacidades gráficas, 44-46
- CASE toolkits (ver juego de herramientas CASE)
- CASE tools (ver herramientas CASE)
- CASE workbench (ver banco de trabajo CASE)
- CASE workstation (ver estación de trabajo CASE)
- CASE y el código reutilizable, 358
- CASE y la cuarta generación, 263-265
- CASE y la quinta generación, 267
- CASE, casos de estudio, 209-229
- CASE, causas de los fallos en la utilización de la, 228-229
- CASE, componentes de la, 171-176
- CASE, definiciones, 36
- CASE, determinación de las necesidades, 230-233
- CASE, evaluación, 242-243
- CASE, historia, 26-33
- CASE, implantadores de la, 238-239
- CASE, objetivos, 33-34
- CASE, plan de implantación, 234
- CASE, relación con otras tecnologías de software, 259-270
- CASE, selección de las herramientas, 239
- CASE, utilización de un proyecto piloto, 238
- CASE, vender la, 239
- CASE, ventajas de la, 23
- CASE, ventajas sobre la cuarta generación, 263-265
- ciclo de vida CASE de software, 250-257
- ciclo de vida CASE de software, cambios en el, 251-254
- ciclo de vida CASE de software, problemas con el, 254-257
- ciclo de vida (ver ciclo de vida de software)
- COBOL, 107, 108, 190-191, 349
- compañeros de metodología CASE, definición de los, 37
- compañeros de metodología CASE, ejemplos, 203
- componente front-end, 171, 172, 285, 286
- comprobación de errores, 66-80
- comprobación de la consistencia, 69-71
- conductor de metodología, 281-282, 323-342
- conductor de metodología inteligente, 338-339
- conductor de metodología y conocimiento a nivel de procesos, 324-326
- conductor de metodología y conocimientos a nivel de documentación, 324
- conductor de metodología y desarrollo de software, 338-342
- conductor de metodología, características, 338-342
- conductor de metodología, estrategias relativas a, 334-338
- conductor de metodología, heurísticos relativos al, 337-338
- conductor de metodología, principios básicos, 327-330
- conductor de metodología, reglas y hechos relativos al, 330-334
- conductor inteligente de metodología, 338-342
- conocimiento a nivel de documentación, 324
- conocimiento a nivel de guía, 324
- conocimiento a nivel de proceso, 324-326
- consistencia, validación de la, 69-71
- constructores de menús, 103
- Cortex, 112
- Cortex Corporation, 211, 214
- CORVISION, 112
- coupling (ver acoplamiento)
- cuarta generación, herramientas de, 260-265
- Cullinet, 182, 188

CH

- CHEN toolkit, 188
- Chen & Associates, 188

DeMarco, metodología de análisis estructurado de, 116, 126, 127, 324, 332
 Departamento de Defensa (Department of Defense, DOD), 350
 depósito CASE, 32, 33, 64, 80-97, 171-175, 268, 270, 346, 358, 361, 366
 depósito CASE, características, 93-94
 depósito CASE, consideraciones en la implantación del, 93
 depósito CASE, contenido del, 85-87
 depósito CASE, gestión de la información, 89-93
 depósito CASE, informes, 87-88
 depósitos múltiples, 172-173
 descomposición funcional, 337
 descomposición funcional, validación de la, 71-72
 DESIGN MACHINE, 182
 DESIGNAID, 54, 105, 131, 159, 182
 DEVELOPER, 182
 diagrama de contexto, 52, 125
 diagrama de dependencia, 157
 diagrama de descomposición, 49, 51, 67, 153
 diagrama de entidad, 161
 diagrama de entidad/relación, 27, 31, 55, 57, 110, 153, 154
 diagrama de estructura de datos, 110
 diagrama de estructura jerárquica en árbol, 153
 diagrama de flujo de control, 49, 52, 82, 122, 125
 diagrama de flujo de datos, 27, 30, 45, 48, 50, 59, 60, 64-82, 90, 129, 130, 138, 141
 diagrama de flujo de datos, símbolos utilizados en los, 135
 diagrama de modelo de datos, 56, 57, 59, 73
 diagramación automática, 59-63
 diagramas, necesidad de los, 46-47
 diccionario CASE, 64
 diccionario de datos, 129
 diccionarios, 27
 diccionarios, entradas de, 73
 diseño en el ciclo de vida del software, 246
 diseño orientado a objetos, 334
 Disus, 182
 divide y vencerás, principio de, 328
 documentación en línea, 307
 documentación, requerimientos de la, 47
 DOD y el software reutilizable, 350
 DOD (Department of Defense), 350
 do-what-I-mean (DWIN), sistemas, 305
 DSSD (Data Structured Systems Development), metodología, 158-163
 DuPont, experiencias con APPLICATION

