FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS INSTITUCIONALES

No. 70

METODOLOGIA PARA EL DESARROLLO DE LOS SISTEMAS DE INFORMACION

DEL 9 al 29 DE SEPTIEMBRE

SEPOMEX

INTRODUCCION

MEXICO, D.F.
PALACIO DE MINERIA
1992

TALLERES DE METODOLOGIA Y ESTANDARES PARA DESARROLLO DE SISTEMAS.

1

T E M A 1: Introducción.

La metodología y los estandares para el desarrollo de sistemas se situan en un contexto más amplio de la administración.

Historia de la Administración:

La evolución de la administración la podemos dividir en cinco etapas: empírica, científica, de las relaciones humanas, clásica y moderna.

La etapa empírica se puede considerar hasta el año 1900, en todo este periodo se empleó la administración eficientemente pero sin una teoría base: para el gobierno de los paises, la administración de la iglesia, para la administración de los gremios, de las primeras industrias y del comercio en general. Su característica principal es la de que los supervisores decían que hacer, pero no como hacerlo.

La etapa de la administración científica esta marcada por las contribuciones de Taylor, Gilberth, Gantt y Emerson, que comprenden de 1870 a 1930. En esta epoca se destacó la importancia de tiempos y movimientos, concentrandose pues en el método y los estandares.

En la etapa de la administración de las relaciones humanas que comprende de 1930 a 1940, el sicólogo Mayo y el sociologo Roethlesberger señalaron la importancia de la cohesión y solidaridad de grupo; así como, de los factores sicológicos del trabajo.

Simultaneamente se desarrollo la etapa de la administración clásica de 1900 a 1940. Fayol destaca la importancia de los elementos de la administración (planear, organizar, mandar, coordinar y controlar) por su parte Mooney derivó sus principios.

Es hasta la etapa de la administración moderna de 1940 a la fecha, que se profundiza en temas como la motivación, los estilos de liderasgo, la dinámica de grupos, la dinámica organizacional y la toma de decisiones con metodos cuantitativos.

Proceso Administrativo:

La división más aceptada con respecto a las fases del proceso administrativo es la que señala cuatro funciones: planeación, organización, dirección y control.

La PLANEACION incluye la previsión en la que se elaboran pronósticos. Los planes de uso constante incluyen: objetivos, políticas y procedimientos, los de uso por única ocación son los presupuestos y los programas.

- Pronósticos.- Apreciación de lo que juiciosamente se espera que ocurra.

- Objetivos.- Determinación de los resultados que se pretenden y deben lograrse, dentro de límites de variación razonables.

- Políticas.- Decisiones permanentes sobre asuntos importantes y recurrentes.

- Procedimientos.- Serie de actividades concatenadas que definen el orden cronólogico y la forma establecida de ejecutar el trabajo.

- Programas.- Es el establecimiento por escrito de los objetivos, la secuencia de las operaciones, el tiempo requerido para realizar cada una de las partes, la persona (u órgano) responsable y los recursos necesarios para su realización.

- Presupuestos.- Es la expresión en unidades físicas y/o monetarias de los programas.

La ORGANIZACION abarca la elaboración de una estructura formal; agrupación de funciones por dependencia (u órgano); especificar líneas de autoridad, reporte, responsabilidad, comunicación y toma de decisiones; delegar autoridad y responsabilidad; y diseño de puestos.

- Organograma.- Representación gráfica de la estructura formal de una institución, que muestra los niveles jerárquicos y las líneas de dependencia.

- Descripción funcional de las áreas.-
- Descripción de puestos.-

La DIRECCION es una fase dinámica que implementa los lineamentos para el manejo del personal; integra las necesidades individuales con las institucionales creando un clima de motivación; realiza la supervisión bajo un estilo de liderazgo orientado a los resultados con: comunicación eficáz, delegación efectiva de autoridad, orientación y desarrollo del personal.

El CONTROL es otra fase dinámica de la administración que utiliza las normas, reglas y patrones para identificar las realizaciones exitosas de acuerdo a los reportes de resultados; toma las acciones correctivas donde y cuando sea necesario. En otras palabras se asegura de que lo planeado corresponda a lo realizado.

G L O S A R I O.

METODO.- Modo de hacer con orden una cosa.

METODOLOGIA.- Ciencia del método.

PROCEDIMIENTO.- Método de ejecutar algunas cosas.

PROCESO.- Conjunto de fases sucesivas de un fenómeno.

TECNICA.- Conjunto de procedimientos y recursos se que se sirve una ciencia.

DOCUMENTAR.- Probar, justificar la verdad de una cosa con documentos.

ESTANDARD.- (Anglisismo) Norma, medida, patron, modelo, regla fija.

REGLA.- Modo de ejecutar una cosa.

NORMA.- Regla que se debe seguir o a que se deben ajustar las operaciones.

MODELO.- Ejemplar o forma que uno se propone y sigue en la ejecución de una obra.

MEDIDA.- Unidades que se contemplan para medir un trabajo.

PATRON.- Que sirve como muestra para sacar otro igual.

Metodología y Estandares:

La metodología para desarrollo de sistemas es el conjunto organizado de procedimientos que integran el proceso para el desarrollo de sistemas.

Los estandares para desarrollo de sistemas son el conjunto de normas, reglas, medidas y modelos utilizados en la planeación y control del proceso de desarrollo de sistemas.

Un proceso es el conjunto de fases sucesivas en este caso - las fases sucesivas para el desarrollo de sistemas.

Un proyecto es una forma de planeación sinónimo de un programa.

La primera estandarización que podemos realizar es en la terminología; para evitar confusiones con los terminos del proceso de administración de la función informática en general.

Llamaremos: metodología a los procedimientos integrados y organizados para el desarrollo de sistemas; estandares a sus normas, reglas, medidas y modelos y proyectos a sus programas de trabajo.

Ciclo de Desarrollo de Sistemas:

Al proceso de desarrollo de sistemas lo podemos llamar ciclo dado que todos los sistemas de cómputo son sustituidos periodicamente, repitiendose su proceso de desarrollo en forma cíclica.

Las fases que conforman el ciclo de desarrollo de sistemas dividen un proyecto en subproyectos cada uno de los cuales da la oportunidad de revisar (controlar) el cumplimiento de los requerimientos estipulados (planeados).

Cuales fases son establecidas no es tan importante como e hecho de que sean fijas y bien definadas. Para mejorar los resultados todos los proyectos y áreas funcionales deberían adoptarlas.

Algunos de los beneficios de estandarizar las fases de los proyectos de desarrollo de sistemas son:

- Control.- Permite la revisión en varios puntos, en los que -- se mide la calidad y se ajustan los planes.

- Comunicación.- Estandarizar la terminología permite que los - integrantes del proyecto, los usuarios y los directivos hablen el mismo lenguaje.

- Participación del usuario.- Al respaldar la metodología aumenta el compromiso de participación.

- Documentación.- Cada fase requiere una salida que debe producirse antes de terminar.

- Calidad.- La precisión en las salidas de cada fase aseguran - que se termine el producto y su documentación.

- Estimaciones.- Permite calendarizar cada véz con mejores bases.

Las fases son a su véz divididas en tareas estandarizadas o actividades genéricas. Estas se presentan como actividades - específicas para cada parte de un sistema en un proyecto particular.

HISTORIA DE LA ADMINISTRACION.

PERIODO	ETAPA	CARACTERISTICAS
HASTA 1900	EMPIRICA	DECIAN QUE HACER PERO NO COMO HACERLO.
1870 - 1930	CIENTIFICA	CONCENTRACION EN EL METODO Y LOS ESTANDARES
1930 - 1940	RELACIONES HUMANAS	VALORARON EL GRUPO Y LOS FACTORES SICOLOGICOS DEL TRABAJO.
1900 - 1940	CLASICA	ELEMENTOS (PLANEAR, ORGANIZAR, MANDAR, COORDINAR Y CONTROLAR) PRINCIPIOS DE LA ADMINISTRACION.
1940 A LA FECHA	MODERNA	MOTIVACION, LIDERAZGO, DINAMICA DE GRUPOS Y ORGANIZACIONAL TOMA DE DECISIONES CUANTITATIVAS.

fig. I.1

ELEMENTOS QUE ESTRUCTURAN A LAS FUNCIONES DE PROCESO ADMINISTRATIVO

PLANEACION	ORGANIZACION	DIRECCION	CONTROL
- PRONOSTICOS	- ORGANOGRAMA	- COMUNICACION	- REPORTES
- OBJETIVOS	- DESCRIPCION ORGANICA FUNCIONAL	- DELEGACION	- NORMAS, REGLAS, MODELOS
- POLITICAS	- DESCRIPCION DE PUESTOS	- MOTIVACION	- EVALUACION
- PROCEDIMIENTOS		- LIDERAZGO	- ACCIONES CORRECTIVAS
- PROGRAMAS			
- PRESUPUESTOS			

fig. I.2

PROCESO ADMINISTRATIVO

FUNCIONES	PREGUNTAS	RESULTADO
PLANEACION	QUE ES LO QUE SE NECESITA QUE CURSOS DE ACCION DEBEN ADOPTARSE, COMO Y CUANDO - DEBEN SEGUIRSE.	OBJETIVOS, POLITICAS, PROCEDIMIENTOS Y METODOS.
ORGANIZACION	CUANDO DEBEN TENER LUGAR LAS ACCIONES Y QUIEN DEBE HACER ESE TRABAJO.	DIVISION DEL TRABAJO, DISTRIBUCION DEL TRABAJO.
DIRECCION	POR QUE Y COMO EJECUTAN SUS TAREAS LOS MIEMBROS DEL GRUPO.	COMUNICACION, DELEGACION Y SUPERVISION.
CONTROL	ESTAN SIENDO EJECUTADAS LAS ACCIONES- CUANDO, DONDE Y COMO DE ACUERDO CON LOS PLANES.	INFORMES, COMPARACIONES Y COSTOS.

fig. I.3

PROCESO ADMINISTRATIVO

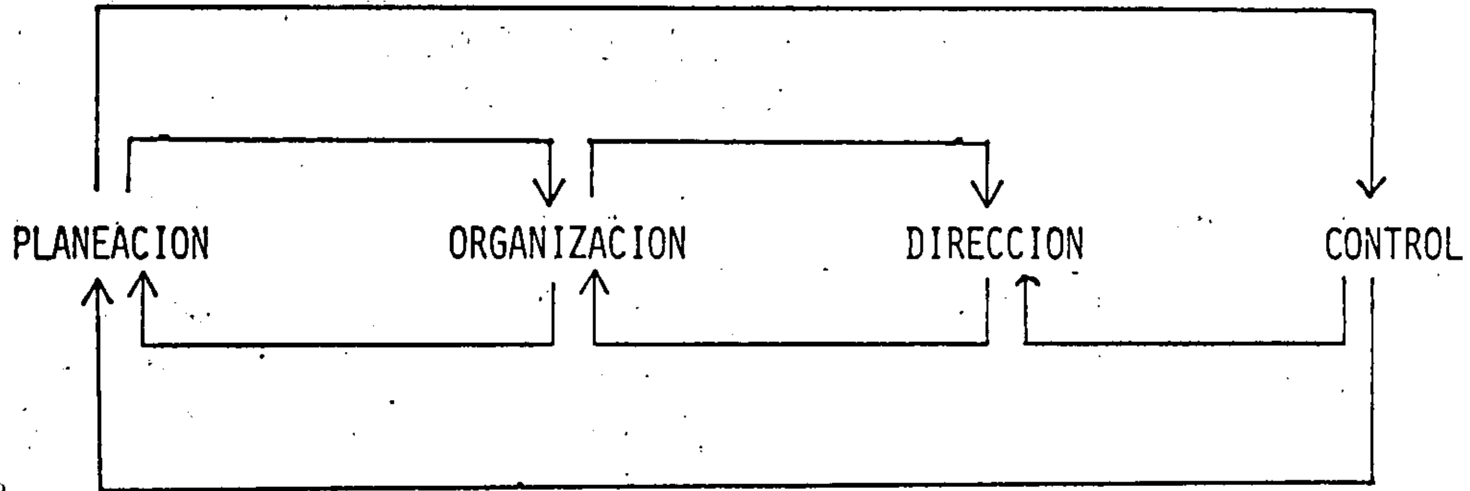
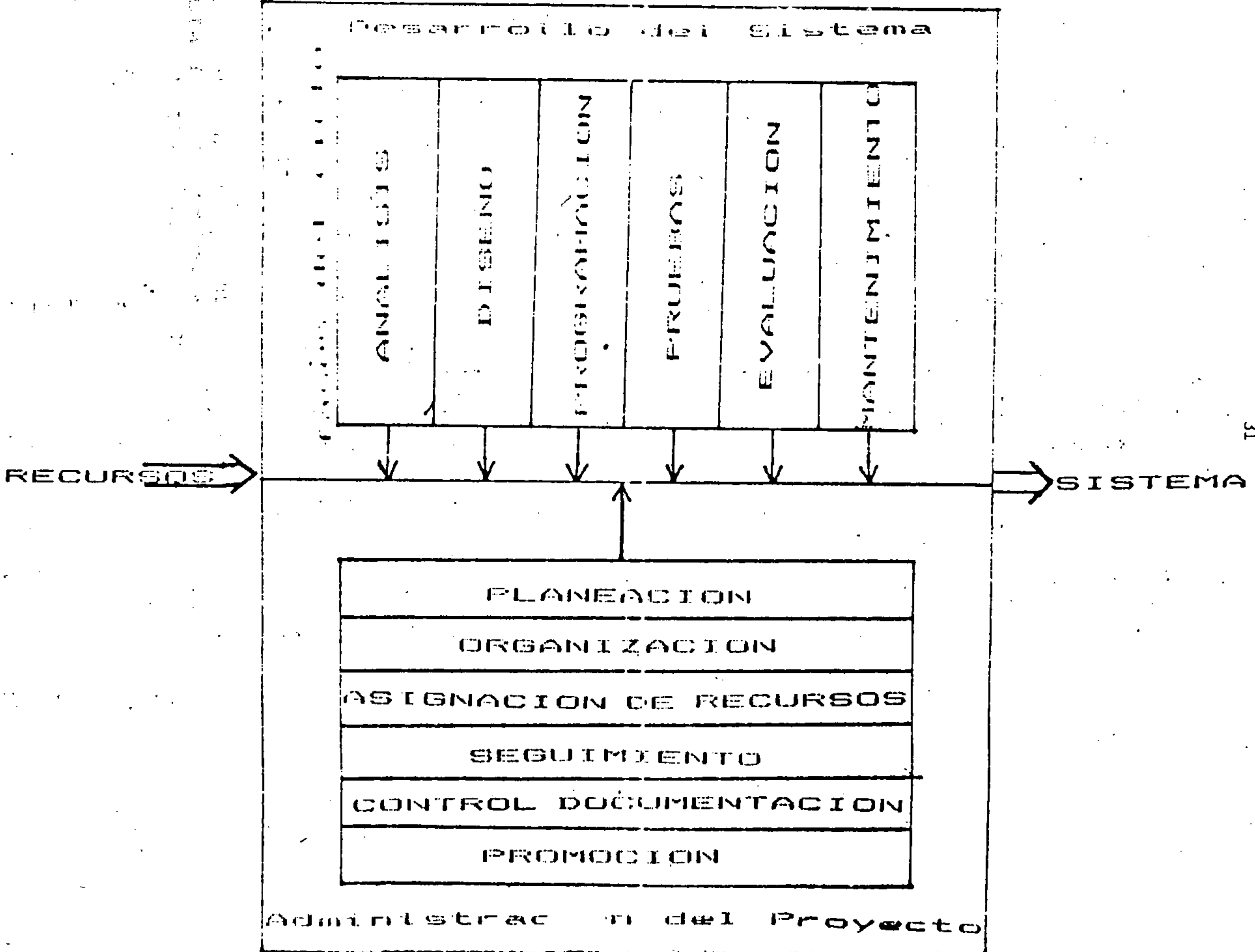


fig.

AMBITO DE LAS FUNCIONES ADMINISTRATIVAS

NIVEL ADMINISTRATIVO	PLANEACIÓN	ORGANIZA-CIÓN	DIRECCIÓN	CONTROL	LA DIRECCIÓN SE CARACTERIZA POR:
SUPREMO					AMPLIA Y CREATIVA
MEDIO SUPERIOR					REGULARMENTE AMPLIA Y CREATIVA
MEDIO INFERIOR					LIMITADA Y ALGO RUTINARIA
INFIMO					DETALLES Y RUTINA

fig. I.5

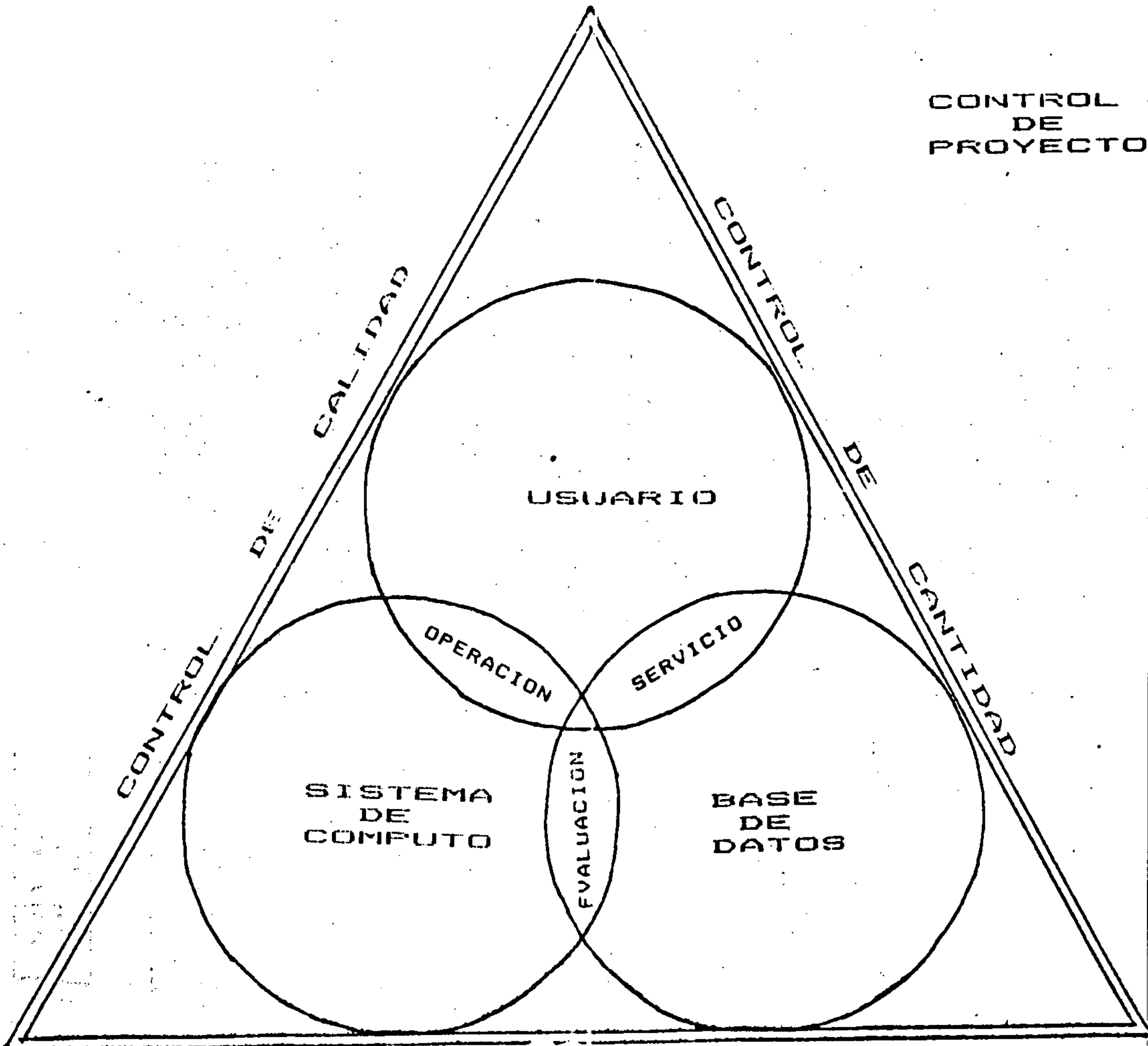

DESARROLLO DE PROYECTOS
Desarrollo del Sistema
ANALISIS
DISENO
PROGRAMACION
PRUEBAS
EVALUACION
MANTENIMIENTO
RECURSOS
SISTEMA
PLANEACION
ORGANIZACION
ASIGNACION DE RECURSOS
SEGUIMIENTO
CONTROL DOCUMENTACION
PROMOCION
Administrac n del Proyecto

FUNCIONES Y PRODUCTOS DE VIDA DE UN SISTEMA

PASOS	FUNCION	PRODUCTOS
INICIACION	ARRANQUE DEL PROYECTO	-REQUERIMIENTOS DEL -PRESUPUESTO Y EST. COSTOS -ESTUDIO DE FACTIBILIDAD -PLAN DEL PROYECTO
ANALISIS	ANALIZAR REQUERIMIENTOS	-DESCRIPCION FUNCIONAL -REQ. DE INFORMACION
DISENO	DISENO DEL NUEVO SISTEMA	-ESPECIF. SISTEMA/SUBSISTEMA -ESPECIF. BASE DE DATOS -ESPECIF. DE PROGRAMAS
DESARROLLO	DES. SOFTWARE	-DOCUMENT. DE PROGRAMAS
[illegible]	INSTALACION DEL NUEVO SISTEMA	-PLAN DE PRUEBAS. -REPORTE DE ANAL. DE PRUEBAS -MANUAL DEL USUARIO
OPERACION	OPERACION DEL SISTEMA	-MANUAL DE OPERACION -MANUAL DE MANTENIMIENTO -REPORTE EVALUACION

CARPETAS PARA DOCUMENTACION DE PROYECTOS

CARPETA	CONTENIDO
DEL PROYECTO	-CORRESPONDENCIA -REQ. DEL USUARIO -PRESUPUESTO Y EST. COSTOS -PLAN DEL PROYECTO
DEL SISTEMA	-EST. DE FACTIBILIDAD -DESCRIP. FUNCIONAL -DOCTO. REQ. INFORMACION -REPORTE DISENO SISTEMA: *ESP. SISTEMA/SUBSISTEMA *ESP. BASE DE DATOS *ESP. PROGRAMAS *PLAN DE PRUEBAS *MANUAL DEL USUARIO -REPORTE DE EVALUACION
DEL PROGRAMA	-DOCUMENTACION DEL PROGRAMA -REPORTE ANALISIS PRUEBAS
DE PRODUCCION	-MANUAL DE OPERACION -MANUAL DE MANTENIMIENTO
DE USUARIO	-MANUAL DEL USUARIO


CONTROL
DE
PROYECTO
CONTROL DE CALIDAD
CONTROL DE CANTIDAD
USUARIO
OPERACION
SERVICIO
EVALUACION
SISTEMA
DE
COMPUTO
BASE
DE
DATOS
A D M I N I S T R A C I O N

Software

Programas de computadora y su documentación asociada, requerida para su desarrollo operación, y mantenimiento.

Ingeniería de Software

La aplicación practica del conocimiento científico en el diseño y construcción de programas de computadora y la documentación asociada requerida para desarrollarlos, operarlos y mantenerlos.

Es la aplicación de la ciencia y matemáticas por medio de la cual la capacidad del equipo de cómputo se hacen útiles al hombre por medio de programas de computadora, procedimientos y documentación asociada.

Que es el Software?

El Software es información:

1). Estructurada con propiedades lógicas y funcionales.

2). Creada y mantenida en varias formas y representaciones durante su ciclo de vida.

3). Fabricado para una maquina en su estado de desarrollo completo.

Existe en 2 formas básicas:

No ejecutable

Documentación

procesable en máquina

Ejecutable

La Ingeniería de Software determina el costo y la calidad del Software producido.

COSTO

El Software es caro y su costo tiende a ser mayor.

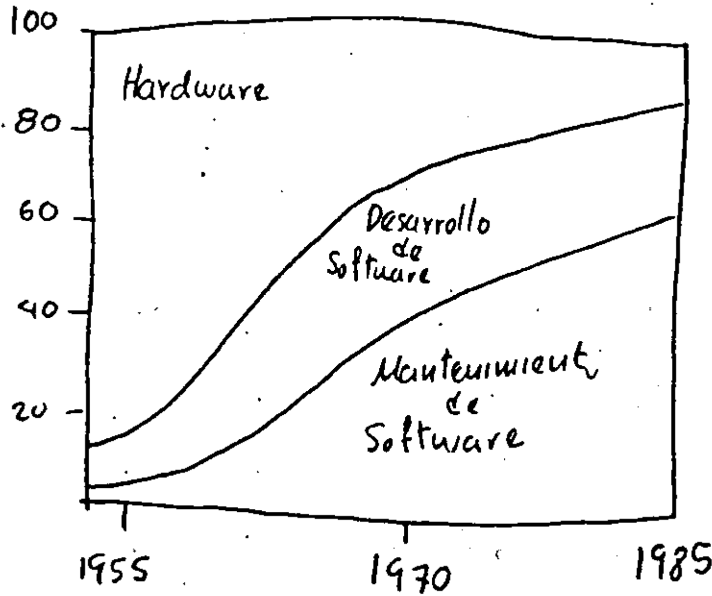
En 1980 en U.S.A. el gasto en Software fué de -- 40,000 millones de dólares 2% del PIB.

Para 1990 podría llegar a ser el 13% del PIB.

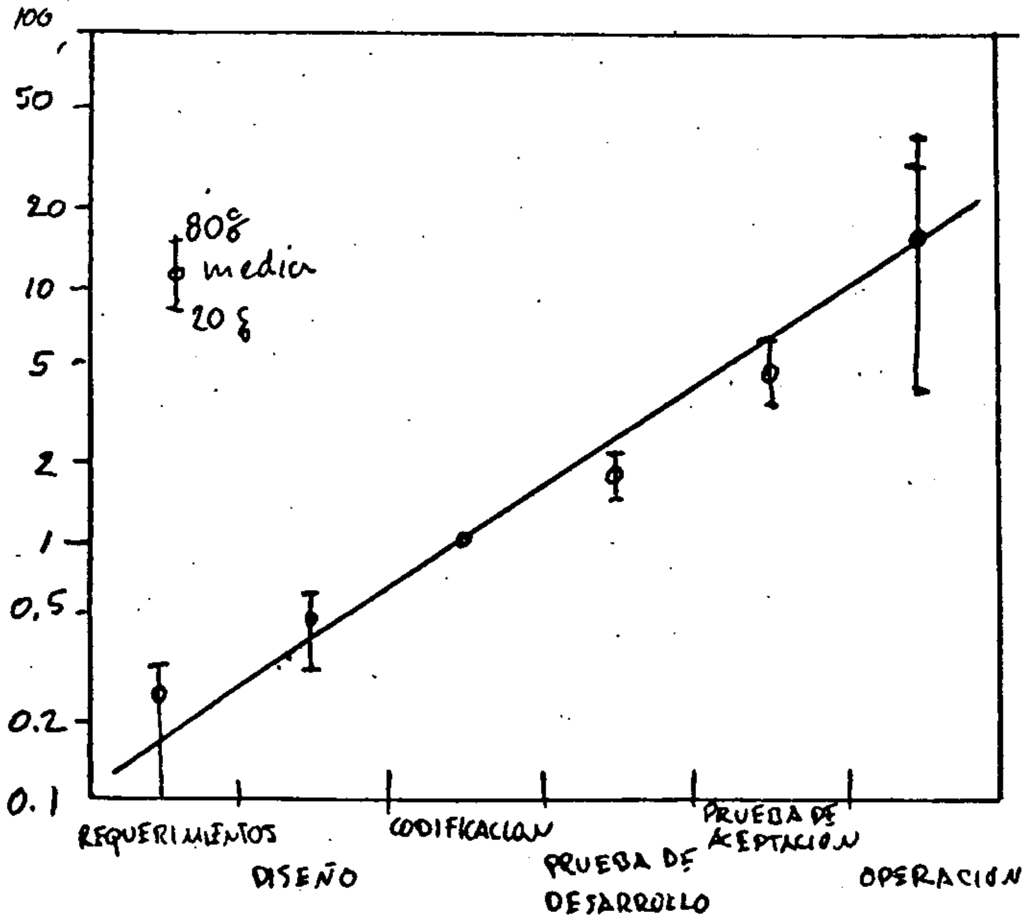
El reto aquí es de dos tipos:

a) Incrementar significativamente la productividad del desarrollo de Software.

b) **Incrementar la eficiencia del mantenimiento del - Software.**



TENDENCIAS DEL COSTO HARDWARE-SOFTWARE



FASE EN LA CUAL SE DETECTA EL ERROR

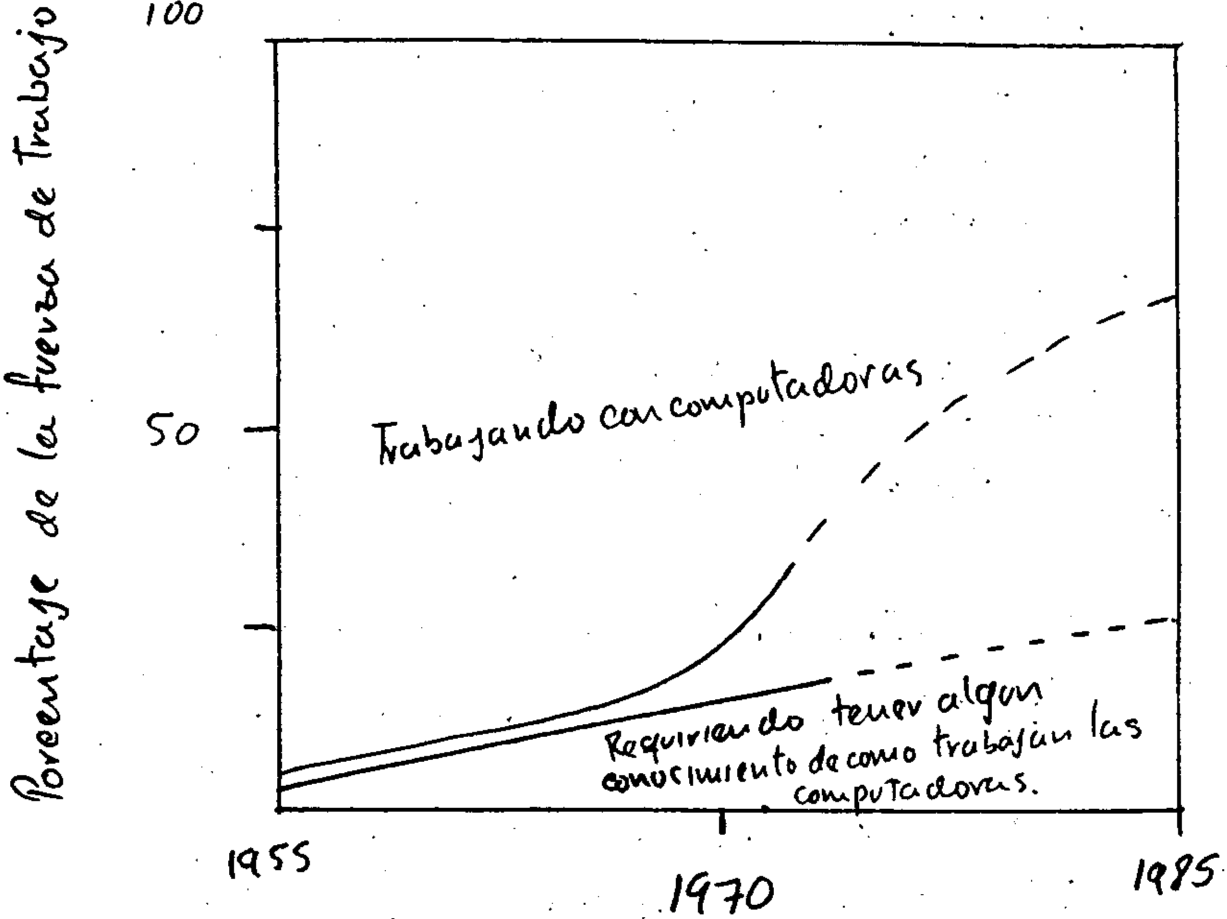
IMPACTO SOCIAL

El papel que juega el Software dentro de la sociedad cada vez es más importante.

El crecimiento de la demanda de Software tiene su origen en el hecho que conforme el Hardware se hace más económico, confiable y poderoso, se encuentran mayores ventajas para automatizar las partes mecánicas de las tareas de los humanos.

El incremento en el impacto en el bienestar humano requiere que se desarrolle y mantenga Software que sea:

- Extremadamente Confiable
- Humano
- Fácil de usar
- Difícil de usar mal
- Auditable



Crecimiento de la confianza en las computadoras y el Software.

II. METODOLOGIAS DE ANALISIS Y DESARROLLO DE SISTEMAS

Existen tres orientaciones dentro de las cuales se pueden clasificar las metodologías que han surgido para la solución de problemas de diseño de sistemas de información, siendo las más comunes: [2,5]

- Orientadas a procedimientos
  - Descomposición funcional
  - Diseño de Flujo de Datos
  - Diseño estructurado
- Orientadas a Datos
  - Diseño de Estructura de Datos
  - Relación de Entidades
  - Desarrollo de Sistemas para Datos Estructurados (DSSD)
- Orientadas a Información
  - Ingeniería de la Información (James Martin)

Las metodologías en las dos primeras orientaciones han evolucionado a partir de la década de los 70's cuando surgió la preocupación sobre alternativas más eficientes para el desarrollo de software.

Ambas orientaciones tienen exponentes que defienden sus particulares puntos de vista. A continuación se describen algunas de ellas:

Una de las técnicas más antigua y simple con orientación a procedimientos es la de descomposición funcional. Esta técnica se basa en el principio de "divide y vencerás", fue popularizada por gentes como: Dijkstra y Wirth.

El diseño bajo este enfoque intenta llevar el desarrollo del sistema desde la visión general hasta las unidades mínimas del problema mediante refinamientos sucesivos del problema (Top-down approach).

El análisis y diseño pueden empezar en las más altas esferas funcionales de la organización y avanzar a niveles de mayor detalle hasta que son representados módulos de programas, continuando hasta mirar en detalle los componentes de un programa. En la mayoría de los casos, la descomposición de los niveles jerárquicos altos no es realizada y se inicia el proceso a partir de niveles intermedios o bajos, perdiendo la visión general de la problemática de la organización.

La estrategia de diseño en este tipo de métodos se divide en los siguientes pasos:

(1) Definición de la función deseada.

(2) División, conexión y verificación de cada una de las subfunciones definidas.

(3) Subdivisión, conexión y verificación hasta llegar a módulos que puedan ser enteramente controlables.

Existen varios problemas en la aplicación de esta técnica, primero, se dice que se debe descomponer el problema en módulos pero no especifica con respecto a que deberán

ser descompuestos, la descomposición puede ser respecto al tiempo (orden en que se dan los procesos), respecto al flujo de los datos, respecto a los grupos lógicos, acceso a recursos comunes, o cualquier otro criterio. Si se descompone respecto al tiempo, se obtienen generalmente módulos tales como Inicialización, Proceso, Terminación, con lo cual sólo obtenemos una cohesión temporal[12].

La mayor ventaja de este método es su aplicabilidad general. Es la técnica que ha sido utilizada por más gente durante más tiempo y en la cual se apoyan todas las otras técnicas que han surgido posteriormente.

Diseño de Flujo de Datos

Un diagrama de flujo de datos muestra principalmente procesos (obviamente su orientación es a procesos) y el flujo entre ellos. Al nivel más alto es utilizado para mostrar los procesos de la organización y las transacciones resultado de los mismos, ya sean manuales o automatizadas. A niveles inferiores es utilizado para mostrar programas o módulos de programas y el flujo entre esos módulos.

Un diagrama de este tipo es utilizado como primer paso de diseño estructurado. Mostrando una panorámica del flujo de la información a través del sistema o programa. Su utilidad principal es como herramienta de análisis que permite dibujar los componentes(procesos) básicos y los datos que fluyen entre ellos.

Este método fue propuesto originalmente por Constantine y extendido por Yourdon y Myers, sin embargo, la versión de DeMarco es la más popular. En su forma más sencilla

---

12 *Es aquello que mantiene unido a un módulo. Asociación entre los componentes elementales de un módulo.*

no es mas que una descomposición funcional con respecto al flujo de datos. Cada bloque de estructura se obtiene por aplicación sucesiva de la definición de caja negra que transforma un conjunto de datos de entrada en otro conjunto de datos de salida.

El método puede ser dividido en las siguientes etapas:

(1) Modelar el problema como flujo de datos.

(2) Identificar los elementos de transformación de datos.

(3) Definir la estructura jerárquica de los procesos.

(4) Refinamiento y optimización.

Este tipo de herramientas es muy valiosa para representar el flujo de documentos y los datos de computadora en sistemas complejos. Algunos extienden su uso a la estructuración interna de los programas, siendo éste uso muy cuestionable ya que existen técnicas mucho mejores para representar programas.

Diseño Estructurado

El método de Diseño Estructurado de Yourdon es el más ampliamente conocido hoy en día. El producto principal es la carta estructurada, que define globalmente la arquitectura de control de un programa mediante la visión de los procedimientos y sus interrelaciones.

Las cartas estructuradas son construídas a partir de diagramas de flujo que representan procedimientos conectados jerárquicamente y junto a éstos la definición de los mecanismos de paso de datos llamados "acopladores".

Yourdon, Constantine y otros definen el diseño estructurado como formado por 4 pasos \3:

(1) Dibujo de diagramas de flujo de datos del sistema que representen el problema.

(2) Esta etapa está formada por dos estrategias -análisis de transformación y análisis de transacción- que guían la traducción de diagramas de flujo de datos a cartas de estructura.

(3) Medición del grado de cohesión y acomplamiento \4.

(4) Empaquetamiento, prepara el diseño para implantación. Aquí se divide el diseño lógico en unidades fisicas.

Diseño de Estructura de Datos

Con ciertas diferencias en forma este método fue desarrollado por Jackson en Inglaterra y Warnier en Francia.

\3 *Aunque existen otras variaciones y actualizaciones hechas por gentes como Ward & Mellor y Hatley.*

\4 *Acoplamiento. Grado de independencia entre módulos.*
Cohesion. Relación de elementos dentro de un módulo

El método de Jackson es similar al de Yourdon, pero orientado a datos, luego entonces la estructura de los datos se define primero.

Jackson ve un programa como secuencias de registros con entradas y salidas y que consta de cuatro etapas: datos, programas, operaciones, texto.

Este método es más completo que el de Yourdon; Jackson considera como estructuras tanto a los datos, como a los procesos; sin embargo, el método tiene 2 defectos principales: carecer de estructuras lógicas de control en iteraciones y condiciones, y se puede verificar fácilmente que el funcionamiento del sistema sea correcto.

La premisa básica consiste en que el programa ve el mundo mediante estructuras de datos y un adecuado modelo con ese enfoque de datos, puede ser fácilmente transformado en un programa o programas que reflejan correctamente el mundo real. La relación de los diferentes niveles que se forman en la definición del problema son relaciones del tipo "está compuesto de". Por ejemplo, un programa de reporte está compuesto por un encabezado, seguido del detalle y al final de los totales. Esta es una relación estática que no cambia durante toda la ejecución del reporte siendo una buena base de construcción del programa. El método satisface el principio de consistencia [15] en un alto grado.

La estrategia de diseño de este método se puede dividir en los siguientes pasos:

(1) Armar un diagrama jerárquico que represente el entorno del problema.
(2) Definir y verificar las estructuras de los flujos de datos.
(3) Derivar y verificar la estructura del programa.

---

15 *Esta contenido en una sólida estructura.*

(4) Derivar y definir las operaciones básicas

(5) Escribir la estructura y el texto del programa

Normalmente estas etapas son realizadas y verificadas independientemente. Para grandes problemas el objetivo es representar programas simples en un diagrama jerárquico. La premisa de Jackson es "si es difícil hacer un programa sencillo, resulta imposible hacer uno complejo", entonces el truco está en partir los problemas complejos en varios problemas sencillos.

El principal problema con la metodología de Jackson se refiere a su forma de abordar el proyecto, pues empieza de abajo hacia arriba (bottom-up) lo que da buenos resultados para problemas sencillos, pero todavía no se desarrolla una alternativa para cuando los problemas se vuelven demasiado complejos.

Relación de Entidades

El principal exponente de este principio es Chen y su fundamento radica en la definición de datos en un modelo conceptual, identificando los tipos de entidades involucradas en la operación de una organización y determinando las relaciones entre estos tipos de entidades. Un diagrama de relación de entidades es desarrollado de tal manera que pueda ser descompuesto en modelos de datos mas detallados.

La premisa de este método radica en las siguientes afirmaciones:

Para que una organización funcione adecuadamente se deben poseer ciertos datos. Estos datos son necesarios independientemente de que se usen computadoras o no, aunque las computadoras proporcionan un poderío para que los datos adecuados

llegen a las personas correctas. Los datos en cuestión deben ser planeados y descritos. Se necesitan datos acerca de los datos, estos últimos son conocidos como "metadatos". Un modelo de datos contiene metadatos.

En esta alternativa se requiere una gran ayuda de los usuarios finales y de los ejecutivos para entender los datos, ya que se requieren formas claras y concisas de manejar y determinar los datos que son necesarios para la organización.
Para definir estas afirmaciones son muy útiles y precisas los diagramas de Relación de Entidades y para el detalle las cartas de "L invertida" que contienen los atributos de las entidades y las propiedades de los atributos.

Desarrollo de Sistemas con Datos Estructurados(DSSD)

Método basado en la teoría de conjuntos. En la programación normal, un conjunto (grupo ordenado de objetos que comparten características comunes) es denotado por una lista de miembros encerrados en paréntesis o llaves. En forma matemática, los conjuntos DSSD se denotan como una lista vertical de miembros agrupados por una llave a la izquierda.

Sus principales proponentes son Warnier y Orr. El método es parecido a las cartas estructuradas y su principal enfoque se encuentra en la ayuda al diseño de programas bien-estructurados. Este método tiene ciertas ventajas sobre otros modelos estructurados y es fácil de aprender y usar, ya que solo utiliza cuatro técnicas básicas de diagramación.

Un diagrama de Warnier-Orr puede representar gráficamente la estructura jerárquica de un programa, de un sistema o de una estructura de datos, además puede mostrar la estructura de control de esa estructura.

DSSD es un método orientado a datos (como el de Jackson) y arranca con un plan que es desarrollado mediante la liga de las metas de la organización con los objetivos del sistema.

El siguiente paso consiste en involucrar al usuario definiendo los requerimientos de la organización, de cómputo y del usuario.

Un diagrama de relación de entidades determina el alcance del sistema, un diagrama de Warnier-Orr y un diccionario de datos, muestra las salidas o resultados necesarios para soportar los procesos.

Ingeniería de la Información

La alternativa propuesta por Martin intenta conciliar y retomar todas las ventajas y beneficios de las metodologías existentes hasta el momento, moldeándolas dentro de un ambiente uniforme de conceptos y técnicas. Martin ve la problemática de los sistemas de información desde un punto de vista mas general, no solo enfocando el proceso de datos, sino la empresa u organización como un ente con problemas de información el cual hay que abordar de manera integral. Desde luego éste método parte de los diagramas de relación de entidades, pero también se apoya en los diagramas de descomposición funcional y en los de flujo de datos.

Los cuatro pasos básicos de la técnica de Martin contienen lo siguiente:

(1) La creación de un plan estratégico de la organización, donde se definen las metas y objetivos a largo plazo. La definición de un modelo de la organización que contenga las funciones básicas de la misma y sus correspondientes necesidades de información.

(2) La extracción y desarrollo a partir de un modelo de datos totalmente normalizado, de la solución de un área particular de la empresa, llegando a definir los atributos de cada entidad relacionada con el detalle de funciones del área, subdividiéndolas en mecanismos y procesos, utilizando herramientas como los diagramas de descomposición.

(3) La distribución a detalle de los procedimientos necesarios para llevar a cabo los procesos obtenidos en la etapa anterior, así como las estructuras lógicas de datos necesarios para llevarlos a cabo. Aquí suelen ser útiles los diagramas de estructura de datos, los diagramas de descomposición y otras herramientas como los formatos de reporte o de captura y los diagramas de acción.

(4) Finalmente, se completa el diseño físico y se desarrollan los programas. Esta aproximación a la solución normalmente se da mediante la utilización de prototipos en los que se pretende una participación muy activa del usuario final.

Existe todavía mucho por hacer en cuanto a métodos y metodologías para el diseño de sistemas. La descomposición Funcional puede ser ideal para las personas que conocen

la respuesta. Las otras parece ser que necesitan menor información sobre la respuesta final.

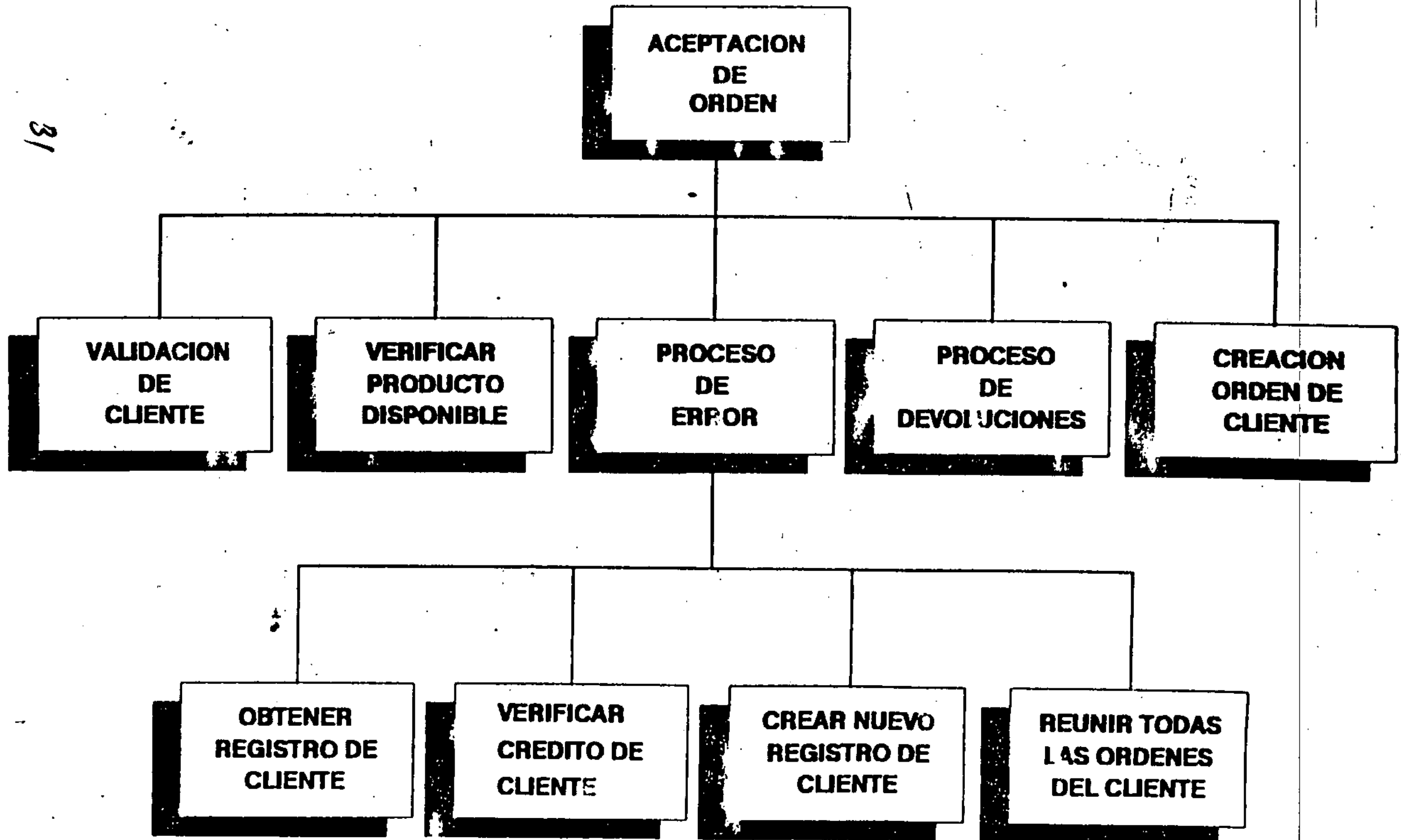
Aunque todas las metodologías tienen en alguna parte algo de mágicas, claramente se puede deducir que en la Descomposición Funcional la magia aparece muy cerca del producto final. En los otros métodos, la magia parece estar presente a nivel de definición del problema. Desde luego es necesario tener un mejor manejo y dominio de esa parte mágica medio desconocida que siempre aparece en todas las metodologías.

Algunas de las características deseables para los esquemas del futuro serían:

- La existencia de un procedimiento racional que permitiera dividir y modelar los problemas.
- Un esquema homogéneo que pueda ser compartido por diferentes personas inmersas en un mismo proyecto.
- Una interacción eficiente con el mundo real para problemas grandes .
- Un esquema que subdivida eficientemente tanto el proceso de diseño como la solución del problema.

