Universidad Tecnológica Nacional

Facultad Regional Resistencia

Diseño de Sistemas

Profesor: *A.U.S. Gustavo Marcelo Torossi*

Diseño Estructurado

Diseño Estructurado De Sistemas

Tabla de Contenidos

CONCEPTOS BASICOS

Unidad 0: Revisión de Conceptos Fundamentales

Revisión del ciclo de desarrollo de Sistemas

Enfoque clásico ⟶ *Desarrollo en cascada o ascendente*

- Las fases se suceden en orden estrictamente secuencial
- Nada está hecho hasta que todo este terminado
- Las fallas más triviales se encuentran al comienzo del período de prueba y las más graves al final
- La eliminación de fallas suele ser extremadamente difícil durante las últimas etapas de prueba del sistema

Enfoque estructurado ⟶ *Desarrollo descendente o iterativo*

- Las actividades pueden desarrollarse en forma paralela con progresivos niveles de refinamiento
- Existe retroalimentación entre actividades
- Las fallas y desvíos se detectan y corrigen tempranamente

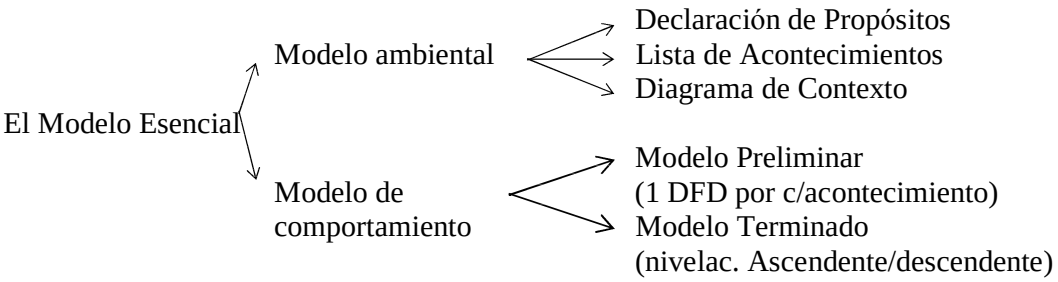
Actividades del Análisis / actividades del Diseño / actividades de Implementación.

La actividad de *Análisis* parte de los requerimientos observados durante el proceso de relevamiento y estudio del *domino del problema* y arroja como resultado lo *QUE debe hacer* el sistema para brindar una *solución* al problema del usuario, independientemente de la naturaleza de la tecnología que se use para su implementación. El análisis transforma las políticas del usuario y el esquema del proyecto en una especificación estructurada.

La actividad de *Diseño* se dedica a asignar porciones de la especificación estructurada resultante del proceso de análisis (también conocida como modelo esencial) a procesadores adecuados (sean máquinas o humanos), y a labores apropiadas (tareas, particiones, etc.) dentro de cada procesador. Dentro de cada labor, la actividad de diseño se dedica a la creación de una jerarquía apropiada de módulos de programas y de interfaces entre ellos, para implantar la especificación creada durante el análisis. Además la actividad del diseño se ocupa de la transformación del modelo de datos de entidad-relación en un diseño de base de datos.

La actividad de *Implementación* o implantación incluye la codificación y la integración de módulos en un esqueleto progresivamente más completo del sistema final. Por eso, esta actividad incluye tanto programación estructurada como implantación descendente.

Los Modelos del Análisis



El Modelo de Implantación del Usuario

Es el punto de inflexión entre la etapa de análisis y la etapa de diseño. El modelo de implementación del usuario especifica un conjunto de *restricciones* que el usuario deseará imponer al grupo de desarrollo y condicionarán al diseñador.
Los aspectos más importantes que se especifican en el modelo de implementación del usuario son:

- *Delimitación de la frontera de automatización:* distribución del modelo esencial entre personas y máquinas: el usuario puede tomar diferentes actitudes frente a este punto, pero lo que debe tenerse presente es que siempre es el usuario el que finalmente tiene la responsabilidad de fijar la frontera de automatización. El usuario puede fijar entre las siguientes alternativas
  - ⇒ Al usuario no le interesa donde está la frontera de automatización, dejando librado al diseñador la desición de establecerla.
  - ⇒ El usuario escoge un sistema totalmente automatizado
  - ⇒ El usuario escoge un sistema totalmente manual

- *Detalle de la interacción humano-máquina:* especifica todos los aspectos del diseño de la interfaz entre el sistema y el entorno. Los aspectos mas importantes a considerar en este punto son:
  - ⇒ Elección de dispositivos de E/S
  - ⇒ Formato de las entradas que fluyen desde los terminadores hasta el sistema
  - ⇒ Formato de las salidas que fluyen desde el sistema hacia los terminadores
  - ⇒ Secuencia y tiempos de entradas y salidas en un sistema en línea, navegaciones de pantalla
  - ⇒ Métodos de codificación a utilizar para el ingreso de datos

- *Actividades de apoyo manual que se podrían requerir:* actividades 'no esenciales' que deben agregarse al sistema por no disponerse de una tecnología perfecta e ideal. Pueden representarse como burbujas adicionales en el modelo esencial. Los casos típicos son:
  - ⇒ Controles de posibles fallas humanas/técnicas (ingreso de datos al sistema, realización de cálculos, dispositivos de almacenamiento, salida de datos del sistema)
  - ⇒ Operación del sistema en producción

- *Restricciones operativas que el usuario desea imponer al sistema:* son restricciones que afectarán la configuración de hw, sistema operativo, telecomunicaciones, lenguaje de programación. Los aspectos típicos son:
  - ⇒ Volumen de los datos
  - ⇒ Tiempo de respuesta en sistemas On-line
  - ⇒ Restricciones políticas sobre modalidades de implantación
  - ⇒ Restricciones ambientales
  - ⇒ Restricciones de seguridad y confiabilidad (mtbf, mttr)
  - ⇒ Restricciones de seguridad (controles de acceso al sistema)

- Agregado de procesos de arranque y apagado del sistema.

Los Modelos del Diseño

La actividad de diseño implica la realización de una serie de modelos.
Los modelos más importantes del diseño son:

- Modelo de Implantación de Sistemas
  - Modelo de Procesadores
  - Modelo de Tareas
- Modelo de Implantación de Programas
- Modelo de Implantación de Almacenes

El Modelo de Procesadores

Asigna el modelo esencial a distintos procesadores y determina la arquitectura de comunicación entre ellos. Implica la asignación de procesos procesos y almacenes a los procesadores.
Según la cantidad de procesadores utilizados y las forma de comunicación entre ellos se tienen distintas configuraciones.

Tipos de configuración típicas:

- Centralizada (host based)
- Descentralizada
- Mixta
- Distribuida / C-S

Centralizada: Asigna el modelo esencial completo a un único procesador central.

Descentralizada: Se asignan partes del modelo esencial a diferentes procesadores los cuales trabajan en forma independiente.
En el caso de almacenes que deban ser compartidos por procesos asignados a diferentes procesadores, los mismos deberán duplicarse, y mantenerse copias actualizadas en cada procesador.

Mixta: Puede darse una combinación de los casos anteriores. Es común la existencia de un sistema central que consolida toda la información de la organización y que en

diferentes unidades operativas que no este conectadas a dicho procesador central existan sistemas satélites que implementan algunos procesos con almacenes con datos locales.

Distribuida: Se asignan partes del modelo esencial a diferentes procesadores los cuales están comunicados de alguna forma y sobre los que corre un sistema operativo distribuido. En este caso el usuario ve al conjunto de procesadores como un único recurso computacional.