FACTORY, 216-220
 DWIN (do-what-I-mean), sistemas, 305

E

EDIF (Electronic Document Interchange Form), 174
 eferentes, ramas, 137, 138
 Electronic Document Interchange Form (EDIF), 174
 enciclopedias, 27
 énfasis en el front-end, 102, 249
 entidad/relación, diagrama de, 27, 31, 55, 57, 110, 153, 154
 entorno habitable, 281, 285-320
 entorno habitable, amigable con el usuario, 287-292
 entorno habitable, ayudante, 306-312
 entorno habitable, características, 319
 entorno habitable, corrector, 305
 entorno habitable, diagnosticador, 303-305
 entorno habitable, dirigido al usuario, 293-302
 entorno habitable, interfaz de usuario, 285-286
 entorno habitable, más allá del entorno CASE, 286-287
 entorno habitable, reactivo, 302-303
 entorno habitable, tutor inteligente, 312-319
 errores de diseño, 102
 especificación de procesos, 129-130
 especificación formal y la reutilización, 362
 estación de trabajo (workstation) CASE, 37, 166-168
 EXCELERATOR, experiencias de productividad con, 211-214
 exceso semántico, 301
 extensiones del diseño, 186-187

F

FOCUS, 262, 296
 formalidad, principio de, 328
 frameworks, 199-200
 frameworks, juego de herramientas CASE, 199-200
 front-end, componente, 171, 172, 285, 286
 front-end, énfasis en el, 102, 249
 funcional, cohesión, 142
 funcional, descomposición, 337

G

Gane-Searson, metodología de análisis
 estructurado, 27, 116, 126, 127, 132-135
 generador CASE de código
 generador de código COBOL, 107, 108,
 190-191
 generadores de código, 106-108, 189-191
 generadores de informes, 103
 generadores de pantallas, 103
 gráficas, capacidades, 44-46
 guía automática del sistema, 324

H

hardware, plataformas de, 165-176
 Hatley, metodología de diseño de, 116, 124
 HELP a nivel de comando, 306, 309
 HELP inteligente, 309-312
 HELP interactivo, 309
 HELP introductorio, 306
 HELP por niveles o escalonado, 306
 HELP sensible al contexto, 306
 HELP, componente de un sistema, 306-312
 herramientas CASE de análisis, 183-187,
 359-360
 herramientas de análisis, 183-187
 herramientas de cuarta generación, 262-265
 herramientas de diagramación, 184
 herramientas de programación, 188-191
 herramientas de prototipo, 103-105
 herramientas de reestructuración automática de
 programas, 195
 herramientas de reestructuración de programas,
 195
 herramientas del mantenimiento automático,
 191-195
 herramientas de gestión del programa, 198
 herramientas CASE, 24-26, 268, 359, 364, 366
 herramientas CASE, categorías de, 179-204
 herramientas CASE, de documentación, 324
 herramientas CASE, definición, 36-37
 herramientas CASE, diagramación, 324
 herramientas CASE, distribución de las, 179
 herramientas (tools) cuarta generación,
 categorías, 262
 herramientas (tools) cuarta generación, futuro
 de las, 270
 herramientas (tools) cuarta generación, la
 CASE versus las, 263
 herramientas (tools) cuarta generación,
 ventajas de la CASE sobre las, 268-269

herramientas (tools) cuarta generación,
 ventajas e inconvenientes, 262-263
 heurísticos, 337-338