DIAGRAMA DE DESCOMPOSICION

(DIJHSTRA - WIRTH)



31

DIAGRAMA DE FLUJO DE DATOS
(CONSTANTINE - YOURDON - MYERS)

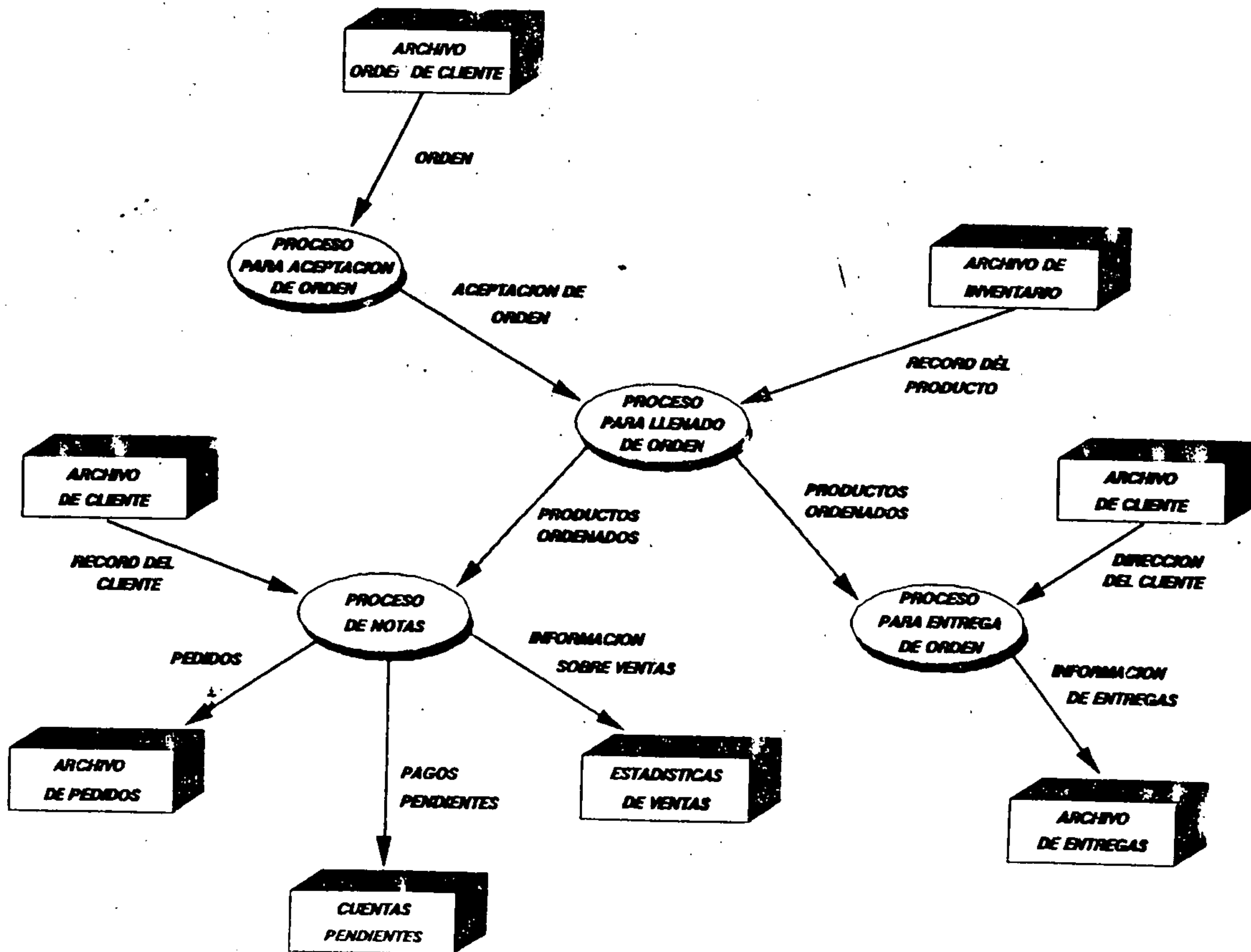
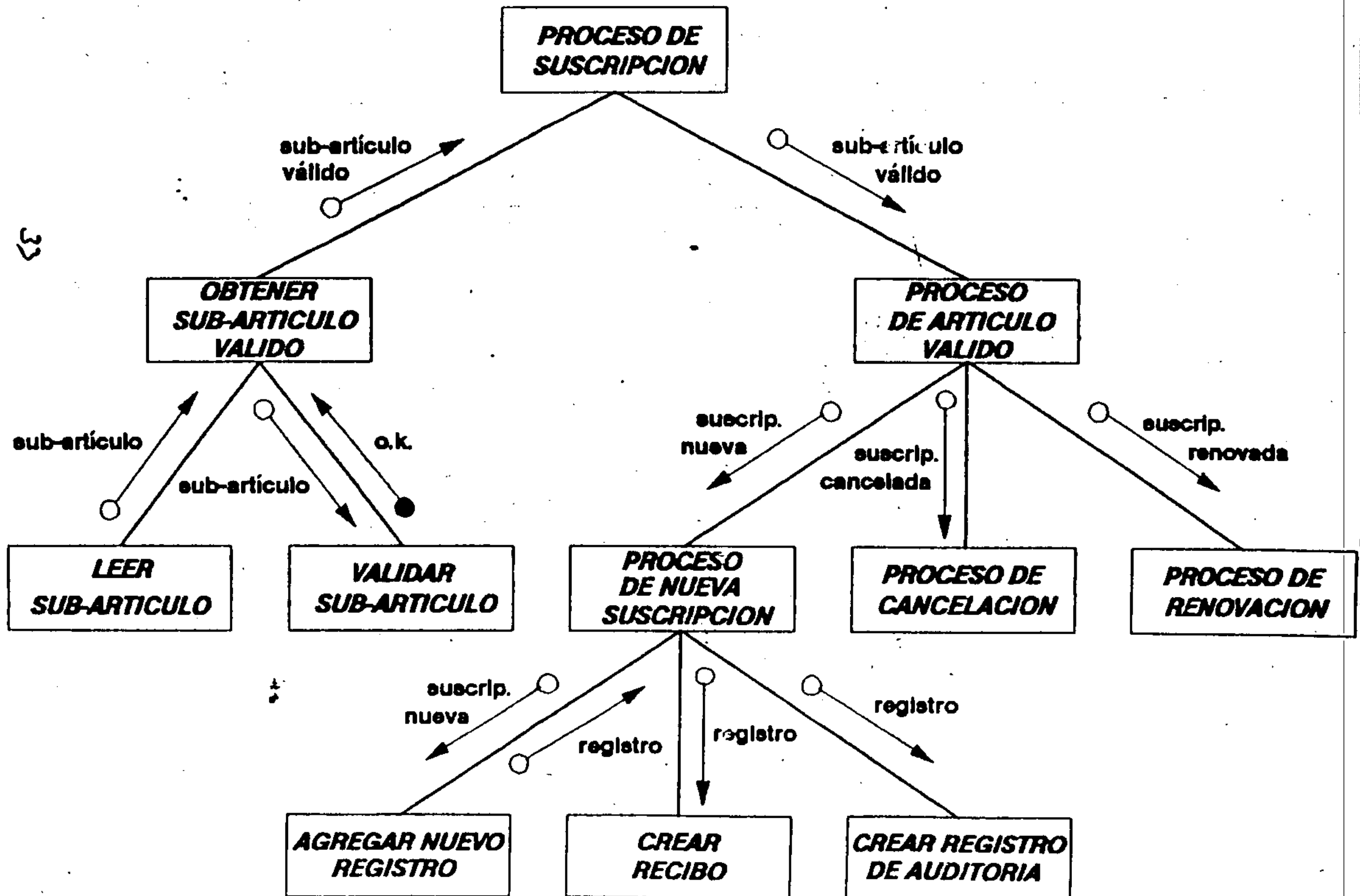


DIAGRAMA DE CARTA ESTRUCTURADA

(YOURDON - CONSTANTINE)



NIVEL

1

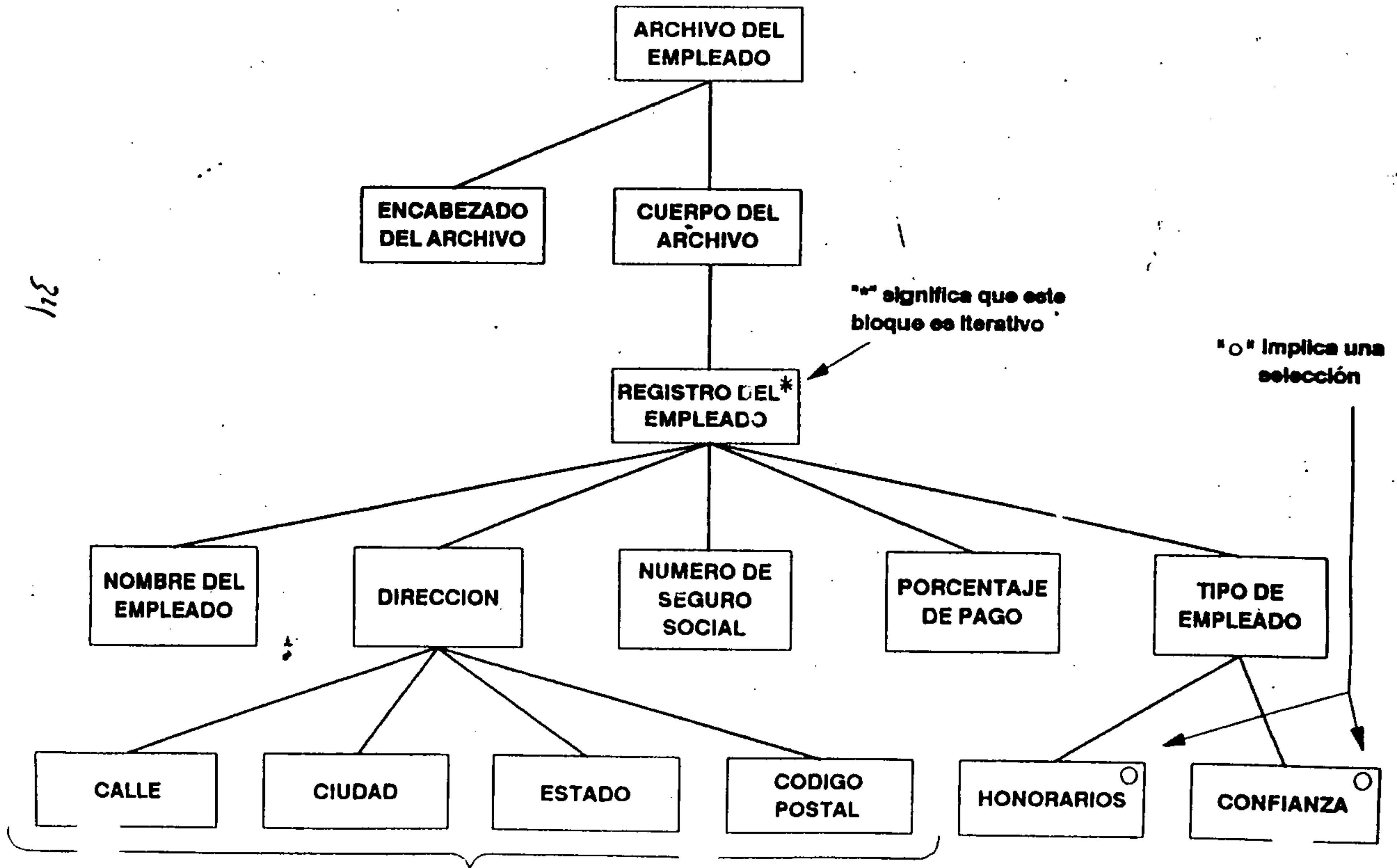
2

3

4

ESTRUCTURA JERARQUICA DE ARBOL

(MICHAEL JACKSON)



RELACION DE ENTIDADES
(PETER CHEN)

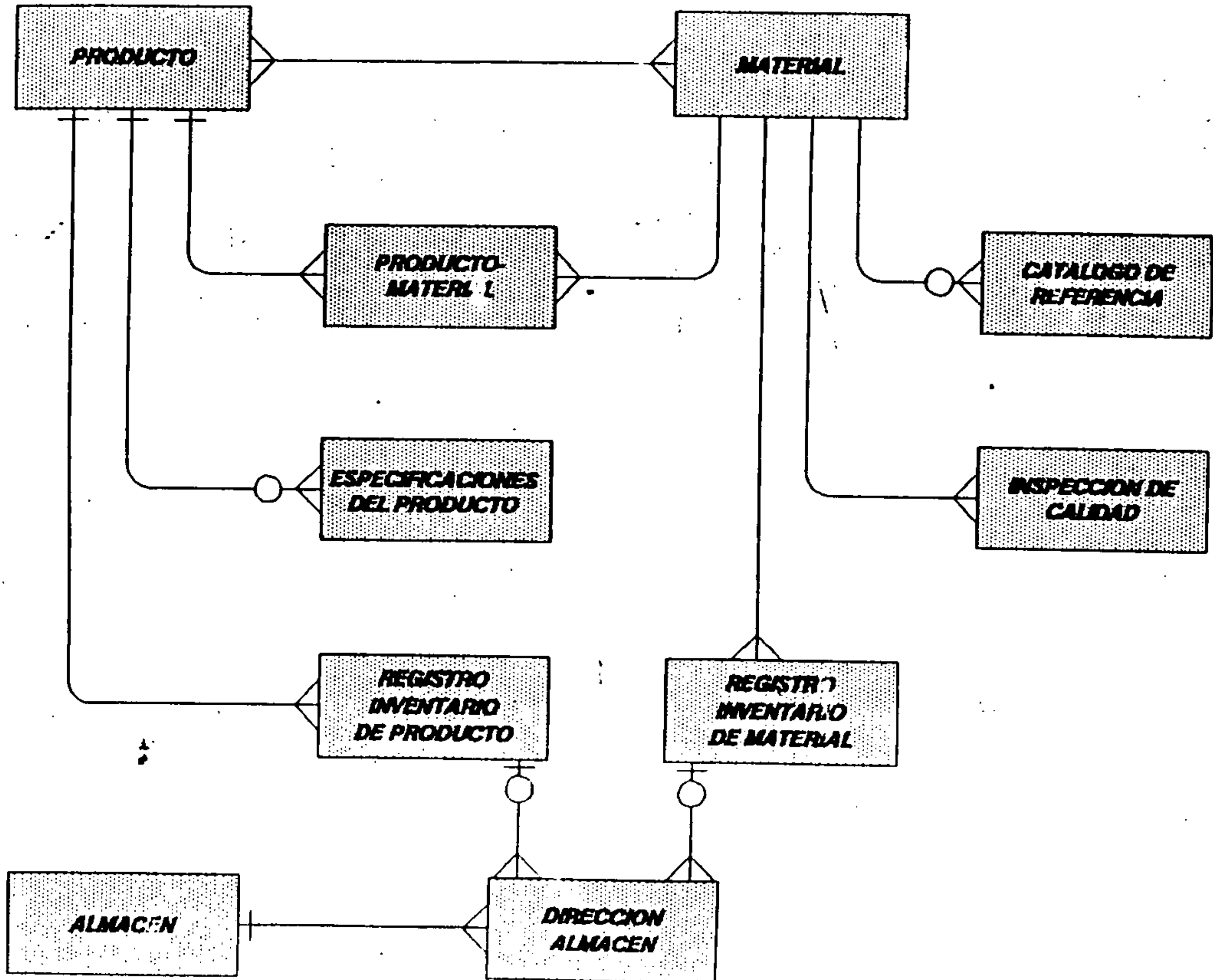
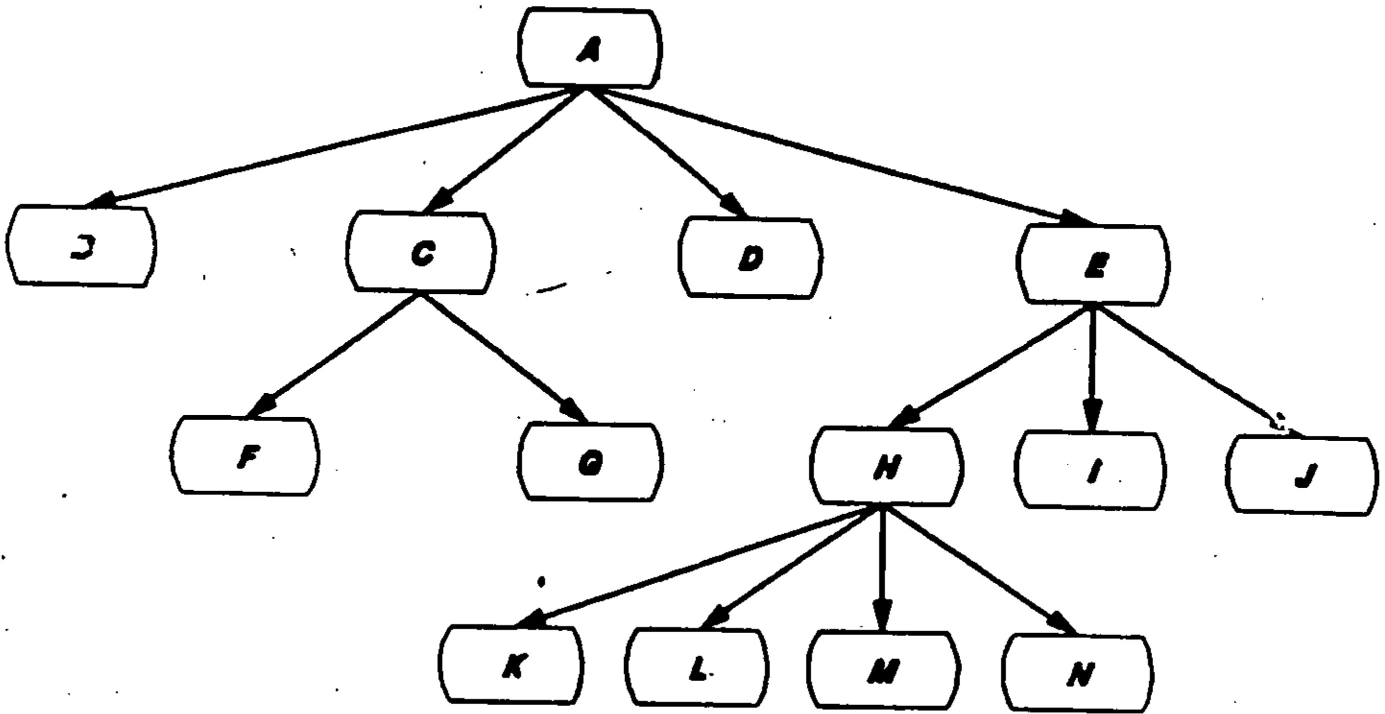


DIAGRAMA DE ESTRUCTURA JERARQUICA
(WARNIER-ORR)



ESQUEMA WARNIER-ORR

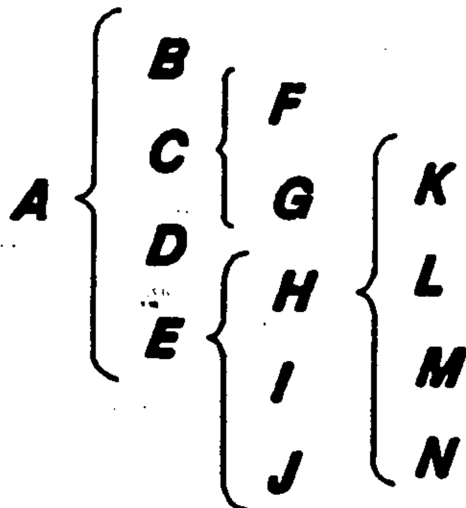


Figure 2.6

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS INSTITUCIONALES

No. 70

METODOLOGIA PARA EL DESARROLLO DE LOS SISTEMAS DE INFORMACION

DEL 9 AL 29 DE SEPTIEMBRE

SEPOMEX

¿CUANDO SE NECESITA CREAR UN PROTOTIPO?

MEXICO, D.F.
PALACIO DE MINERIA

1992

CUANDO SE NECESITA CREAR UN PROTOTIPO?

- CUANDO NO SE CUENTA CON TODA LA INFORMACION O EL SISTEMA ES MUY GRANDE.

- CUANDO SE ESTE INSEGURO DE SUS NECESIDADES.

- CUANDO SU USO SEA INCIERTO O MUY VARIABLE.

2

PARA QUE SE USA UN PROTOTIPO?

- PARA CLARIFICAR LOS REQUERIMIENTO DEL USUARIO
- PARA HACER MAS EFECTIVO EL DISEÑO
- PARA CREAR UN SISTEMA FINAL
- ENSEÑANZA/APRENDIZAJE

PROTOTIPOS

- IDENTIFICAR LOS REQUERIMIENTOS BASICOS DEL USUARIO POR MEDIO DE UN ANALISIS RAPIDO E INCOMPLETO

- DESARROLLAR LA BASE DE DATOS (SI ES RELACIONAL, HACERLO SIN MIEDO YA QUE LOS CAMBIOS SON FACILES DE HACER)

- PLANEAR Y DESARROLLAR MENUS Y SUBMENUS

- COMPLETAR EL PROTOTIPO DESARROLLANDO FUNCION POR FUNCION

- DESPUES DE LA APROBACION DEL USUARIO, ESCRIBIR LAS ESPECIFICACIONES (DOCUMENTACION)

- IMPLANTAR Y USAR EL PROTOTIPO

- REVISAR Y MEJORAR EL PROTOTIPO

Prototyping: Not a Method but a Philosophy

BY DR. PETER G. KAUBER

■ In recent years, a system development methodology called prototyping has emerged as an alternative to the traditional development methodology. While the traditional approach amounts to a serial, phased process in which each major step or phase is completed prior to the start of the next, the prototyping approach tends to merge analysis, design, and implementation into a fluid process that emphasizes extrememly fast initial development, followed by an on-going, evolutionalry elaboration with continuous use, feedback, and modification. Prototyping quickly puts a product into the hands of the user, and then permits that user to direct the alteration and enhancement of the product on a continuous basis and with quick cycling of the desire-satisfaction loop.[1]

Prototyping is often advocated in cases where initial specifications are hard to determine, where the user needs to work with the product before understanding fully what he or she might want from it. Thus, prototyping is often treated in the context of decision-support systems, and is equally often ignored in the context of traditional data processing, where it is assumed that clear specifications can be generated early and, thus, where the serial, phased approach of the traditional development methodology is deemed appropriate.[2] I claim that this misses the real point that prototyping can address; the remainder of this article is an elaboration of this theme: prototyping should be viewed in the context of a general organizational philosophy and not treated as merely one methodology among many. Such a perspective will require that the systems analyst finally be what he/she has almost never been: a knower, appreciator, analyzer, and designer of true *systems*.

Organizations, Environments, and Decisions

If we are serious about our claims that information systems are not ends in themselves but rather are to serve users, then it becomes important to remain aware of the "nest" of systems within which the information system participates. If the organization is treated as the "system of interest" (the "focal" system), then an information system is a subsystem of the focal system. Since the focal system is itself an *open* system, its environment is of crucial importance to it.[3] When environments are stable,

DR. PETER G. KAUBER

Dr. Kauber is a Professor of computer science at North Carolina State University, Raleigh, NC. He has had industry experience as a systems analyst and database analyst and has served as a consultant in database design. Dr. Kauber's interests include logical database design, information systems theory, and the design and development of evolutionary systems.

they can be taken account of initially and — perhaps — be pretty much ignored thereafter. For a contemporary business organization, the environment is increasingly unstable: for most organizations, environmental *change* is the "constant," and the *rate* of change is increasing. This claim has been well documented elsewhere.[4] Since an open system carries on transactions with its environment, in order to attain its goals or to sustain its current state, a viable organization must itself respond to changes in its environment. Decisions about *what* to do and *how* to do it will need to be made, periodically at best, almost continuously in the extreme. Robert Reich's well-known work, *The Next American Frontier*, is, among other things, a sustained argument in defense of the claim that American industry, to survive and flourish, must acknowledge that its environment is changing and that it must respond by altering what it does and how it does it.[5]

For a contemporary business organization, the environment is increasingly unstable.

Decision-making

Clearly the focus here is on *decision*-making: the goals a system seeks to attain, the methods used to attain them; the methods employed to maintain itself as a viable entity. What model of decision-making is appropriate here? The traditional "preferred" model is the so-called "rational" model: the decision-maker selects a goal (for "economic" man, this might well be profit-maximization), examines a large number of alternative means to attain the goal, evaluates these via some sort of cost-benefit analysis, and selects the (an) optimum alternative. This approach assumes that the decision-maker can obtain the appropriate — and complete — information on: what alternatives exist, how much each costs, the consequences of implementing each, and the values of these consequences.[6] In a dynamic environment, such information is purely and simply unavailable. In an environment characterized by rapid change and, hence, by *novelty*, we simply *do not know*, with anything close to complete assurance, what the consequences of our actions will be. This has always been true: our activities have always *altered* the environment, hence our knowledge *about* the environment — based on *past* experience and information — has always been suspect. Now however, the fact is too blatant to ignore. The "rational" model is less and less appropriate as the environment changes more and changes more rapidly.

Human decision-makers aren't really very good at decision-making if you compare their performance with the optimal.

Several alternatives to the "rational" model of decision-making have been proposed: "satisficing," the "organizational procedures" view, the "political" view, the "individual difference" view.[7] And the *cybernetic* view.[8] Back in 1961, while Jay Forrester was describing the "systems dynamics" approach, in his *Industrial Dynamics*, itself based on a cybernetic view of decision-making, he commented, in passing, that human decision-makers aren't really very good at decision-making if you compare their performance with the optimal (in those cases where the optimal can be determined).[9] That this is in some sense true seems to have escaped all those who continue to subscribe to some form of the "rational" view. It is true when by "decision" we mean an individual, isolated decision. But what cybernetics has shown is that decisions really ought to be treated as a *directed sequence* of decision-acts: an organizational decision-maker selects, on the basis of

the available but always inadequate information, an alternative; he/she then monitors the effects of that decision (feedback), and takes whatever corrective action is warranted (the *next* decision-act); then the consequences are again monitored, and corrective action is again taken, and so on. With sufficient experience, wisdom, and — let's admit it — luck, reasonable (not perfect) alternatives will be selected at each step; with adequate monitoring, the reasonableness of the *next* alternative selected will improve. Thus the notion of "decision" as a directed sequence of decision-acts: "directed" because a goal has been selected; "sequence" because the available information, and the time appropriate to response, at any given point is always inadequate to guarantee the correctness of any single decision-act.

Under conditions of uncertainty, the essence of good decision-making becomes learning rather than knowing.

The above account amounts to a view that could aptly be called "decision as experiment." We *try* something; we *see what happens*; we *modify* our activity and try something slightly (or grossly) different; we again see what happens; and so on. Given that the environment with which the organization carries on its transactions is both unknown (to some extent) and changing — both as a consequence of our actions and in response to the activities of other systems — one should expect the decision-process to be non-terminating. The traditional "rational" approach to decision-making simply fails to acknowledge — or at least fails to highlight — the most pertinent aspects of real-life decision-making: we can't possible know all that we need to know in order to make *the* correct decision; and even if we did know and did make the correct decision, the useful life of its "correctness" would probably be short!

Substantial evidence that the "decision as experiment" approach contributes to the success of business organizations is provided in Peters' and Waterman's best-selling *In Search of Excellence*.[10] Whole chapters extoll the virtues — based on actual performance, not hypothesis — of the experimental perspective. Again and again, one encounters the "bias for action" and the "tolerance for failure," both of which characterize the experimental approach to decision-making. The message is obvious: "Learning and progress accrue only when there is *something* to learn from, and the something, the stuff of learning and progress, is any completed action."[11] Under conditions of uncertainty, the essence of good decision-making becomes *learning* rather than *knowing*. It is noteworthy, and hardly a coincidence, that Peters and Waterman pointedly take the "rational" model of decision-making to task, and that Reich criticizes the "business school" approach to management, an approach that has been dominated by the "rational" view.[12]

Decision-making and Information Systems

What has all of this to do with MIS? If we in the information systems profession truly intend to serve our organizations, then the above has profound implications for how we should perceive the design, development, and operation of information systems. For, the dynamism inherent in the decision process impacts the adequacy of information systems that support the decision process *itself* and that support the *implementation* of these decisions. In short, there are potentially *no* time-stable environments for information systems.

Given the systems theoretic perspective taken earlier, the organization becomes the environment for information systems. Since the organization itself is to respond to a dynamic environment, via the decisions made by its managers, the organization becomes dynamic in nature. Since the information system, to be responsive to its users, must accommodate the dynamism of the organization, it too must be dynamic — *fundamentally*, not occasionally or accidentally. The importance of this can be appreciated if we articulate the various approaches to change that have been manifested by systems development personnel and in systems development methodologies during the short history of computerized information systems.

There are potentially no time-stable environments for information systems.

Approaches to Change

There are essentially four approaches that might be taken toward information system change, both within the development process itself and with respect to the "finished" product (the resulting system): (1)*prevention* of change; (2) *control* of change; (3) *assumption* of change; and (4) *exploitation* of change.

1. *Prevention*. This approach has probably never been *consciously* applied to *completed* systems, but has had a glorius career in the context of the development *process*. The traditional development process, with its serial sequence of "phases," emphasized the "sign-off" in an attempt to secure user agreement with respect to system specifications prior to system implementation (specifically, detailed design and programming). Once the sign-off occurred, the overall design specifications were "frozen;" while the delivered system might in fact meet these specifications, the sytsem could fail to meet the true needs of its ultimate users, either because of errors in the specifications or — and this becomes important in the present context — because user needs have *changed* in the meantime. Change "prevention" may be a fine way to assign responsibility in cases of system failure, but it does little to guarantee system success.

With regard to *completed* systems, the "prevention" philosophy tends to function implicity, as a consequence of the "investment of resources" that the completed system constitutes, and due to the difficulty, and resource-consuming character, of traditional "maintenance and improvement" tasks.

2. *Control*. The "control" philosophy addresses the major drawback inherent in the "prevention" approach as it applies to system *development*: that the completed system may meet the specifications but fail to meet the needs of the users. In the "control" approach, specifications are not literally frozen, but any proprosed changes, once the original specifications are accepted by the user, are subjected to system re-analysis, re-design, and resource re-estimation, and all parties once again sign-off as to their agreement on the new specifications and new resource estimates. Often the charge-out, to the user, for the additional estimated resources, serves to constrain (or restrain) the user with respect to change requests. In extreme cases, the "control" philosophy reduces to the "prevention" philosophy in its effects: changes are so expensive to the user that they are not proposed at all. The resulting system once again fails to meet (some of) the true needs of its users.

The "control" philosophy, as applied to *completed* systems, is probably the dominant one in use today. Maintenance and improvement requests are often treated basically as new sytem requests, with abbreviated analysis, design, and implementation phases, complete with project estimates and user sign-offs. The user (or the organization) decides whether the modification is worth undertaking, based on time and other resource estimates. Maintenance backlogs often increase these *elapsed* time estimates dramatically. In such cases, the "control" philosophy again reduces to that of "prevention": change becomes prohibitively expensive.

3. *Assumption*. The "assumption of change" philosophy is characteristic of the prototyping mentality and of the prototyping methodologies that are now coming into use. Here it is *assumed* that no set of specifications is entirely "correct;" or, if correct, that the specifications will reflect reality for only a limited period of time. The development methodology features the quick construction of an initial product (the prototype), easy change, and fast turnaround; on the other hand, "maintenance" is assumed to be continuous and on-going. Change is neither viewed as an evil to be eliminated nor as a costly inconvenience to be carefully controlled (actually: restrained). Rather, change is assumed and is incorporated directly into the development process — in fact, development and maintenance concepts and processes cease being treated as separate phenomena: development *is* on-going mainte-

nance.[13]

The "assumption" approach was originally adopted in response to the perceived need to invent a development methodology that would accommodate the special requirements of decision-support systems. I will argue below that is should *not* be limited to this context.

4. *Exploitation*. To date, the "exploitation" philosophy has been adopted by some organizations as an overall competitive strategy, but is probably relatively rare in the information system development field. Here, change is *embraced*: an organization (or sub-organization) deliberately pursues those product/service areas in which it can exploit its own ability to change more rapidly and more successfully than the competition. "Fad"-oriented companies, like Wham-O (inventor of the hula hoop, the Super-Ball, and the frisbee) exploit an ability to enter a market quickly, dominate it, and get out (if necessary) upon saturation or upon perceived market decline.[14] Whether the "exploitation" philosophy makes sense for adoption by an organization's MIS area reamins to be seen. One can, however, easily imagine an *external* MIS organization (service bureau) promoting its services on this basis.

Prototyping: Specialized Technique or General Approach?

A common argument in support of prototyping places it in the context of decision-support systems: The application of computers to managerial decision-making is new, poorly understood, subject to the idiosyncrasies of the decision-maker, etc., so that determination of precise specifications prior to implementation is impossible or, at best, unrealistic. Thus an "evolutionary" approach to decision-support system development is needed. At best, the final iteration of the prototyping approach will provide a set of solid design specifications — the (modified) prototype becomes the speficiation for its own replacement.[15]

Such an argument misses the important point that the *only* information system development methodology that makes sense is one that can accommodate almost continuous, rapidly-implemented change. This is not to say that every system *will* change. It is to say that every system *may* require changes, both during its development and after its institutionalization within an organization. It is further to say that such change should be *expected*, given the previous account of organizations, their environments, and the decision-making required to make organizations responsive to their environments. A set of specifications that gets changed because the *requirements* change doesn't look all that different from a set that gets changed because someone got the specifications *wrong*: in both cases, the specifications *change*, and the developing or developed system should too.

The only information system development methodology that makes sense is one that can accommodate almost continuous, rapidly-implemented change.

To argue that some applications systems are relatively immune to either problem — to requirements change or to unclairty with respect to requirements — is to deny the dynamic nature of organizational environments and of information system environments. Who would today argue that a payroll system — once the paradigm of a stable data processing application — is immune to change? It will of course be true that, over time, some applications will be found to have required fewer, less dramatic changes than others. But who would wish to predict in advance which ones these will be? Even should such predications turn out correct, the underlying *philosophy* of information system development and maintenance should square with that of the organization itself — and the evidence seems pretty much in regarding the appropriate nature of the latter!

There is little doubt that the complete range of technologies — especially the software — required to support a full-fledged prototyping approach, covering all types and sizes of information system, is not yet available. The support-products that have recently been appearing, however — application and program generators, report generators, sophisticated

database systems with high-level query and development languages (4GLs), modeling languages, and DSS generators — suggest that it will perhaps not be too long before a thorough-

The transition to the prototyping philosophy needs to be made today.

going commitment to such a methodology could realistically be made.[16] Regardless of when exactly the full transition to the prototyping *procedure* can be made in practice, the transition to the prototyping *philosophy* needs to be made today: the viability of our business organizatons may depend upon it.

Conclusions

The recent crisis of American business — mainly its apparent inability to compete effectively in the world market — has led to several penetrating analyses and critiques. What seems to emerge from the best of these studies is the fact that American business inhabits an increasingly dynamic environment and that it has largely failed to respond adequately or quickly enough to the rapid and on-going changes in that environment. Independently of either the crisis or the analses, MIS theoreticians and practitioners have been developing the concepts and tools of the decision-support system, in an attempt — finally — to utilize computerized information systems in support of managerial decision-making. The special nature of such systems — in particular the fact that user requirements can rarely be specified in advance — has led to the realization that a new development methodology is required, namely, prototyping. A systems-theoretic analysis of these two developmetns shows that, despite their independent evolution, they are best consciously and deliberately synthesized: the solution to the problem of organizational nonresponsiveness requires, first, a change in perspective on effective decision-making, but second, a call for information systems that are themselves dynamic enough to provide effective organizational support. Thus prototyping emerges, not as one methodological option, to be employed in special cases, but as the essence of a development and maintenance approach that must become foundational to the organization's information systems. Prototyping — as a concept and as a methodology — must emerge as the philosophy that underlies future systems analysis, design, implementation, and evolution. If it is true that information systems are to support organizations, then information systems should become as dynamic as those organizations must themselves become, in order for the latter to survive, to regain their innovative and competitive edge, to flourish. •jsm

Prototyping must emerge as the philosophy that underlies future systems analysis, design, implementation, and evolution.

References

[1]J.D. Naumann and A.M. Jenkins, "Prototyping: The New Paradigm for Systems Development," *MISQ*, vol. 6, no. 3 (1982), pp. 29-44.

[2]Naumann and Jenkins, "Prototyping," pp. 37-42; P.G.W. Keen and M.S. Scott Morton, *Decision Support Systems: An Organizational Perspective* (Reading, MA: Addison-Wesley, 1978), chapter 6; S.L. Alter, *Decision Support Systems: Current Practice and Continuing Challenges* (Reading, MA: Addison-Wesley, 1980), chapter 7.

[3]Schoderbek, Schoderbek, and Kefalas, *Management Systems: Conceptual Considerations* (3d ed.; Plano, TX: Business Publications, 1985); R.L. Ackoff, "Towards a System of Systems Concepts," *Management Science*, vol. 17, no. 11 (1971), pp. 661-671.

[4]A. Toffler, *Future Shock* (New York: Bantam, 1971).

[5]R.B. Reich, *The Next American Frontier* (New York: Times Books, 1983).

[6]J.D. Steinbruner, *The Cybernetic Theory of Decision* (Princeton, NJ: Princeton University Press, 1974), chapter 2; Keen and Scott Morton, *Decision Support Systems*, chapter 3.

[7]Keen and Scott Morton, *Decision Support Systems*, chapter 3.

[8]Steinbruner, *The Cybernetic Theory*, chapter 3.

[9]J. Forrester, *Industrial Dynamics* (Cambridge, MA: The M.I.T. Press, 1961), esp. chapter 1.

[10]T.J. Peters and R.H. Waterman, Jr., *In Search of Excellence* (New York: Harper & Row, 1982).

[11]Peters and Waterman, *In Search of Excellence*, p. 134.

[12]Peters and Waterman, *In Search of Excellence*, chapter 2; Reich, *The Next American Frontier*, chapters 4, 7, 8.

[13]Naumann and Jenkins, "Prototyping," pp. 32-33.

[14]Toffler, *Future Shock*, pp. 72-73.

[15]Naumann and Jenkins, "Prototyping," p. 33.

[16]J. Martin, *Application Development Without Programmers* (Englewood Cliffs, NJ: Prentice-Hall, 1982); J. Martin and C. McClure, *Software Maintenance: The Problem and its Solutions* (Englewood Cliffs, NJ: Prentice-Hall, 1983).

Prototyping for systems building

Effects on the DP department

by RUSSELL JONES

Many computer systems fail to meet the requirements of the business managers for whom they are written. The fundamental cause is an inability of computer systems designers either to understand the requirements of their business peers, or to translate these requirements accurately into operational computer systems.

Fortunately, there exists a totally new method of designing and developing computer systems which can help overcome this fundamental problem. Increasingly, a new breed of systems developers is using 'prototyping' techniques to form the heart of radical methods of designing and building computer systems. Methods which allow such systems to be built in one third of the time taken by traditional methods.

Abstract: Systems design is often time consuming and mismanaged. Powerful software tools can help design prototype computer systems, which offer the computer department the opportunity of testing their design before developing the final version. It requires a close interaction between business managers and user staff as well as changes in responsibility to obtain the best results.

Keywords: data processing, system development, methodologies, software tools

Russell Jones is a technical journalist.

Prototyping is not a new concept; it is similar to the well tried and tested engineering practice of building prototypes of, for example, cars, before moving to their final production. But how does prototyping differ from those methods currently employed by computer systems designers? In essence, current methods attempt to get the design of such systems right first time, an action which is rarely possible in practice. Consequently, many computer departments spend a large proportion of their time maintaining computer systems which were initially designed badly.

Present design methods

Often, it is not the computer department's 'technical' designs, i.e. the way they utilize the computer, which are incorrect, but their basic 'business' designs, i.e. the way in which they attempt to model the business environment within which they are used.

This 'business' design is important. All computer systems are concerned in some way with modelling business life, and the first task to be performed in designing and building any proposed system must be concerned with gaining a detailed understanding of the specific business environment within which that system is to operate.

Unfortunately, in many cases this task is not being performed well enough. Why is this? The major reason can be found by looking in detail at the techniques and tools used by computer staff to perform this initial business design.

When a new computer system is to be built, system analysts discuss the requirements of the business department and return with the specification for signature. Users start programming against this rigid specification and wait months or years for delivery of the system. No user can comment on the suitability of a computer system until he or she has tried it out in practice. Using a computer system can cause a user to rethink business procedures; most users do not understand the minutiae of the detailed technical system specification they are presented with, and are thus in no position to criticise it.

The result is that many new computer systems must, almost immediately, be altered radically. Such alterations are time consuming and expensive, because the same criteria can be applied to building a computer system as to building any finished product. If the basic design is wrong, all subsequent development work must at best, be reworked; at worst, it must be written off. This is why so many computer departments spend so much time maintaining, more correctly, reworking old computer systems.

The lesson from this is if a method can be found which helps improve the basic business design of computer systems, then better systems will be produced more quickly. To achieve this aim, there is one method available to computer staff, to tap the detailed knowledge of the staff who will eventually use the computer systems. It is

0011-684X/85/070024-03$03.00 © 1985 Butterworth & Co (Publishers) Ltd.

this knowledge which the prototyping method of systems design exploits.

Software tools

The use of prototyping by computer systems designers is an attractive proposition, so why has it not been used before? The reason is technical. Most types of prototyping involve the building of initial 'cut down' versions of computer systems quickly. Until recently, the relatively high cost of computer hardware, together with a lack of system building 'power tools' to build prototypes quickly enough, had precluded its widespread and effective use. The arrival of low-cost and powerful personal computers and powerful, fourth generation software which can generate systems quickly and automatically, has removed these prototyping barriers.

Prototyping uses these powerful software tools to build scaled down versions of eventual computer systems. These 'first cut' prototypes can be developed in a fraction of the time it would normally take to develop full scale systems. User staff can review these prototypes, and can comment on their design far more effectively than they were able to on traditional 'paper' specifications.

During the review, user staff and computer staff can work together in refining the prototype. User staff can amend it as they gain practical experience in its use, with computer staff available to ensure that technological constraints are not exceeded. However, it is users who are in control. They can specify screen formats; use powerful 'screen painting' tools; draw up basic report structures and dictate the scope of the business information to be processed by the computer system.

At the end of this detailed review both user and computer staff will have built an accurate working model of the eventual system. Because the users were involved in the design of the real working system, this model will be more likely to reflect realistic business requirements. User staff will also feel a high level of commitment to the proposed computer system. This commitment can be a critical 'political' factor in smoothing user problems during the difficult first few months of 'live' running.

Development variations

While the basic concept of prototyping is increasingly used by a wide range of systems developers, there are a number of interesting variations in the detailed approach taken by particular developers.

There are some who mimic, almost exactly, the engineering approach to prototyping. They build one, or at most two, prototypes; learn sufficient from that process to be able to draw up a full detailed and costed final specification, and then use a variety of third and fourth generation software development tools to build the final product.

Using this method, development staff initially meet clients to discuss their overall requirements. On the basis of this meeting, a firm quote for developing a prototype system is provided, together with an estimated time and cost for the full system development.

They then develop a full working prototype of the system required, which, while not capable of encompassing every detail of the eventual system, does provide an accurate representation of its overall scope. This prototype of the system required, which, while not capable of encompassing every detail of the eventual system, does provide an accurate representation of its overall scope. This prototype is shown to user staff, who can work with it to gain experience of using the proposed system. Typically, at this review stage, user staff modify their design criteria; often, they add extra functions and increased detail to the proposed system.

Only when this review has been completed successfully, and when user staff are absolutely happy with the overall design of the system, are a timescale and cost agreed for the full system development.

The attractions of even this limited use of the prototyping approach to prospective users of computer systems are:

- They see a system working before being asked to specify details of their requirements.
- They only need to commit a fraction of the expected overall development cost, before seeing a working prototype of their system.
- The prototype gives them confidence that the final system will meet their needs.
- Since the same power tools are used to develop both the prototype and the final system, users typically take delivery of high quality systems, more quickly, which reflect their requirements.

However, there are also a number of organizations which take the prototyping approach further than building one or two 'first cut' systems. There is a whole range of fourth generation application generation tools which enable complete systems to be developed directly from initial prototypes. This method is a process of 'iterative design'. Users are involved almost full-time in the design and building of their systems, from initial concepts to the minutiae of, for example, screen processing of a particular online transaction.

Prototyping has been used in a variety of organizations, and the overall response from users has been most encouraging. Not only do their systems more closely mirror their business requirements, but they are also built quickly and inexpensively. Typically it takes one third of the time it might have taken using other methods.

As a method of building computer systems, prototyping is very different from those techniques currently em-

ployed by most computer departments. In many large-scale organizations, there is poor contact between computer and business staff. Prototyping relies on, for its success, a close interaction between both sets of staff, especially during the review period when the initial design of a system is refined as users gain experience of it.

Changes in responsibility

The introduction of prototyping into an organization requires a number of management changes. These revolve around how the computer department performs its work in dealing with business staff, and also around the type of staff employed by those computer departments.

Within many computer departments there is still a clearly defined distinction between systems analysts and computer programming staff. The former go out and talk to user staff as part of the design process, whereas the latter typically work exclusively within the computer department, turning designs into working systems. This distinction can cause problems.

Systems analysts are often ignorant of the full capabilities of the range of technical tools at their disposal. Equally, many programming staff spend far too much time dealing with essentially mundane technical details, and, consequently, are ill-informed as to the business side of the organization. Such arbitrary distinctions in job content lead to communication problems between the systems analyst, designer and programmer.

Fortunately, the introduction of prototyping techniques lessens these distinctions. One aspect of this is the changed role of the programmer. To succeed, prototyping relies on the use of a range of comprehensive, fourth generation, application generator software tools, to build prototypes quickly. These tools diminish the level of technical skill required of programmers. They produce programs directly from high-level definitions of screen layouts, reports, data validation rules etc.

Programmers no longer need to perform many of those fairly mundane programming tasks. The availability of powerful tools gives the programmer freedom to get on with the more challenging and difficult types of programming, such as complex calculations which are, as yet, outside the scope of the new tools. Prototyping also requires computer programmers to communicate more often and more readily with business staff. This increases the perception by such staff of the problems and working patterns of real business life.

New breed of systems builders

The end result of the changes will see the emergence of a new breed of staff within computer departments. Computer staff will no longer be merely programmers, or analysts, but 'systems builders'. They will possess a variety of different skills, including:

- an understanding of the business culture within which they work,
- a detailed knowledge of the powerful software and prototyping tools available to them,
- a good range of interpersonal skills to enable them to relate to, and understand, the problems of business staff.

In short, any team charged with developing a computer system for a particular business department is likely to comprise a number of multidisciplinary staff. They will be equally at home in dealing with, for example, the mysteries of cash flow as they will with the system building tools at their disposal. Such an alteration will require changes to the way in which computer staff are trained. More emphasis will be placed on business and personal skills, decreasing the emphasis on the more mundane programming skills.

Such teams are also likely to be much smaller than at present, since prototyping enables computer systems to be built very quickly. In turn, this will mean that computer staff will need to exhibit a much wider range of management skills than at present. They will need to be highly motivated, capable of working, either alone, or as part of a two or three person team, directly with user staff. They will also need to be mature and flexible in their response to user requirements, and be able to 'sell' their own ideas and suggestions as part of the design process.

There will also be a number of changes for user staff as a result of the introduction of prototyping. It will put the responsibility with user staff to get the computer design right. Prototyping relies for its success on the full involvement of user staff. To achieve this, user management must be prepared to allocate sufficient time and resources to the task of helping to build their own computer systems.

An awareness of the real business opportunities which computerization offers must become a natural part of business managers work. They must be prepared to make positive suggestions about the use of computer systems, and, via the prototyping approach, become fully involved in their construction. Organizations will certainly obtain better systems more quickly, and the concept of using computers will start to become as natural as using any other piece of office equipment.

Prototyping turns the development of successful computer systems into a joint effort between user and computer staff and the former needs to be fully committed to its use. Unless user staff are prepared to contribute fully to the system building process, prototyping can be no more successful than current methods. Certainly, the use of prototyping places a burden of time and commitment on user staff. However, the results prototyping produce make such an investment well worth making. □

Techniques and Issues in Rapid Prototyping

BY TERESA STEINBOCK HARRISON

In the traditional software development cycle we find that there is a substantial time delay between the initial analysis and the delivery of an operational system. For large projects development time may be well over a year. Figure 1[13] portrays such a traditional development cycle consisting of the phases requirements definition, design, code, test, and maintenance, with the cumulative effort devoted to each phase.

During this development time one will typically find that a designer consults the user only when some specification is unclear. Rarely are there frequent system performance reviews that allow the user to see the progress to date. Rather, the user commonly sees the system for the first time when it is delivered.

After the system is delivered, discrepancies are often discovered between what the users wanted and what is delivered as the final product. Whether the discrepancies result from ill-defined requirements or from misinterpretation on the part of the designer, the result is unsatisfied users. The users may refuse to use the system or use it maliciously until it is either changed or discarded.

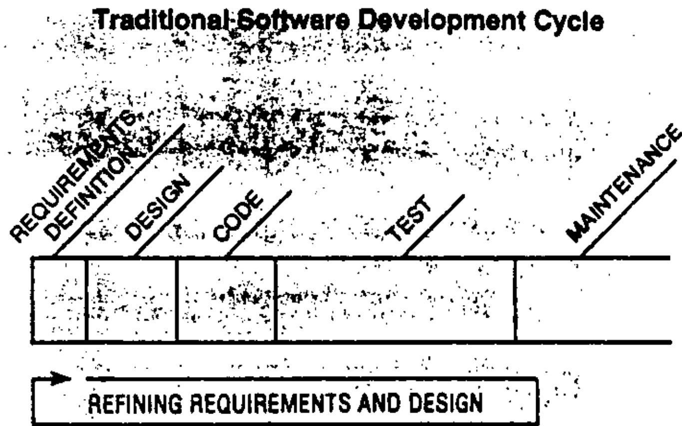


Figure 1

In response to these problems the notion of rapid prototyping of software systems has evolved. A model of the rapid prototyping development cycle is presented in Figure 2.[13] This technique focuses on user involvement throughout the development process. As the development phase procedes, partial models of the proposed system are built.

By building partial models of the system, information is provided about the system's behavior. The users are then allowed to use these partial models. Thus, their comments and responses can be gathered before any design becomes final. If the design is unsatisfactory, then it can be changed *quickly* and *easily*. It is an iterative process of design, testing, and modification with the user continuously involved until all requirements are met.

TERESA STEINBOCK HARRISON

Teresa Steinbock is a systems analyst with Accountants Library of Professional Software, an Oregon based software firm specializing in accounting packages. She holds a BS in Mathematics and Computer Applications Management from the University of Portland and a Master's degree in Computer Science from Oregon State University. She has taught Computer Science courses at Oregon State and previously worked as a Programmer and Systems-Analyst for the Bonneville Power Administration and Tektronix. Her current interests include rapid prototyping of information systems, data base management systems, and decision support systems.

Software Development Cycle with Rapid Prototyping

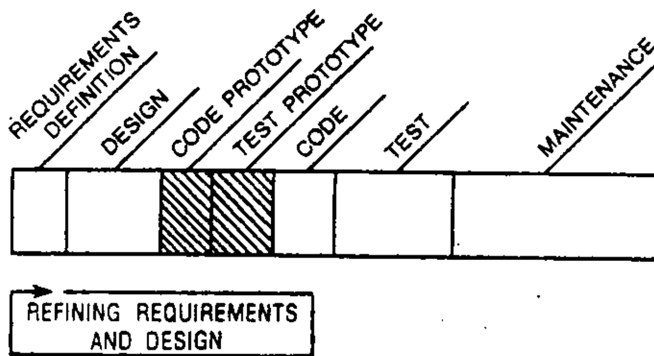


Figure 2

It seems that rapid prototyping will result in more commitment on the part of the users since they had a part in the development and they obtain a better understanding of how the system works. Furthermore, fewer changes that arise from ill-defined requirements need to be made to the system after delivery since the requirements have been well-defined upon completion of the prototype.

Three major issues to be addressed when building a prototype are:

1. What aspects of the system should be prototyped?
2. What technique will be used to build the prototype?
3. Once the prototype has been completed, what will become of it?

Aspects of the System to Prototype

Once the decision has been made to prototype a system, the question of what should be prototyped arises. The areas most likely to be prototyped include the following:[25]

— User Interface
— System Functions
— Time and Memory Requirements

User Interface — In a data processing environment with input and output processes dominant, user acceptance of a system depends on the system interface to the human operator.[18] Here, decisions such as whether to use *menus* or a *command language* to drive the system are made. Also designed may be the dialogue including the prompts, available selections, error messages, and methods of recovering from an error.

By developing a prototype of the user interface, the user is allowed to view the system as if it were the final system. The user is able to interact with the system in its dialogue, test and data entry procedures, and view the sequences of screens. If a design is not satisfactory, then it can easily and quickly be changed. Some changes may only be cosmetic (fields line up, consistent headings among screens, consistent error messages) or some changes may arise due to crucial omissions (a particular field is omitted), but they have a significant impact on the user and in the end they will make the system more acceptable to the user.

System Functions — If a final system does not meet some user requirements, we can assume that there was some uncertainty or misunderstanding between the user's needs and designer's interpretation. Therefore, some aspects of the application will require actual database interactions and computations using limited samples of data. If a prototype is constructed to model areas of uncertainty or important features of the system where questions may arise, then it is possible to rapidly arrive at a program that allows the user and designer to analyze the functionality and correctness. Here, it is important not to be distracted with details such as error routines or efficient algorithms. It is a quick way to determine if the designer understands what the user wants and to test alternative ways of performing certain functions.

Time and Memory Requirements — In this area the concern is whether some function can be performed in a predefined time and/or memory allocation. Therefore, a prototype built in this area

focuses on implementing time constrained aspects of the system or memory consuming portions of the system to determine the feasibility of selected algorithms. While this aspect can be prototyped, it is more properly included in simulation. Thus, we will defer further discussion of this aspect.

Current Directions in Selecting System Aspects — Currently, most rapid prototyping applications have focused on the area of the user interface and system functionality. It should be noted, however, that prototyping one area does not necessarily preclude prototyping in another. One possibility is for several prototypes to be built, each portraying different aspects of the system. An example of this approach can be found in.[26] In this case, one prototype was built to design the user interface and a second prototype was built to design the functions of the final system. Based on these prototypes, the final system was then built. A second possibility is to build a prototype that portrays more than one of the areas. This approach has been taken in a majority of the prototyping applications discussed here. We will now look at the various methodologies currently being employed to quickly generate prototypes.

Rapid Prototyping Techniques — The techniques commonly used to generate prototypes consist of executable specifications, scenario-based design, special purpose languages, automatic programming, reusable software and simplified assumptions. The remainder of this section describes each of these approaches. The use of a particular technique is not limited to use with a single system aspect. Many can be used with both major aspects (See Table 1). Furthermore, these techniques can be combined in developing a prototype.

Table 1
Areas in which various methodologies can be used

Methodology	User Interface	System Functions
Executable Specifications	X	X
Scenario-Based Design	X	X
Special Purpose Languages		X
Automatic Programming	X	X
Reusable Software		X
Simplifying Assumptions		X

Executable Specifications — Executable specifications is a technique used to automate the process of requirements specifications. An executable specification language describes the intended behavior (what) without describing any particular algorithm (how).[1] As a result, there is freedom from implementation concerns. With this approach, one can directly observe the behavior of any system that one can specify in the language.[10] Two approaches to this technique exist: algebraic specifications and finite-state models.

Algebraic Specifications — Algebraic specifications involve a description of objects in terms of sets, functions over those sets and invariant equations over the functions.[21] The invariant equations comprise a specification which can be viewed as reduction rules as in the OBJ language.[10] Given an OBJ specification and an initial expression, the expression may undergo several rewritings determined by the equations of the specification.

Finite-State Models — The most widely used form of executable specifications are based on finite-state models or transition diagrams. When starting in an initial state, an input is accepted, an output produced, and the state changed. Constraints on the properties of elements of the state serve to determine valid state transitions.[21] An example of prototyping the user/program dialogue is shown in Fig. 3.[24]

USE Transition Diagram

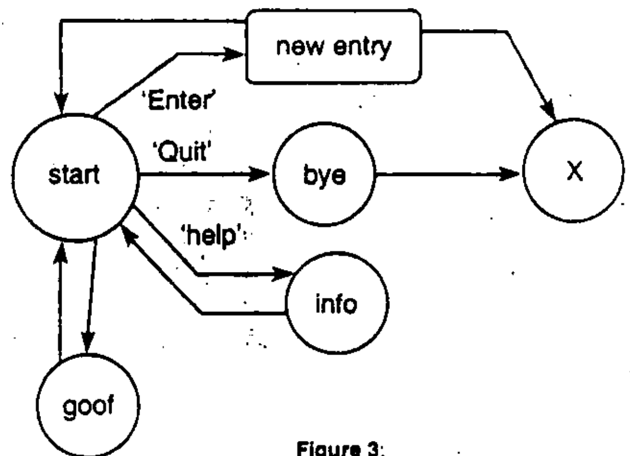


Figure 3:

Starting in the initial state START, the message associated with this state is displayed. If the input is "quit" then the path from state START to BYE is traversed, any associated actions take place and the message associated with state BYE is displayed. Similar actions occur if the string "help" is entered. If the input string is "enter" then a subconversation NEW ENTRY, represented by another USE transition

diagram, is evoked. Any input string that is not defined results in the path to GOOF being traversed.

Once a model contains sufficient operational detail it can be interpreted to produce an executable design.[22] This can be used to display approximate behavior of the intended system for analysis by both user and designer.

One of the major advantages of this method is the ability to verify the correctness and consistency of formal specifications. Once the requirements have been defined in a formal and logical manner the requirements will be more complete and comprehensible. Since the diagrams are interpreted directly the users are able to see resulting actions and developers are able to observe if any requirements clash. If any problems arise, a behavior is changed and the specifications re-executed.

Interpretive models currently being used include Gist[1,4,7], RSP,[5] OBJ,[10] PSLAIR1,[17] and USE.[24]

Scenario-Based Design — It has been recognized that the longer errors remain in software, the more costly they become to fix. Since requirements identification is the first step in the software life-cycle, errors in this phase incur a substantial cost to correct after system installation. Scenario-based design attempts to address the problem of requirements identification.[11] A scenario simulates events the user would experience during system operation. It provides a model of the input-process-output screens and associated dialogue. Consequently, the developers are able to present the user with a realistic view of the system to appear[16] as it is currently conceived.

One suggested approach to the development of scenarios is through the use of a set of reusable software modules.[11] These reusable modules would be generic so as to represent aspects of any given scenario. Modules would be pre-developed so that when initial requirements have been conceptualized, the designer sets up a configuration of modules that model the system. As requirements are modified so too are the scenarios until a satisfactory system evolves.

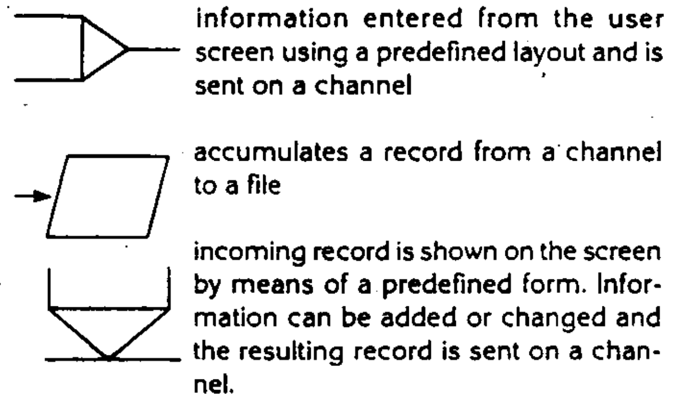
Another more widely used approach to scenario-based design deals with the development of a prototype language such as ACT/[15,16] or HIBOL[18]. The scenario development process consists of defining screens, data entries, and their consequent actions. An example of such an approach is done in HIBOL. For this example we can assume that the screen/report layouts have defined through the use of some screen/report generator provided by the language.

From the external description and appearance of the system, one then works inward to develop details that are consistent with the external view.[15] This would include such things as incorporating a database interface and error handling and recovery procedures.

The advantage of scenario-based design is in the development of data processing applications. In this area much emphasis is placed on the input-output processes and the interface between user and computer. The effectiveness of data processing systems developed rests heavily on the constructed scenarios and dialogue. By providing a design methodology such as scenarios, the interface can be created and modified quickly until it is acceptable.

Scenario-based designs currently in use include ACT/1[15,16] and HIBOL.[18]

Special-Purpose Languages — Special-purpose languages are modeling languages which use the terminology of the application domain. When used in the development of a prototype it facilitates the communication between user and designer about characteristics in the proposed system. One particular modeling language that has been developed is for information flow.[9] Administrative processes are modeled as packages of information which flow between work-stations in an organization. A graphical representation is initially used to describe the system with different symbols for different operations. An example of some of these symbols appear below.



Information Flow Symbols

The information flow graph is converted manually to a format language which is then interpreted to form a prototype system. This method has been used in a project called DASIS for a medical care system.[9]

Automatic Programming — Automatic programming is a proposed strategy for the development of rapid prototypes as an alternative to executable specifications.[2] It offers the advantages of the prototype running more efficiently and the spec-

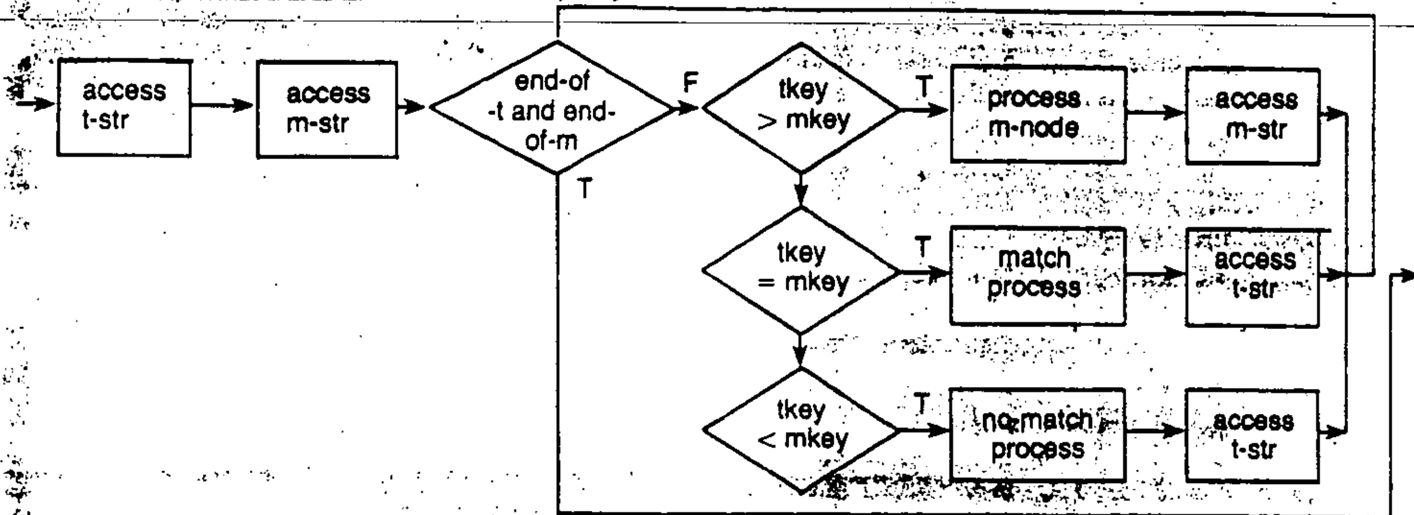


Figure 4

ification language allowing for more informality. The development of automatic programming systems requires the development of specificataion languages and the representation and codification of knowledge about specific programming techniques. Currently, the state-of-the-art techniques do not support this methodology.