Cliente/Servidor: Se distribuyen partes del proceso en diferentes procesadores.
C/S 2 niveles: Servidor de B.D. / Aplicación-Presentación en Estación de Trabajo
C/S 3 niveles: Servidor de B.D. / Servidor de Aplicación / Presentación en Est.Trab.

Tipos de configuración de comunicación entre procesadores:
- Conexión directa entre procesadores (canal / red local / otros)
- Enlace de telecomunicaciones entre procesadores
- Enlace indirecto: los datos son transferidos de un procesador a otro via algún medio de almacenamiento (cinta, cd, dskte, etc)

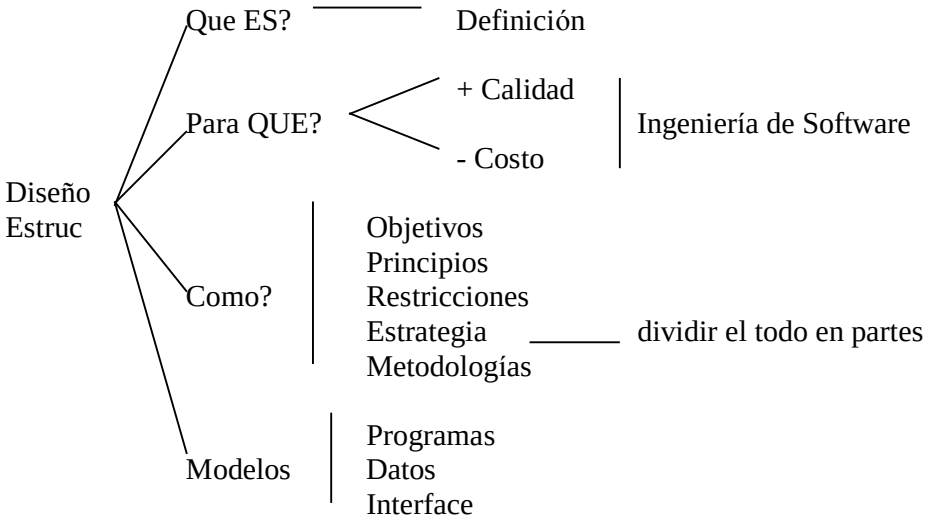
Factores que influyen en la configuración de procesadores:
- Costo
- Eficiencia
- Seguridad (procesadores y datos en lugares seguros)
- Confiabilidad (separar los procesos en varios procesadores, proc.redundantes)
- Restricciones políticas y operacionales.

El Modelo de Tareas
Dentro de cada procesador definido en el modelo anterior, deben asignarse procesos a diferentes tareas o particiones.
En muchos sistemas operativos modernos, el manejo de tareas es transparente al desarrollador.
Las tareas pueden categorizarse típicamente en *Interactivas, Batch,* y en *Tiempo Real.*
Para la mayoría de los sistemas administrativos es importante determinar que partes del modelo esencial se asignaran a tareas interactivas y cuales a tareas batch.
La comunicación entre tareas normalmentes es provista via el sistema operativo.

El Modelo de Programas
Para cada tarea debe desarrollarse un modelo de programa. De esto se encarga principalmente el D.E.

Cuadro Sinóptico

El siguiente cuadro sinóptico resume los aspectos principales del diseño estructurado:



Unidad 1: Introducción al Diseño Estructurado

Conceptos Generales Sobre el Diseño

Definición: "Diseño es el proceso de aplicar distintas técnicas y principios con el propósito de definir un dispositivo, proceso, o sistema, con los suficientes detalles como para permitir su realización física" (E.S.Taylor, *An Interim Report on Engineering Design*, Massachusetts Institute of Technology, 1959)

El objetivo del diseñador es producir un modelo de una entidad que se construirá más adelante. El proceso por el cual se desarrolla el modelo combina:

- la intuición y los criterios en base a la experiencia de construir entidades similares
- un conjunto de principios y/o heurísticas que guían la forma en la que se desarrolla el modelo
- un conjunto de criterios que permiten discernir sobre calidad
- un proceso de iteración que conduce finalmente a una representación del diseño final

La actividad de Diseño se dedica a asignar porciones de la especificación estructurada (también conocida como modelo esencial) a procesadores adecuados (sean máquinas o humanos) y a labores apropiadas (o tareas, particiones, etc.) dentro de cada procesador. Dentro de cada labor, la actividad de diseño se dedica a la creación de una jerarquía apropiada de módulos de programas y de interfases entre ellos para implantar la especificación creada en la actividad de análisis. Además, la actividad de diseño se ocupa de la transformación de modelos de datos de entidad-relación en un diseño de base de datos. (Ed.Yourdon – "Análisis Estructurado Moderno")

.

1.1 Qué es diseño estructurado?

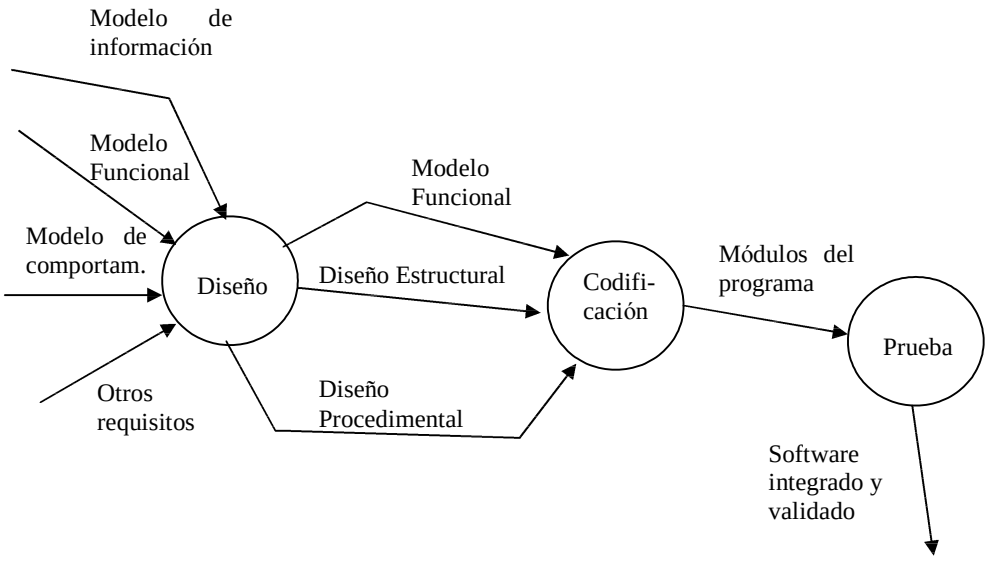
Definición: "Diseño estructurado es el proceso de decidir que componentes, y la interconexión entre los mismos, para solucionar un problema bien especificado".

El diseño es una actividad que comienza cuando el analista de sistemas ha producido un conjunto de requerimientos funcionales lógicos para un sistema, y finaliza cuando el diseñador ha especificado los componentes del sistema y las relaciones entre los mismos.
Frecuentemente analista y diseñador son la misma persona, sin embargo es necesario que se realice un cambio de enfoque mental al pasar de una etapa a la otra. *Al abordar la etapa de diseño, la persona debe quitarse el sombrero de analista y colocarse el sombrero de diseñador*[i].

Una vez que se han establecido los requisitos del software (en el análisis), el diseño del software es la primera de tres actividades técnicas: *diseño, codificación, y prueba.* Cada actividad transforma la información de forma que finalmente se obtiene un software para computadora válido.
En la figura se muestra el flujo de información durante la fase de desarrollo. Los requisitos del sistema, establecidos mediante los *modelos de información, funcional* y

[i] Ver "Seis sombreros para pensar" - Edward De Bono

de *comportamiento*, alimentan el proceso del diseño. Mediante alguna metodología (en nuestro caso, estructurada basada en el flujo de información) se realiza el diseño estructural, procedimental, y de datos.