I

IDEAL, 262
 IDMS ARCHITECT, 182, 188
 IEW (ver INFORMATION ENGINEERING
 WORKBENCH)
 independencia de los datos, principio, 330
 independencia lógica, principio de la, 329
 Index Technology Corporation, 30, 56, 125,
 182, 211
 información de los errores, 80
 INFORMATION ENGINEERING FACILITY,
 31, 51, 105, 107, 110, 154, 155, 181, 200
 INFORMATION ENGINEERING
 WORKBENCH, 29, 53, 60, 65, 67, 81, 90,
 130, 156, 200
 INFORMATION ENGINEERING
 WORKBENCH, experiencias de
 productividad, 221-224
 ingeniería de la información, metodología, 27,
 128, 150-158, 332, 336
 ingeniería de la información, premisas básicas,
 121
 ingeniería de la información, principios, 330
 ingeniería del software, 113-116
 ingeniería del software, pasos básicos a seguir
 en la, 116, 117
 ingeniería del software, premisas básicas, 114
 ingeniería del software, principios básicos,
 327-330
 ingeniería del software, tipos de diagramas
 utilizados en la, 115
 integración de herramientas, 94-97
 integridad conceptual, principio de la, 329
 Interactive Development Environments, 182
 Interlisp, 42
 i-Logix, Inc, 182

J

Jackson, metodología de diseño de, 109, 117,
 124, 128, 143-150
 jerárquica, principio de ordenación, 328, 329
 juego de herramientas de diseño de datos, 188
 juego de herramientas CASE, 179-200
 juego de herramientas CASE de análisis,
 183-187

juego de herramientas CASE de diseño de datos, 188

juego de herramientas CASE de gestión del proyecto, 198

juego de herramientas CASE de mantenimiento, 191-197

juego de herramientas CASE de programación, 188-191

juego de herramientas CASE, definición, 36-37

juego de herramientas CASE, frameworks, 199-200

K

KnowledgeWare, 29, 53, 60, 65, 67, 81, 90, 130, 139, 156, 200, 221

L

LCP (Logical Construction of Programs), 158

Learmonth and Burchett, 182, 188

lenguaje natural, sistemas en, 301

LIFE CYCLE MANAGER, 182

Logical Construction of Programs (LCP), 158

M

Mantenimiento, ciclo de vida de software, 248

mantenimiento, herramientas de, 191-195

Martin, metodología de la ingeniería de la información de, 27, 128, 150-158, 332, 336

McCabe, métrica de la complejidad de, 71

McDonnell-Douglas Company, 75, 76, 77, 83, 92, 182

método de composición para la construcción de programas, 362-365

metodología Data Structured Systems Development (DSSD), 158-163

metodología de desarrollo de software de Orr, 109, 128

metodología de diseño de Hatley, 116, 124

metodología de la ingeniería de la información, 27, 128, 150-158, 332, 336

metodología DSSD (Data Structured Systems Development), 158-163

miniespecificación, 73, 130, 131

modos, 295

módulos de programas, reglas de los, 331

N

Nastec, 54, 105, 131, 159, 181, 182

NATURAL, 262

necesidad de los diagramas, 46-47

O

ocultación, principio de, 329

on-line, documentación, 307

operación, ciclo de vida de software, 247

Oracle, 188

ordenación jerárquica, principio de, 328, 329

orientado a los datos (data-oriented),

desarrollo de software, 117, 118

orientado a objetos, diseño, 334

orientado al procedimiento, desarrollo de software, 117-118

Orr, metodología de desarrollo de software, 109, 128

Orr&Associates, 182

P

Pansophic, 104, 111, 182

plataforma de hardware de estación de trabajo individual, 166-168

plataforma de hardware, consideraciones, 176

plataforma de hardware, datos y los lenguajes de cuarta generación, 171

plataforma de hardware, de niveles múltiples, 168-171

plataforma de hardware, puentes entre las estaciones de trabajo, 168

plataforma de hardware, puentes entre los diccionarios, las bases de, 170

plataformas hardware CASE, 175-176

PL/1, 111

principio de divide y vencerás, 328

principio de formalidad, 328

principio de la independencia de los datos, 330

principio de la independencia lógica, 329

principio de la integridad conceptual, 329

principio de ocultación, 329

principio de ordenación jerárquica, 328-329

principio del acceso del usuario final, 214

principio del análisis de los datos, 330

principio del asombro mínimo, 298

productividad del software, 209-226

productividad e INFORMATION

ENGINEERING WORKBENCH, 221-224

productividad y APPLICATION FACTORY, 214-220
productividad y el software reutilizable, 366
productividad y EXCELERATOR, 211-214
productividad, estadísticas de, 210
Programmer Apprentice Systems, 360
Programmers Workbench, 192-195
Programming Environment Project, 350
PROKIT*ANALYST, 182
PROKIT*WORKBENCH, 75, 76, 77, 83, 92
PROMOD, 182, 201
Promod, Inc., 182, 201
PROMOD/MODULAR DESIGN TRANSFORMATION PROCESS, 201