Reusable Software — Reusable components are common occurring patterns of programming that exist in a specific application. They consist of an I/O specification, control structure, and a generic problem/solution description. To apply reusable components in the development of prototypes it is necessary to capture and correctly represent past experience. Based on these past experiences or knowledge new systems can be built.[19] Patterns are represented by flowcharts, pseudo-code, or other equivalent formalisms. An example of a pattern in the form of a flowchart[19] is shown in Figure 4. This pattern represents the task of file update using a master file and transaction file. The pattern is then interpreted to meet the requirements of the specific application. Using a combination of such existing patterns and, if necessary, the development of a new pattern, it is possible to rapidly create a prototype of the system.

Simplifying Assumptions — Simplifying assumptions are assumptions made in the development process which allow designers to rapidly arrive at a system. Although these assumptions may be false, the designer can focus on the important aspects without distraction of details. For example, in the process of updating a file, the assumptions can be made that the file does exist, there will be no errors when accessing the file, and the records to be updated exist. Once the proposed system meets all requirements, then these assumptions can be retracted and the details added. To support the use of simplifying assumptions a tool called the "programmer's apprentice" is being developed.[20] The PA supports the choosing of assumptions and keeps track of assumptions so they can later be retracted. Therefore, in the development of a system one can arrive at a prototype that demonstrates functions that the users see. When the requirements have been met, the details are added.

Prototype as Throw-away Design or Final System — Once a prototype has been completed that meets the specifications, one must consider what to do with it next. In this situation the main choices that exist are throwing the prototype away or converting it to the final system.

Those that believe the prototye version can or should be thrown away and the final system built from scratch base their arguments on the principle that shortcuts were taken in producing the prototype rapidly.[8] Things such as maintainability, efficiency, portability, and completeness are sacrificed for the sake of the speed of realizing a sample system with testability and modifiability. Trying to convert such a system to the final production version could lead to unnecessary complexity in the final system. The effect of building such a prototype is a better understanding and realization of the proposed system.

However, once the prototype has been completed its usefulness does not have to end. Two further possibilities exist for its use. First, the prototype can be used as a documentation of the design.[6,14] As the final system is being built, the prototype serves as the specification to follow, and when the system is complete some documentation of the design exists. Second, the prototype is used as a way of prototyping system maintenance.[6] Before a change is made to the production system, it is made to the prototype to determine if it is functionally correct. Once correctness is established, the change is incorporated into the production system.

Those who believe that the prototype can be converted to the final system hold that once the prototype is complete, the total system has almost been completed. All that is needed is to add functionality that the user did not see or alter and replace inefficient portions of code.[23] Another view is that the prototype is developed iteratively. Once the prototype meets all specficiations, the prototype has evolved into the final system[3,12].

The decision to throw away or convert to final product is not an arbitrary choice. One must consider the tools used and approach taken in developing the prototype. If tools exist for transforming and refining code into a system that is maintainable, efficient, and complete then the target system can be derived mechanically from the prototype. Also, where the prototype is built from a programming language (HIBOL or ACT/1) it is possible to complete the entire project in the language. IF the prototype can be enhanced with the addition of a few capabilities (error detection and recovery) and performance can be improved at a few crucial points, then it would not be economical to reprogram and entire system.[11] However, if the approach of developing several individual prototypes of the system was taken, then it would be advised to rebuild the final system using the prototypes as a guide line.

Conclusion

Traditionally, the user is seldom involved with a new system until it is delivered for his/her use. Thus many problems are not discovered until after delivery of the product. It has been well established that the earlier errors occur in the software life-cycle, the more costly they are to repair.[16] Rapid prototyping is an attempt to alleviate some of the problems encountered in the traditional lifecycle approach. Rapid prototyping has the following advantages:

- It provides an effective and efficient method of fact finding.[6] Communication is opened between user and developer and both learn what is really needed and wanted.
- It provides a potential learning tool.[6] The user gains experience with the computer as well as with the proposed system by getting hands on experience early in the development process.
- As users work with the system, they can determine if it is functionally correct. If not, changes can be made quickly before final details are added.
- If the prototype is not converted to the production system, it can be used as a documentation tool or as a tool for testing maintenance features.

We have seen a variety of prototyping techniques that have been proposed and implemented. Their success lies in the ability to overcome the communication barrier between user and developer and the ability to quickly incorporate changes. Extra cost may bee incurred with the development of the prototype but the overall costs of system development are lowered. **ejsm**

References

1. Balzer, R., N. Goldman and D. Wile, "Operational Specification as the Basis for Rapid Prototyping", *ACM SIGSOFT Software Engineering Notes*, December 1982, pp. 3-16.
2. Ibid, pp. 33-34.
3. Ibid, pp. 35-38.
4. Ibid, pp. 25-31.
5. Ibid, pp. 39-44.
6. Dearnley, P., and P. Mayhew, "In Favour of System Prototypes and their Integration into the Systems Development Cycle", *The Computer Journal*, Vol. 26, 1983, pp. 36-42.
7. Feather, M., "Mappings for Rapid Prototyping", *ACM SIGSOFT Software Engineering Notes*, December 1982, pp. 17-24.
8. Ibid, pp. 72-74.
9. Ibid, pp. 67-70.
10. Ibid, pp. 75-84.
11. Ibid, pp. 88-92.
12. Ibid, pp. 94-95.
13. Ibid, pp. 96-105.
14. Ibid, pp. 112-118.
15. Ibid, pp. 120-126.
16. Mason, R., and T. Carey, "Prototyping Interactive Information Systems", *Communications of the ACM*, May 1983, pp. 347-354.
17. McCoyd, G., and J. Mitchell, "System Sketching: The Generation of Rapid Prototypes for Transaction Based Systems", *ACM SIGSOFT Software Engineering Notes*, December 1982, pp. 127-132.
18. Ibid, pp. 133-140.
19. Ibid, pp. 141-149.
20. Ibid, 150-154.
21. Ibid, pp. 155-159.
22. Ibid, pp. 167-168.
23. Ibid, pp. 160-166.
24. Ibid, pp. 171-180.
25. Ibid, pp. 181-185.
26. Zelkowitz, M., "A Case Study In Rapid Prototyping", *Software-Practice and Experience*, Vol. 10, 1980, pp. 1037-1042.

Prototyping and the Systems Development Life Cycle

BY RALPH HARRISON

■ Designing and developing systems to satisfy user requirements is not an easy process. Since the mid-1970's, we have been developing systems following various System Development Life Cycle (SDLC) methodologies. These SDLC's may vary in content, but they all have one common purpose: to guide the successful development of the systems project.

Developing a system following an SDLC methodology is far better than ad hoc development. However, these methodologies are paper-based. They do not allow the users to:

- Get a hands-on feeling for the system prior to acceptance testing,
- Develop their requirements while gaining experience with the system.

Together, these two problems can result in a system that does not meet the expectations of the users. The consequences of this can result in:

- Expensive changes being made to implemented programs to process transactions correctly,
- New functions being added to the system, and
- The system not being accepted by the users.

This article discusses prototyping and its use in developing systems. It presents the benefits of prototyping in overcoming problems associated with the traditional SDLC, discusses the impact prototyping has on the SDLC, and presents the software tools that can be employed in prototyping.

RALPH HARRISON

Ralph Harrison is a senior consultant with Peat, Marwick and Partners' System Development Group in Ottawa, Ontario. He has participated in many aspects of developing on-line and large-scale systems. His primary area of interest is the control of the system's development during the project life cycle from the users' perspective. He holds a BASc in Industrial Engineering and an MASc in Management Information Systems from the University of Toronto.

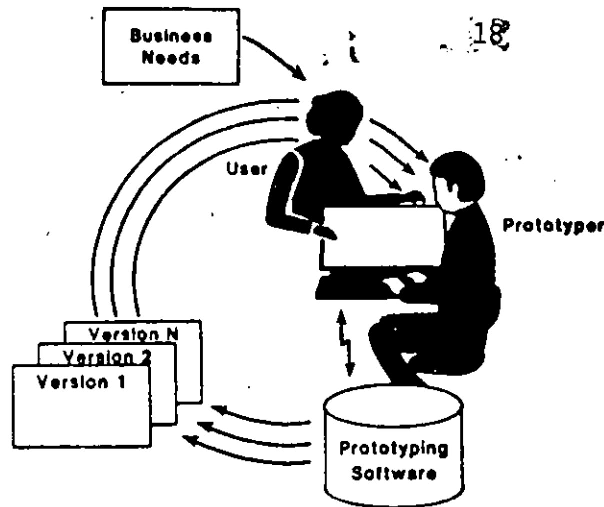
Prototyping enables the system to be designed from the users' perspective.

Prototyping

Prototyping for an on-line system is:

- The creation of on-line screens and reports,
- The simulation of interaction among on-line screens, reports and files,
- The implementation of a working model or subset of a system with real data.

Prototyping enables the system to be designed from the users' perspective. It is an iterative process in which screens, screen flow logic and transaction and processing logic are developed with the users through successive scenarios. It allows the users to work with the inputs and outputs of the system and experience the interactions among inputs, files and reports prior to system implementation.



Levels of Prototyping

There are three levels of prototyping. Each level can be used successively to contribute to the design and development of the system. The first level consists of on-line screens. The objectives of this level are to determine the screen and report layouts and contents, the screen flow sequence and the method for sequencing the screens. The systems analyst would develop scenarios to illustrate the basic processes of the application. Several iterations would be performed until the users were satisfied with the screen representations and screen logic flow.

The second level of prototyping expands upon the first by introducing database interactions and data manipulation. The main purpose of this level is to demonstrate the operation of key areas of the system. Users would enter specific sets of transactions, upon which simulated processing would be performed. Features such as error handling could also be incorporated. This level of prototyping assists in verifying the first level screen design and screen flows, and clarifying application processing logic.

The third level is a working model of the system. It is a subset of the system in which application logic transactions and database interactions operate with real data. The objective at this level is to develop a model that will evolve into the final systems specification. This objective is attained by allowing the users to experience the interaction with a live system. In many instances, if it were expanded by adding missing or incomplete functions or transactions, part or all of the model could be implemented.

The Benefits

Prototyping addresses the two shortcomings inherent in the traditional paper-based SDLC. Prototyping: is a user friendly technique; requires active user participation throughout the design of a system; and allows the users' requirements to evolve throughout the project.

Prototyping is a process in which the development of the system is driven by the users' interactions with the system's externals. Users can experience their interfacing with the system and can verify its operations. It enables the users to play an active role in the design stages of the life cycle and contribute to the design of the aspects that most affect them.

Prototyping reduces the risks associated in all systems development projects. It can be used for both large and small projects. However, its advantages are more evident when applied to large projects; for, the associated risks are greater. Large projects have more users, functions and interorganizational relations. The benefits of prototyping are greater when there are potentially more things that can be misunderstood by more people.

The concept of prototyping is applicable to both new development of and changes to systems. When modifications are minor, the existing implemented system becomes the prototyping vehicle. However, when changes are major, such as incorporating a new subsystem or new functionality, prototyping tools are best employed as they would be for new systems development.

Prototyping is not limited to solely designing the computer aspects of the systems development project. The third level of prototyping, a working model of the system, can be used to

represent the entire environment, including business system, manual procedures, system interaction and training. Prototyping is beneficial in determining and correcting problems associated with:

- Awkward, incomplete or wrong manual procedures,
- Disruption of the work flow,
- Interfacing and interaction with the system, and
- Preparing staff for the system.

In order to illustrate the benefits of prototyping, let us consider this simplified analogy to buying a custom built car. Following the traditional development life cycle approach, the builder would meet with us and determine our needs. He would then return with drawings illustrating his interpretations of our requirements and a list of specifications to be incorporated in the final product. The drawings and specifications would include items such as seat angle and height, legroom, headroom, tires, acceleration and braking. Following our review and subject to any adjustments, we would give approval to build.

Prototyping reduces the risks associated in all systems development projects.

A first level prototype would allow us to work with the builder in determining our requirements. A prototype would be developed consisting of a mock-up of the body with doors and an interior. We could open the doors, assess capabilities and sit in the interior. There, we could judge the adequacy of seat support, angle and height, legroom and headroom.

The second level prototype would expand upon the first. It would include the body and interior, and such items as the steering wheel, shift lever and pedals. It would allow us to test the steering, shifting or braking, although these actions would be simulated.

The third level would be a stripped down working model, complete with engine, transmission and mechanical components. It would allow for testing acceleration, cornering, and braking but would be absent of frills, such as air conditioning, carpets, and finished exterior and interior.

With the traditional SDLC, our involvement would be limited to the first stages, although we would be asked to review and approve refinements of the drawings and specifications. The next point where we would participate to any great extent is our testing and accepting the finished product.

On the other hand, prototyping requires our participation throughout the design and development processes. Clearly, in this analogy, the end product developed through prototyping will more adequately meet our requirements. Similar benefits can also be attained when prototyping is applied to the system development process.

Impact on the SDLC

Prototyping can be used in conjunction with the traditional SDLC to augment the users' participation in and understanding of the requirements, conceptual and detailed design stages. However, the use of prototyping in this manner tends to blur the distinction between the traditional stages. The three levels of prototyping respectively bring forward components from the conceptual design to requirements definition, detailed design to conceptual design and development to detailed design.

With the traditional SDLC, screens and reports are designed during the conceptual design. However, the first level of prototyping introduces the design of screens and reports as an integral component of the requirements definition stage.

In the traditional SDLC, the requirements definition stage is concerned with what should occur; while, the conceptual design stage focuses on how it should be done. Prototyping combines the design of the system's externals with the development of user requirements. The functions of the system are not developed separately from the methods of performing them. The defining of business processes is impacted by the introduction of on-line screens and is dependent on the users' interfacing with and operation of the system.

The second level of prototyping introduces components of the traditional detailed design stage to the conceptual design stage. This level augments the design of the file structures through the users' manipulation of screens and screen flows. The simulation of and interaction

with files requires a level of design detail not normally encountered until detailed design.

The third level of prototyping merges detailed design and development, for, it requires that a working model be developed. The prototyping of a subset parallels the phased design and development for a large system.

Prototyping involves the systems developers in the design stages. In the development of a large system following the traditional SDLC, the developers or programmers are often not the same personnel who design the system. Prototyping provides an opportunity for the developers to become more familiar with the system through active participation in earlier stages.

Prototyping also impacts the development stage. By the time the traditional development stage is reached, a considerable portion of the system has been produced. At this stage it is often possible to implement the model as a first phase and complete development as a second phase.

User acceptance testing is also impacted by prototyping. Through their involvement in the design and development of the system, the users will have gained significant knowledge of the system and its capabilities. The users will not be facing an unknown entity at acceptance testing. Since they will understand how the system operates, they will be able to test the operations of the system, rather than first having to familiarize themselves with its concepts. With prototyping, the users confirm the operations of a system for which they designed the user interfaces.

Software Tools

Just as there are different levels of prototyping, there are different types of software tools to address these levels. An on-line editor such as IBM's ISPF can be used to generate the input and inquiry screens. A fourth generation language such as FOCUS or MAPPER, or a prototyper can accommodate the second level prototyping requirements for simulating the operation of screens and the logic flow interaction between them.

Existing prototypers are based on one of two approaches, those that are internals driven and those that are externals driven. Internals driven prototypers require a data base and data elements to be created prior to developing screen scenarios.* The iteration process requires modifications to be made to the data base and elements while the screen formats and flows are refined.

Externals driven prototypers are dependent upon the user interface.** They do not require a data base to be established in order to develop the scenarios. The iteration process requires changes to be made only to screen formats and flows, and not to an internally supporting database or data elements.

With prototyping, the users confirm the operations of a system.

A working model or subset should be implementable as part of the final system, subject to completion of missing or incomplete components. Thus, the third level prototype should be developed in the production system's language or with a prototyper. Prototypers can be expanded from a second level simulation model to a working model and production system by adding linkages to the database using standard high level calls through user exits.

Summary

Prototyping is an iterative process in which the systems' externals, screen flow logic and application processing logic are successively developed. It allows the users to participate actively throughout design and development and refine their requirements as they become more familiar with the concepts and operation of the system. Prototyping can be used with the traditional SDLC, but blurs the distinction of the stages. When employed on a development project, this user friendly technique can positively contribute to a successful system implementation. •jsm

*RAMIS II is an information centre tool that can be used as an internals driven prototyper. The development of the users' interface follows the establishment of the data base and its relations, and the data dictionary.

**An example of an externals driven prototyper is Synerlogic's ACT/1. It enables the screens, flows, edits, error presentation and data reformatting to be developed without establishing a database or writing code.

Application Prototyping: A Life Cycle Perspective

BY BERNARD BOAR

■ It is unusual to pick up a data processing magazine without finding an article on the urgent need to improve the productivity of application development. Whether the impetus be competitive pressure, government deregulation, user demands, or the evolution from a batch to on-line (interactive) world, management is keenly sensitive to the need to not only build applications faster but to build them in sync with active operational user needs.

Inevitably, as one reviews the various suggestions and merges them with their own experience, it quickly becomes obvious that any substantive improvement in development productivity must equate to directly solving the requirements definition problem. Regardless of the benefits to be accrued by better design, improved testing, structured reviews, and so forth, their contribution will be marginal (if not irrelevant) if the business problem is not clearly understood in the first place.

Application prototyping, the building of software models, is gaining a growing and vocal following in the data processing community as a superior method to define user requirements. Prototyping is a strategy for determining requirements wherein user needs are extracted, presented, and refined by building a working model of the ultimate system — quickly and in context. It differs significantly from the traditional approaches in that once an initial (good guess) understanding of the problem is achieved an attempt is made to immediately implement that understanding. The consequential first-cut model serves as an anchor point to permit meaningful and unambiguous communication among all project participants. Exact definition of the system occurs through gradual and evolutionary discovery as opposed to omniscient foresight. It is a heuristic response to the definition problem as opposed to historical deterministic approaches. It is a "bias forection" technique that trades the conjecture of prespecification for "hands-on" experimentation.

Prototyping assumes that the risk of poor (ambiguous, unworkable, incomplete, inconsistent) requirements is best controlled and minimized by using a definition strategy that allows gradual learning and incremental change as part of the normal discovery process. It places the user in the familiar and comfortable position of critical consumer.

The need for an animated definition technique has emerged from the failure of prespecification methods to solve three outstanding problems that plague and debilitate definition:

- It is extremely difficult for users to prespecify their requirements in total and final detail.
- Graphic and narrative documentation techniques have proven inadequate to communicate the dynamics of an application and its ultimate acceptability.
- Miscommunication is common among project members. Poor communication precludes developing a successful system. Prototyping solves the fundamental communication problem by permitting (encouraging) the users to "touch" the system early in the life cycle. As a consequence, changes can be made while the proposed system is in a

highly malleable state. Users receive a system which is not only one that they want, but one that they recognize.

In spite of the common sense appeal of prototyping and the pressing need for an interactive definition methodology, data processing executives remain concerned about the apparent lack of structure to the prototyping process. Much of the literature provides an extremely simplistic view of prototyping in which:

- User and prototyper meet to discuss requirements
- The prototyper almost instantly creates a model
- After a hands-on review, the prototype is quickly revised
- The prototype is moved to production status
- Everybody lives happily ever after

Somehow, experienced data processing management knows that it's not quite that simple; especially, when getting the maximum return on prototyping means they need to apply it to their large record management applications. Additionally, they are concerned about how prototyping will integrate with their overall system development methodology and the potential problems of implementing prototypes. Though management desperately wants productivity, they desire it within an orderly and controllable context.

Prototyping offers an exceptional opportunity to solve the definition quagmire, but to be deployed successfully. It needs to be implemented within the structure of a prototype life cycle (PLC). The PLC not only addresses the needs of the users and prototypers for an orderly prototype development process but, as importantly, addresses management concerns that prototyping is not (or won't become) a cover-up for ad-hoc or chaotic development.

Prototyping offers the first workable solution to the definition problem. Debate and discussion can focus on animated and malleable models rather than passive piles of narratives and graphics. There is, however, no magic. If not deployed with foresight, prototyping can degenerate into another missed opportunity as management becomes disillusioned with an apparently uncontrollable process. The PLC offers the needed structure to make prototyping not only responsive, but controllable.

Why a Prototype Life Cycle?

Figure 1 is a common introduction to the prototyping process. Though incomplete as a complete definition strategy, it shows the placement of prototyping within the conventional system development life cycle (it's a definition technique) and the core activities which compose the prototyping process:

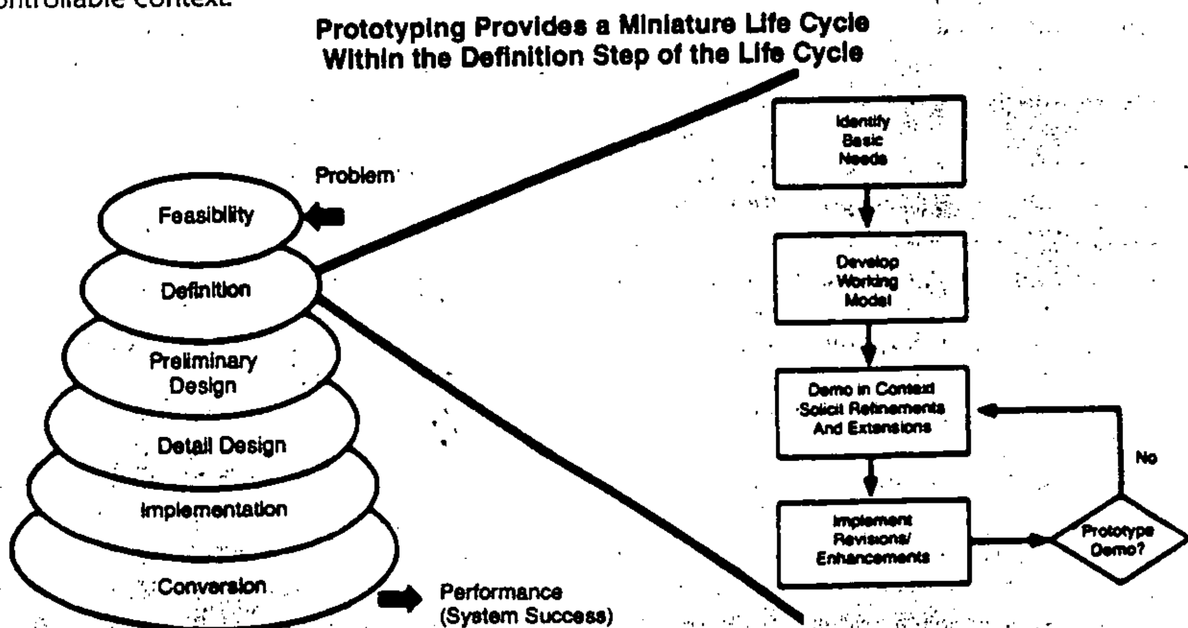


Figure 1

- *Identify Basic Needs* — Determine the basic goals, objectives, data, and functions of the application.
- *Develop Working Model* — Quickly build a first-cut model of the application.
- *Demonstrate In Context* — Demonstrate the model to all affected parties and aggressively solicit additional requirements.
- *Implement Revisions/Enhancements* — Revise the model to harmonize with the new set of user requirements.
- *Prototype Done?* — Continue the strategy of iteration (successive approximation) until demonstrated functionality has been understood and deemed satisfactory by all participants.

Many proponents of prototyping get immediately concerned at the idea of structuring the prototyping process. After all, they suggest "Won't such formality defeat the entreprenurial and dynamic attributes of prototyping and slow it down to traditional pre-specification techniques?" Though the concern is certainly legitimate, we don't wish to merely replace one protracted definition technique with another, so there appears little alternative to formalizing the prototyping process. There are four reasons for structuring the prototyping process.

1. Construction of Large Record Management or Transaction Processing Systems — To accrue maximum benefit from prototyping, it will be necessary to apply it to large transaction processing applications. For these types of applications, it is necessary to minimize risk. With such applications, accomplishing "quick build" is challenging enough without the expectation of "instant build." The PLC provides a structure to apply prototyping where it is most needed.

2. Prototyper Has Traditional Building Activities — Prototyping is a physical, not logical, modeling activity. In building a model, the prototyper has to work with all the traditional construction components; records, screens, programs, etc. It is one thing to create a working screen; it's quite another thing to create and change a working model of GO screens. The PLC provides the prototyper with orderly construction periods to build/refine the prototype.

3. Prototyping Encourages Evolution, but Evolution must be Managed — Prototyping is a license to change one's mind, and out of the dynamics of role ploying and debate a set of workable requirements is discovered. The project manager needs an orderly process through which to structure review sessions to obtain a lasting consensus. The PLC provides the project manager with clear control points to examine the prototype and reach an orderly consensus on change.

4. Prototyping Naturally Divides Into Discreet Steps — Simply put, when one observes the prototyping process, it naturally and simply divides itself into the core steps. The PLC is now the documenting of the natural process, then imposing an artificial structure on it.

In summary, the PLC provides an efficient strategy to insure delivery of a prototype with sufficient verified functionality to the representative of the ultimate system. Rather than potentially restricting the environment, the PLC provides an orderly approach through which users, prototypers, project managers, and management can all be productive and comfortable. Identified as such, organizations can put in place support activities to optimize each prototype activity. Unplanned prototyping, as "a big glob," will nevertheless reduce in practice to the same steps but without the same support of management comfort. Prototyping is not an excuse for chaos and, to the contrary, the PLC provides a highly efficient, effective, and controllable approach to model building.

Roles and Responsibilities

Prototyping is a sub-project within the overall development plan. As is the case with any other project task, successful execution requires project members to understand their roles and responsibilities. In executing the PLC, there are four key players; the prototyper, application user, project manager, and the systems analyst. In simple cases, an individual may execute multiple roles. For example, in building small prototypes the analyst and prototyper could be the same individual.

Each of the primary players have the following responsibilities:

Prototyper
- Provides prototyping expertise
- Builds/evolves prototype
- Documents prototype
- Conducts demonstrations
- Explains, not defends, model

Application User
- Provides expertise in defining operational requirements
- Verifies acceptability of prototype through role playing
- Reviews non-prototyped requirements

Project Manager
- Manages the overall prototyping process within the complete project plan
- Schedules/organizes prototype reviews
- Plans for "leveraging" of prototype
- Provides a project catalyst

Systems Analyst
- Works with users to scope/analyze/organize overall requirements
- Review models
- Documents non-prototyped requirements

Supporting these primary players can be any other organizational entity whose participation/review can further the quality/utility of the prototype. For example, in applications requiring voluminous and repetitive input, it could be advantageous to have a human factors analyst review the model for ergonomic sensitivity. For application with large databases and/or high transaction volumes, a performance/sizing analyst could evaluate the prototype for operational feasibility. In essence, as required, various organizational experts can selectively support the primary prototyping team to create a functionally correct model.

A Walk Through the Prototype Life Cycle

The purpose of this section is to provide a closer look at each step of the PLC. The PLC, as exhibited in Figure 2, provides a total approach to requirements definition, though the same core steps as illustrated in Figure 1, will be briefly reviewed.

The Complete Prototype Life Cycle Provides Actual Approach to Requirements Definition

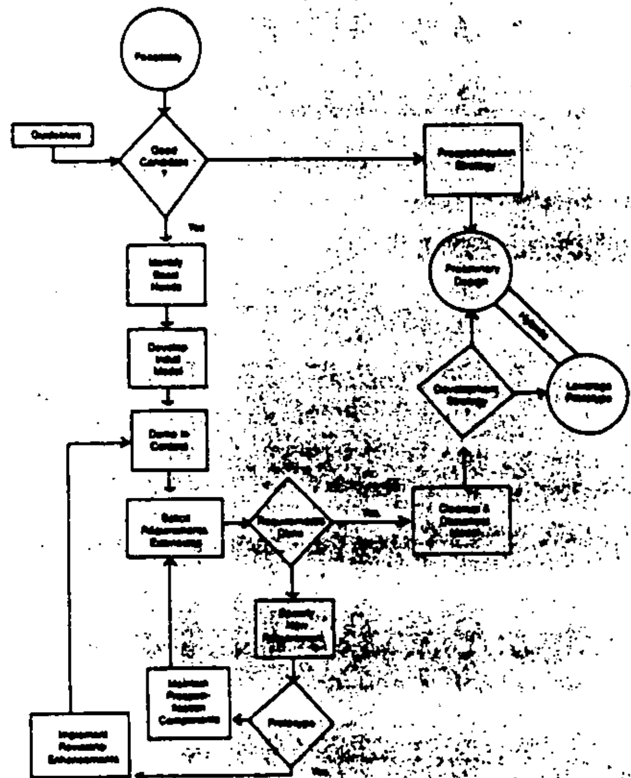


Figure 2

Step: Prototype Candidate Selection — The objective of this step is to determine which methodology is best suited as the primary definition strategy: prototyping or pre-specification. Whether prototyping is the best primary definition strategy should be decided by analyzing key determinants such as system structure, logic structure, user willingness, and requirements ambiguity. The decision should be made carefully; not dogmatically assumed. This step may be exited when a consensus decision has been reached on the suitability of prototyping as the definition strategy.

Step: Identify Basic Needs — The objective of this step is to develop a sufficient (though incomplete) understanding of the problem to enable the rapid design and construction of the initial model. Needs analysis is the first step in any definition technique. The analysis must emphasize the quick discovery of the applications basic requirements. For most applications there is a core set of data and functions that define its essence. The remainder of the application is ancillary to these core requirements and is directly derivable from them.

Needs analysis must focus on quick identification of those items that form the nucleus of the system. This step may be exited when the prototypers have sufficient understanding of the application to develop a "good guess" initial model.

Step: Develop Initial Model — The objective of this step is to build the first-cut model of the application. A working model should be constructed with sufficient depth and breadth of function and ergonomics to permit meaningful discussion and refinement to begin. The initial thrust should attempt to demonstrate coherent understanding of the whole problem. This step may be exited when the model is sufficiently complete and demonstrates sufficient functions to permit discussion and evaluations.

Step: Demonstrate In Context/Solicit Refinements and Extensions/Specify New Requirements — The objectives of this step are to develop additional requirements by having all affected parties observe, experience, and critique the model. It should be operated by all participants. It should be fully explained, but not defended. All the reviewers should be encouraged to explain their reaction to the model. The users should "role play" with the model to assure themselves that they can perform the business with it. This step may be exited when revised requirements have been generated.

Step: Maintain Prespecification Components — The objective of this step is to specify all definition deliverables that require traditional definition. All requirements which are collected, but not prototyped, must still be recorded. Prototyping does not eliminate the need to catalog all the requirements of the specific application. This step may be exited when the additional documentation has been collected.

Step: Cleanup and Document Prototype — The purpose of this step is to put the prototype into a form so that it can serve as a understandable baseline for further development. A prototype has to be documented like any other software system or it will be incomprehensive to anyone other than the original prototyper. An automated documentation capability of the prototyping software is consequently a prime software requirement. This step may be exited when documentation is complete.

The prototype life cycle provides a complete and flexible approach to animating requirements definition. It has the following crucial attributes of a disciplined methodology.

- It is comprehensive in providing all the necessary deliverables — prototyped plus pre-specified
- It can be easily tailored to the immediate characteristic of a project
- It meets the diverse needs of users, project managers, and prototypers
- It is orderly and controllable

Prototyping, as delivered through the PLC, offers state of the art technology to streamline the business of requirements definition. It's animated, responsive, and user centered; what has always been needed to obtain a verified set of requirements.

Integrating the PLC into the Structured Development Life Cycle

Most companies have adopted some form of structured development life cycle (SDLC) to provide the infrastructure of a phased and controllable approach to software development. Inevitably, as illustrated in Figure 3, the question arises "Where does prototyping fit?"

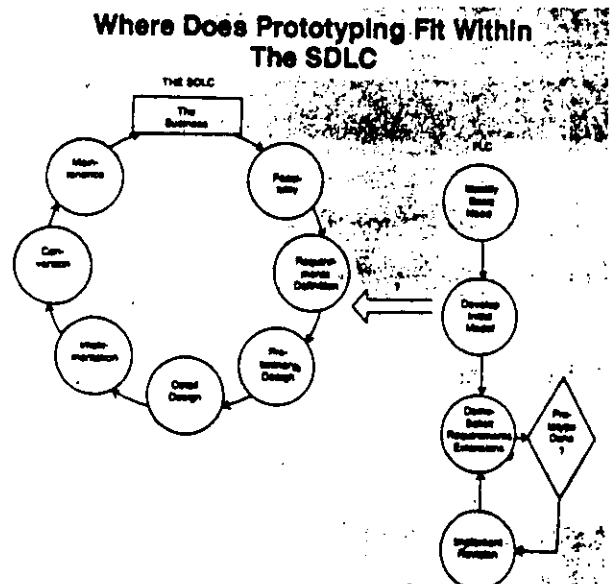


Figure 3

BERNARD H. BOAR

Mr. Boar is the author of "APPLICATION PROTOTYPING: A REQUIREMENTS DEFINITION STRATEGY FOR THE 80'S" as well as numerous articles published in Computerworld, Auerbach, Journal of Systems Management and American Management Association Perspectives.

Some advocates of prototyping would suggest that prototyping, as illustrated in Figure 4, can replace definition, design steps, and implementation. Though certainly understandable, such ambition would seem infeasible. Unfortunately, there is too much missing at the end of the PLC to directly implement the prototype. At minimum, critical production system features such as complete functionality, quality assurance testing, operational run books, application performance ad sizing, training materials, and preprinted forms are all missing. The PLC does not permit magical "lifecycle by-pass."

As illustrated in Figure 5, the PLC can replace traditional requirements definition. At one extreme, the prototype can be viewed as a pure requirements document that happens to execute. Preliminary design would serve to now build the application using the prototype as the specification. At the other extreme, the prototype can be viewed as a partially complete final application that requires optimization and function completion before production execution. In essence, where the prototyping technology equates to the production technology, the completed prototype can be highly leveraged into a production system.

Prototyping Replaces Definition, Design and Implementation Steps

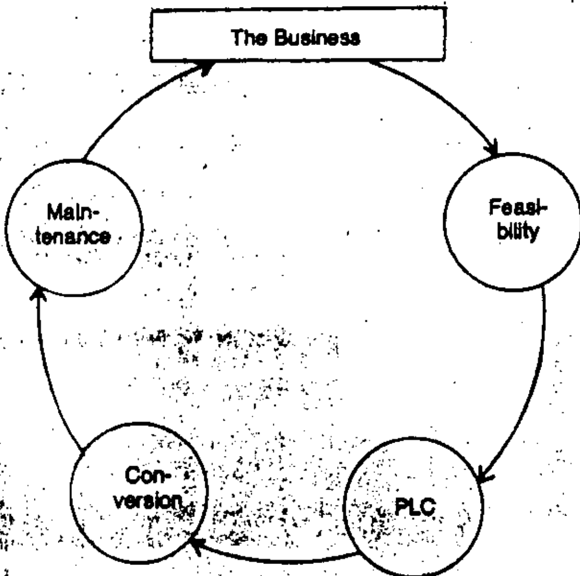


Figure 4

Prototyping Integrates Within the SDLC as the Definition Step

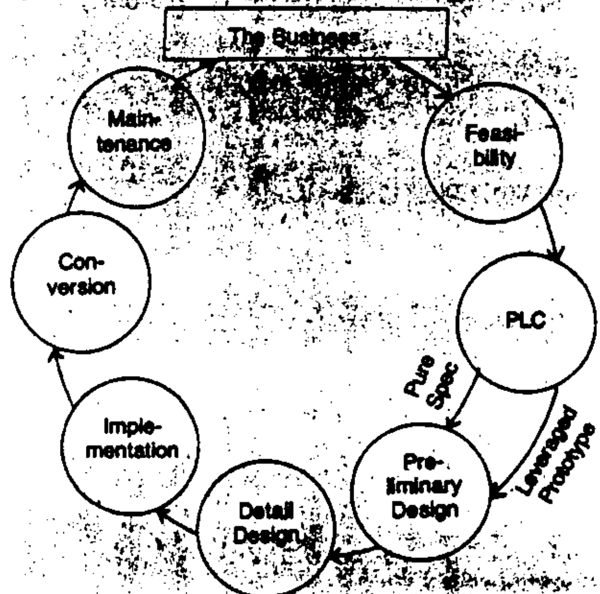


Figure 5

The prototype can provide numerous productivity benefits to the SDLC beyond being a pure specification or leveragable model. The prototype can improve productivity by:

- serving as an ongoing education vehicle for new project members
- providing excellent input to performance/sizing models to assure operational feasibility
- serve as a training aid for the training organization and users while the real system is being built
- becoming part of a "Systems-by-Example" portfolio of applications used to educate new users about alternative possible system functionality
- contributing parts to an on-going "software parts" department

Prototyping not only integrates well within the overall SDLC, but when analyzed offers potential benefits above and beyond requirements.

Summary

The sad truth is that, in spite of all the rhetoric to the contrary, requirements definition remains a most imperfect task. Though not intended as such, most applications at conversion demonstrate all the imperfections of an initial model. If productivity is to be improved, first and foremost, the requirements problem must be solved.

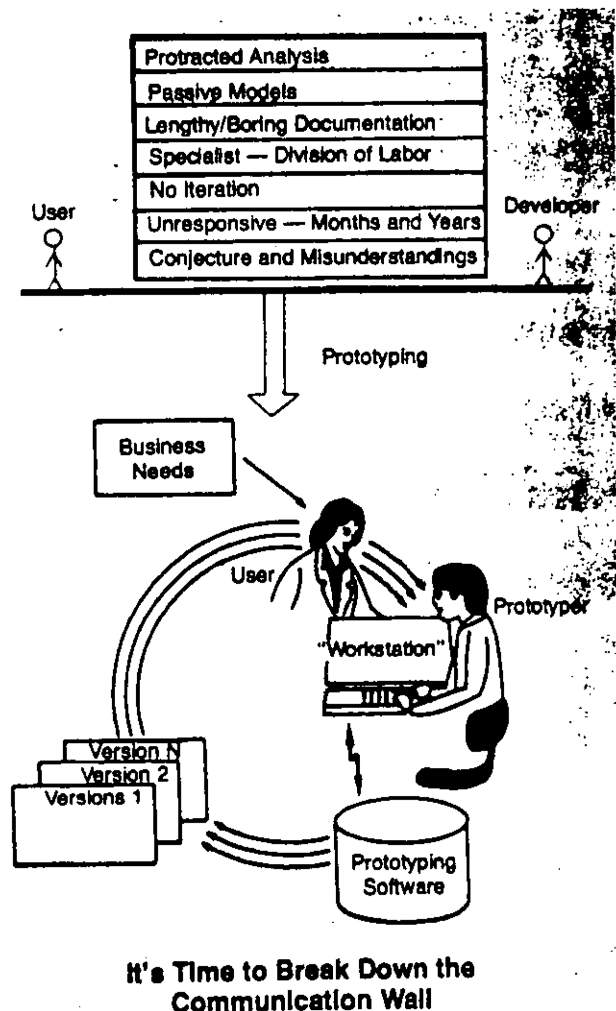
Prototyping's key to success is that it focuses on the user. For the first time, users can actively and meaningfully participate in definition because they clearly understand what they will receive. Prototyping works because it fits naturally the way people naturally decide on what they want.

The strategy is simple: invest neither substantial human nor machine resources until a working model has been examined, operated, and accepted by the user. "Plan to throw one away, you will anyhow."

Though there are many tactics that contribute to the success of prototyping, user involvement, integrated prototyping software, virtuoso prototypers, and proactive project management, clearly the PLC is also a major success factor. The PLC:

- provides an orderly and controllable strategy for performing prototyping
- formalizes roles and responsibilities
- trades passive communication for interactive experience
- anchors pre-specification components by the model
- meets the needs of users, prototypers, project managers, and management
- provides an efficient and effective framework for iteration
- evolves the existing investment in the SDLC
- provides productivity benefits beyond definition itself

The solution to the communication problem was never to make everyone a professional specifier but to permit everyone to review specifications in a familiar medium. It's time to breakdown the communications wall (Fig. 6).



It's Time to Break Down the Communication Wall

Figure 6

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS INSTITUCIONALES

No. 70

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

DEL 9 AL 29 DE SEPTIEMBRE

SEPOMEX

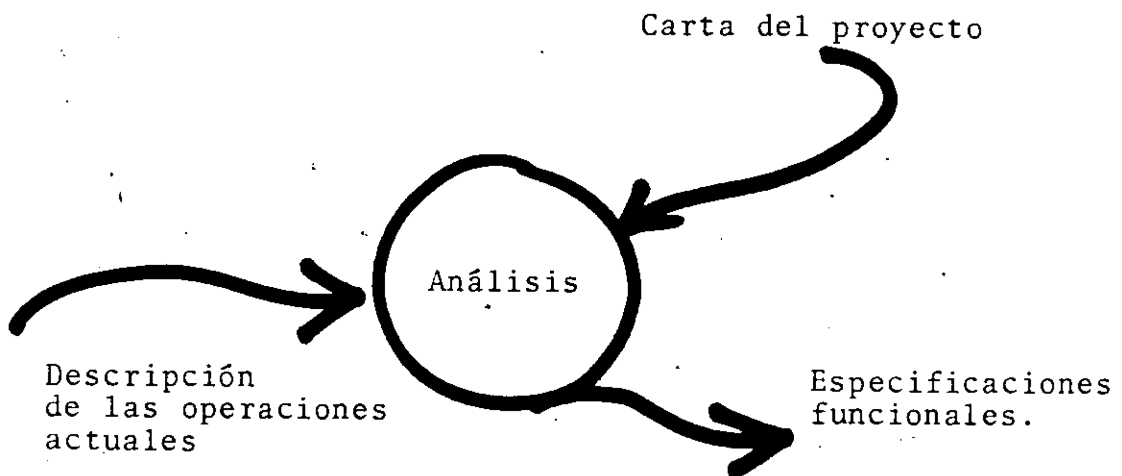
ANALISIS ESTRUCTURADO

MEXICO, D.F.
PALACIO DE MINERIA
1992

Análisis estructurado

Cuando Ed Yourdon habló de análisis estructurado, la idea fue largamente especulada. El no tenía resultados actuales sobre proyectos completos para reportarlo. Sus ideas estaban basadas de la simple observación de algunos principios de particiones de arriba hacia abajo (top-down) usadas por los diseñadores y que se podían aplicar en la fase de análisis. Desde este tiempo ha habido una revolución en la metodología del análisis.

¿Qué es el análisis?



Análisis es el proceso de transformar una cadena de información acerca de las operaciones corrientes o actuales y de los nuevos requerimientos a una descripción ordenada y rigurosa de un sistema para ser construido. Esta descripción es también llamada especificaciones funcionales o especificación del sistema.

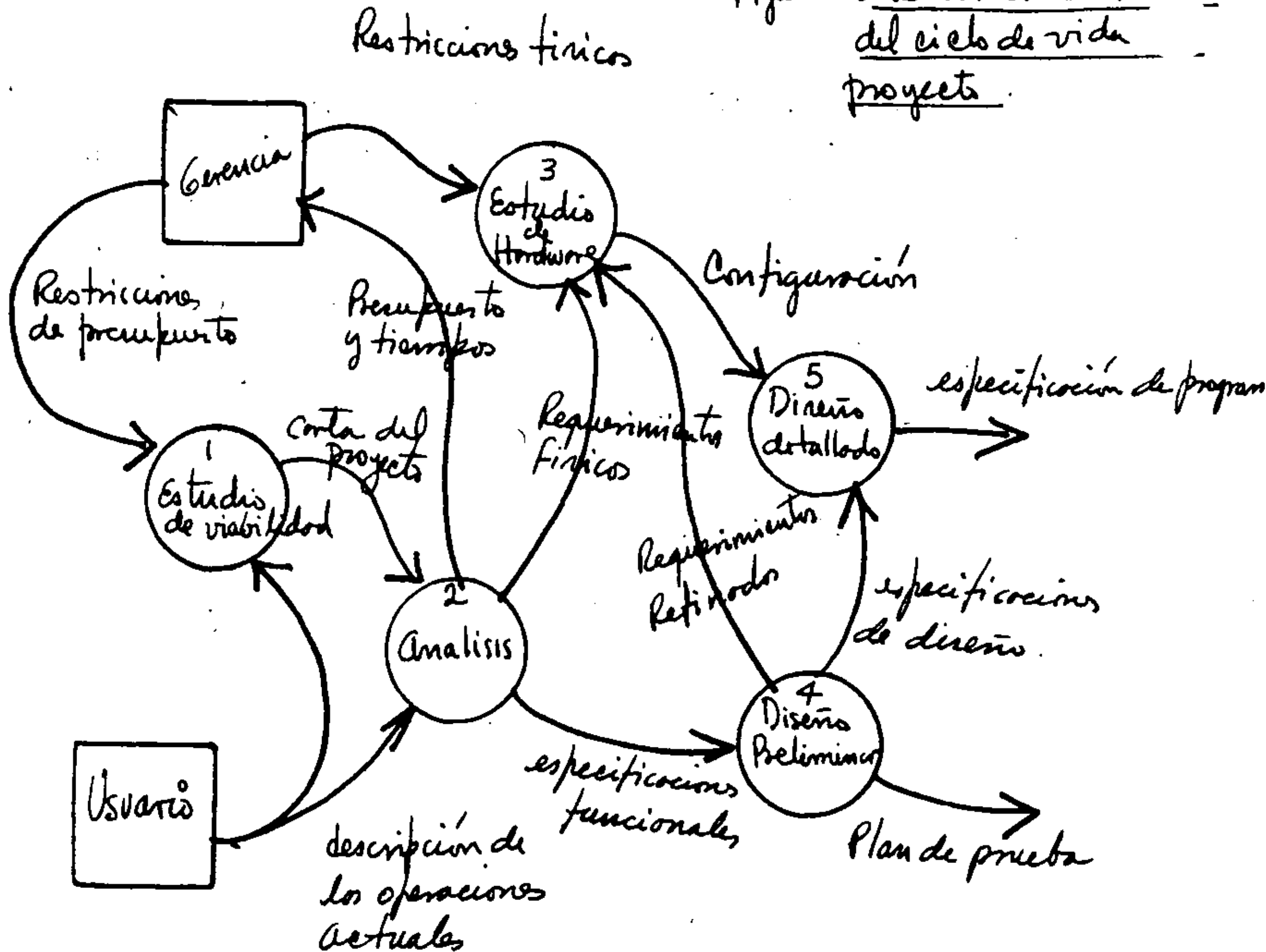


Figura: Analisis en el contexto del ciclo de vida proyecto

En el contexto de el ciclo de vida del proyecto para el desarrollo de sistemas, el análisis se encuentra muy al principio, solamente precedido por el estudio de viabilidad durante el cual la carta del proyecto es generada. La carta de proyecto contiene consideraciones que gobiernan el desarrollo, cambios que deben ser considerados etc. La fase de análisis concierne principalemente con la generación de especificaciones de el sistema para ser construído.

Especificamente la tarea esta compuesta de las siguientes actividades:

- interacción con el usuario
- estudio del medio ambiente actual
- negociación
- diseño externo del nuevo sistema
- diseño de formatos de E/S

. estudio de costo beneficio
. escritura de las especificaciones y
. estimación

¿Que es el análisis estructurado?

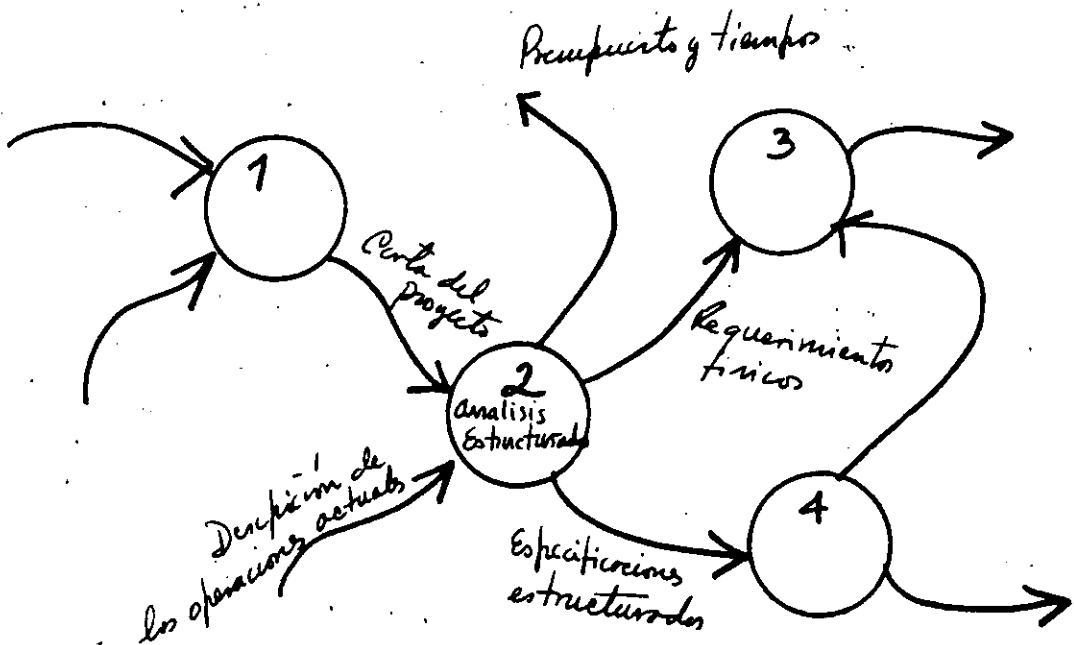


Figura:
analisis estructurado en el contexto de ciclo de vida del proyecto.

El análisis estructurado es una disciplina moderna para conducir la fase de análisis. En el contexto de el -- ciclo de vida del proyecto su única diferencia aparente es un nuevo producto llamado "especificaciones estruc-- turadas". Esta nueva clase de especificación tiene las

siguientes características:

- . es gráfica, compuesta mayormente de diagramas
- . es particionada, no es una sola especificación sino una red conectada de "mini especificaciones".
- . es de arriba-abajo (top-down), presentada en modo -- jerarquico y progresivamente de los niveles superiores mas abstractos hasta los niveles inferiores mas detallados.
- . es mantenible, una especificación puede ser actualizada para reflejar cambios en los requerimientos.
- . es un modelo en papel del sistema, el usuario puede trabajar con el para perfeccionar su visión de las operaciones tal y como seran en el nuevo sistema.

En la siguiente figura se representa en un solo proceso el análisis estructurado. Veamos en detalle el proceso.

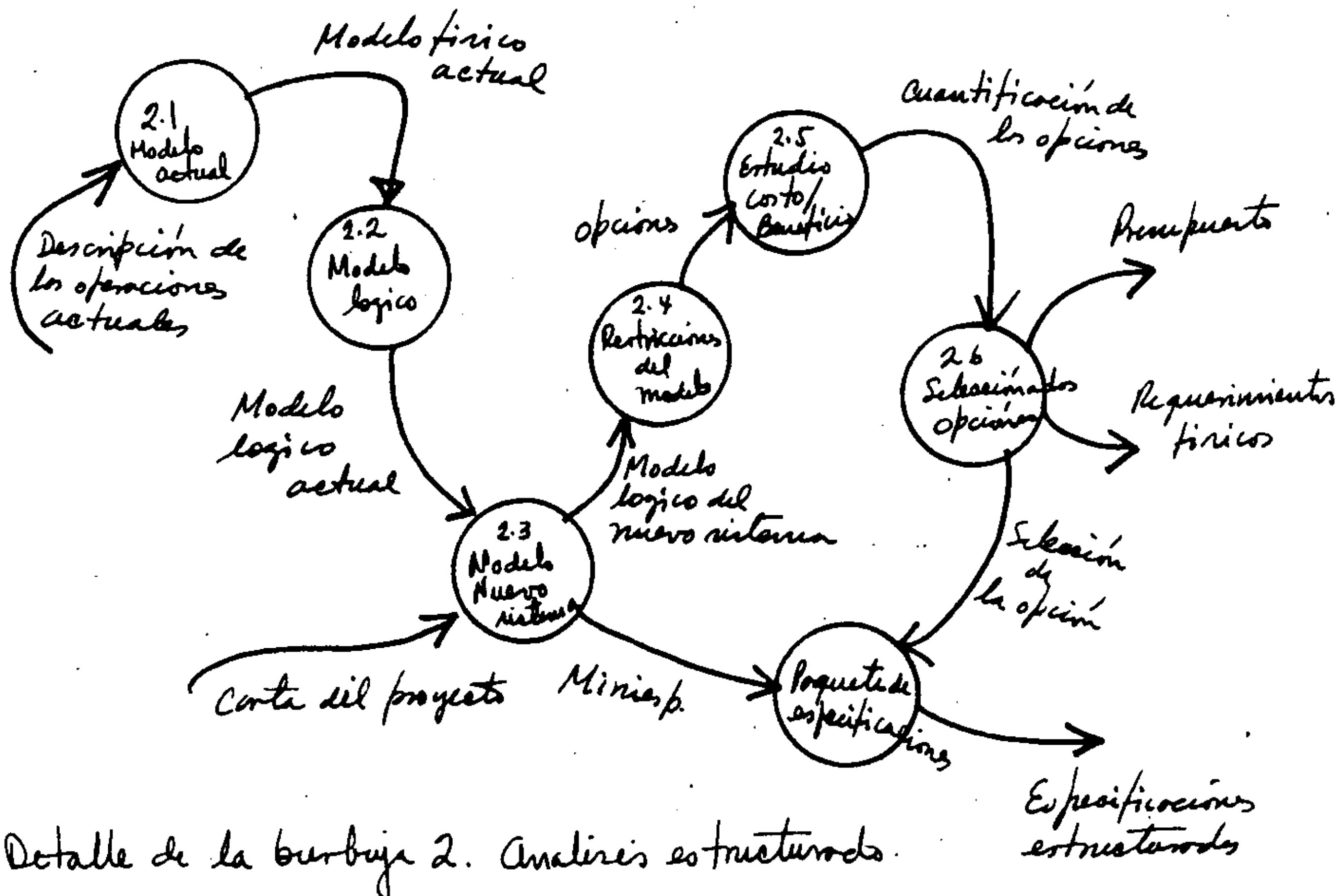


Figura: Detalle de la burbuja 2. Analisis estructurado

Note que la figura tiene las mismas entradas y salidas que la burbuja 2 de la figura anterior. Pero muestra las transformaciones en considerable mas detalle. Consideran los autores que esta figura da mas información que cientos de palabras. Bastará una definición de los componentes y del flujo de información para terminar de comprender el proceso.

Proceso 2.1. Modelo del sistema actual: Hay casi siempre un sistema actual. El análisis estructurado nos tendrá que construir un modelo "fisico" en papel del sistema actual y usarlo para perfeccionar el entendimiento del medio ambiente actual. La justificación para la naturaleza física de este primer modelo es que su propósito es ser una representación verificable de las operaciones actuales y deberán ser fácilmente entendibles por el grupo de desarrollo.