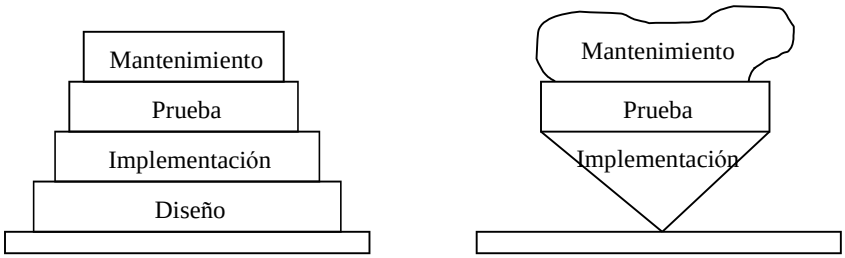


El *diseño de datos* transforma el modelo del campo de información, creado durante el análisis, en las estructuras de datos que se van a requerir para implementar el software.

El *diseño estructural* define las relaciones entre los principales elementos estructurales del programa. El objetivo principal del diseño estructural es desarrollar una estructura de programa modular y representar las relaciones de control entre los módulos.

El *diseño procedimental* transforma los elementos estructurales en una descripción procedimental del software. El diseño procedimental se realiza después de que se ha establecido la estructura del programa y de los datos. Define los algoritmos de procesamiento necesarios.
Concluido el diseño se genera el código fuente y para integrar y validar el software, se llevan a cabo pruebas de testeo.

Las fases del diseño, codificación y prueba absorben el 75% o más del coste de la ingeniería del software (excluyendo el mantenimiento). Es aquí donde se toman *decisiones* que afectarán finalmente al éxito de la implementación del programa y, con igual importancia, a la facilidad de mantenimiento que tendrá el software. Estas decisiones se llevan a cabo durante el diseño del software, haciendo que sea un paso *fundamental* de la fase de desarrollo.
La importancia del diseño del software se puede sentar con una única palabra: *calidad*. El diseño es el proceso en el que se asienta la calidad del desarrollo del software. El diseño produce las representaciones del software de las que puede evaluarse su calidad.
El diseño sirve como base para todas las posteriores etapas del desarrollo y de la fase de mantenimiento. Sin diseño nos arriesgamos a construir un sistema inestable, un sistema que falle cuando se realicen pequeños cambios, un sistema que pueda se difícil de probar, un sistema cuya calidad no pueda ser evaluada hasta más adelante en el proceso de ingeniería de software, cuando quede poco tiempo y se haya gastado ya mucho dinero.



Con Diseño *Sin Diseño*

1.2 Objetivos Del Diseño Estructurado

"El diseño estructurado, tiende a transformar el desarrollo de software de una práctica artesanal a una disciplina de ingeniería".

- Eficiencia
- Mantenibilidad
- Modificabilidad
- Flexibilidad
- Generalidad
- Utilidad

"Diseño" significa planear la forma y método de una solución. Es el proceso que determina las características principales del sistema final, establece los límites en performance y calidad que la mejor implementación puede alcanzar, y puede determinar a que costos se alcanzará.
El diseño se caracteriza usualmente por un gran número de decisiones técnicas individuales. En orden de transformar el desarrollo de software en una disciplina de ingeniería, se debe sistematizar tales decisiones, hacerlas más explícitas y técnicas, y menos implícitas y artesanales.

Un ingeniero no busca simplemente *una* solución, busca la *mejor* solución, dentro de las *limitaciones* reconocidas, y realizando *compromisos* requeridos en el trabajo del mundo real.
En orden de convertir el diseño de sistemas de computadoras en una disciplina de ingeniería, previo a todo, debemos definir *objetivos técnicos claros* para los programas de computadora. Es esencial además comprender las *restricciones* primarias que condicionan las soluciones posibles.
Para realizar decisiones concisas y deliberadas, debemos identificar los *puntos de decisión* .
Finalmente necesitamos una *metodología* que nos asista en la *toma de decisiones*.
Dadas estas cosas: objetivos, restricciones, decisiones reconocidas, y una metodología efectiva, podemos obtener soluciones de ingeniería, y no artesanales.

Diseño estructurado y calidad del software

Un concepto importate a clarificar es el de *calidad.* Desafortunadamente, muchos diseñadores se conforman con un sistema que "funcione" sin reparar en un *buen* sistema.

Una corriente de pensamiento estima que un programa es bueno si sus algoritmos son astutos y no obvios a otro programador; esto refleja la "inteligencia" del programador.

Otra escuela de pensamiento asocia calidad con incremento de la velocidad de ejecución y disminución de los requerimientos de memoria central. Estos son aspectos de un concepto más amplio: *eficiencia*. En general, se busca diseños que hagan un uso inteligente de los *recursos*. Estos recursos no incluyen solamente procesador y memoria, también incluyen almacenamiento secundario, tiempo de periféricos de entrada salida, tiempo de líneas de teleproceso, tiempo de personal, y más.

Otra medida de calidad es la *confiabilidad.* Es importante notar que si bien la confiabilidad del software puede ser vista como un problema de depuración de errores en los programas, es también un problema de diseño. La confiabilidad se expresa en como MTBF (mean time between fairules: tiempo medio entre fallas).
Un concepto muy relacionado a la confiabilidad y de suma importancia es el de *mantenibilidad*. Podemos definir la mantenibilidad como:

$$\text{Mantenibilidad del sistema} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}.$$

donde:
MTBF: tiempo medio entre fallas
MTTR: tiempo medio de reparación (mean time to repair)

Diremos que un sistema es mantenible si permite la detección, análisis, rediseño, y corrección de errores fácilmente.

En tanto la mantenibilidad afecta la viabilidad del sistema en un entorno relativamente constante, la *modificabilidad* influye en los costos de mantener un sistema viable en condiciones de cambio de requerimientos. La modificabilidad es la posibilidad de

realizar modificaciones y extensiones a partes del sistema, o agregar nuevas partes con facilidad (no corrección de errores).

En estudios realizados se determinó que las organizaciones abocadas al procesamiento de datos invierten aproximadamente un 50% del presupuesto en mantenimiento de los sistemas, involucrando esto corrección de errores y modificaciones, razón por la cual la mantenibilidad y la modificabilidad son dos objetivos primarios en el diseño de software.

La *flexibilidad* representa la facilidad de que el mismo sistema pueda realizar variaciones sobre una misma temática, sin necesidad de modificaciones.
La *generalidad* expresa el alcance sobre un determinado tema.
Flexibilidad y generalidad son dos objetivos importantes en el diseño de sistemas del tipo de propósitos generales.

La *utilidad* o facilidad de uso es un factor importante que influye en el éxito del sistema y sus aceptación por parte del usuario. Un sistema bien diseñado pero con interfaces muy "duras" tiende a ser resistido por los usuario.

Finalmente diremos que eficiencia, mantenibilidad, modificabilidad, flexibilidad, generalidad, y utilidad, son componentes de la *calidad* objetiva de un sistema.
En términos simples también diremos que nuestro objetivo primario es obtener sistemas de *costo mínimo.* Es decir, es nuestro interés obtener sistemas económicos para desarrollar, operar, mantener y modificar.