Q

QBE, 262
QMF, 262
quinta generación versus la CASE, 267
quinta generación, herramientas de, 260, 265-267
quinta generación, futuro de la, 270

R

Rapid Iterative Production Prototyping (RIPP), 216-220
rastreo de los requerimientos, 74-80
Raytheon Company, 349
reactivo, sistema, 301-302
repository ver depósito
requerimientos de la documentación, 47
REQUIREMENTS MANAGEMENT SYSTEM, 182
reutilización (ver software reutilizable)
RIPP (Rapid Iterative Production Prototyping), 216-220

S

Sage, 182
SCHEMAGEN, 182
seudocódigo, 57, 73
simulación, 106
sistema autocorrector, 305
sistema consistente, 296-298
sistema flexible, 298-300
sistema robusto, 289
sistema sensible, 302-303
sistema simple, 294-296

sistemas de información, 121-124
sistemas de manipulación directa, 288
sistemas de tiempo real, 121-124
sistemas de tiempo real/embebidos, 106
sistemas expertos, 265-266
sistemas expertos, características, 266
sistemas transparentes, 288
software, ciclo de vida CASE del, 250-257
software, ciclo de vida del, 267-269
software, ciclo de vida del s. como concepto de unificación, 246-248
software, ciclo de vida tradicional, 248-250
software, estación de trabajo (workstation) para el, 42
software, productividad del, 209-210
software, reutilización del, 345-367
software, reutilización del s. componentes de la, 338, 351, 352-356, 357
software, reutilización del s. con partes de programas, 355-356
software, reutilización del s. con sistemas CAD/CAM, 345
software, reutilización del s. de alto nivel, 361
software, reutilización del s. y la construcción de programas, 362-365
software, reutilización del s. y la productividad, 365
software, reutilización del s. y las especificaciones, 361-362
software, reutilización del s. y paquetes de aplicación, 352-355
software, reutilización del s., estudios y proyectos, 349-350
software, reutilización del s., formas de la, 348
software, reutilización del s., problemas con la, 350-352
software, reutilización del s., ventajas de la, 348
SPADE-O, 315, 317, 318
SPD (Structured Program Design), 158
SQL, 262
SQL*DESIGN DICTIONARY, 188
STARS, 350
STATEMATE, 182
structure charts (ver diagrama de estructura)

T

TAGS, 182
TEAWORK, 52, 63, 82, 182
TekCASE ANALYST/RT, 201
TekCASE DESIGNER, 182, 201
Tektronix, 182, 201

TELON, 104, 111, 182
 tercera generación, herramientas de, 259,
 269-270
 Texas Instruments, 31, 51, 105, 107, 110, 154,
 155, 181, 200
 tiempo real, herramientas de análisis, 185
 tiempo real, sistemas de, 121-124
 tiempo real/embebidos, sistemas de, 106
 Touche Ross, experiencias con
 EXCELERATOR, 213-214
 transformación central (diseño estructurado de
 Yourdon), 137
 transición de estado, diagrama de, 123
 tutor inteligente, capacidades, 314
 tutor inteligente, características, 313
 tutor inteligente ejemplos de, 315-319

U

unidades de carga, 142
 Unix, 42
 UNIX pipeline, 363
 usuario, ilusión del, 298
 usuario, interfaz del, 285-286
 usuario, sistema controlado por el, 300
 usuario, sistema orientado al, 293-302
 usuario, características del sistema orientado
 al, 293-294

V

validación de errores, 65-80
 validación de la consistencia, 69-71
 validación de la totalidad, 69-71
 validación de metodologías específicas, 72
 validación semántica, 72
 verificación de los tipos, 68-69
 verificador de especificaciones, 186
 verificador de sintaxis, 68-69

W

Ward/Mellor, metodología de diseño de, 116
 Warnier-Orr, diagrama de, 54, 56, 158-163
 Warnier-Orr, metodología de diseño, 109, 118

X

Xerox Star, 294-296

Y

Yourdon Software Engineering Company, 50,
 55, 70, 84, 91, 123, 182
 Yourdon, metodología de diseño estructurado
 de, 28, 71, 116, 127-128, 134-142, 332, 335
 Yourdon, pasos en la metodología de diseño
 estructurado, 136-142