Proceso 2.2. Derivar un equivalente lógico.- El equivalente lógico del modelo físico es uno que esta divorciado de los "comos" de las operaciones actuales y en su lugar se concentra en los "que". En lugar de una descripción de la forma como se lleva a cabo la política se hace una descripción de la política en sí misma.

Proceso 2.3. Modelo del nuevo sistema.- Aquí es donde el trabajo mas importante de la fase de análisis se realiza, la invención del nuevo sistema. La carta de proyecto que ha recogido las diferencias y las diferencias potenciales entre el medio ambiente actual y el nuevo y el modelo lógico del sistema existente son los elementos del análista para construir el nuevo modelo junto con la documentación del futuro medio ambiente. El nuevo modelo presenta al sistema como un grupo particionado de procesos elementales. El detalle de estos procesos son ahora especificados utilizando una "miniespecificación" por proceso.

Proceso 2.4. Restricciones del modelo. El nuevo modelo es demasiado lógico para nuestros propósitos debido a que no se ha establecido cuanto de lo declarado se hace dentro y cuanto fuera de la máquina. Es en este punto donde el analista establece los límites entre el hombre y la máquina y se limita el alcance de la automatización.

Típicamente estos se hace mas de una vez con el objeto de dar varias alternativas para la selección de procesos.

Proceso 2.5 Medidas de costo y beneficio.- El estudio de costo-beneficio es realizado para cada una de las opciones. Cada uno de los modelos tentativos junto con sus parámetros asociados de costo-beneficio son presentados en forma cuantificada.

Proceso 2.6. Selección de la opción.- Las opciones cuantificadas son analizadas y una de ellas es seleccionada como la mejor.

Proceso 2.7. Presentación de especificaciones.- Ahora todos los elementos de la especificación estructurada son armados y presentados. El resultado es: el nuevo modelo -- físico seleccionado, el conjunto integrado de miniespecificaciones y posiblemente agunas tablas de contenido, ---- resumenes, etc.

¿Qué es el modelo?

El modelo al que se ha referido para la representación del sistema es un modelo escrito. En la convención de Análisis estructurado este modelo se logra con el uso de un - - "diagrama de flujo de datos" y de un "diccionario de da-- tos".

1.- Diagrama de flujo de datos.- Es una representación en forma de red de un sistema. Representa al sistema en términos de los componentes de los procesos y declara todas las interfases entre los componentes.

2.- Diccionario de datos.- Es un conjunto de definiciones de las interfases declarados en el diagrama de flujo de datos. Se define cada una de esas interfases en términos de sus componentes.

El modelo del sistema tiene diferentes usos en el análisis estructurado.

1.- Es una herramienta de comunicación.- Los analistas y los usuarios tienen fallas en la comunicación, el modelo es esencial para las discusiones y que estas puedan conducir a un entendimiento común. Se le considera al modelo como la mas importante ayuda para la comunicación.

2.- Es el marco de referencia para las especificaciones. El modelo declara las piezas que componen el sistema y las partes de esas piezas, hasta esas que ya no puede ser subdivididas. Los niveles básicos, son acompañados de una miniespecificación para completar su especificación.

3.- Es el punto de inicio para el diseño. Debido a que el modelo es el mas elocuente elemento que dió pie a los requerimientos tendrá una gran influencia en el trabajo que se haga en la fase de diseño.

Algunas fallas del análisis clásico según Yourdon.

Las principales fallas del análisis clásico los encontramos en el proceso de especificación del sistema. Algunos de estos son:

1.- Las especificaciones funcionales en el análisis clásico son generalmente una narrativa en un documento solamente leible en forma serial del principio al fin.

2.- Las especificaciones funcionales son imposibles de actualizar.

3.- Debido a que las especificaciones funcionales no son entendibles por si mismas no podemos mostrarle nada al usuario sino hasta el final del análisis, esto no permite la interacción que podría servir para refinar y perfeccionar el producto.

Como ayuda el análisis estructurado según Yourdon.

El análisis estructurado ayuda a resolver los problemas de la fase de análisis ya que:

1.- Ataca el problema del tamaño por particiones
2.- Ataca el problema de la comunicación en forma interactiva y con una inversión de los puntos de vista.
3.- Ataca el problema del mantenimiento de especificaciones por redundancia limitada.

El concepto de particionar o descomponer funcionalmente puede ser familiar para los diseñadores como el primer paso para crear un diseño estructurado. Su potencial valor en el análisis fue evidente desde el principio, sistemas grandes no podían ser analizados sin alguna forma de particionar concurrentemente. La directa aplicación de las herramientas del diseño como HIPO causaron mas problemas en lugar de resolverlos. Despues de algunos experimentos el análisis estructurado fue apoyado con una nueva herramienta llamada diagrama de flujo de datos.

Sus ventajas son las siguientes:

1.- La apariencia del diagrama de flujo de datos no es para todos espantosa. Parece ser una simple fotografía del elemento que esta en discusión. Nunca se tendrá que explicar una convensión arbitraria, nunca se explicará una cosa a todos. Simplemente se usan diagramas.

2.- Los modelos tipo red como los que se construyen para el diagrama de flujo de datos puede resultar familiar para algunos usuarios. Pueden los usuarios tener nombres diferentes para este tipo de herramientas pero el concepto es similar.

3.- El hecho de particionar con el diagrama de flujo de datos llama la atención de las interfases que resultan del proceso de particionar. La complejidad de los interfases es un indicador importante del esfuerzo que se haga por particionar.

El término "comunicación iterativa" esta usado para describir el intercambio de información en una doble dirección es una de las características de las sesiones mas productivos de trabajo. El periodo de dialogo entre el analista y los usuarios debe reducirse y procurar realimentación. Un entendimiento temprano es siempre imperfecto. También se menciona como parte del análisis la "inversión de puntos de vista". Las especificaciones clásicas describen que hace la computadora, en que orden lo hace, usando términos que son releventas para la computadora y para las gentes de la computadora. Frecuentemente se limita también a la discusión de los procesos internos de la máquina y la transferencia de datos y casi nunca se especifica los casos que suceden fuera de los límites entre el hombre y la máquina.

Desde el punto de vista de la máquina es natural y útil para el grupo de desarrollo pero para los usuarios y su grupo les concierne lo que sucede fuera de la máquina. - El análisis estructurado adopta un punto de vista dife-- rente, el diagrama de flujo de datos sigue las trayecto- rias de los datos por donde ellos pasen, ya sean procesos manuales o automáticos.

El uso de la "redundancia limitada". Uno de los aspec tos mas importantes de esta profesión y que representa - un cuello de botella es el congelamiento de las especifi caciones. No aceptar cambios en las especificaciones porque estas no pueden ser actualizadas o aceptarlas, pe ro sin actualizar las especificaciones resulta un grave peligro para el desarrollo del sistema.

El concepto llave que el análisis estructurado propone - para especificar los requerimientos es tener poca o nada de redundancia para obtener especificaciones considera-- blemente mas consistentes.

¿Qué es una especificación estructurada?

El análisis estructurado se involucra en la construcción de una nueva clase de especificación, una especificación estructurada hecha a base de diagramas de flujo de da-- tos, diccionario de datos y miniespecificaciones.

1.- Diagrama de flujo de datos (DFD) sirve para particio nar el sistema. El sistema que es tratado con un diagrama de flujo de datos puede incluir partes ma-- nuales y automatizadas. El propósito del DFD es - declarar los componentes de los procesos que integran el sistema y las interfases entre los componentes.

2.- Diccionario de datos.- Define las interfases que -- fueron declarados en el DFD. Esto se hace con una notación convencional que permita representar los -- flujos de datos y guardarlos en términos de sus componentes.

3.- Las miniespecificaciones definen un proceso elemental declarado en el DFD. Un proceso es considerado ele mental o primitivo cuando no puede ya ser subdividido en niveles mas bajos. Antes de que la especificación estructurada este completa deberán estar todos los procesos primitivos acompañados de una miniespecificación. El método usado para escribir miniespecificaciones utiliza: Ingles estructurado, tablas - de decisión, árboles de decisión, etc.

Resumen

Los fundamentos del análisis estructurado no son nuevos. Muchas de las ideas han sido ya usadas por años. ¿Que_ es lo nuevo de esta disciplina emergente? Podemos decir que intenta sintematizar el proceso de análisis para especificar requerimientos.

CONSTRUCCION DEL DIAGRAMA DE FLUJO DE DATOS

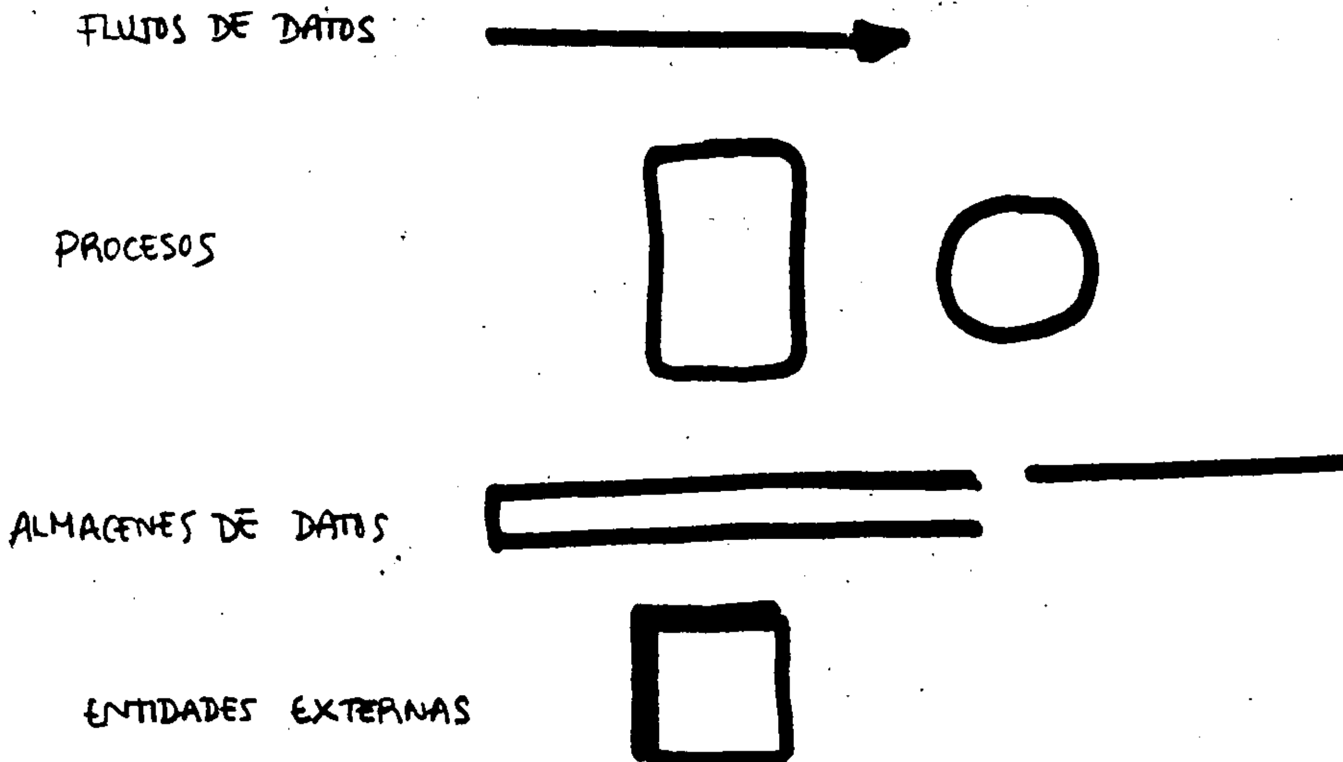
Definición

Un diagrama de flujo de datos (DFD) es una representación en forma de red de un sistema. El DFD es una herramienta para modelar un sistema que puede ser automatizado, manual o mixto. El DFD presenta al sistema en términos de sus piezas componentes con todas las interfases entre los componentes indicados.

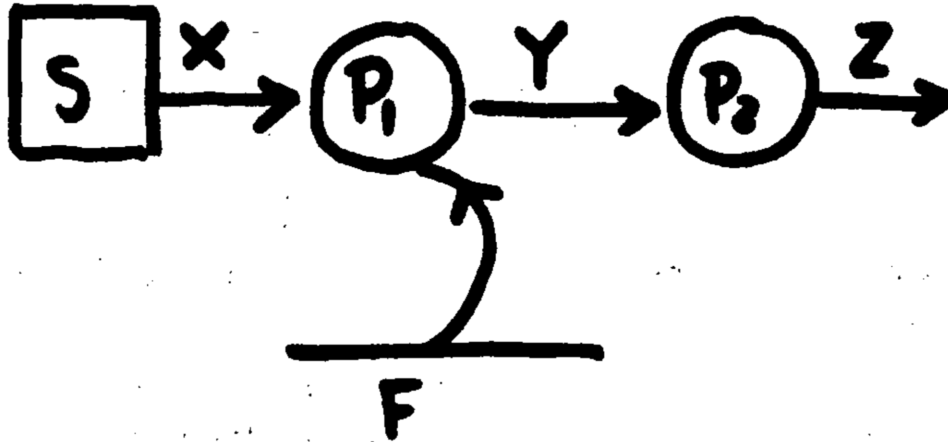
Elementos del DFD

Los diagramas de flujos de datos se construyen con cuatro elementos básicos:

1.- Flujo de datos - representados por vectores
2.- Procesos - representados por círculos o rectángulos redondeados.
3.- Archivos - representados por líneas rectas o rectángulos abiertos.
4.- Datos fuentes o destino - representados por cajas o cuadros.



Ejemplo

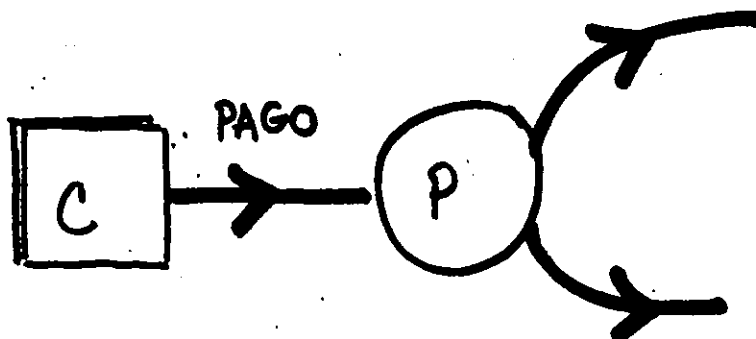


El diagrama de flujo de datos se leería:

X llega de la fuente S y es transformada a Y por el proceso P_1 el cual requiere acceso al archivo F para hacer su trabajo. - Y es posteriormente transformada a Z por el proceso P_2 .

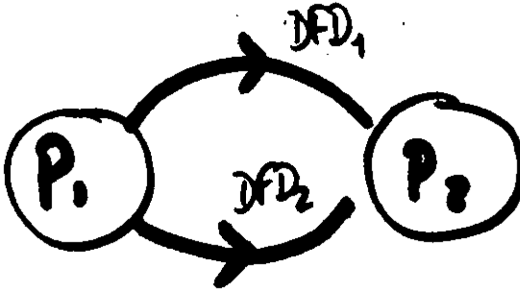
1.- Flujo de datos

El flujo de datos → representa a una interfase entre los -- componentes del diagrama de flujo de datos. Estos, se encuentran entre procesos y van o vienen de archivos o de fuentes y destinos.



El flujo de datos → es una línea a través de la cual, fluye un paquete de información conocida. El pago (flujo de datos del ejemplo anterior) puede consistir del cheque y de la copia de la factura y no por esto, se colocan 2 líneas entre la fuente y el proceso.

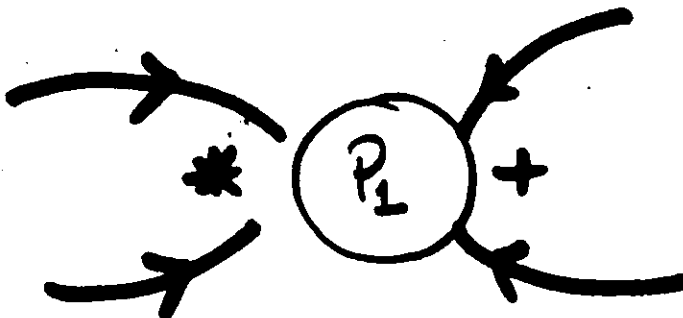
En algunos casos si es posible encontrar que de un proceso a otro existan 2 líneas, pero estas llevan información que juntas, no constituirán un paquete. (El tiempo podría ser una causa)



Convenciones para asignar un nombre al flujo de datos:

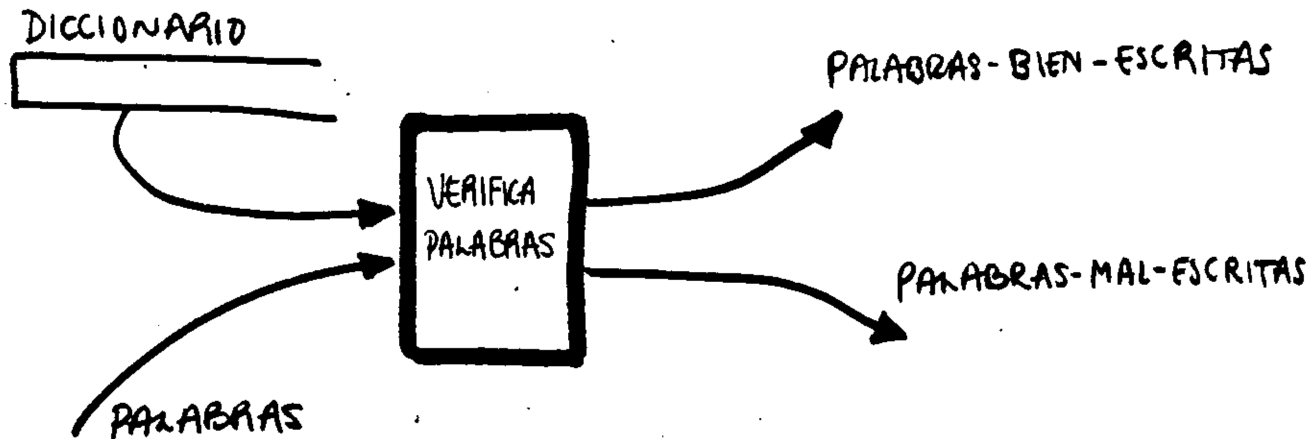
- Los nombres estan ligados con guiones y se usan letras mayúsculas NUMERO-DE-CUENTA
- No hay dos flujos de datos con el mismo nombre
- Los nombres no solamente deben representar los datos que se muevan en el flujo de datos, sí sabemos más de ellos pueden aparecer en el nombre.
- La misma línea de flujo de datos puede llegar a 2 procesos o dos salidas pueden juntarse en una sola línea.
- Las líneas que salen o llegan a archivos no requieren nombre.

En algunos casos, es necesario indicar que dos flujos de datos, deben presentarse al proceso o que uno o el otro pero no ambos.



2.- El Proceso

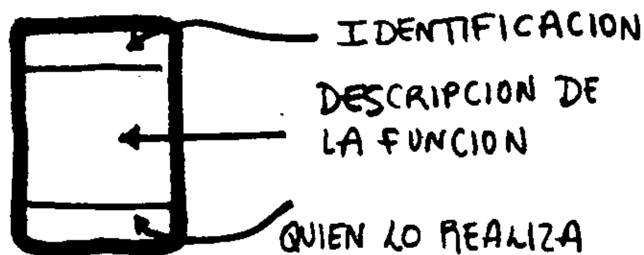
El proceso invariablemente muestra alguna cantidad de trabajo - realizado sobre los datos.



Aquí se muestra un proceso cuya tarea es dividir el flujo de entrada de datos en 2 flujos de datos de salida (Palabras bien -- escritas - Palabras mal escritas).

Un proceso es una tranformación del flujo de datos de entrada a un flujo de datos de salida. Cada proceso, debe tener un nombre.

Convenciones para referenciar el proceso:



Identificación: es un número cuya función es identificar al -- proceso

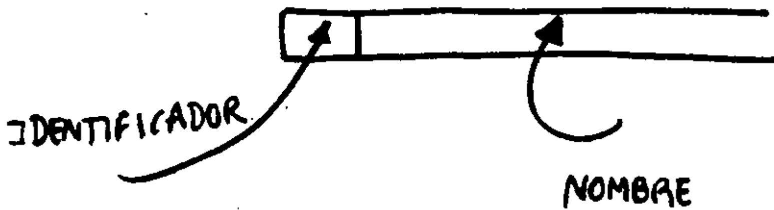
Descripción de la función: es una oración corta que describe al proceso.

Quien lo realiza: Es el nombre de un departamento, persona, programa, etc., que realiza la función.

3.- El archivo

Para los propósitos del DFD el archivo es considerado como un depósito temporal de datos.

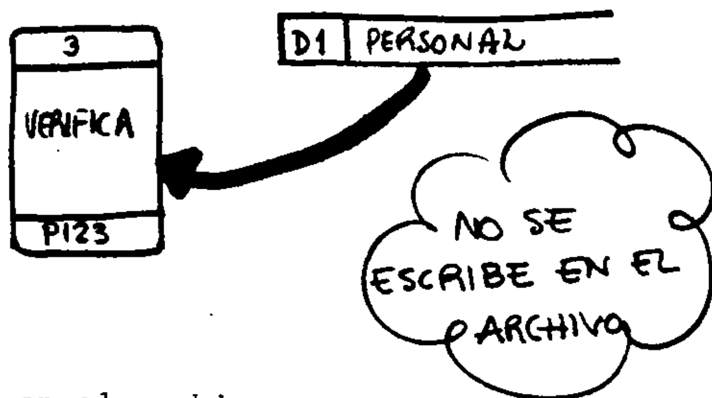
Convenciones:



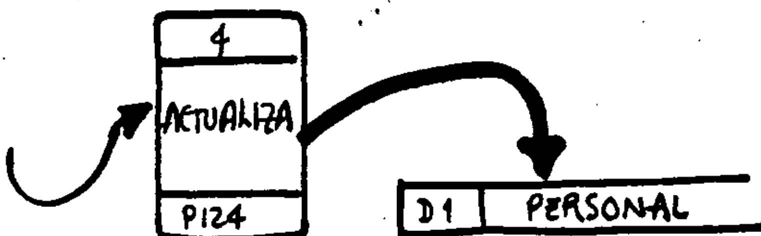
Identificador: Para facilitar su referencia y se utiliza la letra D seguida de un número.

Nombre: Un nombre lo mas descriptivo que se pueda.

El flujo de las flechas que salen o llegan de o hacia los procesos es significante por ejemplo:



Este proceso escribe en el archivo



¿Cómo actualiza sin leer?
Es cierto, pero debe en el diagrama solo indicarse el propósito fundamental que en este caso es hacia afuera.

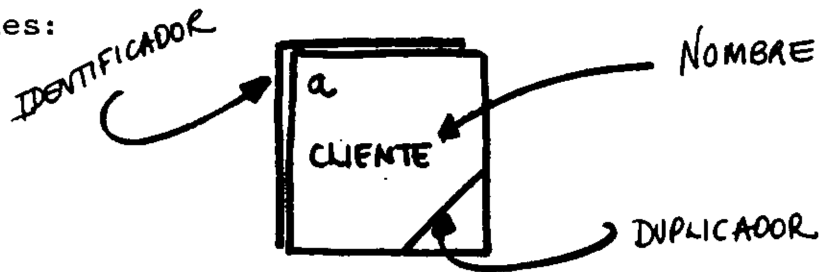
este proceso solo escribe el archivo.

(Si es necesario leer y escribir la flecha puede ser ↕)

4.- La fuente o el destino

La fuente o el destino es una persona o una organización situado fuera del contexto del sistema, es el origen o el receptor de los datos del sistema.

Convenciones:



Identificador: letras minúsculas asociadas a las entidades externas

Nombre: nombre de la entidad

Duplicador: indica que el símbolo se repite en el diagrama.

Diagramas de flujo de datos jerárquicos.

Este concepto se refiere a explotar un diagrama en la modalidad de TOP-DOWN. Estableciendo una relación padre-hijo por cada - proceso de explosión. Por ejemplo:

Diagrama 0

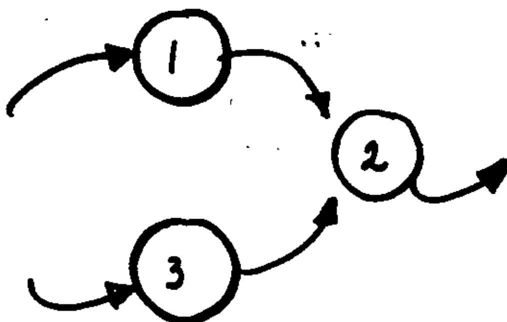
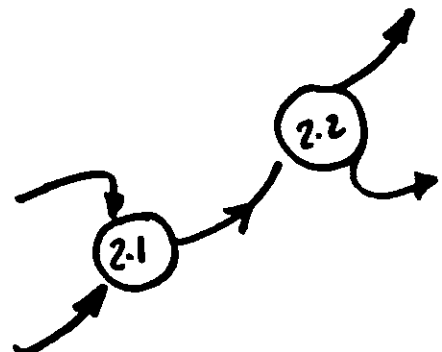
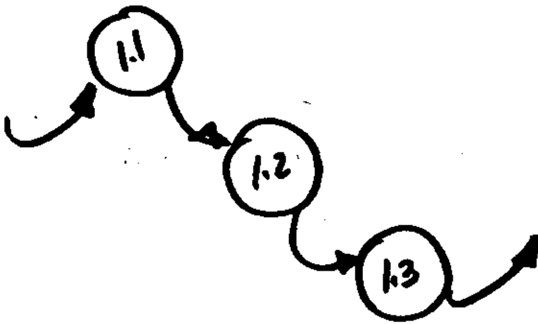
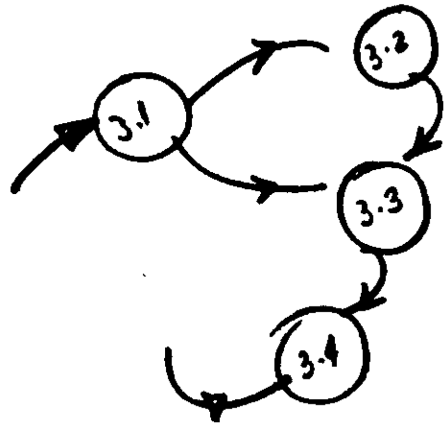


Diagrama 2







Diagramas de flujos de datos balanceados

Es un concepto en el que al hacer alguna explosión la entrada y la salida en el nivel inferior se mantienen

Por ejemplo

El diagrama 2 es desbalanceado y los diagramas 1 y 3 son balanceados.

Guía para la elaboración del diagrama de flujo de datos.

1) Identificar las entidades externas involucradas. Hay que recordar que los flujos de datos, son creados cuando un evento se desarrolla fuera de los límites del sistema:

   Una persona decide comprar alguna cosa
   Un camión llega a la terminal.

2) Identificar las entradas y salidas. Si la persona decide -- comprar algo las entradas pueden ser: Una orden de compra - por teléfono, una carta o pedido por correo, presentarse personalmente a comprar algo, etc. Se sugiere hacer una lista de las entradas y las salidas.

3) Identificar las preguntas y los requerimientos de información que puedan presentarse. Hay que especificar un flujo de datos que define la información dada al sistema y otro para indicar que es lo requerido.

4) Hacer un diagrama colocando las entidades externas y el flujo de datos que se origina de cada uno de ellos, los procesos y los almacenes necesarios.

   En este punto, no hay que poner mucho énfasis en las consideraciones de tiempo, excepto en la procedencia natural de los eventos. Hay que dibujar un diagrama que nunca inicia y nunca termina. No hay que poner decisiones.

5) El primer diagrama puede ser hecho a mano libre

6) Es posible que se requiera al menos 3 variaciones del dibujo inicial. Ya que el primero a lo mejor no es claro.

7) Cuando se tiene el primer diagrama, verificar que todas las - entradas y salidas se han incluido. Excepto aquello que resultan salidas de error o excepciones.

8) Ahora producir un nuevo diagrama:
   - Tratando de minimizar el número de cruces de líneas de flujo.
   - Duplicar entidades externas si es necesario
   - Duplicar archivos si es necesario

9) Validar el diagrama con alguno de los usuarios, mostrarle el diagrama y pasearlo por el y explicarle que es solo una aproximación y si tiene alguna observación que la haga.

10) Hacer una explosión de cada uno de los procesos incorporando errores y salidas de excepción.
Es posible que esto pueda hacer que se modifique el diagrama del nivel anterior.

11) Si es posible, construir el diagrama final en una hoja de -- 36 x 48 pulgadas. Que servirá como ayuda invaluable en la presentación a los usuarios, diseñadores, revisores y a todos los que tengan que ver con el sistema.

Ejemplo

Consideremos el caso de automatizar el proceso de enseñanza-aprendizaje tomando como punto de partida el modelo de Anderson y Faust. Este modelo sugiere la aplicación de las siguientes etapas:

1) Especificación de objetivos
Los objetivos son entregados al profesor por el comité de carrera como la entrada principal al sistema. El profesor los revisa y establece en definitiva cuales serán los objetivos que guiarán su proceso de E.A.

2) Elaboración de instrumentos de Medición
Tomando a sus objetivos el profesor elabora los instrumentos de medición (cuestionarios, prácticas, elaboración de programas, exámenes, etc.) y establece las siguientes clases:

- Evaluación diagnóstica
- Evaluación formativa
- Evaluación Sumaria

La evaluación diagnóstica es la que le permite saber el estado inicial del alumno, la formativa es complementaria a instrucción y la sumaria le permite establecer en que grado se lograron los objetivos.

3) Aplicación de la evaluación diagnóstica.

Antes de iniciar la instrucción el maestro aplica la evaluación diagnóstica al alumno para conocer el estado inicial, de este resultado el maestro puede optar por lo siguiente:

- No se le instruye porque ya conoce el material
- No se le instruye porque no tiene los antecedentes
- Se le instruye

4) Planeación de la instrucción

El maestro de acuerdo a sus objetivos selecciona y/o elabora el método de instrucción y las técnicas de enseñanza, la selecció de las actividades de aprendizaje y la selección y/o elaboración de los materiales y medios educativos.

5) Aplicación de la instrucción

En este punto, el maestro realiza el acto de enseñar a sus alumnos.

6) Aplicación de la evaluación sumaria

Cuando se ha concluído con la instrucción el maestro aplica una evaluación sumaria, la cual le permitirá establecer el grado con el que se lograron los objetivos. Ahora bien si se lograron los objetivos podemos dar por terminados el proceso. Si no se lograron los objetivos hay que investigar las causas en las etapas del modelo.

7) Administración del proceso de E.A.

La administración del proceso de E.A incluye una fase de inscripción, de seguimiento y de emisión de una calificación para el alumno. Otra, en la que se inscribe, se dan antecedentes y consecuentes, temarios, guías de estudios de los cursos.

PROCESO DE CONSTRUCCION DEL DFD.

1) Identificar las entidades externas involucradas.

COMITE DE CARRERA
ALUMNO

2) Identificar entradas y salidas

ENTRADAS: COMITE DE CARRERA

- OBJETIVOS DE APRENDIZAJE
- DATOS DEL CURSO

: ALUMNO

- DATOS DEL ALUMNO
- CONTESTACION DIAGNOSTICA
- CONTESTACION FORMATIVA
- CONTESTACION SUMARIA

SALIDAS: COMITE DE CARRERA

- RESULTADOS SUMARIOS

: ALUMNO

- RESULTADOS SUMARIOS

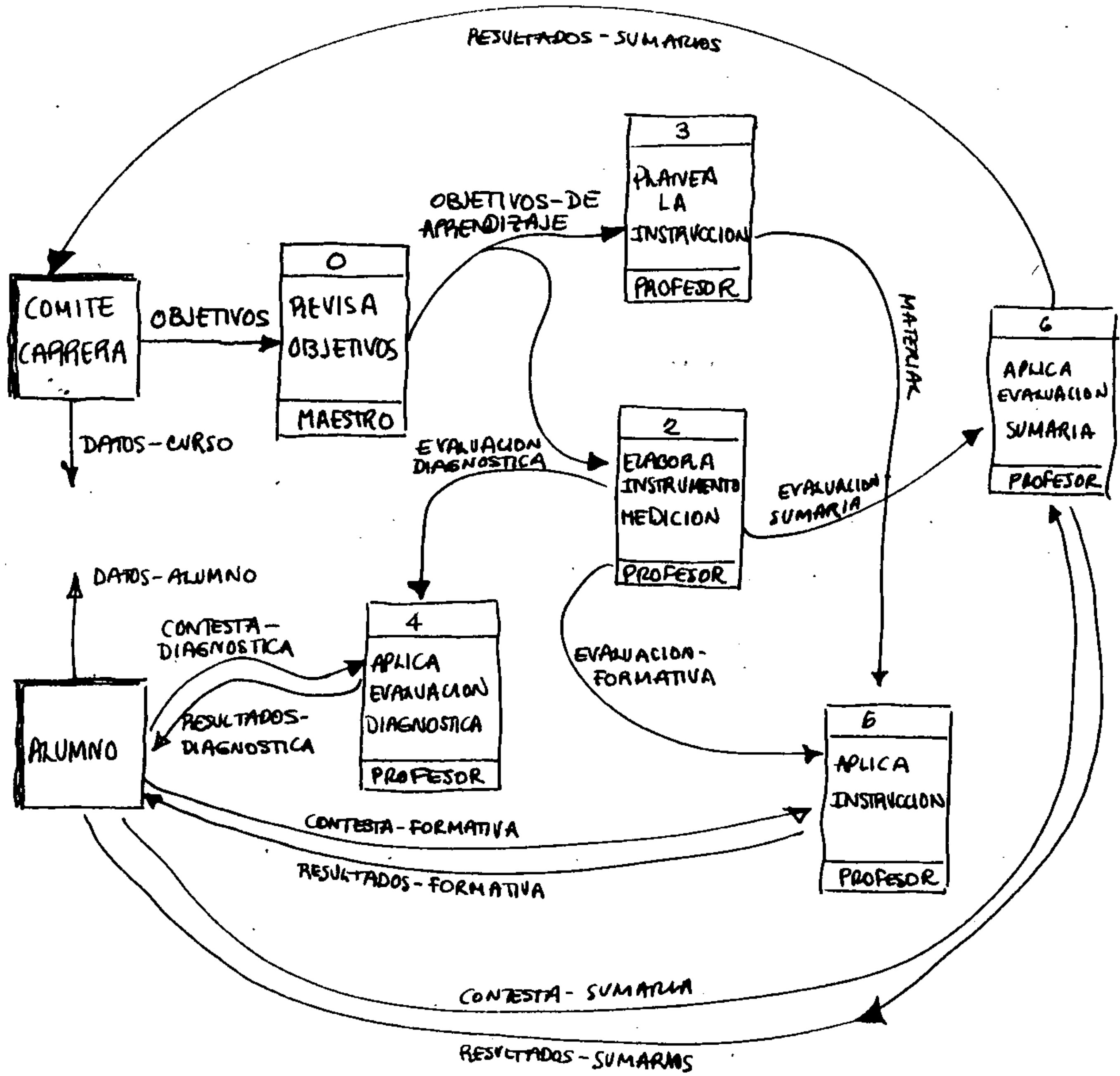
3) Preguntas al sistema

COMITE DE CARRERA:

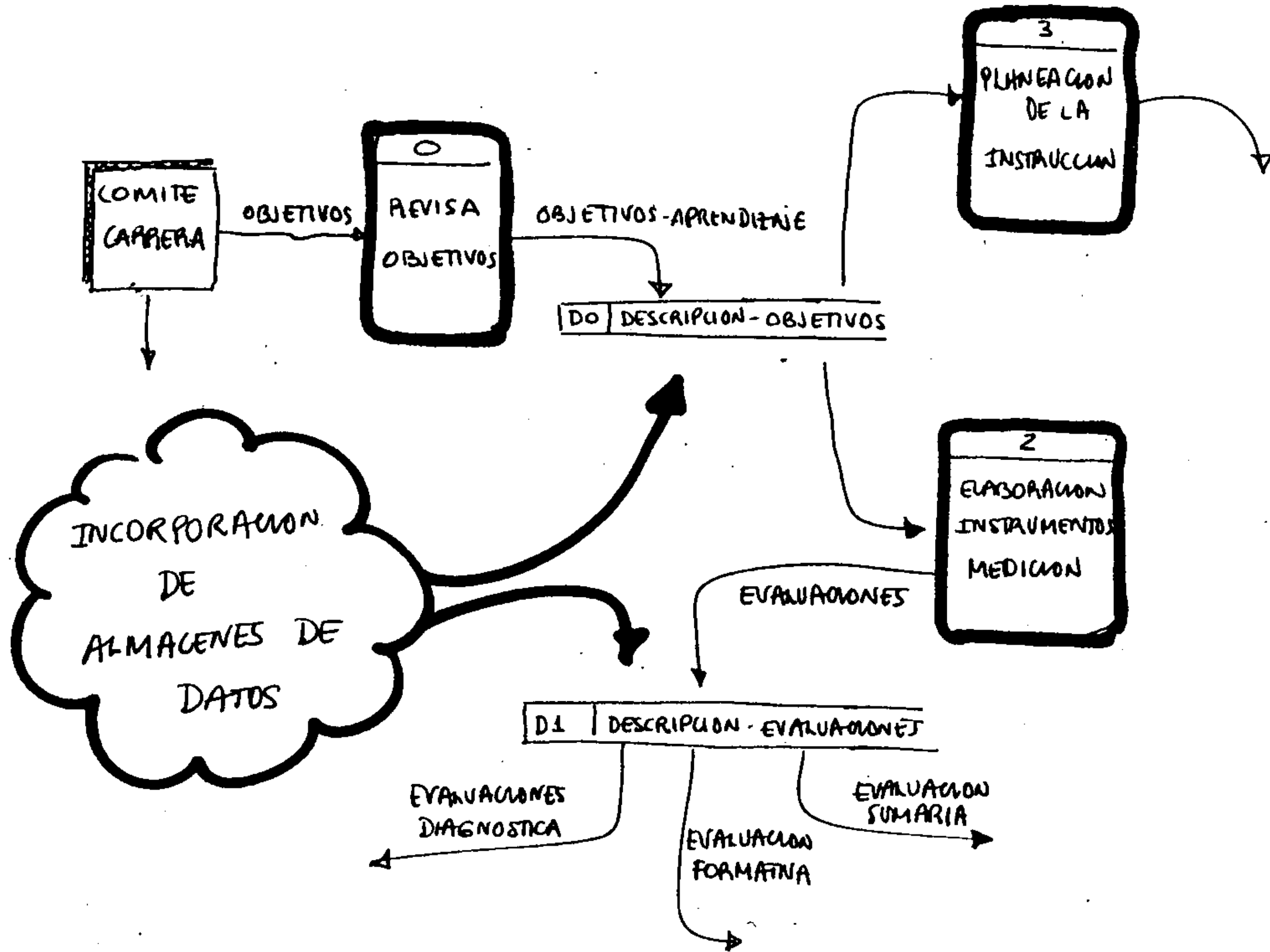
- ¿QUE CURSOS ESTAS IMPARTIENDO?
- ¿QUE ALUMNOS TIENES INSCRITOS EN CADA CURSO?
- ¿CUAL ES EL PROMEDIO DE CALIFICACIONES?
- ¿CUANTOS ALUMNOS ABANDONARON EN CADA CURSO?
- ¿QUE CAMBIOS SUFRIERON LOS OBJETIVOS?

ALUMNO:

- ¿CUAL ES MI PROMEDIO?
- ¿CUAL ES MI CALIFICACION FINAL?
- ¿DONDE PUEDO ENCONTRAR MAS INFORMACION?
- ¿QUIEN ME PUEDE ASESORAR?
- ¿CUALES SON LOS OBJETIVOS DEL CURSO?
- ¿CUALES SON LOS OBJETIVOS DE ESTA UNIDAD?
- ¿CUALES SON LOS ANTECEDENTES DEL CURSO?



(4,5,6,7)


COMITE CARRERA
OBJETIVOS
0
REVISA OBJETIVOS
OBJETIVOS-APRENDIZAJE
3
PLANEACION DE LA INSTRUCCION
D0 DESCRIPCION-OBJETIVOS
2
ELABORACION INSTRUMENTOS MEDICION
INCORPORACION DE ALMACENES DE DATOS
EVALUACIONES
D1 DESCRIPCION-EVALUACIONES
EVALUACIONES DIAGNOSTICA
EVALUACION FORMATIVA
EVALUACION SUMARIA

En los diagramas de flujos de datos, les hemos dado nombre a los flujos de datos, a los almacenes de datos y a los procesos. Pero que significan exactamente los nombres que se les han asignado, la respuesta se encuentra en el diccionario de datos (DD).

El diccionario de datos es la herramienta con la cual, documentaremos cada uno de los elementos del DFD.

Para documentar los flujos de datos y los archivos es necesario establecer que son: un dato elemental y una estructura de datos.

Dato elemental.-

Dato elemental, es sinónimo de data-item, campo y elemento. Cualquier dato elemental, toma valores atómico o que no es posible su descomposición.

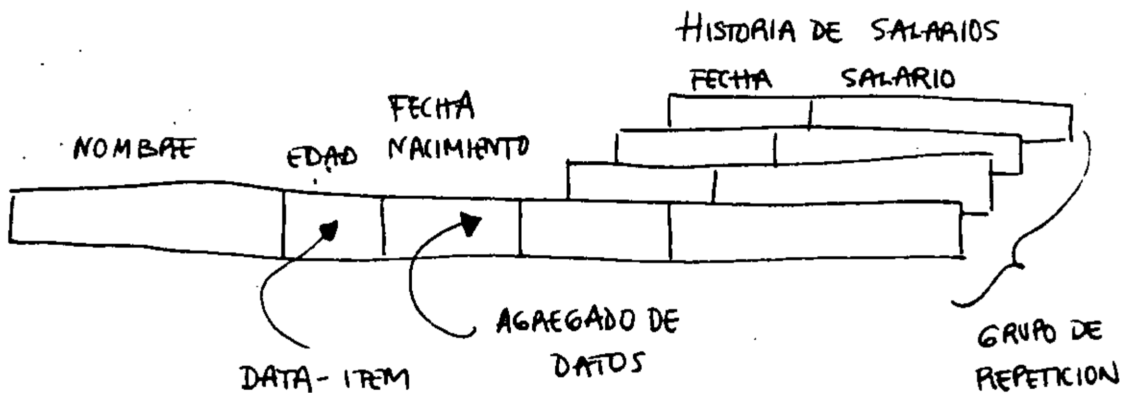
El patron de unos y ceros que lo representa debe ser considerado como un todo.

Estructura de datos.-

Es un agregado de estructuras elementos o de otro tipo de datos (agregado de datos elementales, grupos de repetición, etc.) o de una mezcla de ambos.

Ejemplo:

ENTIDAD: PERSONA   ATRIBUTOS: NOMBRE
EDAD
FECHA DE NACIMIENTO
HISTORIA DE SALARIOS
FECHA
SALARIO



Convenciones para especificar las estructuras de datos:

=	es equivalente a
+	and
[ ]	or
{ }	interación de los componentes
( )	opcional

para mostrar como serían utilizados definiremos la estructura de datos PERSONAL

PERSONAL = NOMBRE+
EDAD+
FECHA DE NACIMIENTO+
HISTORIA DE SALARIOS

FECHA DE NACIMIENTO = DIA+
MES+
AÑO

HISTORIA DE SALARIOS = FECHA+
SALARIO

NOMBRE = DATO-ELEMENTAL
EDAD = DATO-ELEMENTAL
DIA = DATO-ELEMENTAL
MES = DATO-ELEMENTAL
AÑO = DATO-ELEMENTAL
FECHA = DATO-ELEMENTAL
SALARIO= DATO-ELEMENTAL

Para documentar cada uno de los elementos del DAD se sugiere - elaborar formas para describirlas. Las siguientes son un ejemplo en las que se muestra los atributos que deben definirse en cada caso:

La forma F1 es utilizada para especificar las estructuras de - datos elementales dando para ella una descripción el tipo, sinónimo, valores discretos o continuos etc., la forma F2 es para - especificar estructuras de datos considerando una descripción, los componentes o atributos, los flujos y almacenes en donde se utiliza y los volumenes de datos que involucra; F3 es para especificar los flujos de datos especificando de donde y hacia donde fluyen los datos, una descripción del contenido del flujo, - las estructuras que fluyen y los volúmenes de datos; la F4 es - para describir los almacenes de datos dando una descripción de los flujos que llegan y salen y el contenido del almacen en término de las estructuras de datos y la forma F5 para especificar procesos dando en este caso una descripción, las entradas y salidas y un resumen del proceso.

F1

Nombre del dato elemental

descripcion

tipo A AN N

sinonimos

valores discretos

valor	significado

valores continuos

rango

valores tipicos

longitud

estructuras de datos donde aparece

F2

Nombre de la estructura

descripción

Componentes

flujos y almacenes donde aparece

Volumen de información

F3

flujo de datos

fuente ref. ______ descripción ______

destino ref. ______ descripción ______

descripción ______

estructuras de datos que fluyen

volumen de información

F4

Archivo Ref ______

descripción ______

flujos de datos que llegan

flujos de datos que salen

Contenido ______

F5

proceso Ref:

descripción

entradas	resumen lógico	salidas

EJEMPLOS:

SALARIO — Nombre del dato elemental

descripcion PERSEPCION DIARIA POR UNA JORNADA DE 8 HORAS

tipo A AN X

sinonimos

valores discretos

valor	significado

valores continuos

rango

valores tipicos LOS DEL SALARIO MINIMO

longitud 7 DIGITOS

estructuras de datos donde aparece PERSONAL

PERSONAL — Nombre de la estructura

descripción DATOS SOBRE UN EMPLEADO QUE LABORA EN LA EMPRESA

Componentes

- NOMBRE
- EDAD
- FECHA DE NACIMIENTO
- HISTORIA DE SALARIOS

Flujos y almacenes donde aparece

Volumen de información

400 EMPLEADOS

OBJETIVOS — flujo de datos

fuente ref. a — descripción COMITE DE CARRERA

destino ref. o — descripción REVISION DE OBJETIVOS

descripción DESCRIPCION DE LOS OBJETIVOS QUE DEBEN LOGRARSE AL FINALIZAR EL CURSO

estructuras de datos que fluyen — volumen de información

DESCRIP-OBJETIVO — 10 OBJETIVOS/CURSO

DESCRIPCION - EVALUACIONES — Archivo Ref D1

descripción CONTIENE LOS TEXTOS Y LAS IDENTIFICACIONES DE CADA UNO DE LOS REACTIVOS QUE COMPONEN LAS EVALUAC.

flujos de datos que llegan

EVALUACIONES

flujos de datos que salen

EVALUACIONES-DIAGNOSTICAS
EVALUACIONES-FORMATIVAS
EVALUACIONES-SUMARIAS

Contenido

IDENTIFICADOR
TEXTO-REACTIVO

PLANEACION DE LA INSTRUCCION — proceso Ref: 3

descripción: ANALISIS DE CADA OBJETIVO CON EL PROPOSITO DE ESTABLECER LOS RECURSOS, ESTRATEGIAS, TEXTOS QUE NOS LLEVEN AL LOGRO DEL MISMO.

entradas	resumen lógico	salidas
OBJETIVOS-DE APRENDIZAJE	PARA CADA OBJETIVO ESTABLECER: ANALISIS DE CONCEPTOS SECUENCIA JERARQUICA RECURSOS ESTRATEGIAS DEFINICIONES	MATERIAL

CONSTRUCCION DE MINIESPECIFICACIONES

Miniespecificaciones

Una miniespecificación es un documento en el que se da la descripción de la política que gobierna la transformación de una o mas entradas a un proceso a sus correspondientes salidas.

Las herramientas que presenta el análisis estructurado como -- una alternativa al texto clasico son las siguientes

1. Español estructurado
2. Tablas de decisión
3. Arboles de decisión

Español estructurado

La figura principal para describir los procesos es:

Si Condición 1
entonces Acción-A
o bien (no se cumple condición -1)
hacer Acción-B

Ejemplo: Sumar A con B siempre que A sea mayor que B si A es menor que B restar A de B.

Si A es menor que B
entonces restar A de B
o bien (A no es menor que B)
hacer suma de A con B

Existen una gran cantidad de políticas en donde es necesario especificar rangos.

Ejemplo: hasta 20 artículos no existe descuento mas de 20 tienen el 5%

Si el número de artículos es mayor que 20
  entonces dar el 5% de descuento
o bien (el número es menor o igual que 20)
  hacer no dar descuento

En la especificación de rangos los comparadores mas utilizados son:

Mayor que
Mayor o igual que
Menor que
Menor o igual que

En algunos casos es necesario establecer mas de una condición, los cuales deben satisfacerse todos o algunos de ellos para -- tomar una acción.

Ejemplo:

Clientes que nos han comprado mas de 1 000 000.00 en el año y tienen una buena historia de pagos tienen prioridad en el servicio y también clientes con mas de 20 años de - antiguedad.

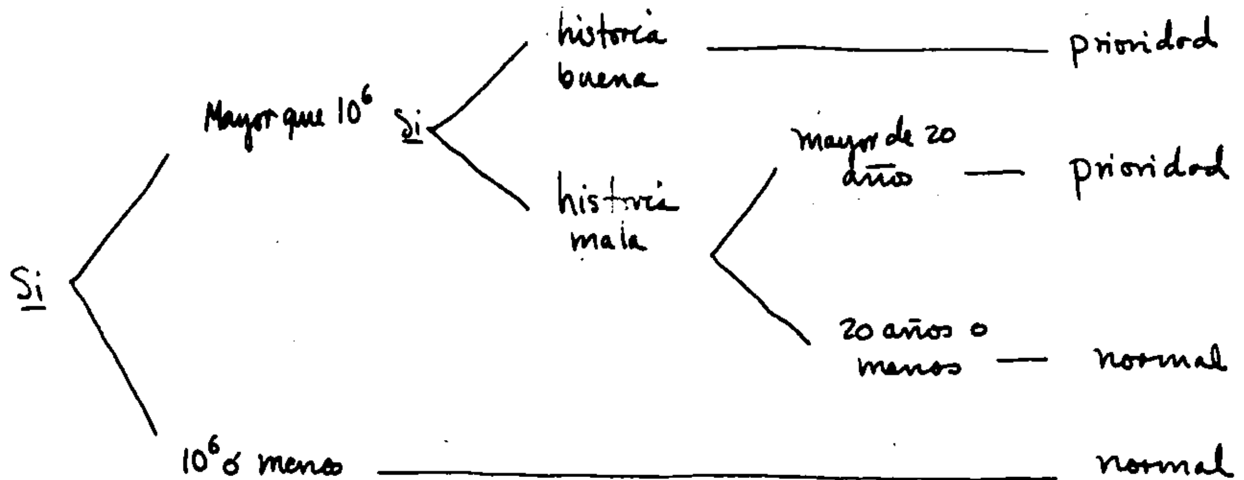
("mas de 100 000" y "buena historia de pago") o "mas de 20 años"

"mas de 100 000" y ("buena historia de pagos" o "mas de 20 años")

Arboles de decisión.

En algunos casos, resulta ser mas clara la especificación, si - construimos un arbol de decisión. Por ejemplo:

En el ejemplo anterior cualquier otra combinación implica un tratamiento normal al cliente, todas estas posibilidades quedan en un arbol de decisión de la siguiente manera:



Español estructurado

<u>Si</u> que 1000000
  <u>entonces si</u> buena historia
    <u>entonces</u> tiene prioridad
  <u>o bien</u> (mala historia)
    <u>hacer</u> si mas de 20 años
      <u>entonces</u> tiene prioridad
    <u>o bien</u> (20 a menos)
      <u>hacer</u> tramiento normal
<u>o bien</u> (1000000)
  <u>hacer</u> tratamiento normal

Tablas de decisión

Una tabla de decisión se forma, haciendo una lista de las condiciones y al final una lista de las acciones.

para el ejemplo anterior tenemos

C.1	mas de 1000000	si	si	si	si	no	no	no	no
C.2	buenos pagos	si	si	no	no	si	si	no	no
C.3	mas de 20 años	si	no	si	no	si	no	si	no
a.1	prioridad	X	X	X					
a.2	normal				X	X	X	X	X

En algunos casos, la acción está determinada por el valor de una o de dos condiciones sin importar cual es el valor de las otras. Esto hace posible reducir la tabla de decisión.

Para reducir es necesario observar lo siguiente:

1) Encontrar un par de reglas para los cuales
   - la acción es la misma
   - los valores de las condiciones son los mismos excepto -- una y solo una condición que es diferente.

2) Reemplazar el por por una sola columna sustituyendo la condición diferente por el símbolo (-) que significa "no importa"

3) Repetir este criterio los veces que sea necesario

Resultado del ejemplo anterior

	1/2	3	4	5/6/7/8
C.1	SI	SI	SI	NO
C.2	SI	NO	NO	-
C.3	-	SI	NO	-
A.1	X	X		
A.2			X	X

Recomendaciones:

1) Hay que usar un arbol de decisión cuando el número de acciones es reducido y no cualquier combinación de las condiciones es posible.

2) Hay que usar una tabla cuando el número de acciones es grande y muchas combinaciones de las condiciones es posible.

Bibliografía

Awad M. Elias. Systems Analysis and Design.
Richard D. Irwing Inc. 1979

Burch, Strater, Gridnitski. Information Systems:
Theory and practice.
John Wiley and Sons 1979.

De Marco. Structured Analysis and System Specification
Prentice Hall. 1979.

Gane and Sorson. Structured Systems Analysis:
tool and techniques. Prentice Hall. 1979

Yourdon N. Edward. Classics in Software Engineering.
Yourdon Press. 1979

Yourdon N. Edward. Structured Analysis/Design Workshop.
Edition 7.2 1981.

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS INSTITUCIONALES

No. 70

METODOLOGIA PARA EL DESARROLLO DE SITEMAS DE INFORMACION

DEL 9 AL 29 DE SEPTIEMBRE

VARIOS TEMAS

MEXICO, D.F.
PALACIO DE MINERIA
1992

Contenido

1.0 EL DISEÑO 1
1.1 Objetivos del Diseño Estructurado 2
1.2 Filosofía General. 4

2.0 LAS IDEAS COMPLEMENTARIAS 6
2.1 El Modelo. 6
2.2 El Control de Complejidad. 7
2.3 Las Herramientas. 8
2.4 La Metodología. 9

3.0 EL DIAGRAMA DE ESTRUCTURA 10

4.0 ACOPLAMIENTO 13
4.1 Factores que intervienen en el Acoplamiento 14
4.2 Tipos de acoplamiento 15
4.2.1 Acoplamiento de Datos. 15
4.2.2 Acoplamiento de Estampado. 15
4.2.3 Acoplamiento de Control. 16
4.2.4 Acoplamiento de Area Común. 16
4.2.5 Acoplamiento de Contenido. 17

5.0 COHESION 18
5.1 Tipos de Cohesión. 18
5.1.1 Cohesión Funcional. 19
5.1.2 Cohesión Secuencial. 19
5.1.3 Cohesión Comunicacional. 19
5.1.4 Cohesión de Procedimiento. 19
5.1.5 Cohesión Temporal. 20
5.1.6 Cohesión Lógica. 20
5.1.7 Cohesión Coincidental. 20

6.0 ANALISIS DE TRANSFORMACION. 21
6.1 La estrategia. 21

7.0 ANALISIS DE TRANSACCIONES 23
7.1 La estrategia. 25

8.0 CASOS PARTICULARES 26
8.1 Programas interactivos 26
8.2 Programas Batch. 28

1.0 EL DISEÑO

En los últimos años, se ha prestado creciente atención a una nueva filosofía y a un grupo de técnicas para el desarrollo de Software. Técnicas como Programación Estructurada, Programación Modular, Diseño top-down, etc. se han utilizado frecuentemente con resultados satisfactorios en la mayoría de los casos.