1.3 Restricciones, compromisos, y decisiones del Diseño

Podemos ver los objetivos técnicos del diseño como constituyendo una "función objetivo" de un problema de optimización, la cual se desea maximizar, sujeta a ciertas restricciones.
Como regla, las restricciones sobre un proceso de diseño de un sistema, caen en dos categorías: *restricciones de desarrollo* y *restricciones operacionales*.
Las restricciones de desarrollo son limitaciones al consumo de recursos durante el período de desarrollo, y pueden ser expresadas en términos generales o descomponerla en su partes como ser tiempo de máquina y horas-hombre. Dentro de las restricciones de desarrollo, entran también las restricciones de *planificación.* Estas se refieren a metas y plazos a ser cumplidos ("el módulo X debe terminarse para Febrero").
Las restricciones operacionales pueden ser expresadas en términos técnicos, como ser máximo tamaño de memoria disponible, máximo tiempo de respuesta aceptable, etc.

El carácter de muchas decisiones de diseño no fija límites rígidos, si no un intervalo de tolerancia, dentro del cual el diseñador puede moverse a costa de variaciones en otros aspectos del sistema. Por ejemplo se puede priorizar eficiencia, en detrimento de facilidad de mantenimiento, o velocidad de ejecución contra tamaño de memoria utilizada.
La esencia del diseño en el mundo real y las decisiones inherentes al mismo es obtener una solución de *compromiso.*

El diseño total es el resultado acumulativo de un gran número de *decisiones técnicas* incrementales.

1.4 Principios utilizados por el diseño estructurado[ii]

1.4.1 Abstracción

La noción psicológica de abstracción permite concentrarse en un problema al mismo nivel de generalización, independientemente de los detalles irrelevantes de bajo nivel. El uso de la abstracción también permite trabajar con conceptos y términos que son familiares al entorno del problema, sin tener que transformarlos a una estructura no familiar.
Cada paso de un proceso de ingeniería de software es un refinamiento del nivel de abstracción de la solución de software.
Conforme nos movemos por diferentes niveles de abstracción, trabajamos para crear *abstracciones* de datos y de procedimientos. Una *abstracción procedural* es una determinada secuencia de instrucciones que tienen una función limitada y específica.
Una *abstracción de datos* es una determinada colección de datos que describen un objeto.

- Rumbaugh: O.O.Modeling and Design

La abstracción es el exámen selectivo de ciertos aspectos de un problema. El objetivo de la abstracción es aislar aquellos aspectos que son importantes para algún propósito y suprimir aquellos aspectos que no son importantes. La abstracción debe realizarse siempre con un propósito, ya que el propósito determina que es y que no es relevante. Muchas abstracciones son posibles sobre una misma cosa, dependiendo de cual sea su propósito.

- Alan Cameron Wills - (Object Expert Jan/Feb 1996)

La abstracción, para mi, es cercana a palabras como "teórico", "esotérico", "académico", e "impráctico". Pero en un sentido en particular, significa la separación de los aspectos más importantes de un determinado problema, del detalle. Este es el único camino que tengo para abordar con mi mente finita cualquier tema complejo.

1.4.2 Refinamiento sucesivo

El *refinamiento sucesivo* es una primera estrategia de diseño descendente propuesta por Niklaus Wirth. La arquitectura de un programa se desarrolla en niveles sucesivos de refinamiento de los detalles procedimentales. Se desarrolla una jerarquía descomponiendo una declaración macroscópica de una función de una forma sucesiva, hasta que se llega a las sentencias del lenguaje de programación.

1.4.3 Modularidad

La arquitectura implica modularidad, el software se divide en componentes con nombres y ubicaciones determinados, que se denominan *módulos,* y que se integran para satisfacer los requisitos del problema.

[ii] Ver 'Ingeniería de Software' de R.Pressman

El tema de la modularidad es tratado extensamente más adelante.

1.4.4 Arquitectura del software

La *arquitectura del software* se refiere a dos características importantes del software de computadoras:

- la estructura jerárquica de los componentes procedimentales (módulos)
- la estructura de datos

1.4.5 Jerarquía de control

La *jerarquía de control*, también denominada *estructura de programa*, representa la organización (frecuentemente jerárquica) de los componentes del programa (módulos) e implica una jerarquía de control. No representa aspectos procedimentales del software, tales como secuencias de procesos, o la repetición de operaciones.

1.4.6 Estructura de datos

La *estructura de datos* es una representación de la relación lógica existente entre los elementos individuales de datos. Debido a que la estructura de la información afectará invariablemente al diseño procedimental final, la estructura de datos es tan importante como la estructura del programa en la representación de la arquitectura del software.

1.4.7 Procedimientos del software

La estructura del programa define la jerarquía de control, independientemente de las decisiones y secuencias de procesamiento. El procedimiento del software se centra sobre los detalles de procesamiento de cada módulo individual.
El procedimiento debe proporcionar una especificación precisa del procesamiento, incluyendo la secuencia de sucesos, los puntos concretos de decisiones, la repetición de operaciones, e incluso la organización/estructura de los datos.

1.4.8 Ocultamiento de la información

El principio de *ocultamiento de la información* sugiere que los módulos se han de caracterizar por decisiones de diseño que los oculten unos a otros. Los módulos deben especificarse y diseñarse de forma que la información (procedimientos y datos) contenida dentro de un módulo sea accesible a otros módulos únicamente a través de las *interfaces* formales establecidas para cada módulo.

Unidad 2: Conceptos Básicos de Diseño Estructurado

2.1 Estrategia del Diseño Estructurado

Cuando se trata con un problema de diseño de reducida envergadura, por ejemplo un sistema que pueda ser desarrollado en un par de semanas, no se tienen mayores problemas, y el desarrollador puede tener todos los elementos del problema "en mente" a la vez. Sin embargo cuando se trabaja en proyectos de gran escala, es difícil que una sola persona sea capaz de llevar todas las tareas y tener en mente todos los elementos a la vez.
El diseño exitoso se basa en un viejo principio conocido desde los días de Julio Cesar: *Divide y conquistarás.*

Específicamente, diremos que el costo de *implementación* de un sistema de computadora podrá minimizarse cuando pueda separarse en partes

- *manejablemente pequeñas*
- *solucionables separadamente*.

Por supuesto la interpretación de manejablemente pequeña varía de persona en persona. Por otro lado muchos intentos de particionar sistemas en pequeñas partes arribaron incrementos en los tiempos de implementación. Esto se debe fundamentalmente al segundo punto: solucionables separadamente En muchos sistemas para implementar la parte A, debemos conocer algo sobre la B, y para implementar algo de B, debemos conocer algo de C.

De manera similar, podemos decir que el costo de *mantenimiento* puede ser minimizado cuando las partes de un sistema son

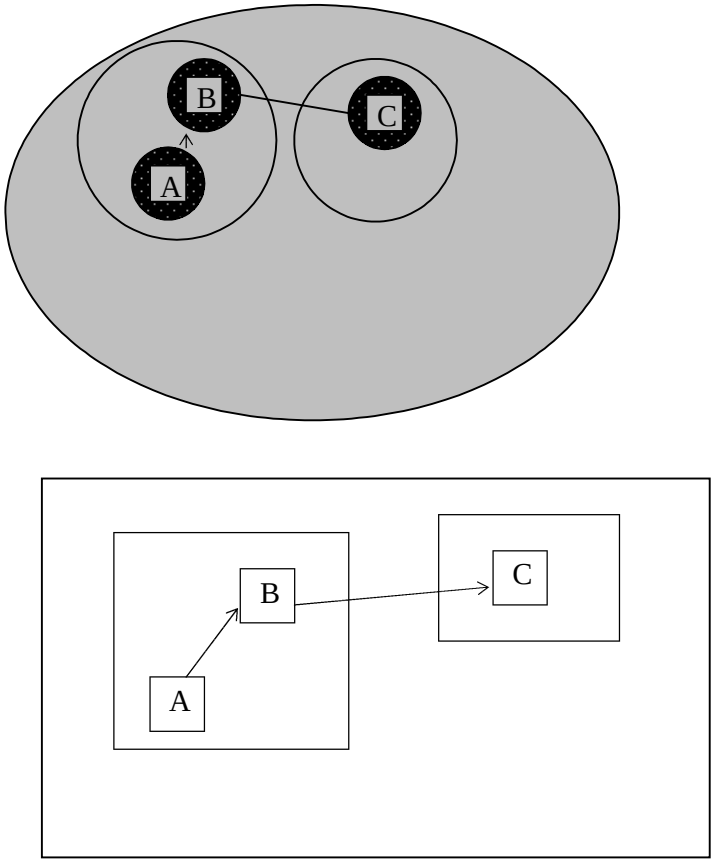
- *fácilmente relacionables con la aplicación*
- *manejablemente pequeñas*
- *corregibles separadamente*

Muchas veces la persona que realiza la modificación no es quien diseño el sistema.
Es importante que las partes de un sistema sean manejablemente pequeñas en orden de simplificar el mantenimiento. Un trabajo de encontrar y corregir un error en una "pieza" de código de 1.000 líneas es muy superior a hacerlo con piezas de 20 líneas. No solo disminuye el tiempo de localizar la falla sino que si la modificación es muy engorrosa, puede reescribirse la pieza completamente. Este concepto de "módulos descartables" ha sido utilizado con éxito muchas veces.
Por otra parte, para minimizar los costos de mantenimiento debemos lograr que cada pieza sea independiente de otra. En otras palabras debemos ser capaces de realizar modificaciones al módulo A sin introducir efectos indeseados en el módulo B.

Finalmente, diremos que el costo de *modificación* de un sistema puede minimizarse si sus partes son

- *fácilmente relacionables con la aplicación*
- *modificables separadamente*

En resumen, podemos afirmar lo siguiente: los costos de implementación, mantenimiento, y modificación, generalmente serán minimizados cuando *cada pieza del sistema corresponda a exactamente una pequeña, bien definida pieza del dominio del problema, y cada relación entre las piezas del sistema corresponde a relaciones entre piezas del dominio del problema.*
En la siguiente figura apreciamos este concepto



2.2 Particionamiento y Organización

Un buen diseño estructurado es un ejercicio de *particionamiento* y *organización* de los componentes de un sistema.
Entenderemos por *particionamiento*, la subdivisión de un problema en subproblemas más pequeños, de tal forma que cada subproblema corresponda a una pieza del sistema.
La cuestión es: Dónde y cómo debe dividirse el problema? Qué aspectos del problema deben pertenecer a la misma pieza del sistema, y cuales a distintas piezas? El diseño estructurado responde estas preguntas con dos principios básicos:

- *Partes del problema altamente interrelacionadas deberán pertenecer a la misma pieza del sistema.*

- *Partes sin relación entre ellas, deben pertenecer a diferentes piezas del sistema sin relación directa.*

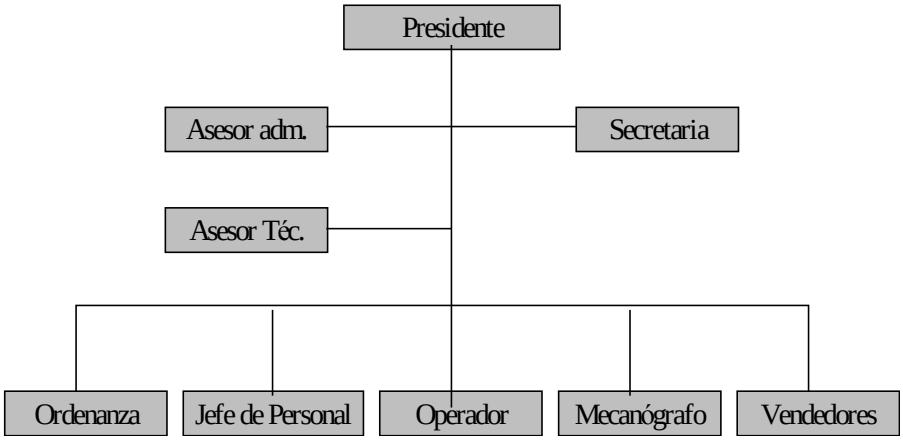
Otro aspecto importante del diseño estructurado es la *organización* del sistema. Debemos decidir como se interrelacionan las partes, y que parte está en relación con cual.
El objetivo es organizar el sistema de tal forma que no existan piezas mas grandes de lo estrictamente necesario para resolver los aspectos del problema que ella abarca. Igualmente importate, es el evitar la introducción de relaciones en el sistema, que no existe en el dominio del problema.

2.3 El concepto de Cajas Negras

Una caja negra es un sistema (o un componente) con entradas conocidas, salidas conocidas, y generalmente transformaciones conocidas, pero del cual no se conoce el contenido en su interior.
En la vida diaria existe innumerable cantidad de ejemplos de uso cotidiano: una radio, un televisor, un automóvil, son cajas negras que usamos a diario sin conocer (en general) como funciona en su interior. Solo conocemos como controlarlos (entradas) y las respuestas que podemos obtener de los artefactos (salidas).
El concepto de caja negra utiliza el principio de ***abstracción.***
Este concepto es de suma utilidad e importancia en la ingeniería en general, y por ende en el desarrollo de software. Lamentablemente muchas veces para poder hacer un uso efectivo de determinado módulo, el diseñador debe revisar su contenido ante posibles contingencias como ser comportamientos no deseados ante determinados valores. Por ejemplo es posible que una rutina haya sido desarrolla para aceptar un determinado rango de valores y falla si se la utiliza con valores fuera de dicho rango, o produce resultados inesperados. Una buena documentación en tales casos, es de utilidad pero no transforma al módulo en una verdadera caja negra. Podríamos hablar en todo caso de "cajas blancas".
Los módulos de programas de computadoras pueden variar en un amplio rango de aproximación al ideal de caja negra. En la mayoría de los casos podemos hablar de "cajas grises".

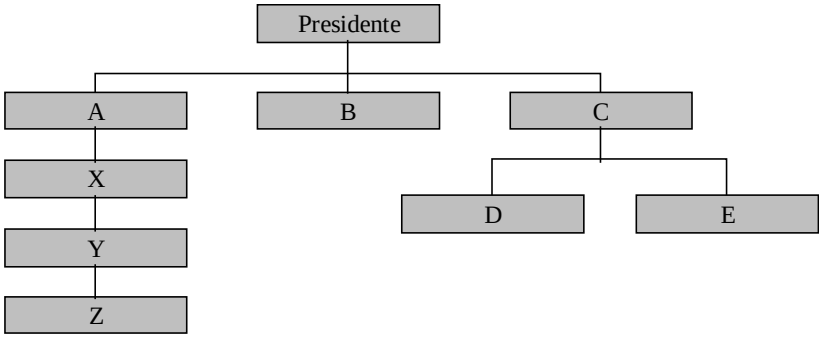
2.4 Comparación entre las estructuras administrativas y el diseño estructurado

Uno de los aspectos más interesantes del diseño de programas es la relación que puede establecerse con las estructuras de organización humanas, particularmente la jerarquía de administración encontrada en la mayoría de las grandes organizaciones.
Observemos por ejemplo el siguiente diagrama de organización de una empresa



A simple vista podemos apreciar que el presidente de la empresa tiene demasiados subordinados, y consecuentemente su trabajo involucrará el manejo de demasiados datos y la toma de demasiadas decisiones, demasiada complejidad, que lo llevará a cometer posibles errores.
Podemos establecer una analogía con la estructura de programas y es razonable pensar que un módulo que tenga demasiados módulos subordinados a quienes controlar, sea sumamente complejo, y susceptible a fallas.