Sin embargo, uno de los problemas que mayor confusión generan es el de terminología, como lo demuestra el uso del calificativo "Estructurado". Hay Programación Estructurada, Análisis Estructurado, Diseño Estructurado, Implantación Estructurada, Pruebas Estructuradas, etc., y, aunque se tiende a una mayor estandarización en el significado de la terminología, es necesario aclarar cuál es el significado que daremos al Diseño Estructurado.

> Diseño Estructurado es el proceso de transformar (decidir) lo que se tiene que hacer (el qué) en la manera de hacerlo (el cómo) para resolver un problema bien especificado.

Una definición más precisa:

> Diseño Estructurado es la elaboración (utilizando herramientas de modelado de sistemas) de una solución jerárquica del sistema, con los mismos componentes e interrelaciones que el problema que se intenta resolver.

Por tanto, el Diseño Estructurado es una actividad que se realiza después de decidir lo que el usuario desea, y antes de implementar estas necesidades en terminos de código.

1.1 Objetivos del Diseño Estructurado

Existen una serie de problemas comunes a la mayoría de los proyectos de desarrollo de software, los que a continuación se mencionan se deben a la falta de técnicas apropiadas para su desarrollo:

1. Difíciles de Administrar.

La principal consecuencia de este problema es que cuando se detecta un retraso o una desviación en lo presupuestado, es demasiado tarde para corregirlo. Además las soluciones convencionales aplicables a proyectos de otras áreas no son aplicables al desarrollo de Software, por ejemplo, añadir recursos humanos tiene un efecto que se puede predecir con la ley de Brooks (1):

> Adding manpower to a late Software project makes it later.

2. Poco Satisfactorio.

Comparado con otras disciplinas de Ingeniería, el desarrollo de Software es poco profesional, frecuentemente los sistemas terminados dejan pocas satisfacciones a usuarios y diseñadores y muchas frustraciones. Esto lleva a que el desarrollo de Software sea, en general, una actividad poco productiva.

3. Poco Confiable.

Es muy frecuente que una vez que el sistema se "libera", falle una y otra vez. Las fallas pueden no tener razones obvias o peor, se pueden producir resultados incorrectos que pasen inadvertidos.

4. Inflexible.

Cuando un sistema llega a trabajar se considera un milagro. Quién supone que se podrán realizar futuros cambios sin problema?

5. Difícil de Mantener.

Para modificar un sistema, se requiere:

- Entender cómo trabaja el sistema actual.
- Concebir el cambio.
- Calcular las ramificaciones del cambio.

6. Ineficientes.

La idea generalizada de eficiencia está influenciada por la época en que el costo del procesador central era muy superior a los otros costos, de aquí que se considere eficientes a los programas que optimizan el uso del procesador central en lugar de aquellos que hacen uso eficiente de recursos escasos.

Con el Diseño Estructurado se pretende hacer sistemas cuyas características eviten los problemas antes mencionados (y otros no mencionados).

El objetivo del Diseño Estructurado es hacer Sistemas que sean:

- Utiles
- Mantenibles
- Modificables
- Flexibles
- Eficientes
- Generales

Todos estos elementos son componentes de un objetivo de calidad.

1.2 Filosofía General.

Podemos resumir los objetivos del Diseño Estructurado en uno solo: Producir sistemas económicamente convenientes (baratos).

Un diseño exitoso se basa en un principio conocido desde los días de Julio César:

Divide y vencerás.

Veamos como este principio se aplica a los siguientes aspectos:

1. Implementación.

   El costo de implementar sistemas de cómputo será minimizado cuando las partes del problema sean:

   - suficientemente pequeños
   - solucionables separadamente

2. Mantenimiento.

   De manera similar, el costo de mantenimiento será minimizado cuando las partes del sistema sean:

   - fácilmente relacionadas a la aplicación
   - suficientemente pequeñas
   - corregibles separadamente

3. Modificaciones.

   Finalmente, el coste de modificar un sistema será minimizado cuando sus partes sean:

   - fácilmente relacionadas al problema
   - modificables separadamente

En resumen, podemos establecer la siguiente filosofía:

> Implementación, Mantenimiento y Modificación serán generalmente minimizados cuando cada parte del sistema corresponda a exactamente una parte bien definida y pequeña del problema, y cada relación entre partes del sistema corresponda solo a relaciones entre partes del problema.

Esto significa que un buen diseño es un ejercicio consistente en dividir y organizar las partes de un sistema.

2.0 LAS IDEAS COMPLEMENTARIAS

Para alcanzar los objetivos del diseño estructurado, se requiere:

- Una herramienta de modelado para planear el sistema.
- Alguna manera para controlar la complejidad de sistemas no triviales.

2.1 El Modelo.

Un modelo es simplemente un cuerpo ordenado de hipótesis acerca de un sistema complejo; es un intento por entender algún aspecto de la infinita variedad de ellos que presenta el mundo, seleccionando a partir de percepciones y de experiencias pasadas, un cuerpo de observaciones generales aplicables al problema en cuestión.

Los modelos se presentan de varias maneras como son gráficas, ecuaciones, modelos a escala, maquetas, simuladores, etc. Los modelos son importantes en las disciplinas de ingeniería y su selección depende del problema en cuestión.

Para alcanzar los objetivos del Diseño Estructurado, requerimos de un modelo con las siguientes características:

- Gráfico.
- Divisible (top-down).
- Riguroso.
- Capaz de predecir el comportamiento del sistema.
- Que sea una consecuencia natural del Análisis Estructurado.
- Que sea una entrada natural para la Implementación Estructurada.
- Documentación básica del sistema.
- Ayuda para mantener y/o modificar el sistema.

2.2 El Control de Complejidad.

> The basic pattern of my approach will be to compose the program in minute steps, deciding each time as little as posible. As the problem analysis proceeds, so does the further refinement of my program. E. W. DIJKSTRA

La herramienta más útil para el control de complejidad en el diseño de sistemas es la Caja Negra.

Una Caja Negra es un mecanismo del cual se conoce:

- Sus entradas
- Sus salidas
- Lo que hace
- No se necesita saber cómo lo hace

Cuando los diseños se hacen con cajas negras pequeñas e independientes éstas son fácilmente entendidas, probadas, corregidas, mantenidas y modificadas.

El uso de la Caja Negra para controlar la complejidad del sistema, radica en dividir el sistema en Cajas Negras conectadas de tal forma que:

- Cada Caja Negra corresponda a una parte bien definida del problema.
- Cada Caja negra sea fácil de entender.
- Las conecciones entre Cajas Negras correspondan a conecciones del problema.
- Las conexiones entre Cajas Negras sean lo más simple posible, de tal manera que las Cajas Negras sean independientes entre sí.

2.3 Las Herramientas.

El Diseño Estructurado se apoya en el uso de dos herramientas:

- El Diagrama de Estructura.

- El Diagrama de Datos.

El Diagrama de Estructura muestra la división del sistema, su jerarquía y su organización.

El Diagrama de Datos muestra la división del sistema, sus flujos de datos y su organización.

Existen otras herramientas complementarias a las antes mencionadas como son las heurísticas de diseño, la morfología del sistema, criterios de transformación, etc., las cuales intervienen en el Diseño Estructurado; sin embargo la aplicación de estas herramientas complementarias requiere del Diagrama de Datos y del Diagrama de Estructura.

2.4 La Metodología.

Es posible dividir el proceso de diseño en:

- Diseño general
- Diseño detallado

El diseño general consiste en decidir qué funciones, parámetros y relaciones son requeridas para el programa (o sistema). Diseño detallado es cómo implementar las funciones.

En términos muy generales, la metodología del Diseño Estructurado consiste en la aplicación sistemática de las herramientas de diseño de la siguiente forma:

- Hacer el Diagrama de Datos.
- Aplicando criterios de transformación, hacer el Diagrama de Estructura.
- Factorizar el Diagrama de Estructura.
- Analizar el Diagrama de Estructura utilizando criterios de:
  - Aclopamiento
  - Cohesión
  - Heurísticas de diseño
- Especificar cada módulo

3.0 EL DIAGRAMA DE ESTRUCTURA

El Diagrama de Estructura es una representación gráfica del sistema que se utiliza como herramienta para el diseño, implementación, documentación, modificación y mantenimiento del sistema. Es una herramienta, no un método.

El Diagrama de Estructura es un modelo independiente del tiempo de las relaciones jerárquicas de los módulos de un programa o sistema; es por esto que no se puede inferir de un Diagrama de Estructura, cuál es el orden en que se ejecutan los módulos.

Los elementos que forman el Diagrama de Estructura son los siguientes:

- Módulos (cuadros)
- Conexiones (flechas)
- Interfases (Nombres de datos que entran y salen de cada módulo)

Un Módulo es una secuencia de instrucciones continuas de programa confinadas por variables limítrofes, tienen un identificador.

Los Módulos tienen los siguientes atributos:

El qué:
- Entradas (lo que obtiene de su invocador)
- Salidas (lo que regresa a su invocador)
- Función (lo que hace a las entradas para producir las salidas)

El cómo:
- Mecánica (cómo hace su función)
- Datos internos (espacio privado de trabajo, solo él los referencia)

Nótese que no existe diferencia entre Módulo, Programa y Sistema.

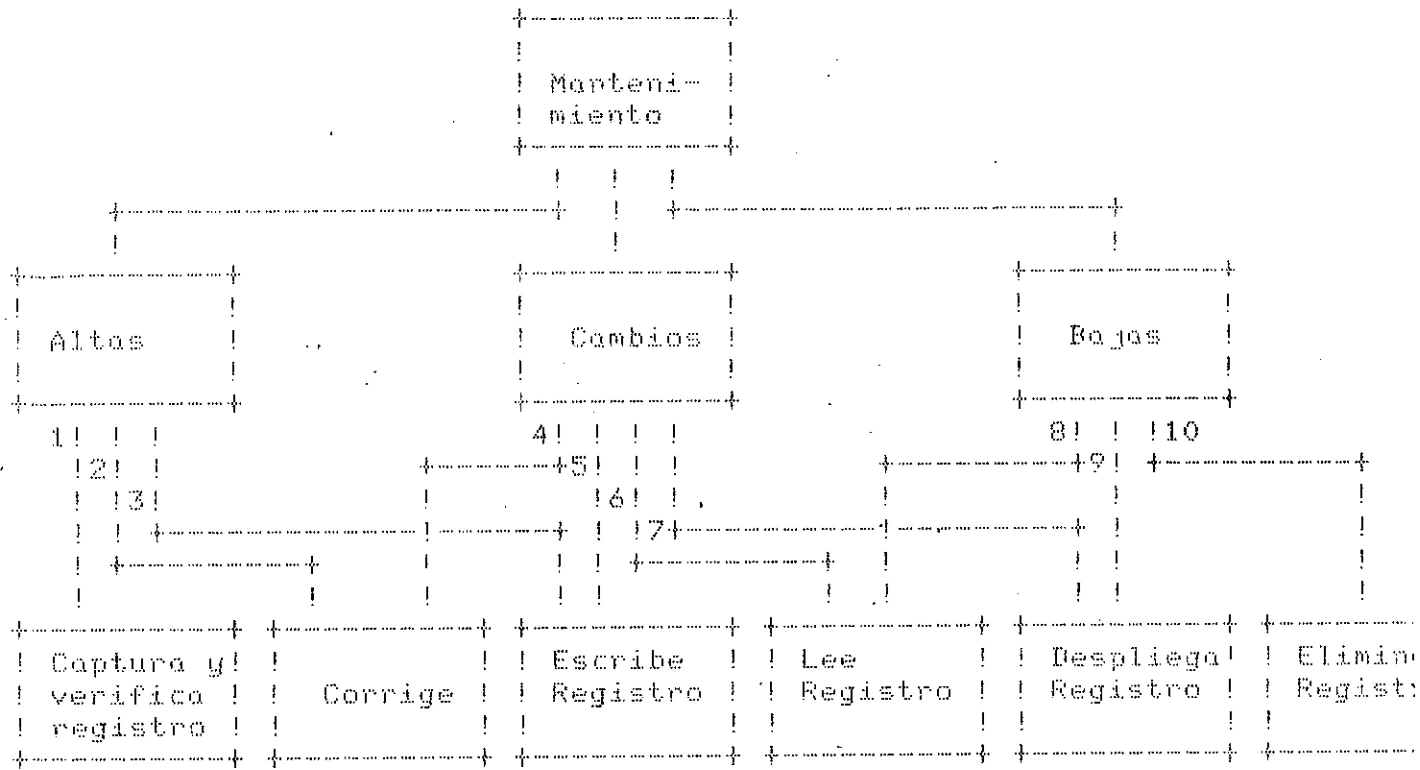
El Diagrama de Estructura muestra lo siguiente:

- La división del sistema en Módulos
- Jerarquía y organización de los Módulos
- Interfases de comunicación entre Módulos
- Nombres (funciones) de los Módulos

El Diagrama de Estructura no muestra:

- Mecanismos internos de los Módulos
- Datos internos de los Módulos

Ejemplo de Diagrama de Estructura:



	Entran (al subordinado)	Salen (del subordinado)
1		Registro
2	Registro	Registro'
3	Registro	
4	Registro	Registro
5	Registro	
6		Registro
7	Registro	
8		Registro
9	Registro	
10	Registro	Resultado

4.0 ACOPLAMIENTO

Dos Módulos son totalmente independientes si cada uno puede funcionar completamente sin la presencia del otro. Esta definición implica que no hay interconecciones directas o indirectas, explicitas o implicitas, obvias u oscuras, entre los Módulos. Esto marca el punto cero en la escala de "dependencia" entre Módulos.

En general, entre más interconecciones existan entre Módulos, serán más dependientes entre sí.

El Acoplamiento es una medida de la interdependencia de un Módulo respecto a otro. Entonces, los Módulos altamente acoplados están unidos por interconecciones rígidas. Los Módulos holgadamente acoplados estan unidos por interconecciones débiles. Los Módulos no acoplados son aquellos que no tienen interconecciones.

El Acoplamiento es el criterio más importante para juzgar las bondades de un diseño.

4.1 Factores que intervienen en el Acoplamiento

Existen cuatro factores principales que pueden incrementar o decrementar el acoplamiento de módulos, son los siguientes:

1. Tipo de conexión entre módulos.

Existen tres tipos de sistemas:

- Conexiones mínimas.
- Conexiones normales.
- Conexiones patológicas.

2. Complejidad de la interfase.

Se puede aproximar por el número de elementos pasados entre los módulos, entre más elementos haya, la interfase será más compleja.

3. Tipo de flujo de información a lo largo de la conexión.

Los sistemas con acoplamiento de datos tienen menor acoplamiento que los de acoplamiento de control y éstos a su vez son mejores que los de acoplamiento híbrido.

4. Momento de unión de la conexión.

Las conexiones unidas a referencias fijas al momento de ejecución, tienen menor acoplamiento que cuando la unión se efectúa al tiempo de carga o de compilación o de codificación.

El concepto de Acoplamiento invita al desarollo de un concepto nuevo: Deacoplamiento.

Deacoplamiento es cualquier técnica o método sistemático para volver a los módulos más independientes.

4.2 Tipos de acoplamiento

Entre menor sea el acoplamiento entre cualesquiera dos módulos, éstos serán más independientes y el diseño será mejor.

Existen cinco tipos de acoplamiento:

- Datos
- Estampado
- Control
- Area común
- Contenido

El mejor acoplamiento es el de datos y el peor el de contenido.

Si se presenta mas de un tipo de acoplamiento entre módulos, es el peor el que aplica.

4.2.1 Acoplamiento de Datos. -

Acoplamiento de datos es cuando solo los datos necesarios son comunicados entre módulos.

Este tipo de acoplamiento es el más deseable, y de hecho, cualquier sistema puede construirse de tal manera que el único acoplamiento sea de datos.

4.2.2 Acoplamiento de Estampado. -

Dos módulos presentan acoplamiento de estampado si referencían la misma estructura de datos (no global).

Una estructura de datos es un compuesto de elementos.

Este tipo de acoplamiento presenta el siguiente problema: un cambio en la estructura de datos afectará a todos los módulos que están "estampados" con la estructura;

sin embargo, usando una buena y natural estructura de datos, este tipo de acoplamiento se acerca al acoplamiento de datos.

4.2.3 Acoplamiento de Control. -

Dos módulos presentan acoplamiento de control si se comunican usando al menos un elemento de control.

Este acoplamiento es indeseable porque uno o ambos módulos no serán una caja negra.

Un peligro relacionado con el acoplamiento de control es el denominado "acoplamiento hibrido".

Acoplamiento hibrido sucede cuando se tienen dos o mas significados para el mismo dato.

4.2.4 Acoplamiento de Area Común. -

Un grupo de módulos presentan acoplamiento de área común si comparten una misma área global de datos.

Existen varios problemas con el acoplamiento de control:

1. Es más dificil reusar los módulos que utilizan áreas comunes ya que usualmente lo hacen por su nombre.

2. El mantenimiento se hace más dificil sobre todo cuando se pasan diferentes tipos de datos através del área común.

3. El uso de áreas globales dificulta la legibilidad de los programas.

4. Es dificil saber qué módulos usan qué datos.

5. En algunos lenguajes como FORTRAN, donde la posición de los datos en COMMON es importante, además del problema de acoplamiento de área común, se tiene un problema de acoplamiento de estampado.

6. Un error en cualquier módulo usando el área común puede aparecer en otro módulo (que tambien use el área común) ya que el área común no está protegida por ningún módulo.

4.2.5 Acoplamiento de Contenido. -

Este es el peor caso, ocurre cuando:

1. Un módulo altera instrucciones en otro módulo.

2. Un módulo referencia o cambia datos contenidos en otro módulo

3. Un módulo brinca a otro.

4. Dos módulos comparten las mismas literales.

Los casos 1, 2 y 3 también se conocen como Patológicos.

El problema principal es que un cambio pequeño y de apariencia inocente a uno de los módulos puede desquiciar a otro módulo en cualquier parte del sistema.

5.0 COHESION

Cohesión es una medida de la consistencia de la asociación de los elementos dentro de un módulo.

Un elemento es:

- Una instrucción
- Un grupo de instrucciones
- Una llamada a otro módulo

Es deseable tener módulos fuertes, altamente cohesivos, es decir, módulos cuyos elementos están altamente relacionados.

Claramente, cohesión y acoplamiento estan relacionados. A mayor cohesión de los módulos del sistema, existirá el menor acoplamiento.

5.1 Tipos de Cohesión.

La cohesión es una segunda medida de que tan bien dividimos el sistema. Existen los siguientes tipos:

- Funcional
- Secuencial
- Comunicacional
- De procedimiento
- Temporal
- Lógica
- Coincidental

Siendo mejor la cohesión funcional y peor la coincidental.

5.1.1 Cohesión Funcional. -

Un módulo con cohesión funcional es aquel en el que todos sus elementos contribuyen a una y solo una tarea completa; cada elemento es parte integral y es esencial para la ejecución de la función del módulo.

5.1.2 Cohesión Secuencial. -

Un módulo con cohesión secuencial es aquel en el que sus elementos están involucrados en tareas en las que datos que salen de un elemento sirven de entrada a otro elemento.

Los módulos con cohesión secuencial son fuertes, usualmente tienen buen acoplamiento. No son ideales porque contienen varias funciones que no forman una sola función completa.

5.1.3 Cohesión Comunicacional. -

Un módulo con cohesión comunicacional es aquel en el que sus elementos contribuyen a diferentes tareas, pero cada tarea se refiere a los mismos parámetros de entrada y/o salida.

Este tipo de cohesión es fuerte ya que el agrupamiento de funciones dentro de un módulo tienen alguna relación con el problema en lugar de la implementación de la solución.

Al dividir estos módulos en módulos separados, se simplifica el acoplamiento y se incrementa la cohesión.

5.1.4 Cohesión de Procedimiento. -

Un módulo con cohesión de procedimiento es aquel en el que el control fluye de un elemento al siguiente, pero, los datos no necesariamente fluyen de la misma manera.

El mayor problema con estos módulos es que manipulan resultados parciales, variables internas, banderas, switches, etc. y que tienen partes de varias funciones.

5.1.5 Cohesión Temporal. -

Los módulos con cohesión temporal son aquellos cuyos elementos están relacionados en tiempo.

Usualmente estos elementos realmente pertenecen a diferentes funciones.

5.1.6 Cohesión Lógica. -

Los módulos con cohesión lógica son aquellos en que sus elementos aparentan estar relacionados a tareas de la misma categoría general (debería llamarase cohesión ilógica).

5.1.7 Cohesión Coincidental. -

Un módulo con cohesión coincidental, tiene elementos sin relación significativa entre ellos. Usualmente ejecutan tareas diferentes y sin relación para diferentes "jefes".

6.0 ANALISIS DE TRANSFORMACION.

Análisis de transformación es un de los estrategias principales para diseñar sistemas altamente balanceados. Tambien se le conoce como Diseño de la Transformación central.

Para el Análisis de transformación se requiere del Diagrama de Datos. El resultado es un Diagrama de Estructura en el que el módulo superior trabaja con datos altamente procesados, datos lógicos.

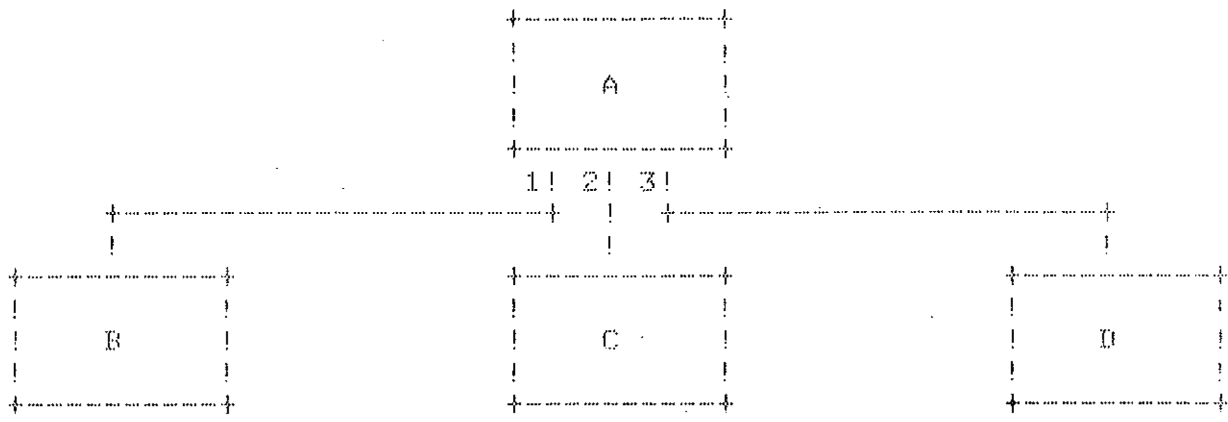
6.1 La estrategia.

1. Dibujar el diagrama de datos del problema.

2. Identificar la transformación central. Esto puede realizarse siguiendo las ramas aferentes y eferentes del diagrama hasta encontrar los puntos en que los datos son independientes de los dispositivos de entrada-salida. Los procesos entre estos puntos forman la transformación central.

3. Identificar las ramas principales de datos de entrada y salida. Determinar los puntos de mayor abstracción.

4. Diseñe la estructura a partir de la información previa con un módulo para cada rama principal de entrada y un modulo para cada rama principal de salida. En general se llegará a la estructura siguiente:



	Entran (al subordinado)	Salen (del subordinado)
1	Usualmente nada	Datos abstractos de entrada
2	Datos abstractos de entrada	Datos abstractos de salida
3	Datos abstractos de salida	Usualmente nada

5. Para cada uno de los módulos de entrada, identificar la última transformación necesaria para producir los datos en la forma en que los regresa el módulo. Después identifique la forma de la entrada justo antes de la última transformación. Para módulos de salida, identifique el primer proceso necesario para acercarse a la salida deseada con el formato deseado. Repita este paso hasta llegar a la forma original de los datos y el formato deseado de los resultados.

7.0 ANALISIS DE TRANSACCIONES

Una transacción es cualquier elemento de datos, control, señal, evento, o cambio de estado que causa, dispara o inicia alguna acción o secuencia de acciones.

En otras palabras, una transacción es una parte de datos que puede ser cualquiera de un número de tipos, cada tipo requiere diferente procesamiento.

La estrategia de análisis de transacciones, simplemente reconoce que los diagramas de datos de este tipo pueden mapearse a una estructura modular particular. El centro de transacciones de un sistema, debe ser capaz de:

1. Obtener (responder a) las transacciones en su forma mas primitiva.

2. Analizar cada transaccion para determinar su tipo.

3. Despachar dependiendo de la transacción.

4. Completar el proceso de cada transacción.

En su forma más factorizada, el centro de transacciones puede ser modularizado de la siguiente forma:

```
                          +---------+
                          !         !
                          ! Transac- !
                          ! ción    !
                          +---------+
                           1!   2!
          +----------------+    +-----------------+
          !                                       !
      +---------+                           +----------+
      !         !                           !          !
      ! Analiza !                           ! Despacho !
      !         !                           !          !
      +---------+                           +----------+
       3!   4!                              ! !   ! !
        !    !                   +-------------+ !   ! +-------------+
      +-+    +----+              !          +---+   +---+            !
      !           !              !          !           !            !
+-----------+ +-----------+ +---------+ +---------+ +---------+ +---------
! Obtiene   ! ! Analiza   ! !         ! !         ! !         ! !
! transac-  ! ! transac-  ! ! Tipo 1  ! ! Tipo 2  ! ! Tipo 3  ! ! Tipo N
! ción      ! ! ción      ! !         ! !         ! !         ! !
+-----------+ +-----------+ +---------+ +---------+ +---------+ +---------
```

	Entran (al subordinado)	Salen (del subordinado)
1		Transacción codificada internamente
2	Transacción codificado internament	
3		Transacción
4	Transacción	Transacción analizada y codificada

7.1 La estrategia.

1 Identificar los orígenes de transacciones.

2. Especificar la organización de transacciones apropiada (la figura anterior es un buen modelo en general).

3. Identificar la transaccion y las acciones que la definen.

4. Note situaciones potecìales en que se puedan combinar módulos.

5. Para cada transacción o grupo cohesivo de transacciones, especificar un módulo de transacciones para procesarla completamente.

6. Para cada acción en una transacción, especificar un módulo de acción subordinado al correspondiente módulo de transacción.

7 Para cada paso detallado en un módulo de acción, especificar un módulo detallado subordinado al módulo de acción que lo necesite.

8.0 CASOS PARTICULARES

8.1 Programas interactivos

Un programa interactivo tiene los siguientes características:

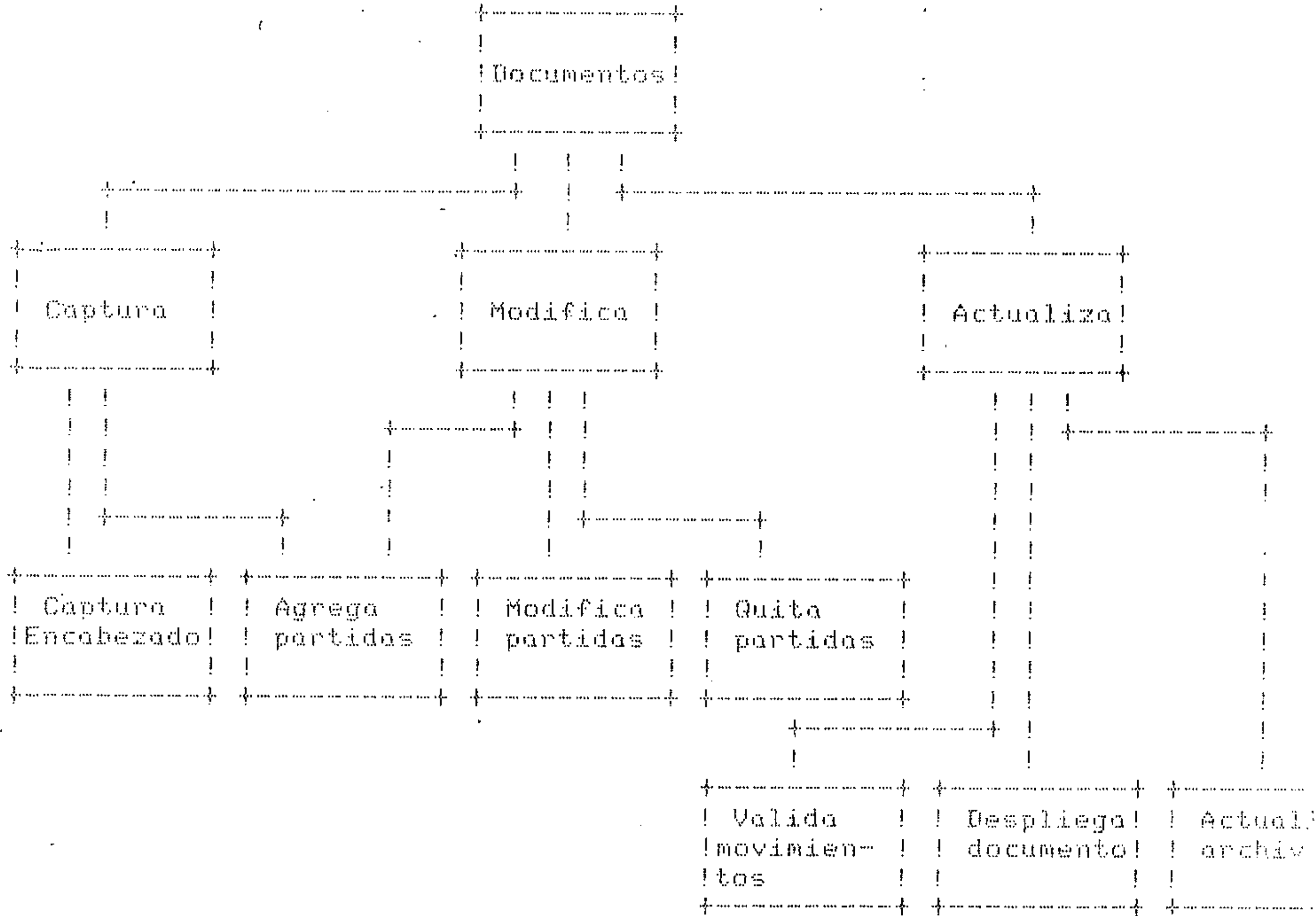
1. Conversacional.
2. Estructurado por tipo función.
3. Cada transacción es validada (consultando archivos) y actualizada instantaneamente.

Estos programas por naturaleza presentan una estructura arborea. Por tanto, la estrategia más apropiada para diseñarlos es a partir del árbol de funciones.

Ejemplo de árbol de funciones:

```
          !--
          !
          ! C)APTURA UN DOCUMENTO
          !
          !
          !
          ! A)CTUALIZA EL ARCHIVO
          !
          !                            !--
OPCIONES (                             ! A)GREGA PARTIDAS
          !                            !
          ! M)ODIFICA EL DOCUMENTO (     Q)UITA PARTIDAS
          !                            !
          !                            ! M)ODIFICA PARTIDAS
          !                            !--
          !
          ! F)IN
          !
          !--
```

Ejemplo de estructura a partir del Árbol de funciones.



8.2 Programas Batch.

Un programa batch tiene las siguientes características:

1. Controlado mediante comandos. Si el comando es incorrecto, se requiere realimentarlo hasta que sea correcto.

2. No es posible hacer correcciones a los errores de validación al momento de ejecutar el programa.

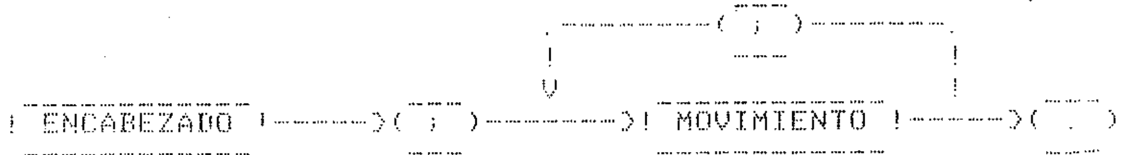
3. Típicamente se utilizan en procesos periódicos.

La manera más adecuada para diseñar estos programas es a partir de una definición sintáctica formal de los comandos y de la información que se va a procesar.

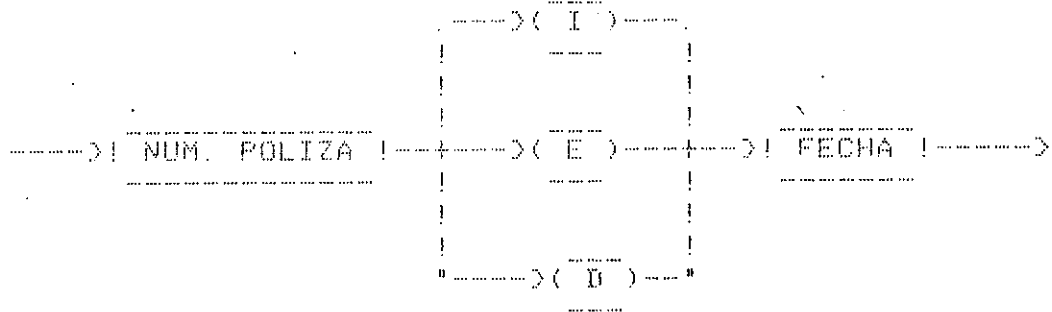
Una de las definiciones sintácticas más convenientes es la notación de Backus Naur (BNF) la cual se puede representar gráficamente con los diagramas de ferrocarril (estilo PASCAL).

Ejemplo de diagrama de sintaxis:

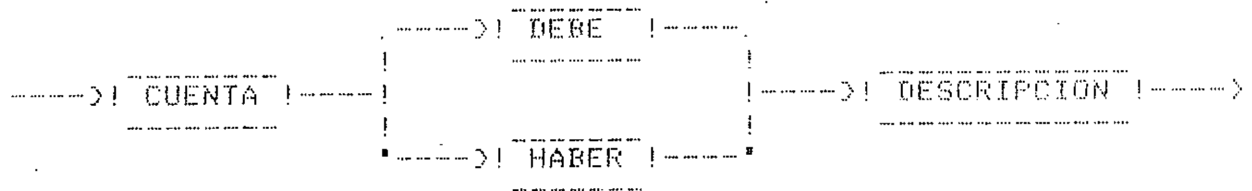
POLIZA:



ENCABEZADO:



MOVIMIENTO:



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS INSTITUCIONALES
No. 70

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

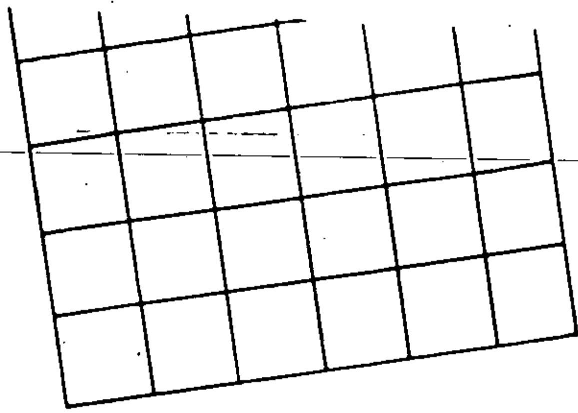
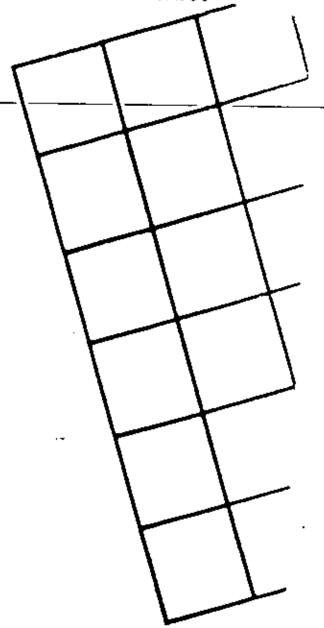
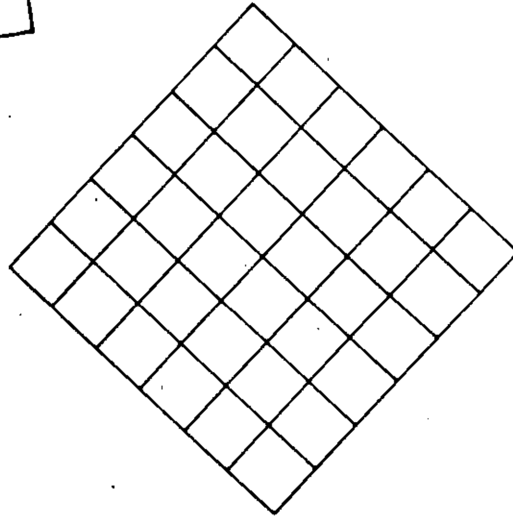
DEL 9 AL 29 DE SEPTIEMBRE

SEPOMEX

DISEÑO DE LA INTERFAZ DEL USUARIO

MEXICO D.F.
PALACIO DE MINERIA
1992

lacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

16 DISEÑO DE LA INTERFAZ DEL USUARIO

Al diseñar un paquete de gráficas, necesitamos considerar no sólo las operaciones de graficación que se efectuarán sino también la forma en que estas operaciones se pondrán a disposición de un usuario. Esta interfaz debe diseñarse de manera que se proporcione un medio adecuado y efectivo para que el usuario accese funciones de gráficas básicas, como el despliegue de objetos, establecimiento de atributos o realización de transformaciones. El paquete de gráficas podría construirse para que produzca diseños de ingeniería, planos arquitectónicos, proyectos de dibujo mecánico o gráficas financieras o bien podría diseñarse como un programa pincel para un artista. Cualquiera que sea el tipo de aplicación al que se destine, necesitamos decidir qué diálogo interactivo sirve mejor al usuario, el tipo de rutinas de manipulación que se utilizarán y los dispositivos de salida que resultan adecuados para el tipo de aplicación implicada. Una interfaz pobremente diseñada aumenta las oportunidades de que el usuario cometa errores y puede incrementar significativamente el tiempo que tarda el usuario en realizar una tarea.

16-1 Componentes de la interfaz del usuario

Existen muchos factores que se incluyen en el diseño de la interfaz del usuario. Además de las operaciones específicas que se pondrán a disposición del usuario, debemos considerar cómo se organizarán los menús, cómo responderá el paquete de gráficas a la entrada y a errores, cómo se organizará el despliegue de salida y cómo se documentará y explicará el paquete al usuario. Para ayudarnos a explorar estos factores, consideramos el diseño de la interfaz de un usuario en términos de las componentes que siguen:

Modelo del usuario
Lenguaje de comando
Formatos del menú
Métodos de retroalimentación
Formatos de salida

El **modelo del usuario** ofrece la definición de los conceptos implicados en el paquete de gráficas. Este modelo ayuda al usuario a entender la forma en que opera el paquete en términos de conceptos de aplicación. Explica al usuario qué tipo de objetos pueden desplegarse y cómo pueden manipularse.

Las operaciones que el usuario tiene a su disposición se definen en el **lenguaje de comando**, el cual especifica las funciones de manipulación de los objetos y las operaciones de archivo. Las funciones comunes de manipulación de objetos son aquellas que sirven para reordenar y transformar objetos de una escena. Las operaciones de archivo pueden ofrecer la creación, cambio de nombre y copiado de segmentos. La forma en que se estructuren los comandos del usuario dependerá del tipo de dispositivos de entrada y salida elegidos para el sistema de gráficas.

Las opciones de procesamiento pueden presentarse a un usuario en **formato de menú**. El menú podría utilizarse para enlistar las operaciones disponibles y los objetos que se manipularán.

Una consideración importante en el diseño de una interfaz es la forma en que el sistema responderá o bien dará **retroalimentación** a la entrada del usuario. La retroalimentación ayuda a un usuario al operar el sistema reconociendo la recepción de comandos, enviándole diversos mensajes y señalándole cuándo se han recibido las selecciones del menú. Algunos tipos de retroalimentación son parte integral de la estructura del lenguaje de comando, en tanto que se ofrecen otras formas de retroalimentación para ayudar a un usuario a entender la operación del sistema.

La forma en que la información se presentará a un usuario se determina por los **formatos de salida**. Una imagen de salida debe organizarse de modo que suministre información al usuario en la forma más eficaz posible. Los factores que se considerarán en el diseño de los formatos de salida incluyen la elección de modelos geométricos que se utilizan para representar objetos y la disposición total de la salida en un dispositivo de despliegue.

16-2 Modelo del usuario

El diseño de la interfaz de un usuario comienza con el modelo del usuario. El modelo determina la estructura conceptual que se le presentará al usuario. El modelo describe para lo que está diseñado el sistema y de qué operaciones de grafica-

ción se dispone. Por ejemplo, si el paquete de gráficas se va a utilizar como una herramienta en diseño arquitectónico, el modelo describe la forma en que puede usarse el paquete para construir y desplegar vistas de edificios. Una vez que se haya establecido el modelo del usuario, las otras componentes de la interfaz pueden desarrollarse. La etapa final en el diseño del modelo consiste en elaborar el manual del usuario, el cual explica el sistema y ofrece ayuda en su uso.

Básicamente, el modelo del usuario define el sistema de gráficas en términos de objetos y las operaciones que pueden efectuarse con los objetos. Para un sistema de proyección de una instalación, los objetos pudieran definirse como un conjunto de elementos de mobiliario (mesas, sillas, etc.) y entre las operaciones a disposición se incluirían aquellas para posicionar y eliminar diferentes piezas de mobiliario dentro del proyecto de la instalación. En forma análoga, un programa de diseño de circuitos podría utilizar elementos eléctricos o lógicos para los objetos, con operaciones de posicionamiento disponibles para agregar o suprimir elementos dentro del diseño total del circuito.

Objetos como los elementos de mobiliario y de circuitos se denominan **objetos de aplicación**. Además de estos objetos, el modelo del usuario podría contener otros objetos que se utilizan para controlar operaciones de graficación. Estos se llaman **objetos de control**. Ejemplos de objetos de control son los cursores para seleccionar posiciones en la pantalla, símbolos de selección del menú y retículas de posicionamiento (fig. 16-1). Las operaciones de que se dispone en un paquete permiten la manipulación de objetos de manipulación y de control. A los usuarios se les puede brindar la capacidad de hacer girar los objetos de control, eliminar estos objetos o bien variar su tamaño.

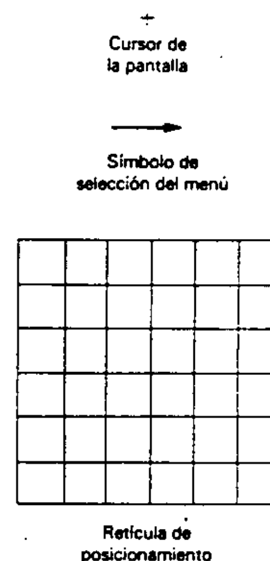


FIGURA 16-1
Ejemplos de objetos de control.

Sólo deben emplearse conceptos bien conocidos en el modelo del usuario. Si construimos un paquete de gráficas como auxiliar en el diseño arquitectónico, la operación del paquete debe describirse en términos del posicionamiento de paredes, puertas, ventanas y otras componentes de construcción para formar las estructuras arquitectónicas. El modelo no debe contener referencias de conceptos que puedan ser desconocidas para un usuario. Por ejemplo, no existe ninguna razón para suponer que un arquitecto conocería los términos de la estructura de datos. La introducción de información detallada concerniente a la operación del sistema en términos de estructuras de árbol, listas ligadas y segmentos produciría una sobrecarga de trabajo innecesaria en el usuario. Toda la información contenida en el modelo del usuario debe presentarse en el lenguaje de la aplicación.

En conjunto, el modelo del usuario debe ser lo más simple y consistente que sea posible. Un modelo complicado es difícil de entender para un usuario y le resulta complicado trabajar con él en forma eficiente. El número de objetos y operaciones de graficación del modelo deben minimizarse a sólo aquellas que se necesiten en la aplicación. Esto hace que al usuario le sea fácil aprender el sistema. Por el otro lado, si el paquete se simplifica demasiado, puede ser fácil de aprender pero difícil de aplicar. El diseñador del modelo del usuario debe buscar asimismo consistencia. Los objetos y las operaciones no deben definirse en diferentes formas cuando se utilicen en distintos contextos. Por ejemplo, un solo símbolo no debe servir como objeto de aplicación y como objeto de control, dependiendo del modo de interacción. Esto hace difícil que un usuario lleve el registro de los significados de los símbolos. Es mucho más fácil para un usuario aplicar el paquete si los objetos y las operaciones se definen y utilizan de manera consistente.

El punto inicial para elaborar un modelo de usuario a menudo consiste en realizar un análisis de la tarea. El análisis de la tarea es un estudio del medio y de las necesidades del usuario. Este estudio comúnmente se realiza entrevistando

a usuarios prospectos y observando la forma en que realizan sus tareas. Las conclusiones del análisis de tareas puede formar la base para decidir qué tipos de operaciones y objetos se necesitan y la forma en que el paquete de gráficas debe presentarse al usuario.

16-3 Lenguaje de comando

El lenguaje interactivo elegido para un paquete de gráficas debe ser lo más natural posible para que el usuario lo aprenda, con todas las operaciones especificadas en términos relativos al área de aplicación. Un paquete de diseño de circuitos especifica operaciones en términos de la manipulación de los elementos del circuito, mientras que las operaciones de un paquete de dibujo mecánico son aquellas que sirven para trazar figuras geométricas. Los comandos deben diseñarse de modo que el usuario no tenga que aprender nuevos conceptos, así como también el nuevo lenguaje.

Minimización de la memorización

Cada operación del lenguaje de comando debe estructurarse de manera que un usuario pueda entenderla con facilidad y recuerde el objetivo de la misma. Deben evitarse los formatos de comandos oscuros, complicados, inconsistentes y abreviados. Sólo confunden a un usuario y reducen la efectividad del paquete de gráficas. Es más fácil que un usuario recuerde un comando como *select_object* que el formato abreviado *so*. Una llave, o botón, que borra cualquier comando es más fácil de utilizar que varias llaves de supresión para diferentes operaciones. Además, el lenguaje de comando debe estructurarse de modo que no se pida al usuario distraer su atención constantemente de un dispositivo de entrada a otro.

Para un usuario menos experimentado, un lenguaje de comando con pocas operaciones fácilmente asimilables es por lo general más efectivo que un conjunto de operaciones grande y general. Un conjunto de comandos simplificado es fácil de aprender y recordar, y el usuario puede concentrarse en la aplicación en vez de en el lenguaje. No obstante, un usuario experimentado podría hallar algunas aplicaciones difíciles de manejar con un conjunto pequeño de comandos diseñado para el novato. Para dar entrada a varios usuarios, los lenguajes de comando pueden diseñarse en varios niveles. Los principiantes pueden utilizar el nivel más bajo, el cual contiene el mínimo conjunto de comandos y los expertos pueden usar los conjuntos mayores de comandos en los niveles superiores. Conforme se gana experiencia, un principiante puede ascender los niveles, ampliando el lenguaje de comando un poco cada vez.

Facilidades de ayuda para el usuario

Es muy importante que se incluyan facilidades de ayuda en el lenguaje de comando. Diferentes niveles de ayuda permiten a los usuarios principiantes obtener instrucciones detalladas, mientras que los más experimentados pueden obtener solicitudes de entrada breves que no interrumpan su concentración.

Las facilidades de ayuda pueden incluir una sesión "tutorial" que ofrezca instrucción acerca de cómo utilizar el sistema. Un principiante puede revisar el paquete y hacer un repaso general de las funciones del paquete y de la forma en que opera el conjunto de comandos básicos. Pueden ofrecerse varias aplicaciones de

ejemplo de manera que el usuario pueda apreciar la forma en que las operaciones trabajan en realidad en aplicaciones comunes.

Si se dispone de diferentes niveles de ayuda, un principiante puede seleccionar el nivel más bajo y recibir solicitudes de entrada y explicaciones detalladas en cada etapa durante la aplicación del paquete. Las solicitudes de entrada pueden indicar al principiante exactamente qué hacer a continuación. Un usuario experimentado podría seleccionar menos solicitud de entrada o bien desactivar la solicitud de entrada por completo. Las solicitudes de entrada pueden darse en la forma de menús o mensajes desplegados. Para el usuario experto, pueden darse solicitudes de entrada más tenues. Un cursor centelleante podría indicar cuándo se requiere una entrada coordenada y podría desplegarse una escala cuando se vaya a seleccionar un valor escalar.

Respaldo y manejo de errores

Durante cualquier secuencia de operaciones, debe disponerse de algún mecanismo sencillo de respaldo o aborto. Con frecuencia, una operación puede cancelarse antes de que se complete la ejecución, con el sistema restituido en el estado en que se encontraba antes de que se iniciara la operación. Con la capacidad del respaldo en algún punto, un usuario puede explorar con confianza las capacidades del sistema, sabiendo que pueden eliminarse los efectos de cualquier error.

El respaldo puede ofrecerse en muchas formas. Las posiciones de ventanas y puertas de visión pueden probarse y los objetos pueden llevarse a diferentes posiciones antes de decidir su ubicación final. Algunas veces un sistema puede respaldarse a través de varias operaciones, permitiendo al usuario volver a colocar el sistema en algún punto especificado. En un sistema con extensas capacidades de respaldo, todas las entradas podrían salvarse de manera que un usuario pueda respaldar y "volver a correr" cualquier parte de una sesión.

Si no hay una facilidad de respaldo, pueden usarse otros métodos para ayudar a los usuarios a superar los efectos de los errores. Podría pedirse a un usuario que verifique algunos comandos antes de ejecutar las instrucciones. Se llevaría demasiado tiempo preguntar al usuario *¿Está usted seguro que desea hacer esto?* después de cada entrada, pero sería adecuado para las acciones que uno u otro no pueda quedarse sin hacer o se necesitaría hacer un esfuerzo considerable para restituirlo.

Los diagnósticos efectivos y los mensajes de error deben incorporarse en el lenguaje de comando para permitir a los usuarios evitar errores y entender lo que estuvo mal cuando se haya cometido un error. Los mensajes de error confusos no pueden ayudar a un usuario a corregir un error. El mensaje de error debe ofrecer una explicación clara de lo que anda mal y de lo que se necesita hacer para corregir la situación. Comúnmente, el sistema de recuperación despreciará una entrada incorrecta e informará al usuario del error en una forma que ayude a éste a determinar la entrada adecuada en ese punto.

Tiempo de respuesta

El tiempo que tarda el sistema en responder a la entrada de un usuario depende de la complejidad de la tarea solicitada. Para muchas solicitudes de entrada de rutinas, el sistema puede responder en forma inmediata. Cuando un usuario mete una solicitud de procesamiento complicada, puede esperarse alguna demora y este retraso podría utilizarse para planificar la siguiente fase de la aplicación.

Sin importar la complejidad de la solicitud de la entrada, los usuarios pueden esperar que los sistemas den algún tipo de respuesta inmediata; de lo contrario, no pueden estar seguros de que la entrada fue recibida y que el sistema realiza el procesamiento. Debe diseñarse un pequete de gráficas para dar esta respues- "instantánea" a la entrada del usuario en cerca de una décima de segundo. tiempo de respuesta mayor que éste puede interrumpir la sucesión de ideas de usuario, ya que éste empieza a preguntarse qué está haciendo el sistema en vez de pensar en la siguiente etapa de la aplicación. Si el tiempo de procesamiento va a ser más largo que una décima de segundo, la respuesta inmediata simplemente permite al usuario saber que la entrada se ha recibido. Para el usuario que empieza, esta respuesta podría ser un mensaje que afirma que la entrada se está procesando. Con usuarios experimentados, un cursor centelleante o un cambio de color o intensidad pueden servir para el mismo fin.

Algunos sistemas están diseñados de manera que la variabilidad en tiempos de respuesta entre diferentes tipos de procesamiento no sea demasiado grande. Un sistema que alterna entre respuestas instantáneas y demoras de varios segundos (o minutos) podría ser más desconcertante para un usuario que un sistema con respuesta más lenta con menos variabilidad en los tiempos de respuesta.

Estilos de lenguajes de comando

Existen varios estilos posibles del lenguaje de comando y la elección del formato de los comandos de entrada depende de varios factores. Entre estos factores se incluyen los objetivos del paquete, el tipo de dispositivo de entrada que se utilizarán y el tipo del usuario.

En conjunto, el lenguaje de comando puede conformarse de modo que la secuencia de acciones de entrada sea dirigida por un paquete de gráficas o bien por un usuario. Cuando el paquete dirige la entrada, se indica al usuario qué tipo de acción se espera en cada etapa. Este es un método particularmente efectivo par los principiantes, donde se utilizan solicitudes de entrada y menús para explic lo que se pide y cómo meter la entrada. En algunos casos, los usuarios pueden verse restringidos a un número limitado de respuestas, como la replicación con *sí*, *no* o bien un valor numérico. Con otros sistemas, puede dirigirse a un usuario a seleccionar una acción de una lista de alternativas. Esta selección podría hacerse a partir de un menú desplegado, utilizando una pluma linterna o el control del cursor de un teclado o quizá la selección podría efectuarse utilizando llaves de función. Una vez que se haya escogido la acción, como la selección de un objeto, se dirige al usuario a meter parámetros adecuados (coordenadas, valores de transformación, etc.).

Para lenguajes iniciados por el usuario, las acciones del sistema son dirigidas por entradas del usuario. Se da poca o nada de solicitud de entrada al usuario, quien selecciona en forma independiente el tipo de acción que se realizará en cada etapa. Este método ofrece la mayor flexibilidad y se adecúa mejor al usuario experimentado. Podrían usarse algunos menús y solicitudes de entrada para recordar al usuario las opciones de que dispone con cualquier operación seleccionada, pero en general al usuario tiene la libertad de explorar las capacidades del sistema sin seguir una secuencia de acciones prefijada.

Cuando un paquete está diseñado solamente para realizar un diálogo, no se pone a disposición del usuario ningún conjunto de comandos. La entrada se logra seleccionando opciones de menús desplegados y dando respuestas simples a solicitudes de entrada. Este método de obtención de la entrada es adecuado para princi-

piantes pero es ineficaz para aquellos con más experiencia, quienes pueden utilizar mejor las capacidades de un paquete con un conjunto de comandos de entrada. Los comandos pueden diseñarse para meterse con llaves de función o tecleándolos en un teclado estándar.

Para minimizar el tiempo de aprendizaje del usuario, la sintaxis de los comandos de entrada debe ser simple y directa. Por ejemplo, para eliminar un objeto de un despliegue, un usuario podría meter el comando *delete_object*, seguido de los valores de parámetros que identifican el objeto que se eliminará. Las designaciones de los parámetros podrían dar un número de objeto o bien el nombre de éste. El usuario también podría señalar el objeto con una pluma linterna o mover el cursor de la pantalla a la localidad del objeto. Los comandos que especifican primero la acción, como en el ejemplo que acaba de darse, se denominan comandos prefijos. Los comandos posfijos especifican primero los parámetros y después la acción. Cuando se utilizan comandos posfijos, la lista de parámetros puede corregirse en caso de que aparezca un error antes de que se dé la acción.

En algunas aplicaciones, la entrada del usuario puede reducirse mediante la selección de una sintaxis de comandos que admita múltiples especificaciones de parámetros. Por ejemplo, en vez de un comando de supresión aparte para cada objeto, un usuario podría introducir un solo comando de supresión que permita que se eliminen varios objetos:

```
delete
  object_name1
  object_name2
  object_name3
```

La acción del comando de supresión llegaría a su fin cuando se meta la siguiente operación.

Otra posibilidad para reducir la entrada del usuario consiste en eliminar la lista de parámetros contenidos en los comandos. En vez del comando *delete_object* seguido de la designación del objeto, el nombre del objeto podría incorporarse en el comando. Esto requiere un comando aparte para cada objeto, pero los comandos podrían implantarse como llaves de función, de manera que sólo se requiera una entrada para meter el comando. Para poner un ejemplo, con un comando aparte, como *delete_worktable*, *delete_shelves*, etcétera.