Veamos otro ejemplo



Podemos apreciar a simple vista que la tarea de los jefes A, X, Y, es relativamente trivial y podría se comprimida en una sola jefatura. Estableciendo un comparación con la estructura de programas, si tenemos un módulo cuya única función es llamar a otro, y este a su vez a otro, el cual llama a uno que finalmente realizará la tarea, los módulos intermedios podrán comprimirse un único módulo de control.

Podemos decir que en una organización perfecta, los administradores no realizan ninguna tarea operativa. Su labor consiste en coordinar información entre los subordinados y tomar decisiones. Los niveles inferiores son los que realizan el trabajo operativo. Análogamente, podemos argumentar que los módulos de nivel alto en un

programa o sistema solamente coordinan y controlan la ejecución de los módulos de menor nivel, quienes son los que realizan los cómputos y procesos que el sistema requiere.
Finalmente observaremos que los administradores suministran a sus subordinados únicamente la información que estrictamente necesitan. Análogamente los módulos inferiores solo deben tener acceso a la información que necesitan, y no a otras.

El establecimiento de un paralelo entre las estructuras organizativas humanas y los programas de computadora nos será muy útil en el estudio del diseño estructurado.

2.5 Manejo de la complejidad

En principio diremos que escribir un programa "grande" generalmente lleva más tiempo que escribir un "pequeño". Esto es valedero si medimos "grande" y "pequeño" en unidades apropiadas. Claramente, el número de instrucciones de un programa no es una medida de complejidad ya que existe instrucciones más complejas que otras, y algoritmos más complejos que otros. En realidad lo que diremos es que ***es más difícil resolver un problema difícil!*** , e intentaremos realizar un análisis sobre como disminuir la complejidad de un determinado problema.
Si asumimos que hemos podemos medir por algún método la complejidad de un problema P (no importa aquí que método), diremos que la complejidad del mismo será M(P), y que el costo de realizar un programa que resuelva el problema P será C(P). Los enunciados previos responderán a la siguiente regla:
dados dos problemas P y Q observaremos lo siguiente

$$Si\ M(P) > M(Q) \quad entonces \quad C(P) > C(Q)$$

es decir el costo de resolver un determinado problema es directamente proporcional al tamaño del mismo.

Intentaremos tomar dos problemas separados y en lugar de escribir dos programas, crear un programa combinado. Si juntamos los dos problemas, obtendremos uno mayor que si tomamos los dos por separado. La razón fundamental para no combinar los problemas, es que los humanos tenemos inconvenientes para tratar adecuadamente grandes complejidades, y en la medida que esta se incrementa somos susceptibles a cometer un mayor número de errores. En virtud de esto podemos afirmar que

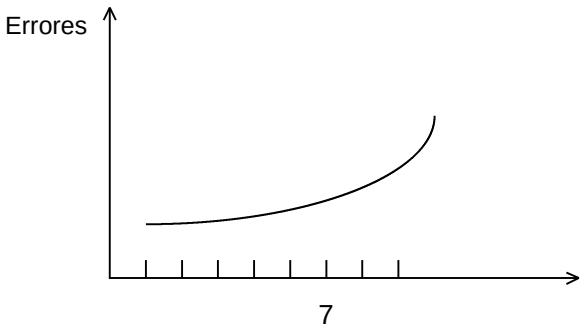
$$M(P+Q) > M(P) + M(Q)$$

y consecuentemente

$$C(P+Q) > C(P) + C(Q)$$

Siempre será preferible crear dos piezas pequeñas que una sola más grande, si ambas solucionan el mismo problema.
Este fenómeno no es solo válida para el campo de la computación. Es verdadero en cualquier campo de la solución de problemas (matemática, física, etc.).
El psicólogo y matemático George Miller, realizó una serie de investigaciones sobre las limitaciones en el procesamiento de información humano. Estos estudios determinaron que la mente humana puede mantener y tratar simultáneamente hasta con siete objetos,

o conceptos. En efecto, la memoria necesaria para la resolución de problemas con múltiples elementos, tiene una capacidad de 7 ± 2 entidades. Por sobre este número la cantidad de errores se incrementa desproporcionadamente en forma no lineal.



Esta es una propiedad de la capacidad de procesamiento de información del cerebro humano bien establecida sobre la que se asienta la descomposición de problemas en subproblemas.
Esto nos lleva a que dado un problema P no trivial, es conveniente descomponerlo en problemas más pequeños ya que se observa que

$$C(P) > C(½P) + C(½P)$$

Ahora bien, cuando se descompone una tarea en dos, si las subtareas no son realmente independientes, al solucionar una de las partes, debe simultáneamente tratarse aspectos de la otra.
Supongamos que descomponemos P en dos partes iguales P' = ½P y P" = ½P, si las partes no son independientes, el costo de resolver el problema entero será:

$$C(P' + I_1 \times P') + C(P'' + I_2 \times P'')$$

donde I_1 es una fracción que representa la interacción de P' con P", e I_2 es una fracción que representa la interacción de P" con P'. Siempre que I_1 e I_2 sean mayores a cero, es obvio que será

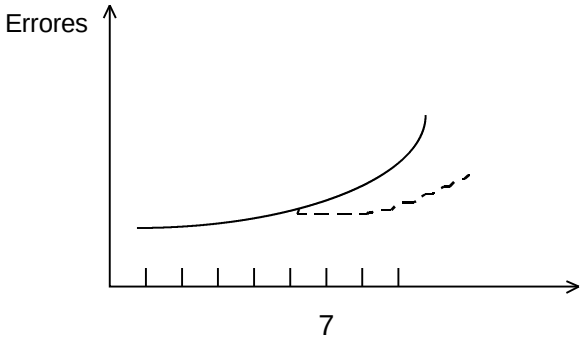
$$C(P' + I_1 \times P') + C(P'' + I_2 \times P'') > C(½P) + C(½P)$$

Si I_1 e I_2 son muy pequeños podremos decir que:

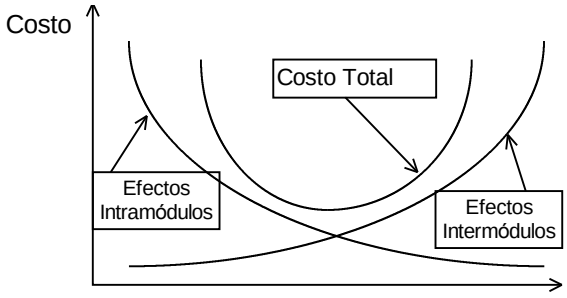
$$C(P) > C(P' + I_1 \times P') + C(P'' + I_2 \times P'')$$

Ahora, la subdivisión de un problema en otros menores tiene un límite. Es obvio que no podemos esperar dividir el sistema en un infinito número de "nadas", y en el caso límite, el desarrollo de un sistema como un número muy grande de piezas independientes es equivalente a desarrollarlo en una sola pieza.
Más adelante analizaremos como determinar el tamaño conveniente de cada parte.