Un patrón de trazo es otra forma de entrada que es de utilidad en muchas aplicaciones de diseño. Aquí, el usuario traza un patrón con un dispositivo de golpe, como una tableta de gráficas; el patrón se despliega en un monitor de video y es procesado por el programa de diseño. El patrón de entrada podría utilizarse con un programa de reconocimiento de patrones para seleccionar una figura o bien este tipo de entrada podría ser la base de un paquete de dibujo que permita a un usuario crear un diseño o imagen.

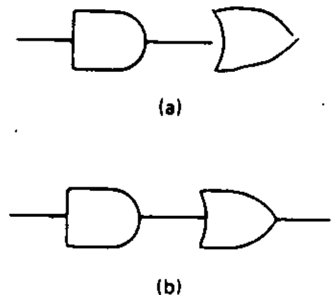


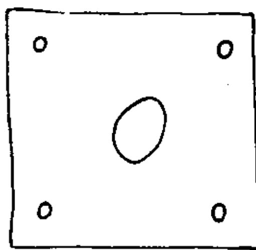
FIGURA 16-2
Un símbolo de entrada (a) trazado por un diseñador es identificado por el programa de reconocimiento de patrones y se sustituye por la figura correspondiente del elemento del circuito (b).

Los programas de reconocimiento de patrones toman un patrón de entrada y lo comparan contra una biblioteca de figuras predefinidas. Un diseñador de circuitos lógicos podría utilizar este tipo de programa para trazar proyectos de circuitos. Cada elemento del circuito es trazado, como en la figura 16-2(a), y el programa sustituye el elemento trazado con la figura de la biblioteca correspondiente, como en la figura 16-2(b). Esta técnica puede ser más rápida que la selección de una figura de un menú y su colocación dentro del circuito si la biblioteca de figuras al-

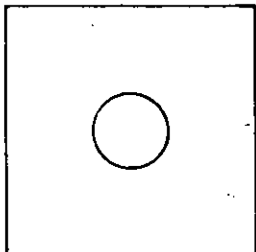
macenadas no es demasiado grande (cerca de 20 veces o menos) y no intervienen rotaciones.

Un programa de reconocimiento de patrones se utiliza con algunos sistemas para permitir que un diseñador de partes determine si una parte que se necesita existe en realidad o podría producirse modificando una parte análoga. La fig 16-3 ilustra el trazo de un diseñador de una placa con cinco agujeros y una pa de ajuste encontrada por el programa. En este ejemplo, el diseñador podría decidir que la parte de ajuste está lo suficientemente cerca para que pudiera ser modificada agregando o sumando los cuatro agujeros menores. Para establecer el tamaño de los modelos de entrada, podría pedirse al usuario que trace objetos dentro de los límites de una retícula desplegada.

Los programas de pintura pueden servir para desplegar algunos tipos de redes (proyectos de tubería y alambrado) y en aplicaciones de arte creativo. En el trazo de redes, el usuario dibuja las líneas de conexión y el programa ajusta el trazo en un proyecto de conexiones horizontales y verticales (fig. 16-4). En aplicaciones de arte, al usuario se le pueden ofrecer dos menús y un área en la cual pueda crear la imagen, como se muestra en la figura 16-5. Un menú presenta un conjunto de pinceles que permiten la creación de diferentes tamaños de línea y texturas de líneas (líneas estrechas, aerógrafo, etc.). El otro menú enlista una paleta de colores y sombras a disposición. Seleccionando diferentes pinceles y colores, un artista puede producir cuadros como los ejemplos de arte creativo del capítulo 1.



(a)



(b)

FIGURA 16-3
Un patrón de entrada (a) trazado por un diseñador de partes se compara contra la biblioteca de partes existentes para localizar una figura similar (b).

16-4 Diseño del menú

Muchos paquetes de gráficas utilizan menús. En algunos casos, toda la entrada del usuario se especifica con selecciones de menú. Cuando se emplean menús en un programa, el usuario es liberado de la carga de recordar opciones de entrada. Esto no sólo reduce la cantidad de memorización que se requiere del usuario enlistando la gama de opciones disponibles, sino que también impide que el usuario seleccione opciones que no sean válidas en ese punto. Además, los menús pueden alterarse fácilmente para dar cabida a diferentes aplicaciones, mientras que las llaves o botones de función deben reprogramarse y volver a rotularse si van a alterarse. Los menús pueden utilizarse como el mecanismo de entrada de operaciones y valores de parámetros.

La selección interactiva del menú puede realizarse con muchos tipos de dispositivos de entrada. La pluma linterna y el pánel de contacto se utilizan para hacer selecciones muy rápidas, ya que el usuario simplemente toca el área de la pantalla que contiene la opción de menú que se seleccionará. Pueden utilizarse teclados, palancas de mando y otros dispositivos para hacer selecciones posicionando un cursor u otro símbolo en una opción de menú. También puede utilizarse un te-

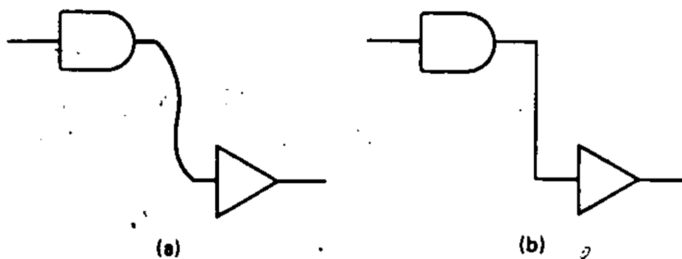


FIGURA 16-4
En un programa de trazo de redes, el trazo de un usuario de una línea de conexión (a) se ajusta a un conjunto de líneas horizontales y verticales (b).

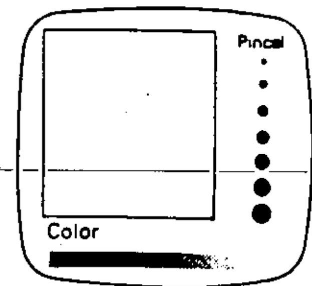
clado para teclear el nombre de identificación o el número de un elemento del menú. Para un número pequeño de elementos de menú, un sistema con registro de voz puede ofrecer un método eficaz para seleccionar opciones.

En general, los menús con menos opciones son más efectivos, ya que reducen la cantidad de tiempo de búsqueda que se necesita para hallar una opción determinada y ocupan menos espacio en la pantalla. Por lo general, los menús se colocan en un lado de la pantalla de manera que no interfieran con la imagen desplegada. Cuando se va a presentar un menú extenso con largas descripciones de cada opción, quizá tenga que ocupar toda la pantalla tal que la imagen y el menú se desplieguen en forma alternada. Esto puede tener un efecto muy disruptivo en la sucesión de ideas del usuario, ya que la continuidad visual con la imagen se pierde cada vez que se hace una selección de menú. Los menús que ocupan toda la pantalla deben evitarse acortando las descripciones de las opciones de menú y dividiendo las selecciones en dos o más submenús.

Un método para organizar un menú en submenús de menor tamaño consiste en disponer las opciones de menú en una estructura de niveles múltiples. Una selección del primer menú sube un menú al segundo nivel, y así sucesivamente. Este es un método efectivo para trabajar con operaciones y parámetros. Una vez que se selecciona una acción del menú de operación, puede presentarse un menú de parámetros (escalas o cuadrantes). Deben evitarse más de dos o tres niveles; de lo contrario, la estructura del menú podría confundir al usuario.

Los elementos listados en un menú pueden presentarse como cadenas de caracteres o bien como íconos gráficos (figuras geométricas). Tales símbolos son de utilidad en la presentación de menús para el diseño de circuitos y la planificación de proyectos de instalaciones. Los íconos pueden usarse asimismo para describir los significados de algunas acciones, como se ilustra en la figura 16-6. En este ejemplo se utiliza una flecha recta para indicar una dirección de traslación y una flecha curva indica la dirección de rotación. Las ventajas de los íconos son que por lo general ocupan menos espacio y se reorganizan más rápidamente que las descripciones de texto correspondientes. Un usuario principiante puede hallar los íconos más difíciles de usar inicialmente, pero una vez que se aprenda el conjunto de íconos, las entradas pueden hacerse más rápido y con pocos errores.

En muchas aplicaciones, los menús siempre se colocan en una posición, como en el lado o en la base de la pantalla. Los elementos siempre se ubican en la misma posición, de modo que el usuario se acostumbre a hacer cada selección en una localidad fija. Otra posibilidad es la de utilizar un menú "movible" que se coloca en la proximidad de la posición regular del cursor de la pantalla (fig. 16-7). Puesto que se supone que el usuario está trabajando con el cursor de la pantalla, los menús movibles permiten que se hagan selecciones con un mínimo de movimiento del ojo y la mano. El menú movible se coloca sobre una parte de la imagen y de este modo debe desplegarse sólo cuando se necesite. Los menús movibles pueden implantarse en un sistema rastreador sustituyendo una parte rectangular de la imagen por la información del menú. Con un sistema vectorial, el menú y la imagen pueden superponerse y dificultar al usuario la identificación de los elementos del menú.



(a)

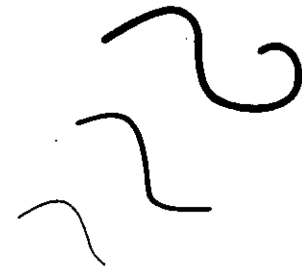
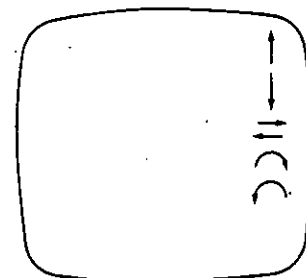

(b)

FIGURA 16-5
(a) Menús utilizados en un programa de pintura para seleccionar colores y texturas y tamaños de pinceles. (b) Líneas trazadas con pinceles de diferente tamaño.

FIGURA 16-6
Iconos que podrían utilizarse para seleccionar direcciones de traslación y rotación.



16-5 Retroalimentación

Una parte importante de cualquier sistema de gráficas es la cantidad de retroalimentación suministrada a un usuario. El sistema necesita realizar un diálogo in-

teractivo e informar al usuario qué está haciendo el sistema en cada etapa. Esto es particularmente importante cuando el tiempo de respuesta es alto. Sin retroalimentación, un usuario puede empezar a preguntarse qué está haciendo el sistema y si la entrada debe volver a darse.

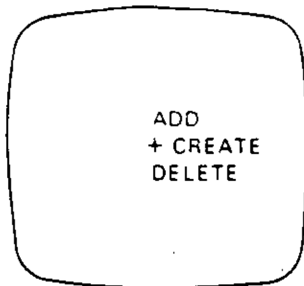


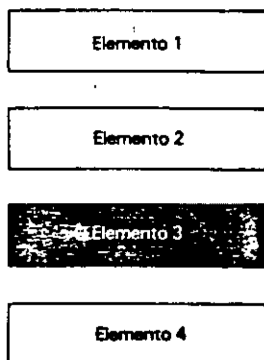
FIGURA 16-7
Menú movible colocado cerca de la posición actual del cursor de la pantalla.

Conforme cada entrada del usuario es recibida, debe aparecer una respuesta inmediatamente en la pantalla. El mensaje debe ser breve e indicar con clarid el tipo de procesamiento en progreso. Esto no sólo informa al usuario que la e. trada se ha recibido, sino que también le indica en qué está trabajando el sistema de manera que pueda corregirse cualquier error de entrada. Si el procesamiento no puede completarse en unos cuantos segundos, podrían desplegarse varios mensajes para mantener informado al usuario del progreso del sistema. En algunos casos esto podría ser un mensaje centelleante para indicar al usuario que el sistema sigue trabajando en la solicitud de entrada. También puede ser posible que el sistema despliegue resultados parciales a medida que se completan, de manera que el despliegue final se construya una pieza a la vez. El sistema podría también permitir al usuario meter otros comandos o datos mientras se procesa una instrucción.

Los mensajes de retroalimentación deben darse con la claridad suficiente para que tengan poca oportunidad de ser pasados por alto. Por el otro lado, no deben ser tan abrumadores que se interrumpa la concentración del usuario. Con llaves de función, la retroalimentación puede darse como un chasquido audible o mediante la iluminación de la llave que se ha oprimido. La retroalimentación de audio tiene la ventaja que no utiliza más del espacio de la pantalla y el usuario no tiene que distraer su atención del área de trabajo con el fin de recibir el mensaje. Cuando se despliegan mensajes en la pantalla, puede usarse un área de mensaje fija de manera que sepa siempre dónde buscar los mensajes. En algunos casos, puede ser útil colocar mensajes de retr alimentación en el área de trabajo del usuario en la proximidad del cursor. Pueden usarse diferentes colores para distinguir la retroalimentación de otros objetos del despliegue.

Para acelerar la respuesta del sistema, pueden elegirse técnicas de retroalimentación para aprovechar las características de operación del tipo de disposit vos en uso. Con un despliegue rastreador, las intensidades de los pixeles se invierten con facilidad (fig. 16-8), de modo que este método puede emplearse para ofrecer una rápida retroalimentación para selecciones de menú. Otros métodos que podrían utilizarse para la retroalimentación de la selección del menú incluyen el realce, cambio de color y centelleo del elemento seleccionado.

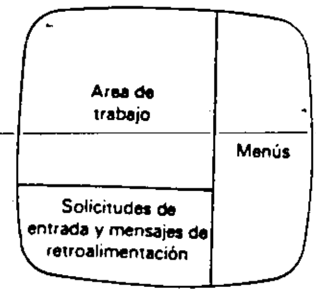
FIGURA 16-8
Inversión de intensidades de pixeles como medio para ofrecer retroalimentación para confirmar una selección de menú.



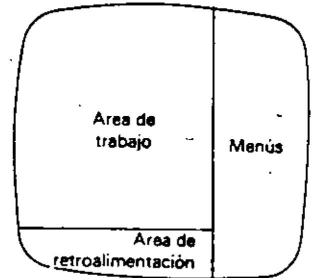
Pueden diseñarse símbolos especiales para diferentes tipos de retroalimentación. Por ejemplo, un símbolo de cruz o de "pulgares abajo" podría usarse para indicar un error y un signo centelleante "en trabajo" podría servir para indicar al usuario que el sistema está procesando la entrada. Este tipo de retroalimentación puede ser muy efectiva con un usuario más experimentado, pero el principiante puede necesitar una retroalimentación más detallada que no sólo indique con claridad lo que el sistema está haciendo sino también lo que el usuario debe meter a continuación.

Con algunos tipos de entrada, se desea la retroalimentación de eco. Los caracteres tecleados pueden desplegarse en la pantalla conforme se introducen de modo que un usuario pueda detectar y corregir errores de inmediato. La entrada con botón y cuadrante puede reflejarse en la misma forma. Los valores escalares que se seleccionan con cuadrantes o bien de escalas desplegadas por lo general se reflejan en la pantalla para permitir que el usuario verifique la exactitud de los valores de entrada. Las selecciones de posiciones podrían reflejarse con un cursor u

otro símbolo que aparezca en la posición seleccionada. Si se seleccionan posiciones con el cursor de la pantalla, pueden desplegarse valores coordenados de la localidad del cursor para indicar exactamente qué posición se ha seleccionado.



(a)



(b)

FIGURA 16-9
Posibilidades del proyecto en la pantalla: (a) área de mensajes grande; (b) área de mensajes reducida.

16-6 Formatos de salida

La información que se presenta al usuario de un paquete de gráficas incluye una combinación de imágenes, menús, mensajes de salida y otras formas de diálogo generadas por el sistema. Existen muchas posibilidades para disponer y presentar esta información de salida al usuario y el diseñador de un paquete de gráficas debe considerar la mejor manera de diseñar los formatos de salida para lograr la mayor efectividad visual. Las consideraciones en el diseño de formatos de salida incluyen estructuras de menú y mensajes, figuras de íconos y símbolos y los proyectos en toda la pantalla. Como se dijo antes, las estructuras de menú y mensajes dependen de varios factores. Además del nivel de experiencia del usuario y del tipo de aplicación, la estructura de menús y mensajes se verá influenciada por el proyecto elegido para la salida de la pantalla.

Formas de iconos y símbolos

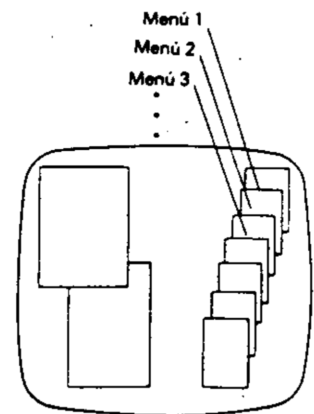
La estructura de muchos símbolos que se utilizan en un paquete de gráficas depende del tipo de aplicación para la cual se dirige el paquete, como el diseño eléctrico, planificación arquitectónica o proyectos de instalaciones. Las formas o figuras de los símbolos se escogen de modo que ofrezcan una imagen todavía más clara y simple del objeto u operación que se supone deben representar. Otros símbolos, como los cursores o apuntadores de menú, deben diseñarse para que sean claramente diferentes de los otros íconos. En algunos casos, un paquete puede permitir al usuario especificar la forma de algunos símbolos.

Proyecto de la pantalla

Tres componentes básicas del proyecto de la pantalla son el área de trabajo del usuario, el área de menú y el área para desplegar solicitudes de entrada y mensajes de retroalimentación. Las secciones fijas de la pantalla pueden designarse para cada una de estas tres áreas, como se muestra en la figura 16-9(a). En este proyecto, los menús siempre se presentan del lado derecho y los mensajes se despliegan en la base de la pantalla. Para hacer el área de trabajo lo más grande posible, las áreas de menú y mensajes deben minimizarse. Si se desea un área de trabajo muy grande, las áres de menú y mensajes podrían suprimirse cuando no se necesiten de manera qué el área de trabajo pueda ampliarse hasta llenar la pantalla. Una manera de permitir un empleo máximo de la pantalla consiste en dar al usuario algún control sobre el tamaño de las áreas de menú y mensajes. Por ejemplo, un usuario experimentado podría reducir el tamaño del área de mensajes eliminando la solicitud de entrada innecesaria, como en la figura 16-9(b).

Puede ofrecerse mayor flexibilidad al organizar proyectos en la pantalla al usuario permitiendo que se forme cualquier número de áreas de ventana en sobreposición. En este esquema, el usuario especifica el área de la pantalla en la cual se desplegará un menú o imagen. A medida que se coloca cada nueva ventana en la pantalla, puede superponerse y oscurecer ventanas creadas con anterioridad. La figura 16-10 muestra un proyecto con varias ventanas en superposición.

FIGURA 16-10
Proyecto de la pantalla con ventanas en superposición.



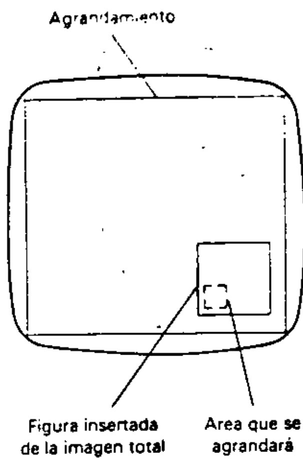


FIGURA 16-11
Sección de una imagen agrandada que contiene una figura insertada que muestra la imagen total.

Debe proporcionarse un conjunto de operaciones al usuario para crear, suprimir y posicionar las áreas de ventanas.

Con algunas aplicaciones conviene tener una capacidad de acercamiento a disposición. Esto permite a un usuario ampliar partes seleccionadas de una imagen o bien mostrar una vista más amplia de una escena, como si el usuario se alejara de la pantalla. La característica de acercamiento puede utilizarse para ampliar una pequeña zona de una imagen mayor de modo que llene el área de trabajo. El agrandamiento puede realizarse para desplegar detalles adicionales que no pueden apreciarse en la imagen total. Algunas veces el usuario desearía observar la imagen en su totalidad y la sección ampliada al mismo tiempo. Una manera de hacer esto consiste en superimponer una imagen total como una pequeña ventana interior en la pantalla, como se muestra en la figura 16-11. Otro método consiste en incluir dos monitores de video en el sistema de gráficas. Puede usarse un DV-ST para desplegar la imagen total y el despliegue de renovación puede emplearse para mostrar agrandamientos.

Además de una capacidad de acercamiento, en algunas aplicaciones de modelado conviene poder simplificar un despliegue eliminando algo del detalle a fin de ofrecer una mejor presentación de la estructura esencial. Por ejemplo, un proyecto de una ciudad podría mostrarse con todas las calles o bien solamente con las principales. Al desplegar sólo las calles más grandes, pueden destacarse las principales calles transitadas de la ciudad (fig. 16-12).

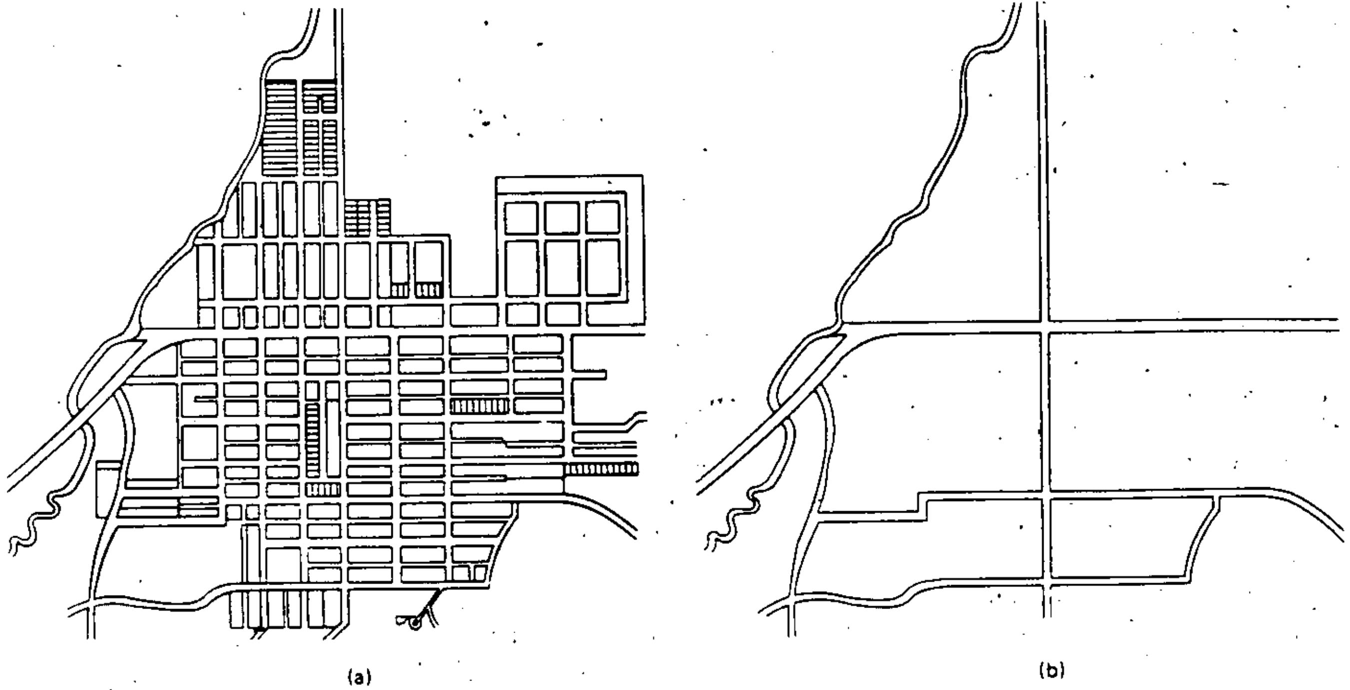


FIGURA 16-12
Despliegue del proyecto de calles de una ciudad: (a) se incluyen todas las calles; (b) sólo se destacan las principales calles transitadas.

12

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS INSTITUCIONALES

No. 70

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

DEL 9 AL 29 DE SEPTIEMBRE

SEPOMEX

PROGRAMACION ESTRUCTURADA

MEXICO D.F.
PALACIO DE MINERIA
1992

4.- PROGRAMACION ESTRUCTURADA

4.1 Origenes y objetivos de la programación estructurada

4.2 El proceso de refinamiento a pasos

4.3 El seudocódigo

4.4 Arboles de decisión

4.5 Diagramas estructurados

4.6 Documentación de programas

4.7 Estructuras de control en FORTRAN IV.

4.1 ORIGENES Y OBJETIVOS DE LA PROGRAMACION ESTRUCTURADA

Dos artículos de Dijkstra marcan el inicio de la programación estructurada, estos son:

- Programming considered as a human activity (Dijkstra - 1965)

- GO TO statement considered harmful (Dijkstra 1968)

En estos dos artículos ya Dijkstra habla de algunos conceptos e ideas que han llegado a ser relevantes en la programación estructurada tales como:

- Argumentos contra GO TO
- La noción del diseño de arriba-a-abajo
- El énfasis en la calidad de programas
- Argumentos contra programas que se modifican a si mismos.
- Otros

La programación estructurada surge como una necesidad de reducir la complejidad de los grandes programas.
Con frecuencia se observa que los programas, debido a su complejidad, no

1.- satisfacen las necesidades del usuario
2.- no se producen a tiempo
3.- cuestan mas de lo estimado
4.- contienen errores y
5.- son difíciles de darles mantenimiento

Es por ello que la programación estructurada se propone lograr los siguientes objetivos:

1.- Que los programas satisfagan los requerimientos especificados.
2.- Programas que se les pueda dar mantenimiento
3.- Minimizar el número de errores que ocurren durante el desarrollo.
4.- Operación resistente a errores.

5.- Programas transportables

6.- Programas cuya lógica sea legible

7.- Minimizar costos

4.2 EL PROCESO DE REFINAMIENTO A PASOS

Nuestra mayor herramienta en el desarrollo de programas es nuestra capacidad de abstracción.
Básicamente esto consiste (cuando resolvemos un problema) en analizar cuidadosamente el problema dado hasta lograr - una representación mental de los elementos escenciales del mismo.
Cada elemento debe tener un propósito bien definido. Este proceso puede aplicarse a su vez a cada uno de los elementos del problema original, es decir, consideramos a cada elemento como un nuevo problema (subproblema). El proceso puede repetirse para los elementos del subproblema y asi sucesivamente.
Inicialmente (en la resolución del problema) estamos concentrados en que funciones (o elementos) descomponer el problema, y a medida que procedemos a descomponerlos sistemáticamente llegamos a concentrarnos en como realizar cada función.
Esta forma de proceder va de lo general a lo particular.
El proceso descrito anteriormente se conoce como proceso de refinamiento a pasos y es central a la programación estructurada.
El proceso de refinamiento a pasos no es directo, ni trivial, como pudiera parecer a primera vista.
Este proceso es escencialmente un proceso de ensayo y error.

Es evidente que necesitamos una notación como herramienta - para ir concretando la solución del proeblema, ya que a medida que se refina sistemáticamente la solución del problema, nuestra limitada capacidad de retención será desbordada por la gran cantidad de detalles involucrados. Una herramienta muy utilizada para este propósito es el seudcódigo o lenguaje de diseño de programas. El seudocódigo debe utilizarse como una extensión natural del proceso de abstracción.

4.3 EL SEUDOCODIGO

El seudocódigo o seudolenguaje es un lenguaje intermedio -- entre el lenguaje nativo del programador y el lenguaje de - programación en que se intenta implantar la solución. El seudocódigo le permite al programador pensar en la lógica y y expresar esa lógica en una forma semiformal sin tener que adentrarse en los detalles particulares de un lenguaje de - programación.

Básicamente el seudocódigo difiere en 2 aspectos de un lenguaje de programación.

1.- No existen restricciones sintácticas para el uso del -- seudocódigo. Solo las estructuras de control básicas y el sangrado para mejorar la claridad del alcance de - dichas estructuras, son las únicas convenciones aceptadas para el uso del seudocódigo.

2.- Cualquier operación se puede expresar a cualquier nivel de detalle, por ejemplo:

```
incrementar   contador
ctdor ← ctdor + 1
```

El seudocódigo permite al programador tratar el problema a diversos niveles de abstracción, ya que esta es la herramienta mediante la cual expresamos la solución de un problema durante el proceso de refinamiento a pasos, esto es, el - seudocódigo es un lenguaje de diseño de programas (PDL).

El seudocódigo es una forma conveniente para documentar los estados de desarrollo del programa, lo cual permite a otros programadores revisar la función del programa antes de implementarlo en un lenguaje de programación, además de facilitar una valoración del estado de desarrollo en cualquier estado del proceso de diseño de programa.

Debido a que el seudocódigo describe detalladamente el -- programa fuente completo, puede mantenerse como parte de la documentación del programa. Siendo así, el seudocó-- digo debe actualizarse cada vez que haya cambios en el -- programa fuente.

Existen pocas guías generales que hacen del seudocódigo - efectivo como una herramienta de diseño de programas. Estas son:

1.- Haga del seudocódigo una extensión del proceso del -- pensamiento.

2.- Sangre el código para resaltar la estructura de la -- lógica.

3.- De nombres a los datos de tal manera que reflejen su_ intención.

4.- Mantenga la lógica simple

5.- Utilice las estructuras de control básico permitidos en el proyecto.

Hay cuatro clases de estructuras básicas con las cuales - es posible especificar un procedimiento en seudocódigo.

a) Secuencia
b) Decisión
c) Iterativas
d) Salida

a) Secuencia

Concatenación

b) Decisión

```
1.- si (predicado) luego
        seudocódigo
    fin
```

```
2.- si (predicado) luego
        seudocódigo-A
    obien
        seudocódigo-B
    fin

3.- si (predicado-A) luego
        seudocódigo-A
    osi (predicado-B) luego
         seudocódigo-B
    osi (predicado-C) luego
              .
              .
              .
    obien
         seudocódigo-X luego
    fin
4.- Casar (selector) con
    (caso 1)
           seudocódigo-A
    (caso 2)
           seudocódigo-B
                .
                .
                .
    (obien)
           seudocódigo-X
    fin
```

c) Iteración

```
1- Entanto (predicado) luego
           seudocódigo
   fin

2- Repetir
           seudocódigo
   Hasta (predicado)
```

3- Desde (vc ← valor-1 hasta valor-2) repetir

seudocódigo

fin

d) Salida

1- escape

2- ciclo

4.4. ARBOLES DE DECISION

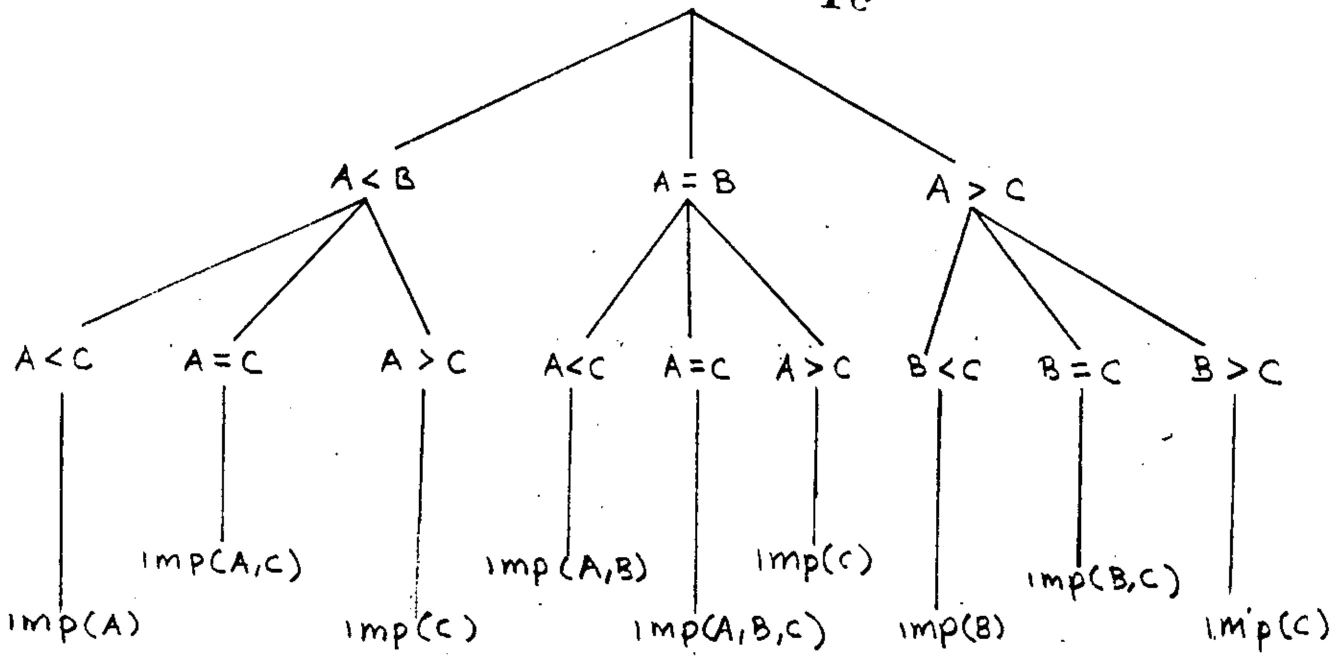
Los árboles de decisión son herramientas útiles que sirven para esclarecer, de manera sistemática, la lógica de segmentos de programa en los que aparecen condicionales anidados. El ejemplo siguiente ilustra el uso de esta herramienta.

Suponga que se quiere imprimir el menor valor de las 3 variables A, B y C de la siguiente manera:
Si una de las 3 variables es menor a las otras dos se imprime. Si dos de ellas son iguales y menores a la tercera se imprimen y si las 3 son iguales se imprimen.

Comparando primero A y B para todos los casos posibles tenemos:

$A<B$ $A=B$ $A>B$

para el primer caso se sabe que A es menor que B y por consiguiente enseguida se debe comparar A con C para todos los casos posibles. Para el segundo caso se sabe que A es igual que B y por consiguiente enseguida se debe comparar A o B con C para todos los casos posibles. Para el tercer caso se sabe que A es mayor que B y por consiguiente enseguida se debe comparar B on C para todos los casos posibles. Estas consideraciones pueden expresarse de forma gráfica de la manera siguiente:



Las líneas indican la relación jerárquica que existe en-- tre las condiciones. Observe que las acciones que se to man una vez dada una condición o condiciones aparecen en el extremo inferior del árbol.

El mapeo de la lógica del árbol de decisiones a seudocódigo es directa.

```
si (A<B) luego
    si(A<C) luego
        imp(A)
    osi (A=C) luego
        impc(A,C)
    obien
        imp(C)
    fin
osi (A=B) luego
    si(A<C) luego
        imp(A,B)
    osi(A=C) luego
        imp(A,B,C)
    obien
        imp(c)
    fin
obien
    si(B<C) luego
        imp(B)
    osi(B=C) luego
        imp(B,C)
    obien
        imp(C)
    fin
fin
```

La correspondencia entre el árbol de decisión y el seudocódigo es evidente.

a

TABLAS DE DECISION

Una tabla de decisión, al igual que un arbol, es una - herramienta que permite escalarece la lógica de condiciona-- les complejos.

A diferencia de un arbol, una tabla se deriva de una - manera mucho mas sistemática, ademas de tener un formato mu- cho mas compacto.

Existen 3 tipos de trablas de decisión

a) Entrada limitada
b) Entrada extendida
c) Entrada mixta

El siguiente diagrama muestra el formato de una tabla de decisión:

PARTE DE CONDICION (1)	ENTRADA DE CONDICION (3)
PARTE DE ACCION (2)	ENTRADA DE ACCION (4)

1. En el cuadrante superior izquierdo esta la parte de condición. Esta área debe contener (en forma de pregunta) todas aquellas condiciones examinadas para un problema dado.

2. En el cuadrante inferior izquierdo esta la parte de acción. Esta área debe contener en forma de narrativa simple todas las acciones posibles resultantes de las condiciones listadas arriba.

3. En el cuadrante superior derecho esta la entrada de condición. En esta área todas las preguntas hechas en la parte de condición deben responderse y todas las combinaciones posibles de estas respuestas deben desarrollarse. Si una respuesta no se indica, puede asumirse que la condición no fue sometida a prueba en esa combinación particular.

4. En el cuadrante inferior derecho esta la entrada de acción. Las acciones apropiadas resultantes de las diversas condiciones de las respuestas a las condiciones de arriba se indican aquí.Una o mas acciones pueden indicarse para cada -- combinación de respuestas de condición.

   Las diversas combinaciones de respuestas a condiciones mostradas en la entrada de condición de la tabla y sus acciones resultantes se llaman reglas. A cada una se le da un número para propósitos de identificación.

   Otro elemento que requiere una tabla como un medio de distinguirla de otra tabla es un nombre.

Tablas de entrada limitada

Como un ejemplo simple de una tabla de entrada limitada, -- considerese el siguiente problema:

Se quiere escribir un procedimiento para una persona que -- indique en forma explicita que acciones tomar cuando sale de - su cada en la mañana a su trabajo de acuerdo a las condiciones del tiempo.

Supóngase que las condiciones del tiempo son dia normal, -- llueve y hace frio y las acciones correspondientes son vestimenta normal, llevar paraguas y llevar sweater.

La tabla de decisión correspondiente queda como sigue:

¿LLUEVE?	S	S	N	N	
¿HACE FRIO?	S	N	S	N	
LLEVAR PARAGUAS	X	X			
HACE FRIO	X		X		
VESTIMENTA NORMAL			X		

Observese que en la entrada de condición sólo se tienen - 2 valores: S(si) y N(no) y en la entrada de acción una X que indica la acción que se toma de acuerdo a la condición o combinación de condiciones que se indica en la parte de condición. Así por ejemplo, la columna R1 (regla 1) se lee como sigue:

si ¿lleve y hace frío? luego

llevar paraguas y sweater

La columna R2 (regla 2)

si llueve y no hace frío luego

llevar paraguas

y así sucesivamente.

Tablas de entrada extendida

Una tabla de entrada extendida tiene en su entrada de condición mas de 2 valores, es decir, no esta limitada a SIS y NOS. Considérese el siguiente ejemplo:

Supóngase que en una línea aérea se tienen 3 clases de pasaje: primera, turista y alterna. Si un pasajero solicita la -- clase turista y hay disponible se le otorga, sino se le asigna la clase alterna. Si la clase alterna se agotó se pone en lista de espera. Si el cliente solicita clase primera se le -- otorga si la hay, en caso contrario se le asigna la clase alterna. Si no hay tampoco clase alterna se le pone en lista de es--

pera.

La tabla correspondiente de decisión, queda como sigue:

¿QUE PETICION?	PP	PP	PP	PT	PT	PT	
¿QUE CLASE ?	CP	CT	CA	CP	CT	CA	
SE DA TURISTA					X		
SE DA PRIMERA	X						
SE DA ALTERNA			X				
PONER EN LISTA DE ESPERA		X		X		X	

PP = PETICION DE PRIMERA
PT = PETICION DE TURISTA
CP = CLASE PRIMERA
CT = CLASE TURISTA
CA = CLASE ALTERNA

Observese que en la entrada de condición se tienen 2 valores para la condición ¿que petición? (PP y PT) y para la condición ¿que clase? se tienen 3 valores (CP, CT y CA). La columna R1 (regla 1) se lee como sigue:

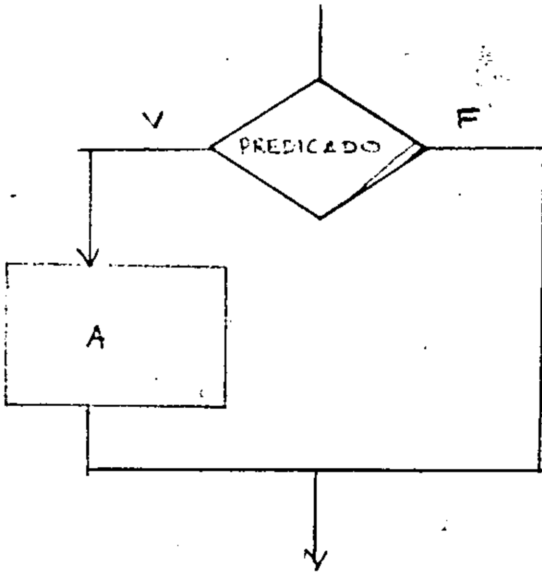
Si la petición es de primera y
la clase primera esta disponble luego
asignar clase primera

Tabla de entrada mixta

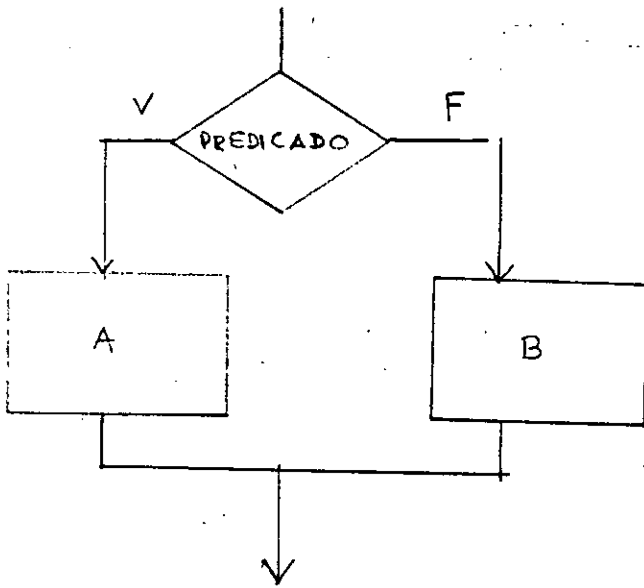
Una tabla de entrada mixta puede tener SIS y NOS y otros valores en la entrada de condición. Sí por ejemplo en la tabla anterior en la primera condición ¿qué petición? se podría poner SIS y NOS en su correspondiente entrada de condición, ya

que esta entrada tiene 2 valores PP y PT.

DESICION

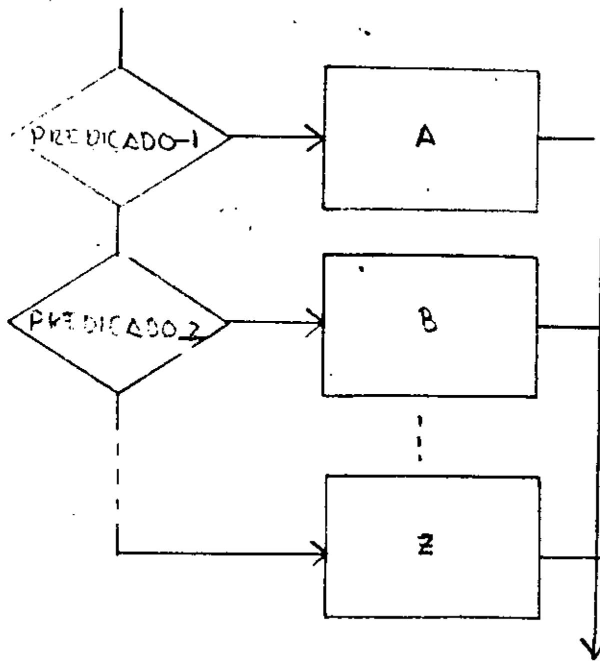



```
si (predicado) luego
      seudocódigo-A
fin
```

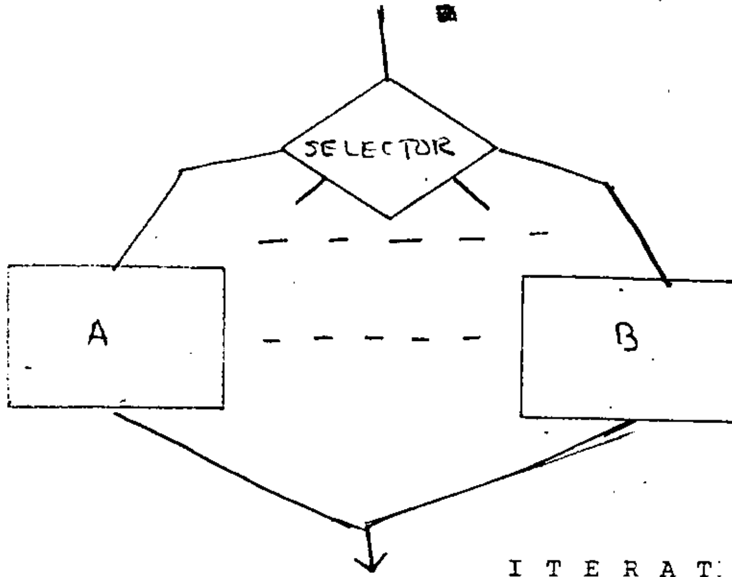



```
si (predicado) luego
      seudocódigo-A
o bien
      seudocódigo-B
fin
```

o

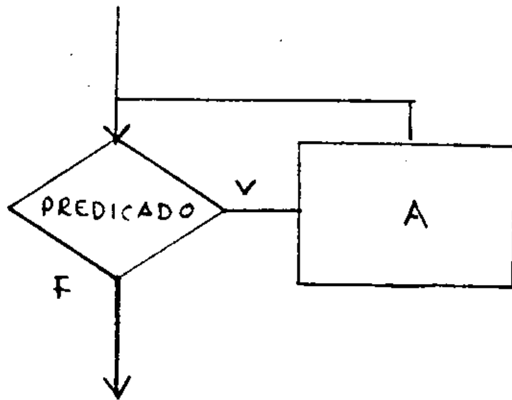



```
si (predicado) luego
      seudocódigo-A
as' (predicado) luego
      seudocódigo-B
.          .
.          .
.          .
o bien
      seudocódigo-Z
fin
```



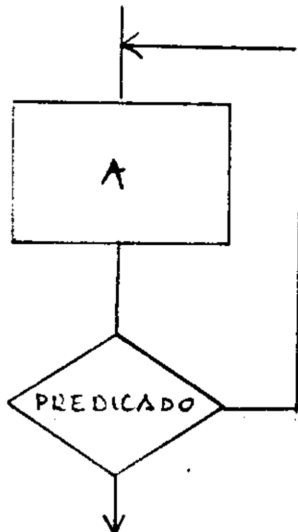

```
casar (selector) con
(caso1)
      seudocódigo-A

   .
   .
   .
(o bien)
      seudocódigo-Z
fin
```

I T E R A T I V A S



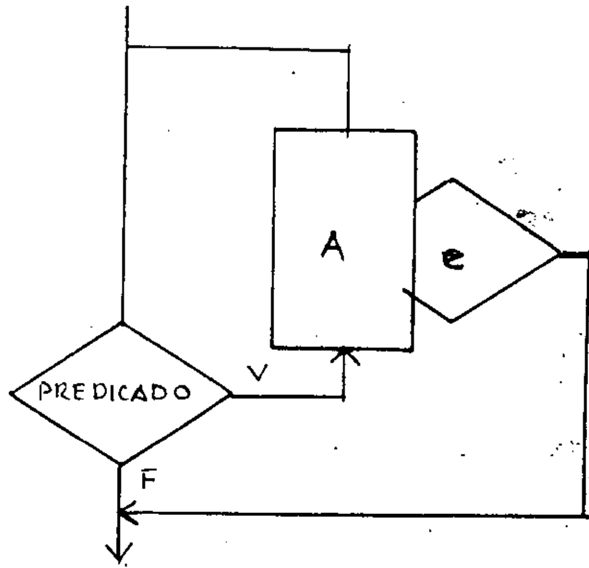

```
entanto (predicado) repetir
      seudocódigo-A
fin
```

o

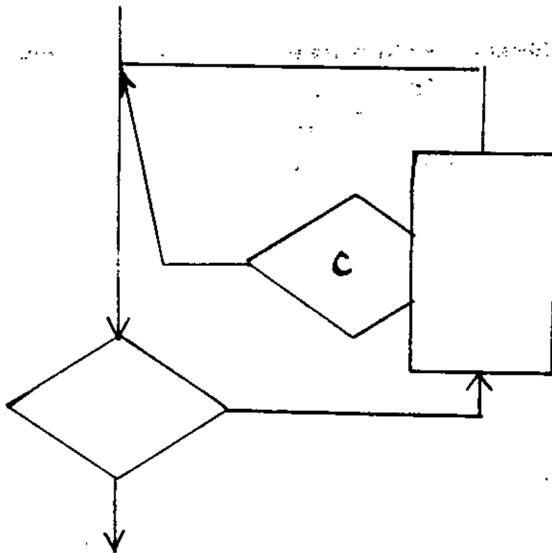



```
repetir
      seudocódigo-A
hasta(predicado)
```

```
desde(ve← valor-1 hasta valor-2)
      repetir
      seudocódigo-A
fin
```

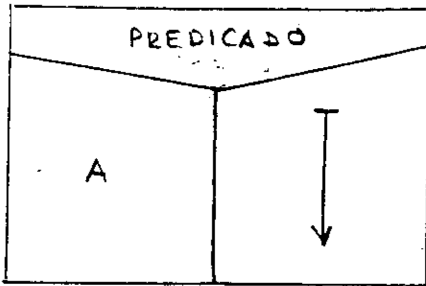


escape

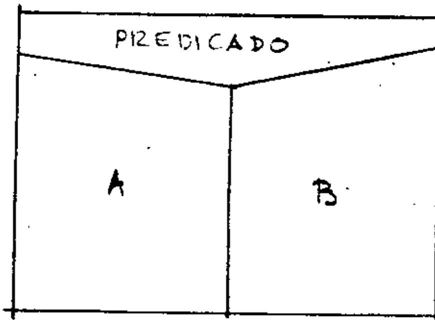


ciclo

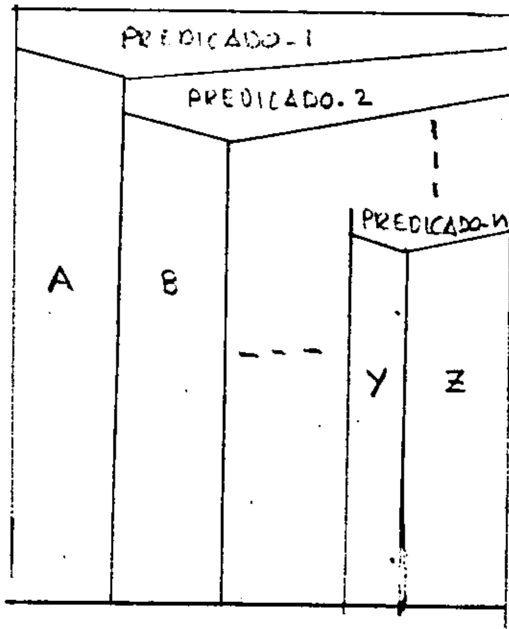
DESICION



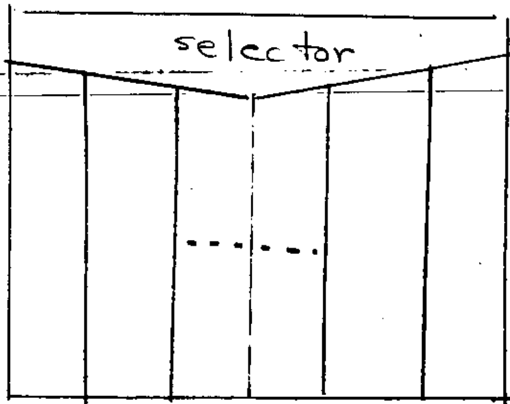

```
si (predicado) luego
      seudocódigo-A
fin
```




```
si (predicado) luego
      seudocódigo-A
o bien
      seudocódigo-B
fin
```

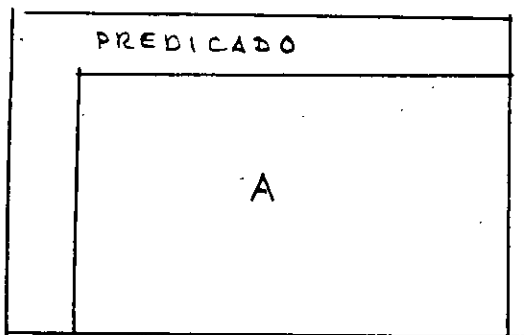



```
si (predicado) luego
      seudocódigo-A
osi (predicado) luego
      seudocódigo-B
.           .
.           .
.           .
o bien
      seudocódigo-Z
fin
```

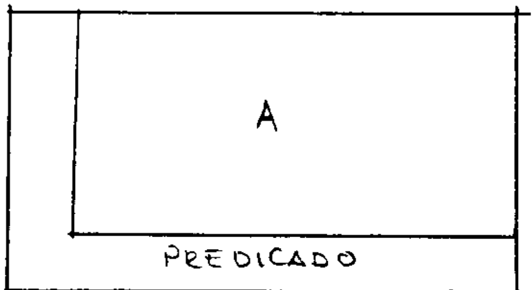



```
casar (selector) con
(caso-1)
      seudocódigo-A
   .           .
   .           .
   .           .
(o bien)
      seudocódigo-Z
fin
```

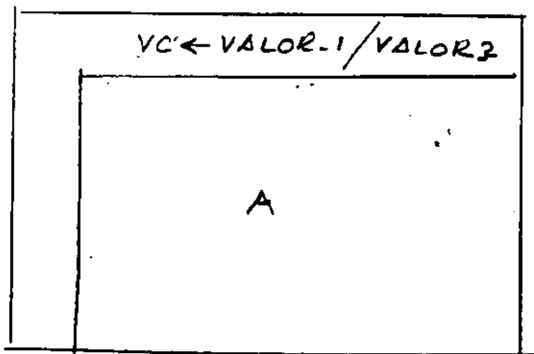
I T E R A T I V A S




```
entanto (predicado) repetir
      seudocódigo-A
fin
```

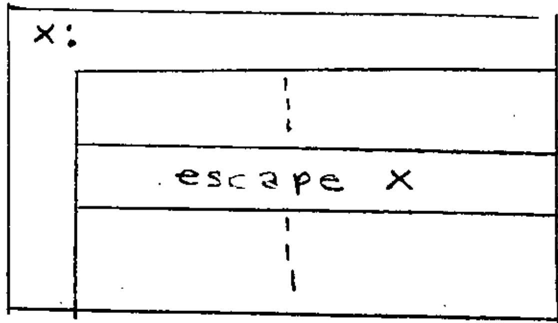



```
repetir
      seudocódigo-A
hasta(predicado)
```

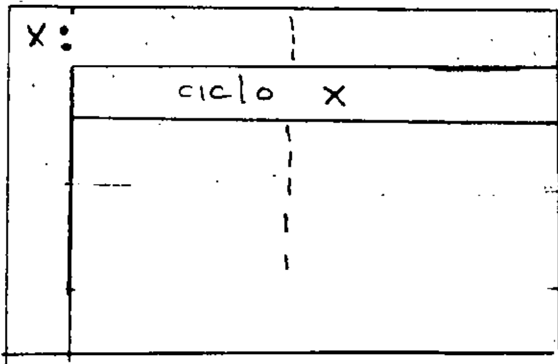



```
desde(ve←valor-1 hasta valor-2
      repetir
      seudocódigo-A
fin
```

S A L I D A

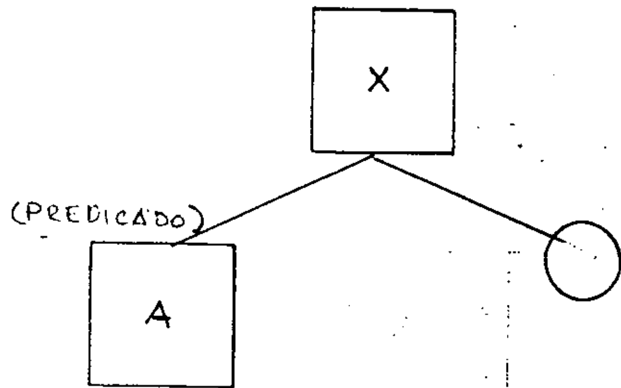


escape

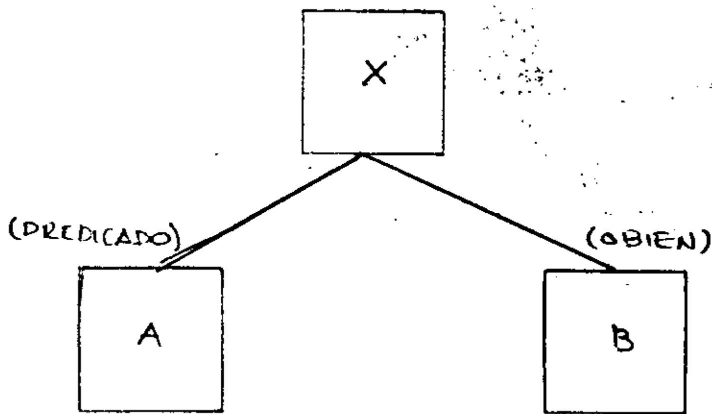


ciclo

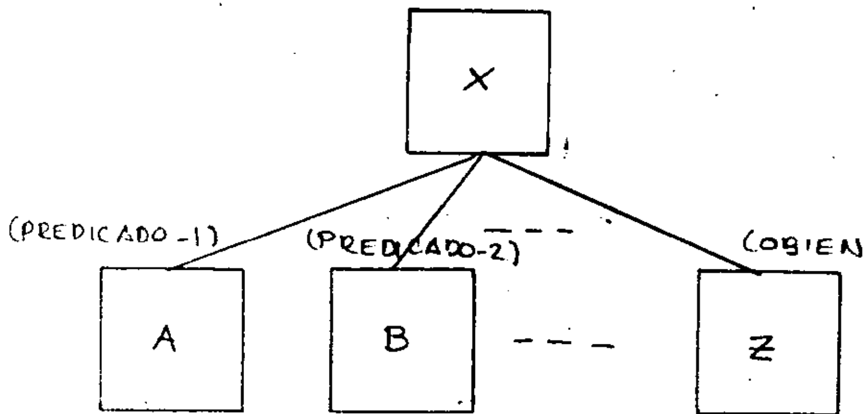
DESICION



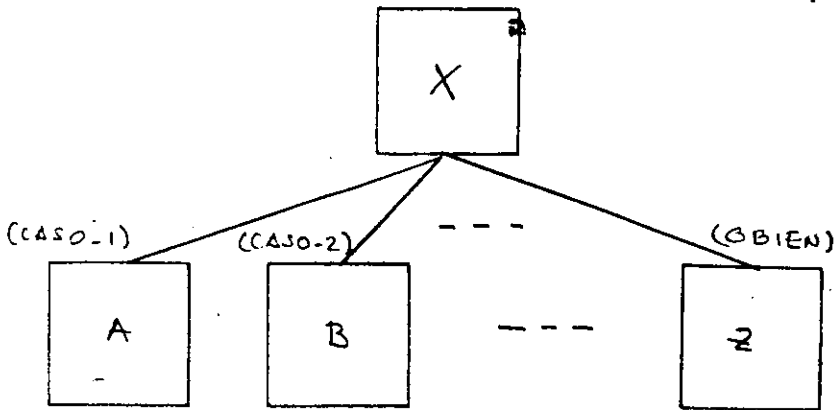

```
si (predicado) luego
      seudocódigo-A
fin
```




```
si (predicado) luego
      seudocódigo-A
o bien
      seudocódigo-B
fin
```



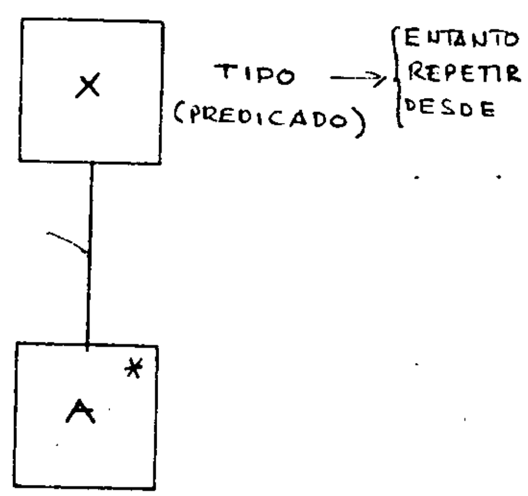

```
si (predicado) luego
      seudocódigo-A
es' (predicado) luego
      seudocódigo-B
.
.
.
o bien
      seudocódigo-Z
fin
```




```
casar (selector) con
(caso1)
      seudocódigo-A
      .      .
      .      .
      .      .
(o bien)
      seudocódigo-Z
fin
```

I T E R A T I V A S

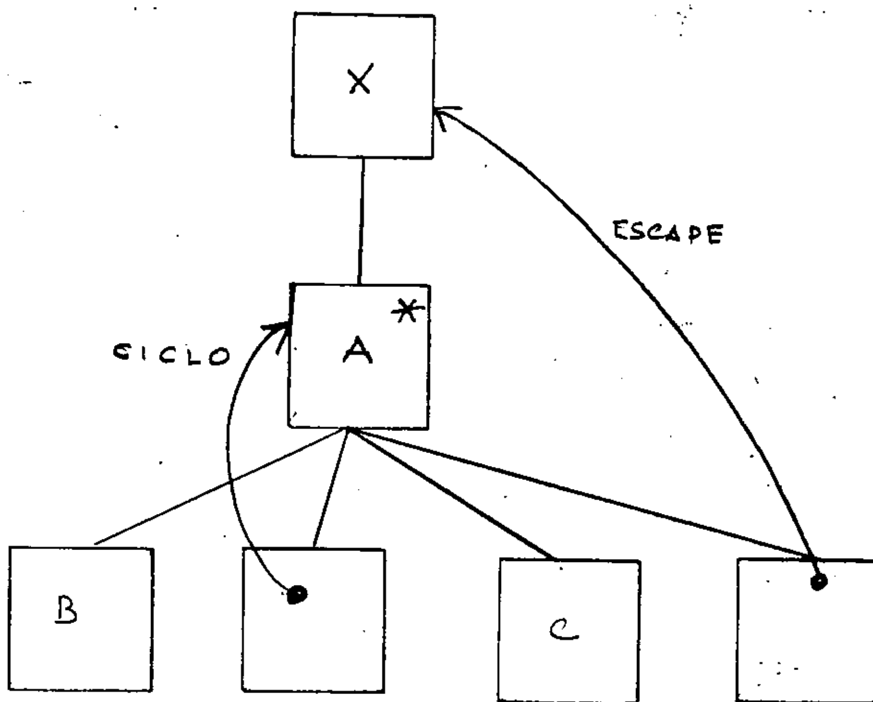
```
entanto (predicado) repetir
      seudocódigo-A
fin
```




```
repetir
      seudocódigo-A
hasta(predicado)
```

```
desde(ve← valor-1 hasta valor-2
      repetir
      seudocódigo-A
fin
```

S A L I D A



escape

ciclo

4.6 DOCUMENTACION DE PROGRAMAS

No es suficiente presentar un programa como una secuencia de instrucciones en un orden dado, ni tampoco es suficiente usar identación para destacar visualmente el alcance de las estructuras de control. Es necesario documentar los programas ya que en muchas ocasiones unas personas codifican un programa y otras le dan mantenimiento.