El proceso de factorización de un problema en partes, puede introducir algunos errores, pero en general si se realiza correctamente tiende a aplastar la curva de errores



Como puede sugerirse, la factorización de un sistema en mil módulos de una línea, es equivalente al costo de un módulo de 1000 líneas (y posiblemente mayor). Claramente estos son los extremos de un espectro de alternativas.
A medida que los módulos sean más pequeños, podemos esperar que su complejidad (y costo) disminuyan, pero además, a mayor cantidad de módulos, tendremos mayor posibilidad problemas debido a errores en las conexiones intermódulos. Estas son dos curvas en contraposición



En este punto no estamos preparados para predecir el tamaño ‘‘óptimamente pequeño” de un módulo. En realidad es muy dudoso que pueda establecerce con precisión, pero veremos una serie de principios que nos conducirán a aproximarnos.

2.6 Complejidad en términos humanos

En el punto anterior realizamos un análisis sobre la incidencia de la complejidad en los costos, y como manejarla a través de la subdivisión de un problema en problemas menores. Vimos que muchos de nuestros problemas en diseño y programación se deben a la limitada capacidad de la mente humana para lidiar con la complejidad.
La cuestión ahora es:

Qué es lo complejo para los humanos?

En otras palabras:

Que aspectos del diseño de sistemas y programas son considerados complejos por el diseñador?

Y por extensión

Que podemos hacer para realizar sistemas menos complejos?

Si realizamos una analogía con un puesto de trabajo en una organización, podemos establecer en primer término que un factor que incide decididamente en la complejidad del puesto esta dado por la cantidad de tareas que debe realizar el empleado. Traducido esto a un programa de computadoras es equivalente al *tamaño de un módulo*. Observaremos como regla general que a medida que aumenta la cantidad de sentencias (cantidad de tareas del puesto), aumenta la complejidad del módulo.

Sin embargo, la cantidad de tareas que debe realizar un empleado, no es el único factor de la complejidad de su puesto, ya que obviamente existen tareas más complejas que otras. Se observa que las tareas que involucran decisiones son más complejas que aquellas que son simplemente rutinarias. El equivalente de esto en los programas de computadora son las *sentencias de decisión* que contribuyen a la complejidad de un módulo.

Además de la cantidad y tipo de tareas que debe realizar un empleado, otro factor que incide decisivamente en la complejidad de su trabajo está dado por la cantidad de factores o elementos que necesita "tener en mente" (variables a considerar) para poder llevarlo a cabo. El equivalente de esto en los programas de computadora está determinado por el "espacio" de vida o *alcance de los elementos de dato*, es decir el número de sentencias durante las cuales el estado y valor de un elemento de datos debe ser recordado por el programador en orden de comprender que es lo que hace el módulo.

Finalmente, un puesto de trabajo será más complejo cuando mayor sea el número de subordinados que debe controlar. Traducido esto a los programas de computadora, es la cantidad de módulos subordinados al módulo considerado. Esto se denomina *alcance o amplitud de control*.

Analizando los factores previamente descriptos, se identifican tres factores comunes que afectan la complejidad de los programas:

- la *cantidad* de información que debe ser comprendida correctamente.
- la *accesibilidad* de la información.
- la *estructura* de la información.

Estos factores determinan la probabilidad de error humano en el procesamiento de información de todo tipo.

Unidad 3: La Estructura de los Programas de Computadora

3.1 Programas y Sentencias

Un programa de computadora es un sistema con componentes e interrelaciones. Intentaremos determinar cuales son dichos componentes y como se relacionan.
En primer lugar definiremos los conceptos de programa y programa de computadora.
Un *programa* puede ser definido como "una secuencia precisa y ordenada de instrucciones y grupos de instrucciones, las cuales, en su total, definen, describen, o caracterizan la realización de alguna tarea".
Un *programa de computadora* es simplemente un programa el cual, posiblemente a través de una transformación, indica a la computadora como realizar una tarea.
En el nivel más elemental, observamos que un programa de computadora está compuesto de sentencias o instrucciones. Estas instrucciones están ordenadas en una secuencia. Podemos por lo tanto identificar a las instrucciones como componentes y a la secuencia como un relación. Esta es la visión clásica o "algoritmica" de un programa.
De esta visión tenemos como consecuencia, que el esfuerzo en el desarrollo de un programa se enfatiza en encontrar un método de solución y su transcripción sentencia a sentencia.
Sin embargo, a los efectos de nuestro estudio nos interesa otra visión sistémica de los programas de computadora, donde los componentes son no las sentencias elementales sino los módulos del programa, y las relaciones entre ellos son la estructura jerárquica (estructura del programa) en la que se encuentran organizados, en vez de su secuencia de activación.

3.2 Definición de Módulo de Programa

Previo a la definición de módulo de programa haremos algunas observaciones. Supongamos un conjunto de sentencias como las que se representan a continuación:

```
          _____
          _____
A1:       BEGIN A
          _____
          _____
B:        _____
          _____
          _____
A2:       END A
C:        _____
          _____
```

Diremos que A1 y A2 son los límites del conjunto o agregado de sentencias llamado A. La sentencia B se encuentra dentro de A, y C se encuentra fuera de A.
Las sentencias se encuentran en el orden en que ingresaran a un compilador. Este orden es conocido como *orden lexicográfico* de un programa. Para nuestro estudio el término lexicográfico siempre significará "como está escrito" o el orden en que aparecen las sentencias de un programa en el listado de un compilador. Volviendo al ejemplo, diremos que la sentencia C está lexicográficamente después que la A2.

Es importante distinguir que el orden lexicográfico casi siempre no se corresponde con el orden de ejecución de las sentencias.

Uno de los propósitos de los elementos de límite (A1 y A2 en el ejemplo) es el de controlar el alcance en el que identificadores son definidos y asociados a objetos (variables).

Estamos ahora en condiciones de definir el término módulo de programa o simplemente módulo: *Un módulo es una secuencia lexicográficamente contigua de sentencias, encerrada entre elementos de frontera, y que poseen un identificador del conjunto de dichas sentencias.*
Dicho de otra manera, un módulo es un grupo de sentencias contiguas que poseen un identificador simple por el cual son referenciadas.
Esta definición es general y dentro de la misma podemos encontrar implementaciones particulares de lenguajes específicos como ser: "párrafos", "secciones", y "subprogramas" de COBOL, "funciones" de C, "procedimientos" de Pascal, etc.
Un lenguaje de programación incluye un determinado tipo de módulo, solo si implementa construcciones lingüísticas específicas que realizan las características de definición y activación de dichos módulos.

3.3 Interface de Módulo

Una interface define un límite del módulo, a través del se lo activa y por el cual fluyen datos y control.
La interface puede considerarse como residente en el módulo referenciado. Puede pensarse como un enchufe (socket) donde la conexión del elemento referenciante se inserta.
Toda interface en un módulo representa cosas que deben ser conocidas, comprendidas, y apropiadamente conectadas por los otros módulos del sistema.
Se busca minimizar la complejidad del sistema/módulo, en parte, minimizando el número y complejidad de las interfaces por módulo.
Todo módulo además debe tener al menos una interface para ser definido y vinculado al resto del sistema. Es posible así mismo que un módulo tenga más de una interface.
Una interface puede cumplir las siguientes cuatro funciones:

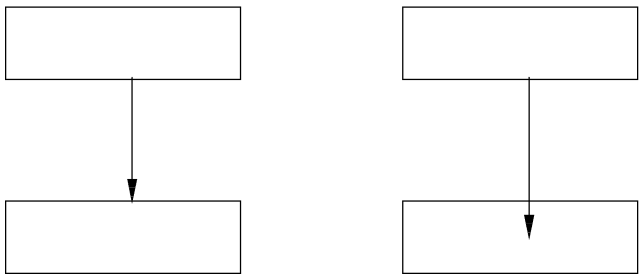
- transmitir datos a un módulo como parámetros de entrada
- recibir datos desde un módulo como resultados de salida
- ser un nombre por el cual ser recibe el control
- ser un nombre por el cual ser transmite el control

Un módulo puede ser identificado y activado por medio de una interfaz de identidad simple. También podemos pasar datos al módulo por medio de la misma interfaz sin agregar otras, haciendo a la interfaz de entrada capaz de aceptar datos. Esto requiere que los elementos de datos sean pasados dinámicamente como argumentos (parámetros) como parte de la secuencia de activación que da el control a un módulo.
Se necesita también que la interface de un módulo sirva para transferir el retorno del control al módulo llamador. Esto puede realizarse haciendo que la transferencia de control desde el llamador sea una transferencia *condicional.* Debe implementarse además un mecanismo para transmitir datos de retorno desde el módulo llamado hacia

el llamador. Puede asociarse un valor a una activación particular del modulo llamado, la cual pueda ser usada contextualmente en el llamador. Tal es el caso de las funciones lógicas. Alternativamente pueden transmitirse parámetros para definir ubicaciones donde el módulo llamado retorna valores al llamador.