Hay 2 tipos de documentación de programas:

a) Documentación interna

b) Documentación externa

La documentación interna consiste en comentarios inmersos en el mismo código ejecutable.

A continuación se dan algunas guías para la documentación interna de un programa.

1) Toda variable debe declararse. El nombre de la variable debe reflejar su propósito. Así, por ejemplo, la asignación

```
x:=2.0c*y**2;
```

no nos dice gran cosa, sin embargo si a las variables que intervienen en la asignación se les da un nombre para que reflejen su propósito.

```
circun:=2.0*PI*RADIO**2;
```

ahora es mas obvia la información que nos da;

El cálculo de la longitud de la circunferencia de un círculo.

2) Todo programa, segmento de programa o subprograma que tenga un propósito muy específico debe comenzar con comentario que describa en forma clara y consisa ese propósito.

3) No hacer comentarios inútiles, así por ejemplo

```
(*se incremnta contador*)
Contador:=contador + 1;
```

en este caso el comentario sale sobrando.

La documentación externa se refiere a toda la información que debe estar asentada en un manual de programación. Esta información consiste en lo siguiente

a) Propósito del programa o subprograma
b) Parámetros de entrada y salida
c) Explicar a grandes rasgos la meçánica del programa
d) Listado del programa
e) Advertancias sobre posibles errores fatales; división por cero, etc.
f) Programas o subprogramas que llamen al programa -- (o subprograma) que se documenta.
g) Programas o subprogramas que llama el programa - - (o subprograma) que se documenta.

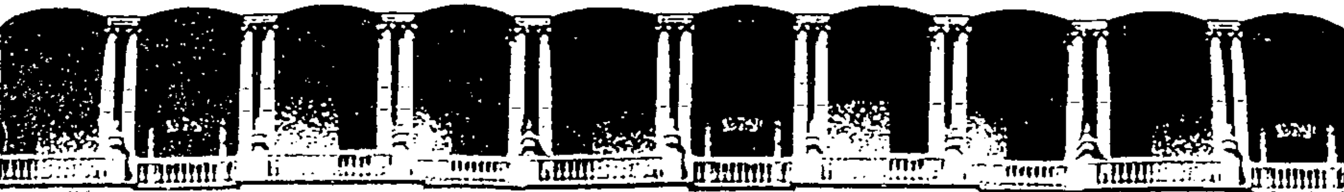
4.7 ESTRUCTURAS DE CONTROL EN FORTRAN IV

DECISION

```
           IF(.NOT. (PREDICADO)) GO TO A
                   CODIGO-A
A          CONTINUE


           IF(.NOT.(PREDICADO)) GO TO A
                   CODIGO-A
                   GO TO B
A          CONTINUE
                   CODIGO-B
B          CONTINUE


           IF(.NOT.(PREDICADO-1)) GO TO A
                   CODIGO-A
                   GO TO X

A          IF(.NOT.(PREDICADO-2)) GO TO B
                   CODIGO-B
                   GO TO X

B          IF(.NOT.(PREDICADO-3)) GO TO C
                   CODIGO-C
                   GO TO X
                      .
                      .
                      .
Y          CONTINUE
                   CODIGO Y
X          CONTINUE
```

```
A      IF(.NOT.(PREDICADO)) GO TO B
              CODIGO-A
              GO TO A
B      CONTINUE

A      CONTINUE
              CODIGO-A
       IF(.NOT.(PREDICADO)) GO TO A
```

```
              SALIDA


              GO TO A
```



FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS INSTITUCIONALES

No. 70

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

DEL 9 AL 29 DE SEPTIEMBRE

SEPOMEX

DESCRIPCION DE LOS PAQUETES CASE

MEXICO, D.F.
PALACIO DE MINERIA
1992

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

III. DESCRIPCION DE LOS PAQUETES CASE

Con el objeto de proporcionar herramientas para el desarrollo de sistemas, durante todo su ciclo de vida, a partir de 1984 aparecen en el mercado los paquetes *CASE*, que han tenido una buena penetración en el mercado. Su principal recurso de venta es el tiempo de desarrollo de una aplicación, estando el ahorro obtenido entre un 35% y un 300%. Adicionalmente se ganaron beneficios tales como sistemas estructurados (en el análisis, el diseño y la programación), estandarizados y completamente documentados, permitiendo un mantenimiento más ágil y sobre todo, inmerso en un plan coherente de desarrollo de la organización. Todo ésto ha acelerado el volumen de ventas como a continuación se muestra: [6,7]

AÑO	VENTAS EN MILLS. DE US. DOLLS.
1984	?
1985	?
1986	$50
1987	$80
1988	$250

Los paquetes *CASE*, son programas que en términos generales proporcionan las siguientes facilidades: [6]

(a) Elaboración de diagramas para representar las especificaciones del sistema en forma gráfica.

(b) Obtención de pantallas y reportes para la creación de especificaciones del sistema y elaboración de prototipos.

(c) **Mantenimiento y administración del depósito de datos e información sobre el sistema para guardar,imprimir y consultar esa información.**

(d) Verificación de especificaciones para detectar errores, de sintaxis, de datos faltantes y de inconsistencias.

(e) Generación de código de programación para poder ejecutar los sistemas, una vez definidos.

(f) Producción y mantenimiento de documentación, tanto técnica como para el usuario de operación.

En la actualidad, existen dos tipos de paquetes *CASE*, los que cubren una etapa del ciclo de vida de los sistemas, (toolkits) y los que intentan proporcionar ayudas en todas las etapas (Workbenches).

En el primer caso (toolkits) podemos clasificar los paquetes en:

1.*Análisis y Diseño*.- Estos paquetes generan automáticamente una descripción del sistema y lo van descomponiendo (top-down) hasta lograr las especificaciones detalladas, obteniéndose las definiciones, la estructura de los datos y los componentes funcionales. Se componen básicamente de 4 partes:

. *Diagramación estructurada*.- Su objetivo es graficar, manipular y guardar diagramas, tales como los de descomposición, de flujo y relación de entidades, mismas que son utilizadas como documentación.

. ***Prototipos*.- Ayudan a determinar los requerimientos del sistema y simular el funcionamiento de la aplicación, produciendo una interfase para que el usuario vea como serán sus resultados (pantallas, reportes, menús, etc.).**

. *Depósito de Información*.- Es el diseño del diccionario de datos, conteniendo las especificaciones y relaciones con el sistema, los diagramas y la documentación correspondiente para planeación, análisis, diseño e implantación, así como la administración del proyecto. En la práctica, debe ser un banco de datos y un sistema de información en toda la extensión de la palabra.

. *Verificación de Errores*.- Los paquetes deben verificar las especificaciones, así como los diagramas. Las cinco verificaciones esenciales son:

.. *De sintaxis*, para que las reglas que gobiernan la construcción de los diagramas se cumplan.

.. *De integridad*, para verificar que los diagramas estén completos en todas sus partes.

.. *De consistencia*, para que la información sea congruente con todas las especificaciones del sistema.

.. *De descomposición funcional*, verifica que los diagramas sean congruentes a través de las jerarquías en la descomposición de un árbol estructurado.

- ***De seguimiento*, demuestra que l trabajo creado en el desarrollo, tiene continuidad con el trabajo producido por anteriores etapas.**

2. *Diseño de Datos*.- Normalmente, estos programas *CASE* soportan el diseño lógico y físico de las bases de datos, obteniéndose modelos lógicos de datos, esquemas para bases de datos particulares y generación automática del código para la descripción de los archivos.

3. *Programación*.- Los paquetes incluyen las siguientes facilidades: diagramación de árbol estructurado con verificación de sintaxis y consistencia, diagrama de procedimientos lógicos, depósito de información como toda una base de datos, generación de código, analizador de código, manipulador de archivos, generación de datos de prueba, analizador de pruebas, comparador de archivos, depurador de programas, monitoreo, generación de corridas y simulador de resultados. Independientemente de todas las bondades de diseño y documentación que dan estos paquetes, lo más importante es la generación automática de código en lenguajes específicos (COBOL, FORTRAN, PL/1, etc.), ahorrando tiempo en la elaboración de programas y se supone, libre de errores, quedando todo conforme a las especificaciones originales.

4. *Mantenimiento*.- En la mayoría de las organizaciones, el principal problema se refiere al mantenimiento de los sistemas, gastando dinero y esfuerzos en tener al día la documentación. Por tal motivo, se han desarrollado paquetes especializados para resolver este problema que, en términos generales llegan a hacer lo siguiente:

. *Analizador de documentación*.- Se lee el código fuente de los programas y se produce su documentación.

. *Analizador de programas.*- Evalúa las rutas que sigue un programa y su funcionamiento.

. *Ingeniería invertida.*- Permite identificar el modelo en que está basado el diseño del sistema.

. *Reestructuración.*- Rehace los programas en forma estructurada y estandariza la documentación.

Una de las metas más importantes del área de Informática, es lograr la estandarización de todos sus sistemas (los existentes y los nuevos) y hasta ahora, los paquetes *CASE* no han logrado resolver estos problemas, (ya que existe una alta intervención humana para rediseñar los sistemas antiguos), sin embargo, se tienen planteamientos sobre las funciones que deben hacer los paquetes, apoyados principalmente en sistemas expertos, bases que los proveedores deberán tomar para llevar a cabo el desarrollo de este tipo de productos [1].

5. *Administración de Proyectos.*- Estos paquetes ayudan al administrador llevando un mejor control, tanto en la etapa de desarrollo, como en la de mantenimiento de un proyecto. Normalmente incluyen: procesador de palabra, correo electrónico, hojas de cálculo, planes y recursos del proyecto, calendario con asignación de tareas, tablas con estimaciones de tiempo, medición de control de calidad y un depósito de información con una bitácora que contiene todo lo relacionado con el sistema.

Un caso especial de paquetes que existen en el mercado son los llamados *Framework*, que permiten trabajar en forma conjunta: paquetes *CASE*, programas tradicionales de

implantación y manejadores de bases de datos que hacen posible presentar una interfaz común, compartir los mismos datos y ejecutar en el mismo equipo, programas de desarrollo aparentemente incompatibles.

Respecto a los paquetes *CASE* que intentan abarcar todo el ciclo de vida de los sistemas (workbench) toman como punto de partida la planeación estratégica de la organización y van pasando por cada una de las etapas del ciclo de vida, sirviendo sus salidas como entradas a la siguiente fase produciendo al final, programas ejecutables y su documentación. En la actualidad, ninguno de estos paquetes ha llegado al grado de sofisticación de los especializados.

Otro aspecto importante que hay que considerar, se refiere a que un gran número de paquetes *CASE* está hecho para ejecutarse en computadoras personales, pero la necesidad de compartir información entre usuarios y los administradores del proyecto, requieren de un depósito de información centralizado, ya sea en una computadora o en un servidor de archivos (file server) de una red, para que exista un desarrollo del sistema congruente y bajo el mismo estandar.

Existen en la actualidad métodos para el desarrollo de sistemas, como ya se indicó en el capítulo anterior. Unos están enfocados primero a describir los procedimientos y luego los datos, y otros primero definen los datos y posteriormente los procedimientos, por tal motivo los paquetes *CASE* normalmente están enfocados a un determinado método; pero en la realidad ninguna técnica por sí sola es suficiente para diseñar un sistema, siendo necesario utilizar varios métodos para tener una visión completa.

Tradicionalmente, las etapas del ciclo de vida de los sistemas que más tiempo se llevan, son el desarrollo y la implantación; en teoría con el uso de paquetes *CASE*, en las etapas que hay que dedicar más empeño son en el análisis y el diseño, reduciéndose los

esfuerzos en el desarrollo y la implantación y pudiéndose lograr reducir el tiempo total de un proyecto y mejorar la calidad del sistema, tal y como se indica en la figura 3.1 [8].

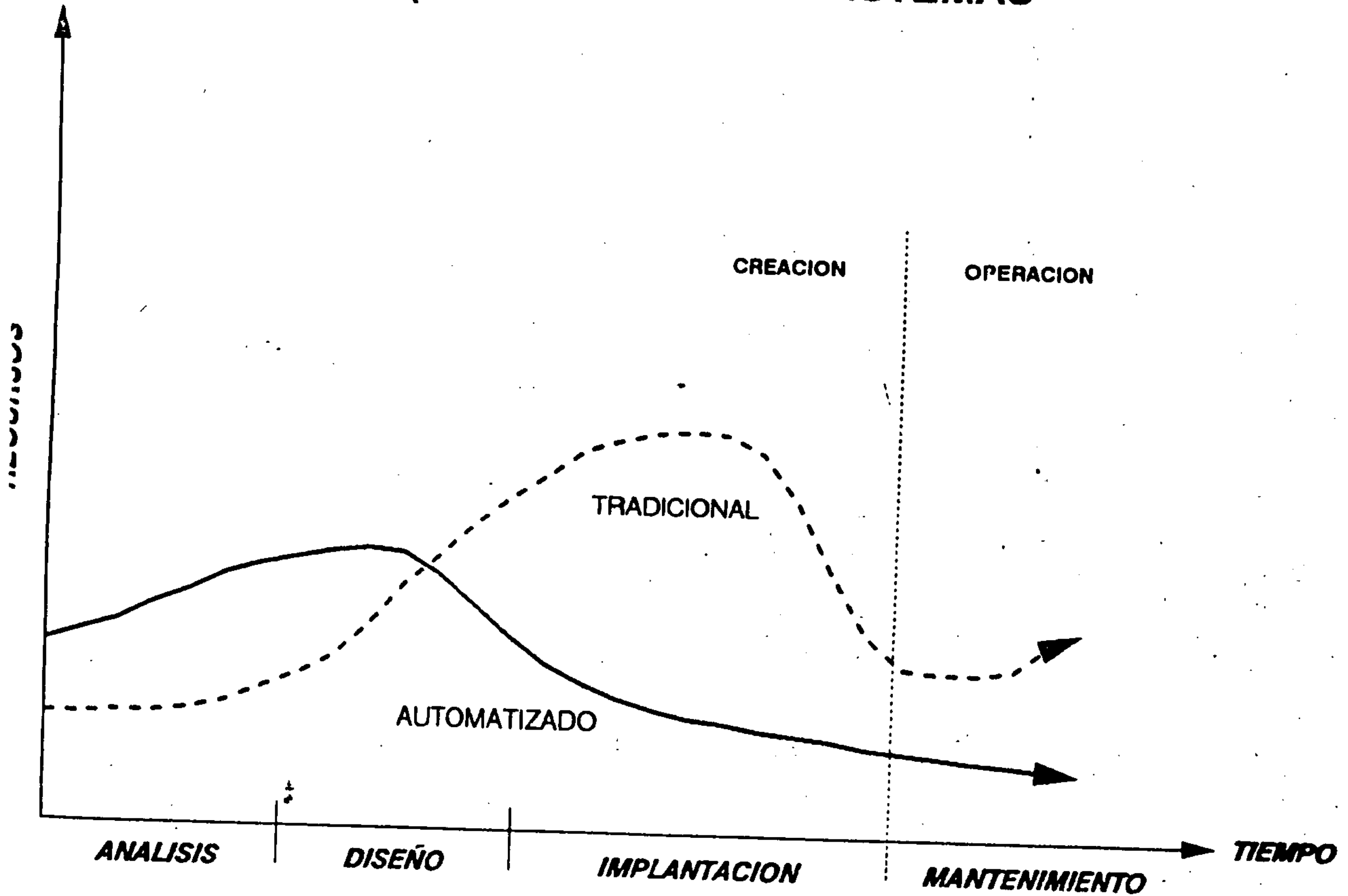
En el anexo 1, se hace una clasificación de paquetes *CASE* que en la actualidad existen en el mercado [7].

En los planes de una organización, para incorporar el uso de los paquetes *CASE*, es necesario llevar a cabo los siguientes pasos:

- Definición del método a utilizar.
- Selección del paquete *CASE*.
- Implantación en la organización.

Etapas que analizaremos en los siguientes capítulos.

CICLO DE VIDA DE LOS SISTEMAS



IV. SELECCION DE METODOLOGIA Y DEL PAQUETE CASE

Antes de pensar en proveedores de paquetes *CASE*, es recomendable hacerse las siguientes preguntas:

(1) ¿Qué ciclo de vida de los sistemas nos interesa abarcar?

Como ya se indicó, ningún paquete *CASE* abarca por sí solo todo el ciclo de vida de los sistemas, por tanto habrá que seleccionar las fases donde se tengan más problemas, así, cuando en una instalación tiene pocos proyectos a desarrollar, es recomendable que enfoque su solución hacia toolkits de mantenimiento, o cuando se requiera mucho desarrollo, ver la conveniencia de utilizar programas enfocados hacia análisis/diseño que abarcan buena parte del ciclo de vida (workbenches). Existe el peligro de seleccionar más de un paquete, con la consecuencia de tener problemas en los estándares y la documentación.

(2) Si en la organización ya se tiene una metodología de desarrollo de sistemas con estándares, ¿es conveniente cambiarla? y ¿por cuáles?.

Para contestar estas preguntas, es necesario considerar varios aspectos, empezando por revisar si el actual método satisface las necesidades de la organización, si no, hay que estudiar cual técnica es la mejor, debido a que si decide cambiar, habrá que reentrenar al personal y los anteriores sistemas quedarían con diferente estandar a los nuevos.

(3) Si no se tiene una metodología formal ¿cuál es la mejor?

9

En este caso, es necesario revisar cual técnica de análisis y diseño, es la más conveniente a la organización.

(4) ¿Qué facilidades debe proporcionar el paquete *CASE*?

Es conveniente analizar que requerimientos debemos de pedir como mínimo al proveedor del paquete, ya que no todos pueden dar las facilidades y muchas veces éstas van ligadas con el precio.

(5) ¿Con qué tipo de equipo se cuenta?

En razón a que los paquetes generalmente están enfocados para procesarse en equipos específicos, por ejemplo: IBM PC compatibles, MAC, UNIX, etc., es importante considerar este aspecto.

Una vez definidas los anteriores parámetros, el universo de paquetes (en la actualidad hay más de 100) que pueden solucionar la problemática particular, se reduce y se puede enfocar la investigación, a través de revistas, información de los proveedores, usuarios que ya utilicen paquetes, etc., sobre todo estudiar los más populares en el mercado y el soporte técnico que se pueda proporcionar en la localidad. En este punto, se pueden utilizar las recomendaciones de autores [4 y 7], que dan guías muy completas sobre las características de los paquetes *CASE* a seleccionar.

Con toda esta información, se procedió a revisar precios contra facilidades y a contratar el paquete *CASE* seleccionado (en el anexo 2, se da una guía de evaluación [7]).

10

V. CONSIDERACIONES PARA LA IMPLANTACION DE UN PAQUETE CASE EN LAS ORGANIZACIONES

Una vez adquirido un paquete *CASE*, su implantación no es tan fácil y rápida como anuncian los vendedores, ya que implica cambios en la organización, en el modo de trabajar y pensar de los profesionales de la Informática, por lo que es necesario llevar a cabo un plan de implantación cuidadoso, que permita obtener beneficios a un mediano plazo, por tal motivo es recomendable tomar en cuenta los siguientes puntos [3]:

(1) Contratar asesores que tengan experiencia en el uso de estos paquetes, de preferencia en el método de desarrollo seleccionado y en el programa a utilizar, creando un grupo que junto con personal experimentado de la organización, puedan llevar a cabo la implantación.

(2) Formular los estándares y procedimientos a utilizar con el paquete *CASE*, para que todos trabajen bajo un solo esquema, los cuales deberán ser constantemente ajustados, en razón a las experiencias que se vayan adquiriendo.

(3) Tomar en consideración las reglas de operación y las facilidades que pueda dar el paquete *CASE*, para la fijación de estándares y procedimientos, y no tratar de hacerlo al revés.

(4) Seleccionar un experimentado grupo de analistas de sistemas, preferentemente líderes de proyectos, para efectos de recibir entrenamiento y sean los primeros que hagan los esfuerzos para la implantación.

//

(5) Crear un grupo interno de capacitación, apoyado en un principio por el grupo de asesores, que elaboren los cursos correspondientes, para siempre tener personal con experiencia en el entrenamiento de los técnicos dentro de la organización.

(6) Llevar a cabo los cursos con el grupo seleccionado, donde se contemple tanto el método de desarrollo de sistemas como el uso del paquete *CASE*, dando prioridad al desarrollo de ejemplos, para que empiecen a dominar el *CASE*.

(7) Seleccionar un sistema a desarrollar que no sea de alta complejidad, donde se apliquen todas las técnicas, estándares y procedimientos. Es importante que el grupo supervise el desarrollo del proyecto, para evitar desviaciones y adquirir experiencia.

(8) Una vez terminado el primer sistema, llevar a cabo los ajustes pertinentes a los estándares, los procedimientos y el temario de los cursos, en función a los resultados obtenidos con el primer proyecto.

(9) Tomando en consideración los planes de la organización en materia de desarrollo de sistemas, asignar proyectos al personal del primer grupo capacitado como responsables y seleccionar un segundo grupo de analistas de sistemas, que trabajarán con el lider.

(10) Proporcionar el entrenamiento al segundo grupo de profesionistas.

(11) Desarrollar los sistemas seleccionados con el segundo grupo de personal, siguiendo con la supervisión por parte de los encargados de la implantación del paquete, para vigilar que los estándares y procedimientos se lleven a cabo y que ayuden a resolver los problemas que se presenten durante el ciclo de vida.

12

(12) Seguir con el mismo ciclo de entrenamiento con el demás personal, apoyados con los analistas que han adquirido experiencia.

Como se observa, el proceso de implantación puede llevar un tiempo considerable, debido principalmente a que el uso de la herramienta, no es fácil y si se trata de hacerlo rápido, puede tenerse una baja en la productividad y no respetarse los estándares y procedimientos establecidos, ya que la herramienta obliga a hacer un análisis más detallado y no permite que los errores pasen de una etapa a la siguiente, malográndose los objetivos planteados originalmente de incrementar la rapidez y calidad en el desarrollo de los proyectos.

Otro punto que hay que tomar en consideración, es el avance tecnológico que puede obtenerse con el paquete *CASE* seleccionado, por lo que es importante ir implantando las nuevas versiones y reentrenar al personal.

Derivado de la competencia, es factible que otros productos *CASE*, diferentes al seleccionado, proporcione con el tiempo, soluciones más acordes a las necesidades de la organización, presentándose el dilema de cambiar de paquete *CASE*, alternativa que hay que evaluar cuidadosamente.

VI. EXPERIENCIA EN EL BANCO DE MEXICO CON LOS PAQUETES CASE

En el Banco de México, formalmente se tiene implantada la programación estructurada, y de manera informal, se utilizan para análisis las técnicas de De Marco y para diseño, la de Yourdon, pero gran parte de la documentación no sigue una norma estricta.

Dentro de los planes inmediatos en la Institución, está el establecimiento de un Plan Estratégico en materia de informática, que abarque por lo menos un período de 5 años. Por otro lado, se requiere establecer formalmente una metodología para el análisis y diseño de sistemas que considere el hecho de contar con un gran número de sistemas que fueron desarrollados con antiguas técnicas diferentes a las que en la actualidad existen, por lo que es importante estandarizarlos con los nuevos proyectos, ya que se han estado desarrollando sistemas desde hace más de 20 años.

Como se observa, la problemática abarca prácticamente todo el ciclo de vida de los sistemas, por lo que se determinó seleccionar en primera instancia una metodología, que brindara el más amplio espectro de solución para el Banco, ésta es la de Martin, debido a que permite llevar a cabo una planeación estratégica global de la Institución y utiliza lo mejor de otras técnicas de análisis y diseño en forma conjunta, lográndose una visión más completa sobre los sistemas.

Debido a la imposibilidad práctica de atacar toda la problemática, sobre todo lo referente a los sistemas actuales, se determinó utilizar el paquete *CASE* solo en las primeras tres etapas del ciclo de vida de los sistemas (planeación, análisis y diseño)

para crear primero la infraestructura metodológica en la Institución y posteriormente exterderla a las otras fases.

Para determinar la adquisición del paquete *CASE*, se hizo un estudio de mercado, principalmente en artículos de revistas y por medio de la información que se solicitó a los proveedores. Se eligieron tres posibles candidatos para hacer la investigación detallada, seleccionándose el que más se apega a la metodología de Martin; otro factor que se consideró fue el incremento de ventas, la mayor en los últimos dos años, la existencia de soporte técnico en México, el cumplimiento con las necesidades planteadas para el Banco de México y la utilización de PC-AT compatible, tipo de equipo se está utilizando actualmente en la Institución.

Las técnicas de diagramación que utiliza el paquete son en la planeación: descomposición funcional, relación de entidades y matrices de propiedades y de asociación; en el análisis: flujo de datos, relación de entidades, diagramas de descomposición y de acción; en diseño: cartas estructuradas, módulo de diagramas de acción, estructura de datos y de base de datos. El paquete permite pasar las especificaciones de un tipo de diagramas a otro, proporcionando gran flexibilidad en el desarrollo de proyectos [10].

Posteriormente, se tomó a un grupo de diez analistas de sistemas y líderes de proyecto para proporcionarles capacitación en la metodología y el uso del paquete durante 48 horas.

Definiendo como prueba piloto, el rediseño del Sistema de Operaciones Internacionales, proyecto con prioridad y alto nivel de dificultad, mismo que se encuentra actualmente en fase de diseño.

Los resultados que se han obtenido a la fecha, se comportan de acuerdo a la teoría, sin embargo, es necesario vencer la curva del aprendizaje, pues toman mayor tiempo en el análisis y el diseño que si se hiciera con papel y lápiz; sin embargo, se logra obtener una cla⁻ˈdad y precisión mayor en las especificaciones, permitiendo una comunicación ágil con el usuario y una definición de programas a un detalle tal, que las dudas por parte del personal son mínimas. Creemos que el tiempo de implantación de este sistema, debe ser más corto de lo que se tiene programado.

Dentro de los planes que se tiene en la implantación del paquete *CASE* y en base a la experiencia que se logre en la prueba piloto, se seguirá trabajando en difundir la herramienta en la Institución, pensando que su introducción podrá llevarnos más de un año, como se indica en el capítulo V.

VII. CONCLUSIONES

Aunque no tienen mucho tiempo en el mercado los paquetes *CASE* han tenido un acelerado avance tecnológico, permitiendo llenar una buena parte de los requerimientos necesarios para llevar a cabo el desarrollo de un sistema en todas sus fases. Estos paquetes son productos muy sofisticados en su diseño interno, presentando facilidades como una interfaz gráfica al usuario muy amigable, logrando con ello un aprendizaje relativamente rápido.

El costo de los paquetes *CASE* es variable, algunos valen menos de $1,000.00 U. S. Dolls., alcanzando otros precios mayores a los $100,000.00 U. S. Dolls., casi siempre en función de la cantidad de facilidades que proporcionan y al equipo al que va dirigido; pero en realidad el costo se encuentra en la implantación, ya que hay que dedicarle recursos humanos especializados, en entrenamiento y en desarrollo de pruebas piloto, obteniéndose al principio logros poco espectaculares, pero una vez entendidas y dominadas las metodologías, los estándares y los procedimientos, a medio plazo se pueden alcanzar resultados más satisfactorios.

En aquellas organizaciones que en la actualidad no tienen contemplado el uso de paquetes *CASE*, es recomendable que empiecen a hacer los estudios correspondientes para su utilización, ya que los beneficios que se pueden alcanzar a mediano plazo, compensan con creces los esfuerzos y costos que implican su implantación; ya que por medio de estas herramientas, podemos lograr estandarizar y documentar los sistemas con mayor rapidez en todo su ciclo de vida, desde la planeación estratégica de la empresa o institución, hasta la generación de código y posteriormente su mantenimiento, obteniendo a la larga ahorros significativos por estos conceptos y si a

ésto agregamos, que nos permita tener un medio de comunicación uniforme, tanto con los usuarios como con los miembros del equipo de trabajo para el desarrollo de un sistema, su utilización es todavía más eficaz.

Un punto que no ha sido resuelto, es la estandarización de los sistemas que actualmente están en operación, y que no estuvieron apoyados en su desarrollo por paquetes *CASE*. En este caso, los toolkits de mantenimiento pueden ser una alternativa en la que hay que estudiar su posible utilidad y esperar la aparición de productos que ayuden a resolver este problema más fácilmente, ya que en la actualidad no existen paquetes que den todas las facilidades.

Definitivamente, el uso de las herramientas *CASE*, pueden coadyuvar al desarrollo de sistemas, en todo su ciclo de vida, aunque por más sofisticadas que sean, no son más que ayudas que los analistas de sistemas y los programadores pueden utilizar en sus labores, pues no sustituyen la capacidad de los individuos, *ya que el hacer sistemas es un arte que se apoya en técnicas* y depende definitivamente de la capacidad creativa de las personas.

BIBLIOGRAFIA

1. Bachman, Charlic, A Case for Reverse Engineering; Datamation; 1o. de julio de 1988; Pag. 49-56.

2. Berland, G. D.; A Guided Tour of Program Design Methodologies Computer, October 1981, IEEE, Computer Society.

3. Gibson, Michael L.; Implementing the Promise; Datamation; 1o. de febrero 1989; Pág. 65-67.

4. Gibson, Michael L.; A Guide to Selecting Case Tools; Datamation; 1o. de julio de 1989; Pag. 65-66.

5. Martin, J., C. McClure; Diagramming Techniques for Analysts and Programmers; Prentice Hall, Inc., 1985

6. McClure, Carma; The Case for Structured Development; PC-Tech Journal; Agosto 1988, Pág. 51-67.

7. McClure, Carma; The Case Experience; Byte, Abril 1989, Pág. 235-244.

8. McClure, Carma; Taking a Closer Look at Software Workstations-The Newest Productivity Tools; Arthur Young International; 1986.

9. Metzger, Phillip W.; Managing a Programing Proyect; Prentice-Hall; 1973.

10. Topper, Andrew; Building up to IEW/WS; PC-Tech Journal; septiembre de 1988; Pag. 110-123.

ANEXO 1

Análisis y Diseño

Analyst/Designer Toolkit
Yourdon, Inc.
1501 Broadway
New York, N.Y.10036
(12) 391-2828

Advanced Logical Software
9903 Santa Mónica Blvd.
Beverly Hills, Ca.90212
(213) 653-5786

CA-Universe/Prototype
Computer Associates
International, Inc.
711 Stewart Ave.
Garden City, N. Y.11530
(516) 227-3300

CasePac
On-Line Software International
2 Executive Dr.
Ft. Lee Executive Park
Ft. Lee, NJ07024
(201) 592-0009

DesignAid - RTrace
Nastec Corp.
24681 Northwestern Hwy.
Southfield, MI48075
(313) 353-3300

Design Generator
Computer Sciences Corp.l
3160 Fairview Park Dr.
Falls Church, VA 22042
(703) 876-1000

Design Machine
Optima, Inc.
1300 Woodfield Rd., Suite 400
Schaumburg, IL 60173
(312) 240-1888

Exelerator
Exelerator/RTS
Index Technology Corp.
One Main St.
Cambridge, MA 02142
(617) 494-8200

Meta Systems Toolset
Structured Architect
Meta Systems, Ltd.
315 East Eisenhower Pkwy.
Ann Arbor, MI 48108
(313) 663-6027

MicroStep
Syscorp International
9420 Research Blvd., Suite 200
Austin, TX 78759
(512) 338-0591

POSE
Computer Systems Advisers
50 Tice Blvd.
Woodcliff Lake, NJ07675
(201) 391-6500

ProKit*Workbench
McDonnell-Douglas
P.O.Box 516
Dept. L515,MS 2812301
St.Louis,MO63166
(314) 232-5715

System Architect
Popkin Software & Systems
111 Prospect St., Suite 505
Stamford, CT 06901
(203) 323-3434

Teamwork
Cadre Technologies
222 Richmond St.
Providence, RI 02903
(401) 351-5950

Diseño de Datos

Auto-Mate Plus
LEMS,Inc.
2900 North Loop W., Suite 800
Houston, TX 77092
(713) 682-8530

Chen Toolkit
Chen & Associates, Inc.
4884 Constitution Ave.
Baton Rouge. LA 70808
(504) 928-5765

IDMS/Architect
Cullinet Software, Inc.
400 Blue Hill Dr.
Westwood, MA 02090
(617) 32? 7700

CASE*Designer
CASE*Dictionary
Oracle Corp.
20 Davis Dr.
Belmont, CA 94002
(800) 345-3267

IDEF/Leverage
D.Appleton Co.
1334 Park View Ave., Suite 220
Manhattan Beach, CA 90266
(213) 546-7575

4Front
Deloitte, Haskins & Sells
200 East Randolph Dr.
Chicago, IL 60601
(312) 856-8168

Programación

APS
Sage Software, Inc.
3200 Tower Oaks Blvd.
Rockville, MD 20852
(301) 230-3200

CoFac
Coding Factory
45 Knightsbridge Rd.
Piscataway, NJ 08854
(201) 981-0100

NETRON/CAP
Netron, Inc.
99 St. Regis Crescent N
Downsview, Ontario
Canada M3J 1Y9
(416) 636-8333

PVCS (Polytron Version Control System)
Poly Make - Polytron Corp.
1700 Northwest 167th Place
Beaverton, OR 97006
(503) 645-1150

COBOL/2 Workbench
Micro Focus, Inc.
2465 East Bayshore Rd.
Palo Alto, CA 94303
(415) 856-4161

DECASE
Digital Equipment Corp.
DECdirect
Continental Blvd.
Merrimack, NH 03054
(800) 344-4825

P-Source - P-Tools
Phoenix Technologies, Ltd.
846 University Ave.
Norwood, MA 02062
(617) 551-4000

Telon
Pansophic Systems, Inc.
2400 Cabot Dr.
Lisle, IL 60532
(312) 572-6000

Transform
Transform Logic Corp.
8502 East Via de Ventura
Scottsdale, AZ 85258
(602) 948-2600

Mantenimiento y Reingeniería

Adpac CASE Tools
Adpac Corp.
340 Brannan St.
San Francisco, CA 94107
(415) 974-6699

Inspector - Recoder
Language Technology, Inc.
27 Congress St.
Salem, MA 01970
(508) 741-1507

Scan/COBOL
SuperStructure
Computer Data Systems, Inc.
1 Curie Court
Rockville, MD 20850
(202) 921-7000

Bachman Re-Engineering Product Set
Bachman Information Systems
Four Cambridge Center
Cambridge, MA 02142
(617) 354-1414

athVu - Retrofit
Peat Marwich Advanced Technology
303 East Wacker Dr.
Chicago, IL 60601
(312) 938-5352

Via/Insight
Via/Smar Test
ViaSoft,Inc.
3033 North 44th St.,Suite 280
Phoenix, AZ 85018
(602) 952-0050

Frame Work

Life Cycle Productivity System
American Management System, Inc.
1777 North Kent St.
Arlington, VA 22209
(703) 841-6060

Software BackPlane
Atherton Technology
1333 Bordeaux Dr.
Sunnyvale, CA 94089
(408) 734-9822

Multi/CAM
AGS Management Systems, Inc.
880 First Ave.
King of Prussia, PA 19406
(215) 265-1550

Sylva Foundry Cadware
50 Fitch St.
New Haven, CT 06515
(203) 397-1853

Administración de Proyectos

Project Workbench
Applied Business Technology.Corp.
361 Broadway
New York, NY 10013
(212) 219-8945

Workbench

AutoCode
Integrated Systems, Inc.
2500 Mission College Blvd.
Santa Clara, CA 95054

EPOS - SPS Software Products
& Services, Inc.
14 East 38th St., 14th Floor
New York, NY 10016
(212) 686-3790

Information Engineering Facility
Texas Instruments
P.O. Box 655621 MS 8474
Dallas, TX 75265
(214) 575-4404

Maestro
Softlab, Inc.
188 The Embarcadero
Bayside Plaza, Suite 750
San Francisco, CA 94105
(415) 957-9175

PACBase
PACBench
PACDesign
CGI Systems, Inc.
1 Blue Hill Plaza
Pearl River, NY 10965
(914) 735-5030

SuperCASE
Advanced Technology International, Inc.
350 Fifth Ave.
New York, NY 10118
(212) 947-4755

CorVision
Cortex Corp.
138 Technology Dr.
Waltham, MA 02154
(617) 894-7000

Foundation
Andersen Consulting
33 West Monroe St.
Chicago, IL 60603
(312) 507-5161

Information Engineering Workbench
KnowledgeWare, Inc.
3340 Peachtree Rd., NE
Atlanta, GA 30026
(404) 231-8575

Manager Family
Manager Software Products, Inc.
131 Hartwel Ave.
Lexington, MA 02173
(617) 863-5800

ProMod
ProMod, Inc.
23685 Birtcher Dr.
El Toro, CA 92630
(714) 855-3046

ANEXO 2

ASPECTOS A CONSIDERAR PARA LA EVALUACION DE PAQUETES

Información General

Proveedor
Producto
Antiguedad prod.
copias vendidas
costo

Equipo Necesario

IBM PC y compat.
Macintosh
Digital Equipment Corp.
Unisys
Honeywell
Wang
Apollo
Sun
Hewlett-Packard
Otros

Categoría

CASE toolkit
CASE workbench

Tipo de Herramienta

Planeación
Análisis
Diseño
Diseño Base de Datos
Diseño Tiempo Real
Generador de código
Programación
Mantenimiento
Framework
Administración de Proyectos

Gráficas

Color
ratón
ventanas

Verificación de Errores

Sintaxis
Consistencia
Completez
Seguimiento
Calidad

Diagramación

Flujo de datos
Control de flujo
Tablas de decisión
Estructura jerárquica
Cartas estructuradas
Diagramas de acción
Warnier/Orr
Diagrama estados
Pseudocódigo
Diseño de pantallas
Flujo de diálogos
Diseño de reportes
Estructura de datos
Relación de entidades
Registros lógicos
Booth
Redes de Petri
Otros

Depósito CASE

Computadores centrales
Computadores personales
Arquitectura DBMS
Reportes
Control de cambio
Auditoría
Control de versión
Descarga
Partición lógica
Consolidación

Generación de Código

Estructura de programa
Programa completo
Lenguaje
Programación en linea
Programación en batch

Soporte al Ciclo de Vida

Planeación
Análisis
Diseño
Implementación
Mantenimiento
Admon. de Proyectos

Metodología

Yourdon
DeMarco
Gane/Sarson
Bachman
Chen
Martin
Merise
Orr
Jackson
Ward/Mellor
Hatley
Object-oriented
SADT
Stradis
Method-1
LSDM
Otros

Prototipos

Diseño de pantallas
Diseño de reportes
Modelo funcional
Simulación

Reingeniería

Analizador estático
Redocumentador
Re-estructura
Ingeniería invertida
Analizador dinámico
Convertidor

Sistema Objetivo

En linea
Lotes
Transacción
Tiempo real
Inmerso

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS INSTITUCIONALES

No. 70

METODOLOGIA PARA EL DESARROLLO DE SISTEMAS DE INFORMACION

DEL 9 AL 29 DE SEPTIEMBRE

SEPOMEX

ENFOQUE ORIENTADO A OBJETOS

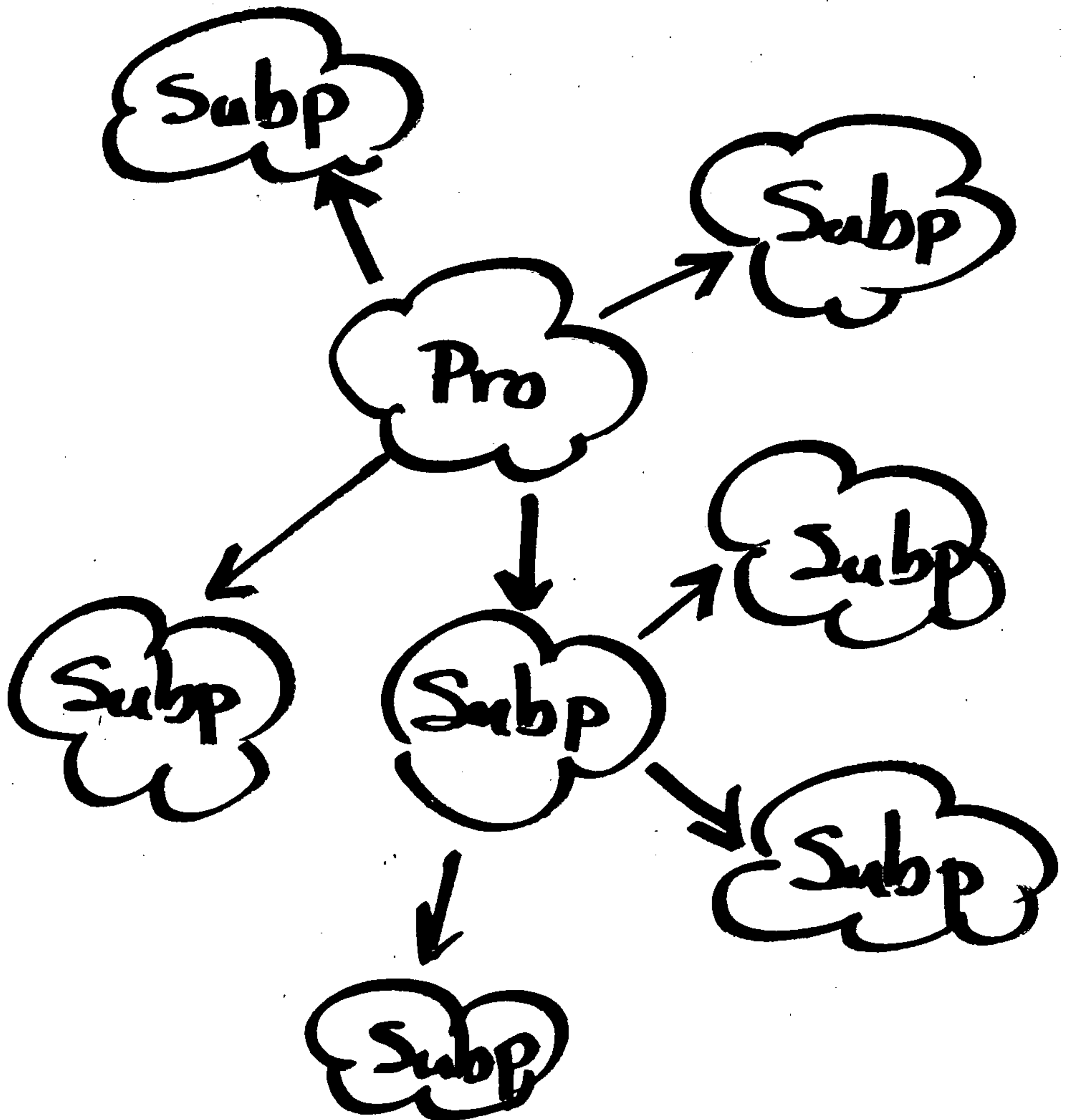
MEXICO D.F.
PALACIO DE MINERIA
1992

lacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

EL ENFOQUE ORIENTADO A OBJETOS

LOS ENFOQUES ESTRUCTURADO Y ORIENTADO A OBJETOS

EL ENFOQUE ESTRUCTURADO



HERRAMIENTAS

(I) SEUDOCODIGO O
PROSA ESTRUCTURADA O
ESPAÑOL ESTRUCTURADO
PDL (PROGRAM DESIGN
LANGUAGE)

a) SECUENCIA
b) DECISION
c) ITERACION
d) SALIDA

(II) ARBOLES DE DECISION

TABLAS DE DECISION

(a) ENTRADA LIMITADA

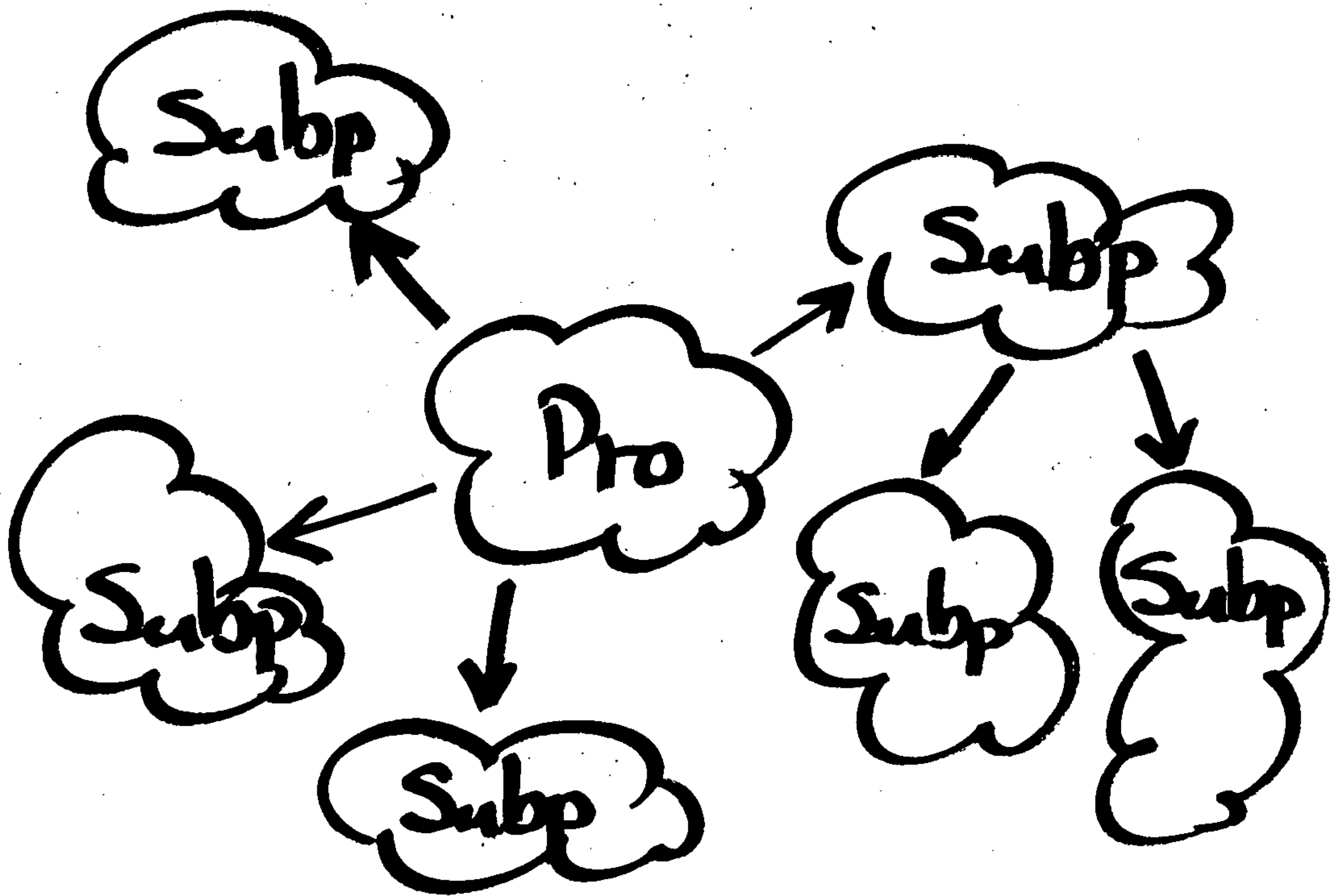
(b) ENTRADA EXTENDIDA

(C) ENTRADA MIXTA

(III) ACOPLAMIENTO

(IV) COHESION

EL ENFOQUE ORIENTADO A OBJETOS



HERRAMIENTAS

(I) OBJETOS
RELACIONES ENTRE OBJETOS

(a) JERARQUICAS
- HERENCIA
- PARTE-TODO

(b) HORIZONTALES
- UNO A UNO
- UNO A MUCHOS
- MUCHOS A MUCHOS

EN EL ENFOQUE ESTRUCTURADO LOS PROGRAMAS SE VISUALIZAN COMO UN CONJUNTO DE RUTINAS EN LAS QUE SE MINIMIZA EL ACOPLAMIENTO Y SE MAXIMIZA LA COHESION. LAS RUTINAS COOPERAN PARA EL LOGRO DE UN FIN.

EN EL ENFOQUE ORIENTADO A OBJETOS UN PROGRAMA O SISTEMA SE VISUALIZA COMO CONSTITUIDO POR COMUNIDADES DE OBJETOS. LOS OBJETOS INTERACTUAN ENTRE SI MEDIANTE MENSAJES AL SOLICITARSE Y PRESTARSE SERVICIOS QUE SON RESPONSABLES DE PROPORCIONARSE. LOS OBJETOS COOPERAN PARA EL LOGRO DE UN FIN.

CONCEPTOS

CLASES y OBJETOS

LA PRINCIPAL RAZON PARA USAR CLASES DE OBJETOS, ES HACER COINCIDIR LA VISION CONCEPTUAL DEL MUNDO REAL CON LA REPRESENTACION TECNICA DE UN SISTEMA.

CLASES y OBJETOS

CLASE

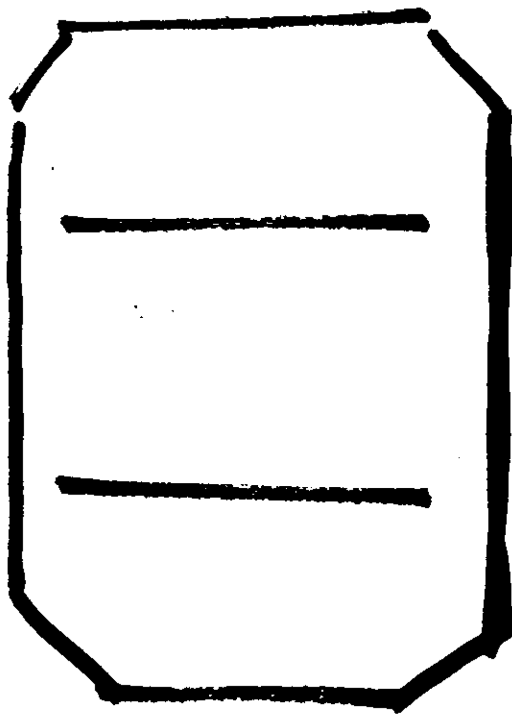
LA ABSTRACCION, EN EL DOMINIO DEL PROBLEMA, DE UNA ESTRUCTURA Y COMPORTAMIENTO COMUNES QUE SE TRATAN COMO UNA UNIDAD.

OBJETO

LA OCURRENCIA DE UNA CLASE.

ESTRUCTURA

ARREGLO O DISPOSICION DE LAS PARTES EN EL TODO.



← NOMBRE DE LA CLASE

← DATOS, ATRIBUTOS
PROPIEDADES

← SERVICIOS
OPERADORES
METODOS
MEMBER FUNCTIONS
(C++)

COMPORTAMIENTO

LA FORMA EN QUE LOS OBJETOS DE UNA CLASE ACTUAN Y/O INTERACTUAN CON LOS OBJETOS DE OTRAS CLASES.

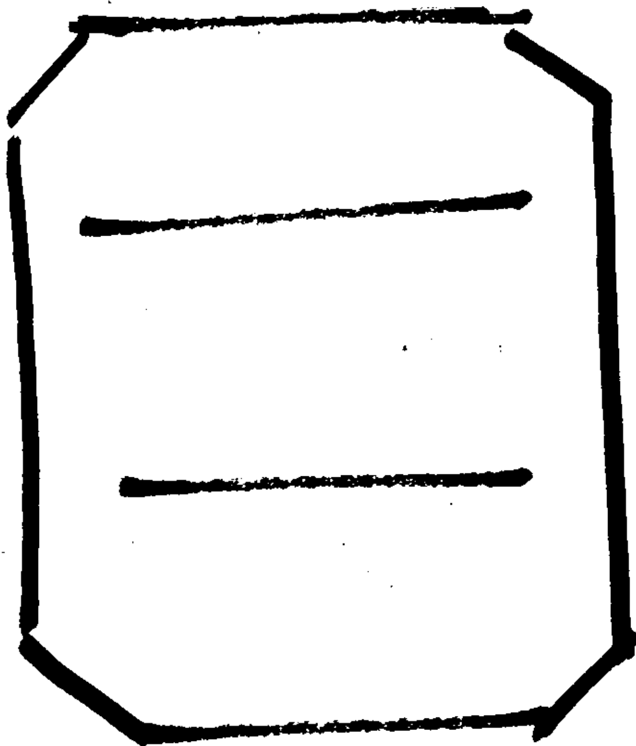
ESTADO DE UN OBJETO

LOS VALORES DE LOS DATOS EN UN MOMENTO DADO.

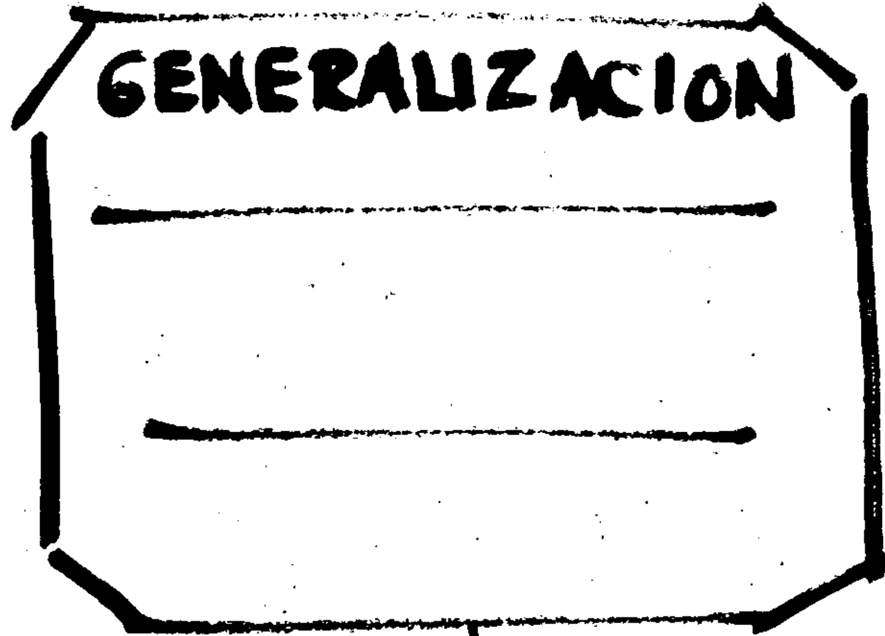
CAMBIO DE ESTADO

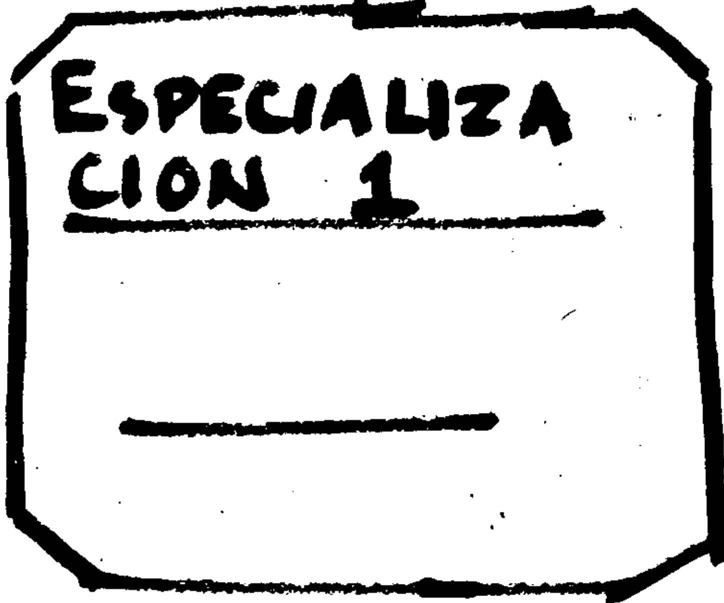
EL CAMBIO, EN EL TIEMPO, DE LOS VALORES DE LOS DATOS.

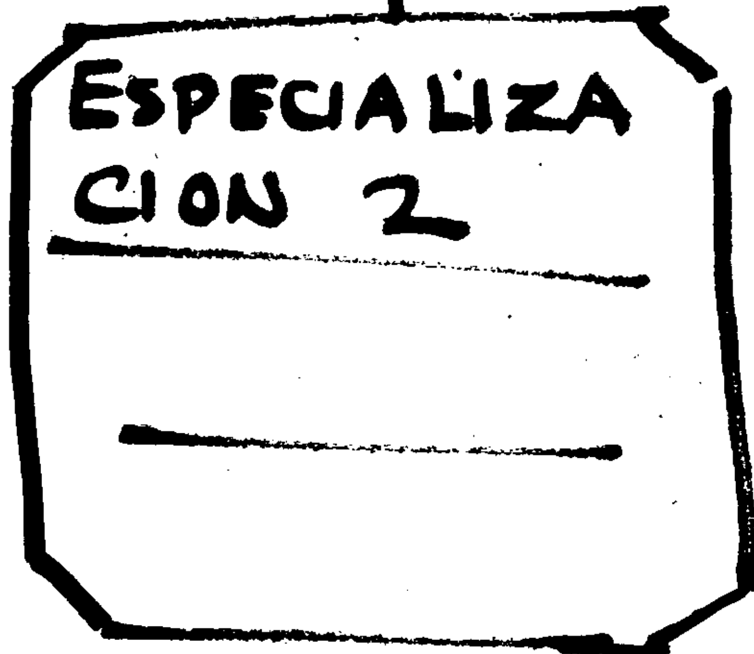
ENCAPSULADO



HERENCIA







AVES

Terre
stres

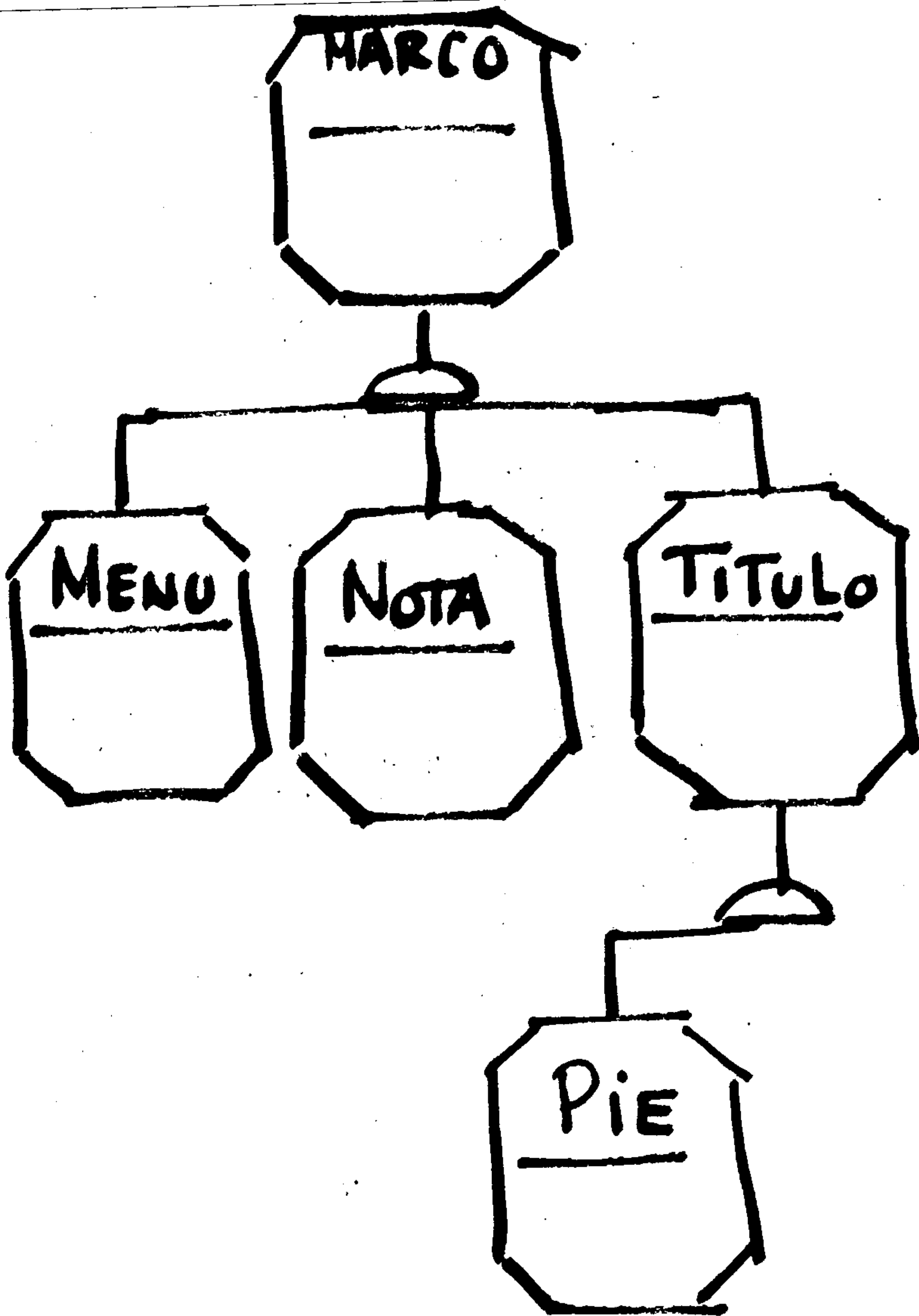
Aere
as

Ave
struz

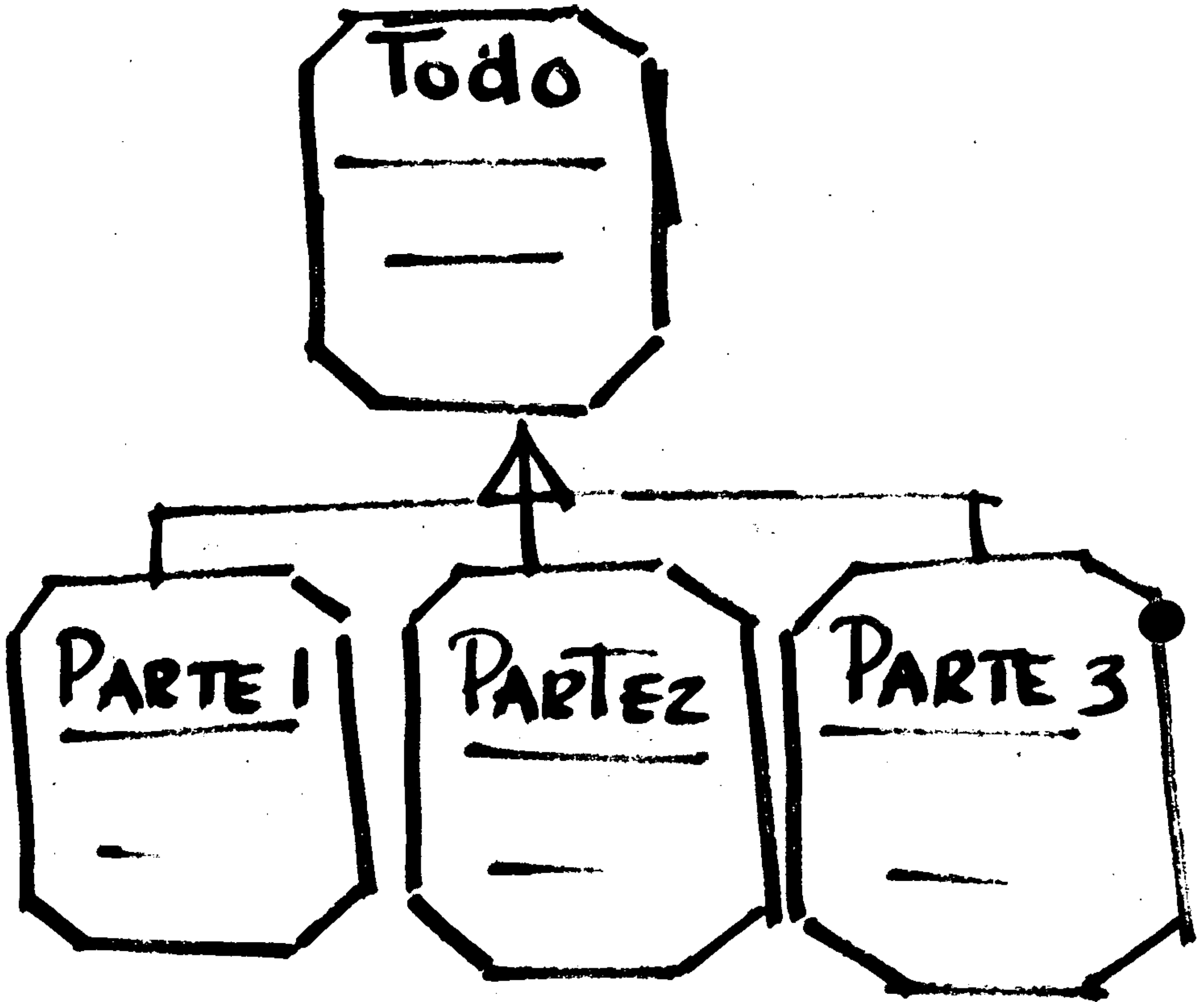
Pin
guino

Agui
la

Halcón


MARCO
MENU
NOTA
TITULO
PIE

AGREGACION



Auto

Volante

llantas

Defensas

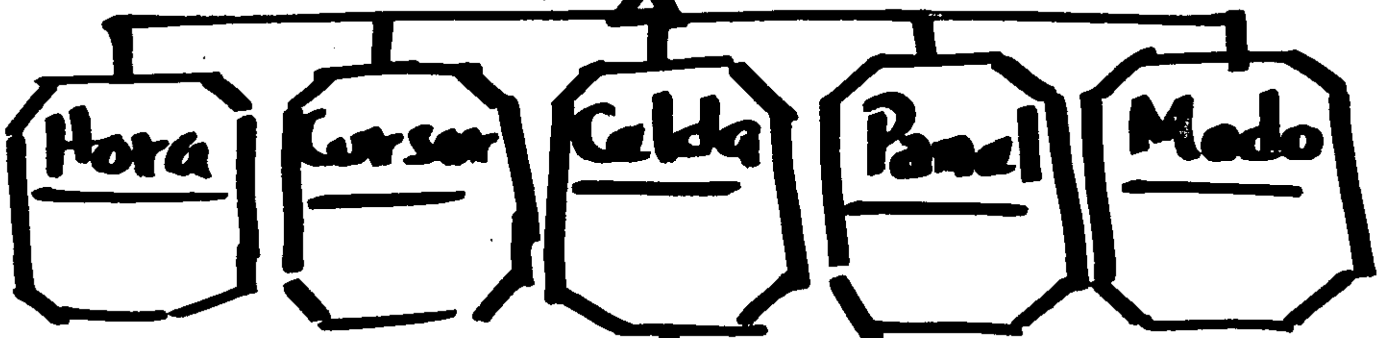
Motor

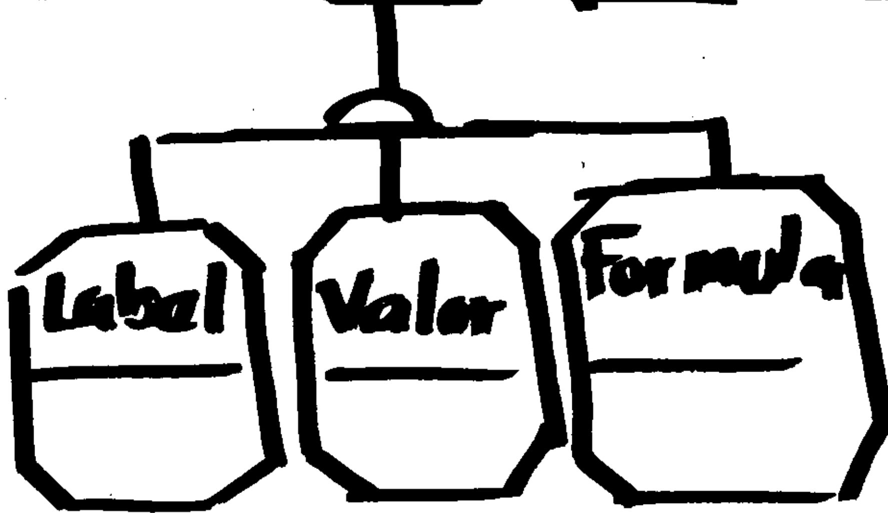
Carburador

Cilindros

Bujias

Hoja





CATEGORIAS DE LOS SERVICIOS

COAD/YOURDON

SERVICIOS ALGORITMICAMENTE SIMPLES

CREAR — CREA E INICIA A UN OBJETO EN UNA CLASE

CONECTAR — CONECTA (DESCONECTA) A UN OBJETO CON OTRO

ACCESAR — LEE Y/O ESCRIBE VALORES DE ATRIBUTO EN UN OBJETO.

LIBERAR — DESCONECTA Y SUPRIME UN OBJETO.

SERVICIOS ALGORITMICAMENTE COMPLEJOS

CALCULAR — CALCULA UN RESULTADO DE LOS VALORES DE LOS ATRIBUTOS DE UN OBJETO.

MONITOREAR — MONITOREA UN SISTEMA O DISPOSITIVO EXTERNO.

JHON G. HUGHES

1. OPERACIONES DE CONSTRUCCIÓN Y DESTRUCCION.

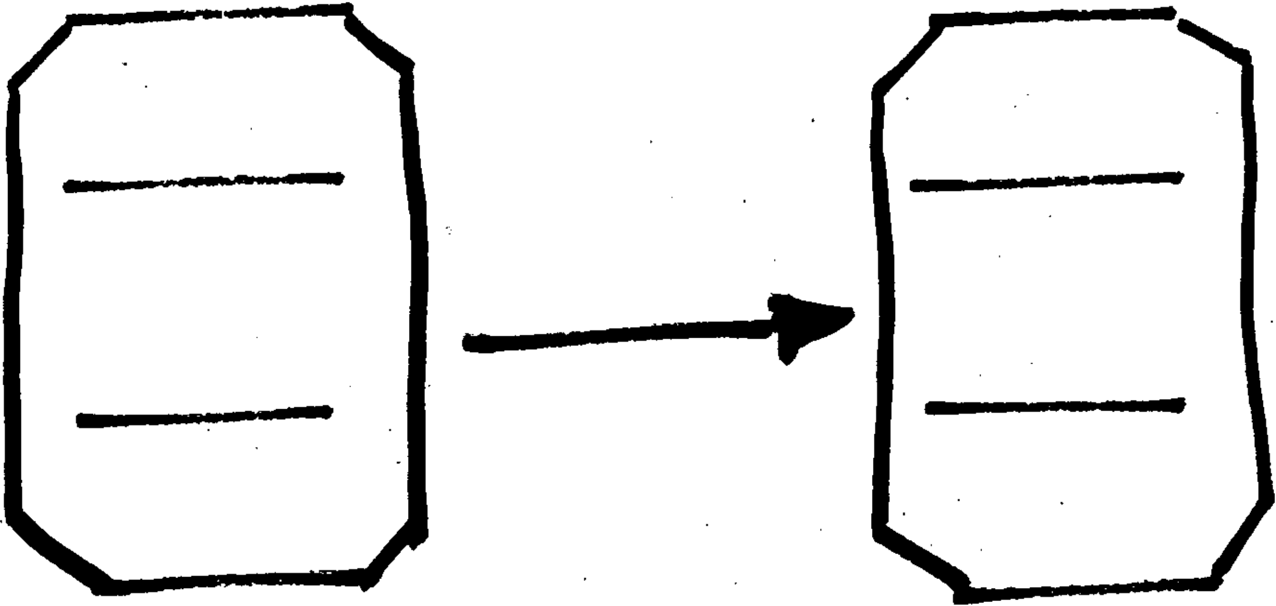
2. OPERACIONES DE ACCESO.

3. OPERACIONES DE TRANSFORMACION.

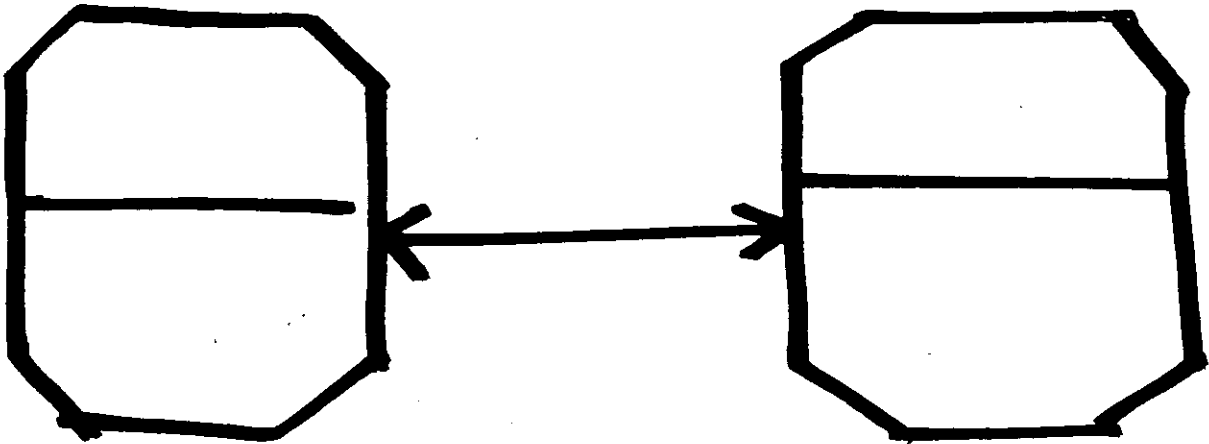
GRADY BOOCH

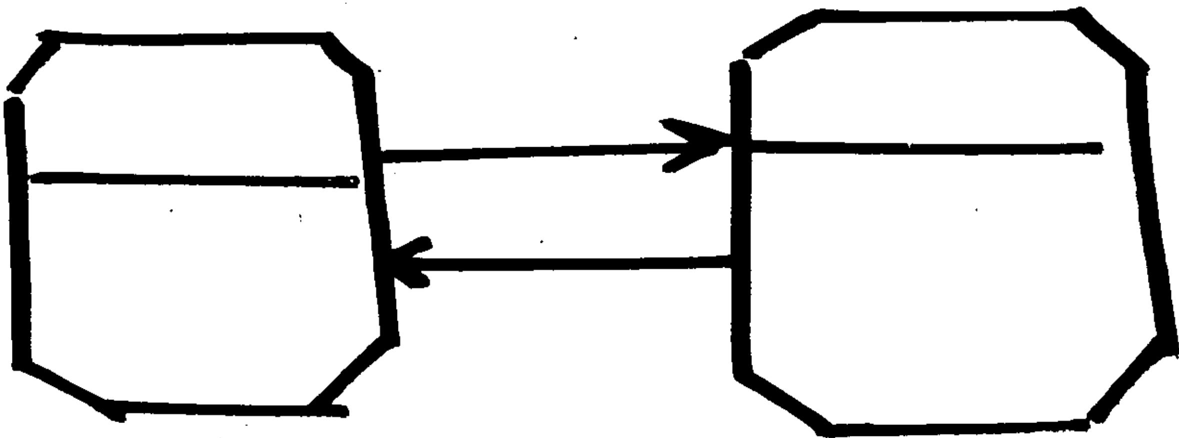
MODIFICAR	UNA OPERACION QUE ALTERA EL ESTADO DE UN OBJETO.
SELECTOR	UNA OPERACION QUE ACCESA EL ESTADO DE UN OBJETO SIN ALTERARLO.
ITERADOR	UNA OPERACION QUE PERMITE QUE TODAS LAS PARTES DE UN OBJETO SEAN ACCESADAS EN ALGUN ORDEN BIEN DEFINIDO

MENSAJES

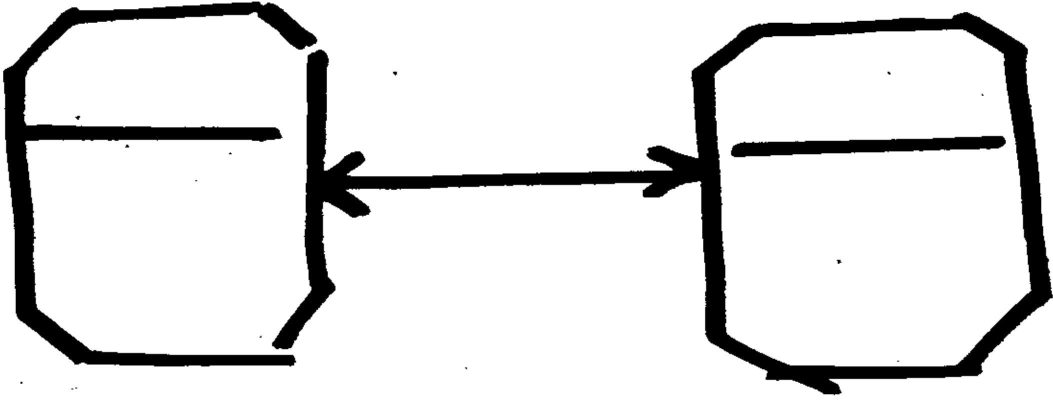


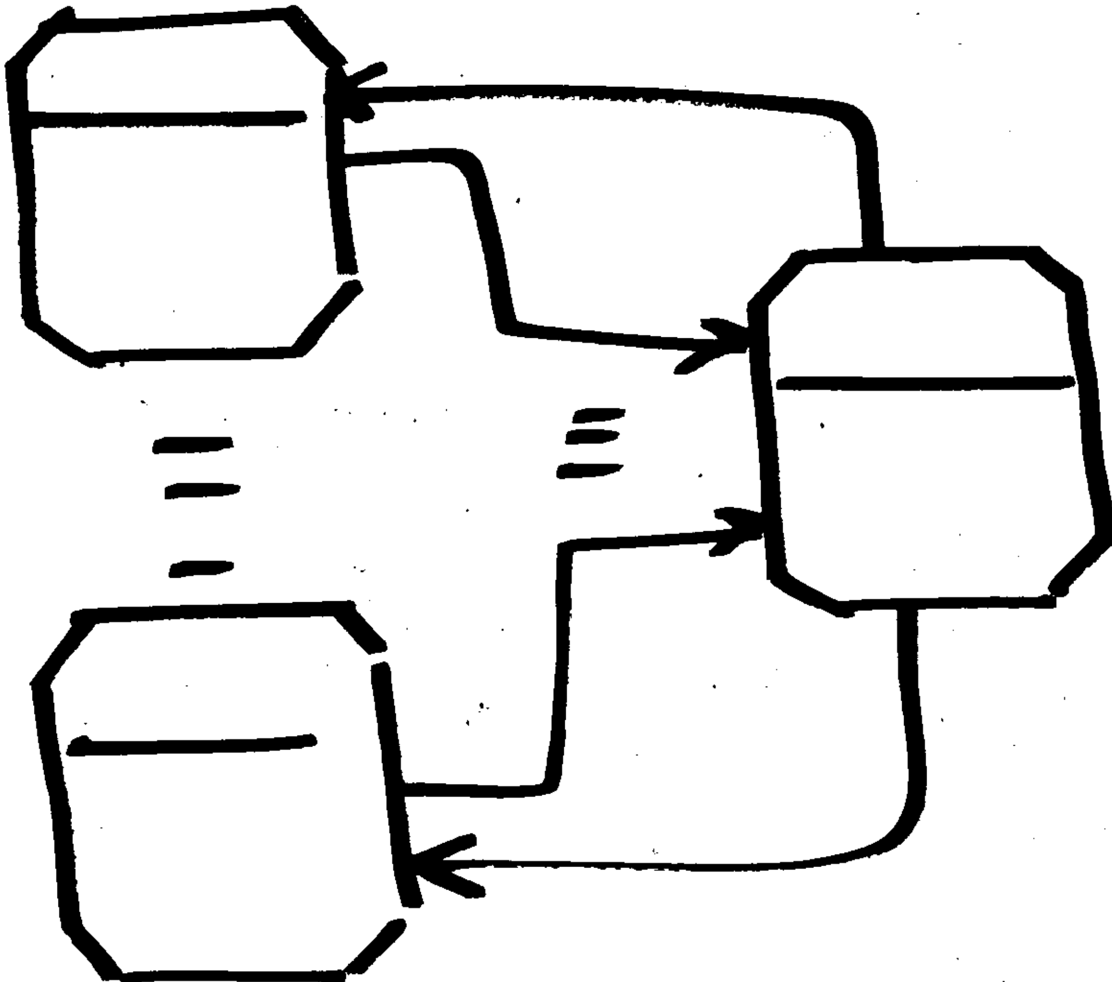
1:1



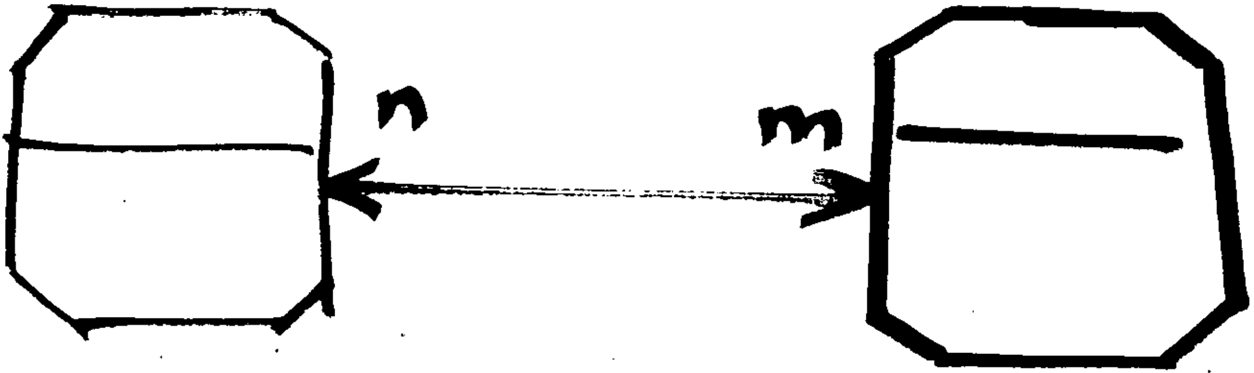


n : 1





n:m



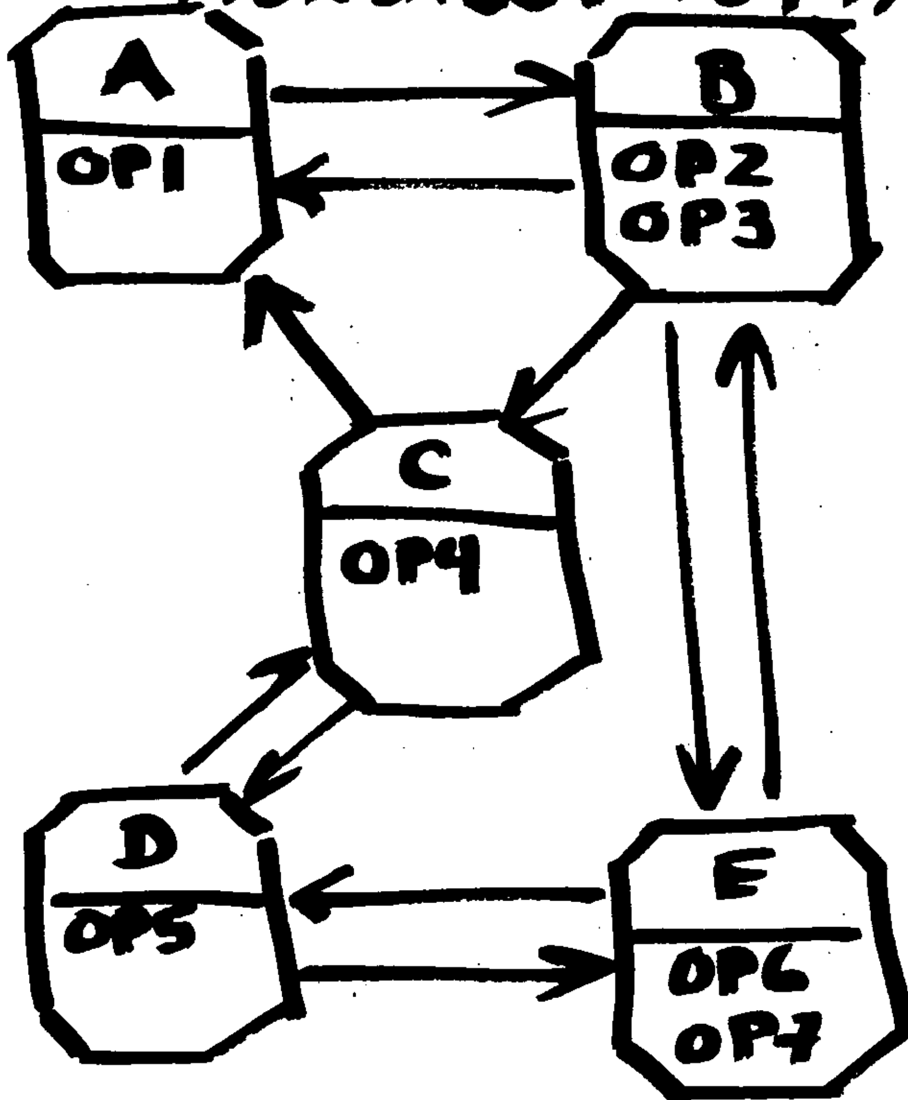
MENSAJE(DESTINO, OPRN,
ARGUMENTOS)

OBJETO B

MENSAJE(D, OP5, <DATOS>)

OBJETO D

MENSAJE(C, OP4, <DATOS>)



EL DIAGRAMA DE ENTIDAD RELACION EXTENDIDO (ERE)

1. ENTIDADES

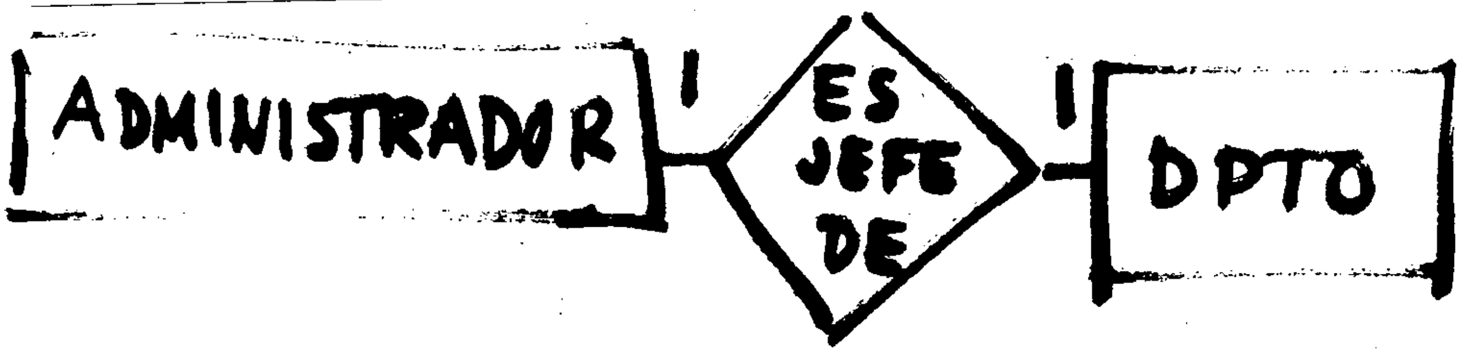
   REPRESENTAN LOS OBJETOS QUE SE MODELAN.

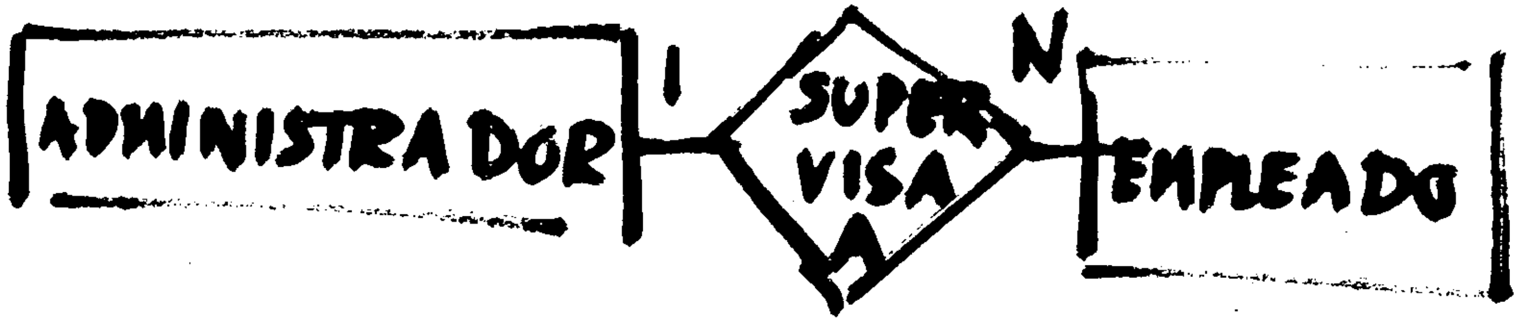
2. ATRIBUTOS

   REPRESENTAN LAS PROPIEDADES DE LOS OBJETOS.

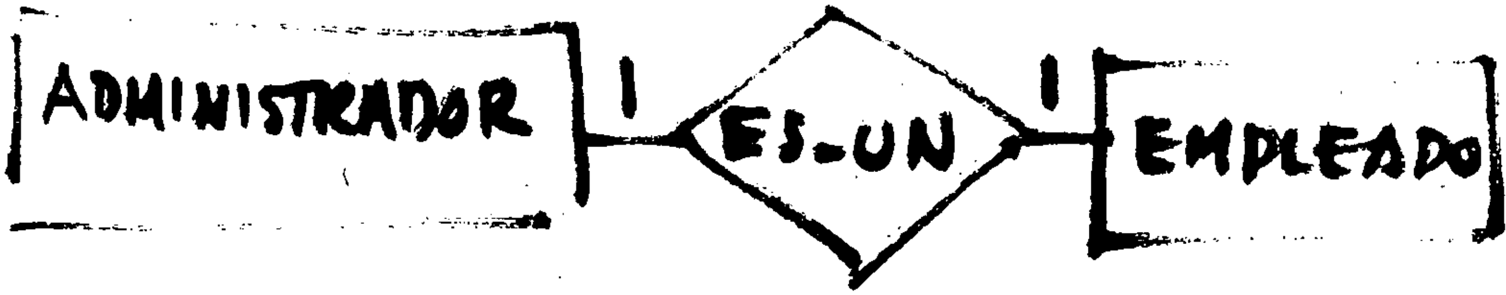
3. RELACIONES

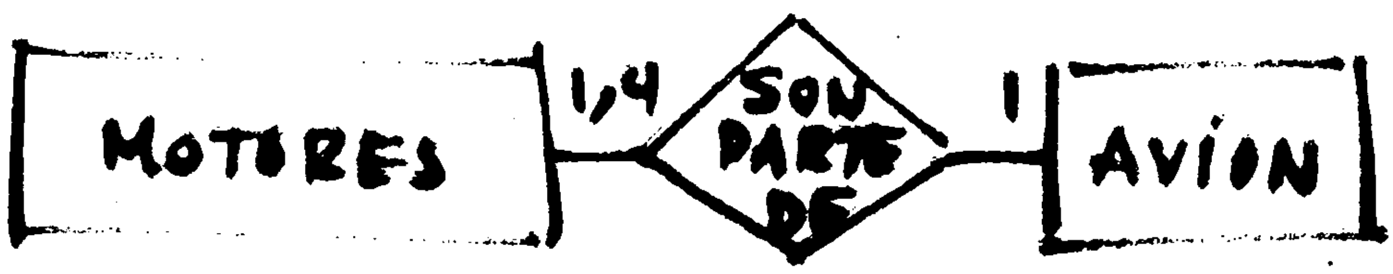
   REPRESENTAN ASOCIACIONES ENTRE LOS OBJETOS.


ADMINISTRADOR
1
ES
JEFE
DE
1
DPTO


ADMINISTRADOR
1
SUPER
VISA
A
N
EMPLEADO

EMPLEADO
N
ESTA
ASIGNADO
A
M
PROYECTO


ADMINISTRADOR
1
ES-UN
1
EMPLEADO


MOTORES
1,4
SON PARTE DE
1
AVION

ANALISIS Y DISEÑO ORIENTADO A OBJETOS

1. IDENTIFICACION DE LOS OBJETOS
2. ESPECIFICACION DE LOS ATRIBUTOS
3. DEFINICION DE LAS OPERACIONES
4. ESTABLECER LAS RELACIONES DE HERENCIA PARTE_TODO Y DE ASOCIACIÓ
5. DEFINICION DE PROTOCOLOS
   - DESCRIPCION
   - IMPLEMENTACION
6. ESTABLECER POLIMORFISMO

CONCEPTOS DE PROGRAMACION ORIENTADA A OBJETOS

P006

```
// Programa 0 ... Objetos

#include <stdio.h>
#include <string.h>
#include <conio.h>

#define BYTE unsigned short int
typedef char STRING[255];


class Letreros {
        BYTE Xl,Yl,Lon;
        STRING c;
public:
        void Iniciar(BYTE x, BYTE y, STRING s);
        void Exhibir(void);
};

class Campos {
        BYTE Xc,Yc;
public:
        void Iniciar(BYTE x, BYTE y);
        void Mostrar(void);
};

// Funciones asociadas de la clase letreros.

void Letreros::Iniciar(BYTE x, BYTE y, STRING s) {
        Xl = x; Yl = y; strcpy(c,s);
}

void Letreros::Exhibir(void) {
        gotoxy(Xl,Yl); puts(c);
}

// Funciones asociadas de la clase campos.

void Campos::Iniciar(BYTE x, BYTE y) {
        Xc = x; Yc = y;
}

void Campos::Mostrar(void) {
        gotoxy(Xc,Yc);
        puts("**********"); gotoxy(Xc,Yc); getch();
}

void main(void) {
//      STRING Texto = "Nombre: ";
        Letreros ElLetrero;
        Campos ElCampo;

        clrscr();
        ElLetrero.Iniciar(10,10,"Nombre: ");  // ...,Texto);
        ElLetrero.Exhibir();
        ElCampo.Iniciar(18,10);
        ElCampo.Mostrar();
}
```

```
// Programa 1 ... Objetos con constructores y destructores

#include <stdio.h>
#include <string.h>
#include <conio.h>

#define BYTE unsigned short int

class Letreros {
        BYTE X1,Y1,Lon;
        char *p;
public:
        Letreros (BYTE x, BYTE y, char *q);
        ~Letreros(void);
        void Exhibir(void);
};

class Campos {
        BYTE Xc,Yc;
public:
        Campos(BYTE x, BYTE y);
        void Mostrar(void);
};

// Funciones asociadas de la clase letreros ..

Letreros::Letreros(BYTE x, BYTE y, char *q) {
        X1 = x; Y1 = y; Lon = strlen(q);
        p = new char[Lon+1]; // p = (char *)malloc(Lon + 1);
        strcpy(p,q);
}

Letreros::~Letreros(void) {
        delete p; // free(p);
}

void Letreros::Exhibir(void) {
        gotoxy(X1,Y1); puts(p);
}

// Funciones asociadas de la clase campos.

Campos::Campos(BYTE x, BYTE y) {
        Xc = x; Yc = y;
}

void Campos::Mostrar(void) {
        gotoxy(Xc,Yc); puts("***********"); gotoxy(Xc,Yc); getch();
}

void main(void) {
        Letreros ElLetrero(10,10,"Nombre: ");
        Campos ElCampo(18,10);

        clrscr();
        ElLetrero.Exhibir();
        ElCampo.Mostrar();
}
```

```
// Programa 2 ... Herencia

#include <stdio.h>
#include <string.h>
#include <conio.h>

#define BYTE unsigned short int

class Letreros {
        BYTE X1,Y1,Lon;
        char *p;
public:
        Letreros (BYTE x, BYTE y, char *q);
        ~Letreros(void);
        void Exhibir(void);
};

class Campos: public Letreros {
        BYTE Xc,Yc;
public:
        Campos(BYTE x1, BYTE y1, char *q, BYTE x2, BYTE y2);
        void Mostrar(void);
};

// Funciones asociadas de la clase letreros.

Letreros::Letreros(BYTE x, BYTE y, char *q) {
        X1 = x; Y1 = y; Lon = strlen(q);
        p = new char[Lon+1]; // p = (char *)malloc(Lon + 1);
        strcpy(p,q);
}

Letreros::~Letreros(void) {
        delete p; // free(p);
}

void Letreros::Exhibir(void) {
        gotoxy(X1,Y1); puts(p);
}

// Funciones asociadas de la clase campos.

Campos::Campos(BYTE x1, BYTE y1, char *q, BYTE x2, BYTE y2)
       :Letreros(x1, y1, q) {
        Xc = x2; Yc = y2;
}

void Campos::Mostrar(void) {
        gotoxy(Xc,Yc); puts("***********"); gotoxy(Xc,Yc); getch();
}

void main(void) {
        Campos LaLinea(10, 10, "Nombre: ", 18, 10);

        clrscr();
        LaLinea.Exhibir();
        LaLinea.Mostrar();
}
```

```
// Programa 3 ... Efecto colateral de la herencia
#include <stdio.h>
#include <string.h>
#include <conio.h>
#   lude <stdlib.h>

#define BYTE unsigned short int

class Mensajes1 {
       BYTE X0, Y0, Lon;
public:
       Mensajes1(BYTE x, BYTE y, char *q);
       ~Mensajes1(void);
       void Exhibir(void);
       void AVideo(void);
protected:
       char *p;
};

class Mensajes2: public Mensajes1 {
public:
       Mensajes2(BYTE x, BYTE y, char *q);
       void AVideo(void);
};

// Funciones asociadas a la Clase Mensajes1

Mensajes1::Mensajes1(BYTE x, BYTE y, char *q) {
       X0 = x; Y0 = y; Lon = strlen(q);
       p = (char *)malloc(Lon + 1); // p = new char[Lon + 1];
       strcpy(p,q);
}

Mensajes1::~Mensajes1(void){
       free(p); // delete p;
}

void Mensajes1::Exhibir(void) {
       gotoxy(X0, Y0); AVideo();
}

void Mensajes1::AVideo(void) {
       putchar('x'); puts(p);
}

// Funciones asociadas a la Clase Mensajes2

Mensajes2::Mensajes2(BYTE x, BYTE y, char *q)
          :Mensajes1(x, y, q) {
}

void Mensajes2::AVideo(void) {
       putchar('*'); puts(p);
}

/   a función principal

void main(void){
       Mensajes1 ElMensaje1(31, 10, "Este es el mensaje 1");
       Mensajes2 ElMensaje2(31, 11, "Este es el mensaje 2");

       clrscr();
```

```
// Programa 4 ... Solución del efecto colateral de la herencia
//                mediante funciones virtuales.
#include <stdio.h>
#include <string.h>
  -lude <conio.h>
  clude <stdlib.h>

#define BYTE unsigned short int

class Mensajes1 {
      BYTE X0, Y0, Lon;
public:
      Mensajes1(BYTE x, BYTE y, char *q);
      ~Mensajes1(void);
      void Exhibir(void);
      virtual void AVideo(void);
protected:
      char *p;
};

class Mensajes2: public Mensajes1 {
public:
      Mensajes2(BYTE x, BYTE y, char *q);
      virtual void AVideo(void);
};

// Funciones asociadas a la Clase Mensajes1

Mensajes1::Mensajes1(BYTE x, BYTE y, char *q) {
      X0 = x; Y0 = y; Lon = strlen(q);
      p = (char *)malloc(Lon + 1); // p = new char[Lon + 1];
      strcpy(p,q);
}

Mensajes1::~Mensajes1(void){
      free(p); // delete p;
}

void Mensajes1::Exhibir(void) {
      gotoxy(X0, Y0); AVideo();
}

void Mensajes1::AVideo(void) {
      putchar('x'); puts(p);
}

// Funciones asociadas a la Clase Mensajes2

Mensajes2::Mensajes2(BYTE x, BYTE y, char *q)
         :Mensajes1(x, y, q) {
}

void Mensajes2::AVideo(void) {
      putchar('*'); puts(p);
}

/   a función principal

void main(void){
      Mensajes1 ElMensaje1(31, 10, "Este es el mensaje 1");
      Mensajes2 ElMensaje2(31, 11, "Este es el mensaje 2");

      clrscr();
```

```
// Programa 06 ... Objetos anidados.
#include <stdio.h>
#include <string.h>
#include <conio.h>

typedef char STRING[255];

class Interna {
        STRING c;
public:
        Interna(STRING t) { strcpy(c,t); }
        void Escribe(void) { gotoxy(32,13); puts(c); }
};

class Externa {
        STRING s;
        Interna LaClaseInterna;
public:
        Externa(STRING t)
                : LaClaseInterna("La clase interna") { strcpy(s,t); }
        void Escribe(void) { gotoxy(32,14); puts(s); }
        void EscribeClaseInterna(void) { LaClaseInterna.Escribe(); }
};

void main(void){
        Externa LaClaseExterna("La clase externa");

        clrscr();
        LaClaseExterna.EscribeClaseInterna();
        LaClaseExterna.Escribe();
        getch();
}
```

```
 // Programa 5 ... Objetos dinámicos
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>

#define BYTE unsigned short int

class Mensajes1 {
        BYTE X0, Y0, Lon;
public:
        Mensajes1(BYTE x, BYTE y, char *q);
        ~Mensajes1(void);
        void Exhibir(void);
        virtual void AVideo(void);
protected:
        char *p;
};

class Mensajes2: public Mensajes1 {
public:
        Mensajes2(BYTE x, BYTE y, char *q);
        virtual void AVideo(void);
};

// Funciones asociadas a la Clase Mensajes1

Mensajes1::Mensajes1(BYTE x, BYTE y, char *q) {
        X0 = x; Y0 = y; Lon = strlen(q);
        p = (char *)malloc(Lon + 1); // p = new char[Lon + 1];
        strcpy(p,q);


Mensajes1::~Mensajes1(void) {
        delete p; // free(p);
}

void Mensajes1::Exhibir(void) {
        gotoxy(X0, Y0); AVideo();
}

void Mensajes1::AVideo(void) {
        puts(p);
}

// Funciones asociadas a la Clase Mensajes2

Mensajes2::Mensajes2(BYTE x, BYTE y, char *q)
                :Mensajes1(x, y, q) {
}

void Mensajes2::AVideo(void){
        highvideo(); puts(p); normvideo();
}

// La función principal

void main(void){
        Mensajes1 *s;
        Mensajes2 *t;

        clrscr();
        s = new Mensajes1(30, 10, "Este es el mensaje 1");
```

```
    t = new Mensajes2(30, 11, "Este es el mensaje 2");
    t -> Exhibir();
    delete t;
    getch();
}
```

```
// Programa 7 ... Funciones asociadas en línea, 1ra forma.

#include <stdio.h>
#include <string.h>
#   lude <conio.h>

#define BYTE unsigned short int
typedef char STRING[255];


class Letreros {
        BYTE X1,Y1,Lon;
        STRING c;
public:
        void Iniciar(BYTE x, BYTE y, STRING s);
        void Exhibir(void);
};

class Campos {
        BYTE Xc,Yc;
public:
        void Iniciar(BYTE x, BYTE y);
        void Mostrar(void);
};

// Funciones asociadas de la clase letreros.

inline void Letreros::Iniciar(BYTE x, BYTE y, STRING s) {
        X1 = x; Y1 = y; strcpy(c,s);
}

inline void Letreros::Exhibir(void) {
        gotoxy(X1,Y1); puts(c);
}

// Funciones asociadas de la clase campos.

inline void Campos::Iniciar(BYTE x, BYTE y) {
        Xc = x; Yc = y;
}

inline void Campos::Mostrar(void) {
        gotoxy(Xc,Yc);
        puts("***********"); gotoxy(Xc,Yc); getch();
}

void main(void) {
//      STRING Texto = "Nombre: ";
        Letreros ElLetrero;
        Campos ElCampo;

        clrscr();
        ElLetrero.Iniciar(10,10,"Nombre: ");  // ...,Texto);
        ElLetrero.Exhibir();
        ElCampo.Iniciar(18,10);
        ElCampo.Mostrar();
}
```

```
// Programa 8 ... Funciones asociadas en línea, 2da forma.

#include <stdio.h>
#include <string.h>
#include <conio.h>

#define BYTE unsigned short int
typedef char STRING[255];


class Letreros {
        BYTE X1,Y1,Lon;
        STRING c;
public:
        void Iniciar(BYTE x, BYTE y, STRING s) { X1 = x; Y1 = y;
                strcpy(c,s); }

        void Exhibir(void) { gotoxy(X1, Y1); puts(c); }
};

class Campos {
        BYTE Xc,Yc;
public:
        void Iniciar(BYTE x, BYTE y) { Xc = x; Yc = y; }

        void Mostrar(void) { gotoxy(Xc, Yc); puts("************");
                gotoxy(Xc, Yc); getch(); }
};

void main(void) {
//      STRING Texto = "Nombre: ";
        Letreros ElLetrero;
        Campos ElCampo;

        clrscr();
        ElLetrero.Iniciar(10,10,"Nombre: ");  // ...,Texto);
        ElLetrero.Exhibir();
        ElCampo.Iniciar(18,10);
        ElCampo.Mostrar();
}
```