3.4 Conexiones Normales y Patológicas

Diremos que entre dos módulos existe una *conexión normal* cuando la conexión se produce al nivel del identificador del módulo invocado, es decir a través de alguna interfaz definida formalmente.
En oposición si la conexión intermodular se realiza a un identificador de un elemento interno del módulo invocado sin pasar por alguna interface definida, diremos que es una *conexión patológica*.



Conexión Normal Conexión Patológica

3.5 Diagramas de Flujo de Datos

Los diagramas de flujo de datos que usaremos en la etapa de diseño son similares a los utilizados para la etapa del análisis.
Las transformaciones son representadas por burbujas (círculos) y los flujos de datos se representan con flechas. Cada flujo se etiqueta con su contenido.
Si dos flujos dibujados adyacentemente son ambos necesarios para realizar una determinada transformación (a la cual arriban), dibujaremos entre ambos un asterisco ("*"). Este símbolo al igual que en otras disciplinas matemáticas representa el operador "Y" o de *conjunción.* De igual manera, si solo uno de los flujos es necesario, utilizaremos el símbolo ⊕, que representa al operador "O" o de *disyunción.*

La cantidad de detalle mostrado en el diagrama de flujo de datos variará de problema en problema y de diseñador en diseñador.
Los diagramas de flujo de datos serán de gran utilidad en el estudio del concepto de *cohesión*, y su utilidad principal es el de herramienta gráfica en la estrategia de diseño estructural llamada *análisis de transformación.*

3.6 Estructura y Procedimiento

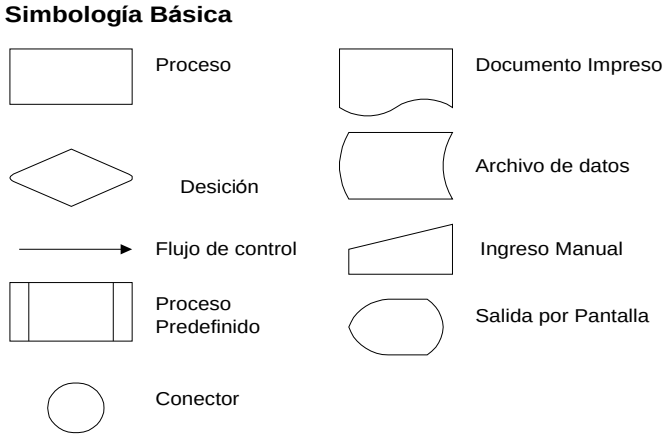
Tanto neófitos como veteranos, muchas veces encuentran difícil de comprender la diferencia entre procedimiento y estructura de programas. Peor aún son las fallas en la diferenciación entre codificación y diseño estructural.

La estructura nos da una relación jerárquica de control existente entre los módulos que conforman un programa.
El diseño estructurado se concentra principalmente en el correcto diseño de dicha estructura y de la correcta formación de los módulos, no asi del aspecto procedimental o algorítmico.

3.7 Diagramas de Flujo y Diagramas de Estructura

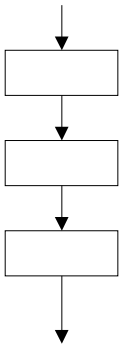
Normalmente los procedimientos se representan con *diagramas de flujo* (no confundir con diagramas de flujo de datos) los cuales modelan la secuencia de operaciones que realiza el programa a través del tiempo.
Un *diagrama de estructura* en cambio no modela la secuencia de ejecución sino la *jerarquía de control* existente entre los módulos que conforman el programa, independientemente del factor tiempo. Existe un módulo raíz de máximo nivel, del cual dependen los demás, en una estructura arborescente.

3.8 Notación de los Diagramas de Flujo de control

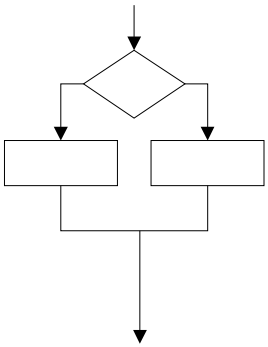


Construcciones Estructuradas

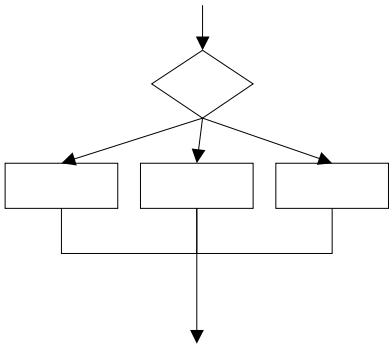
Secuencia



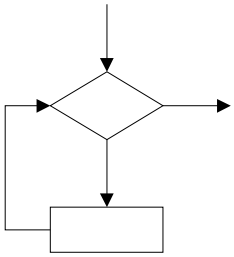
Desición



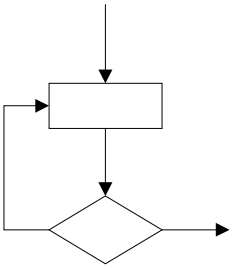
Selección



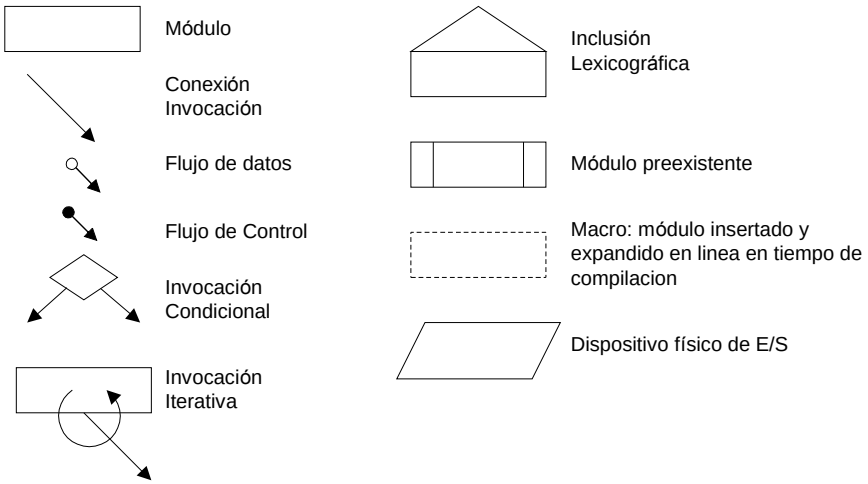
Iteración Superior



Iteración Inferior



3.9 Notación de los Diagramas de Estructura



Ejemplo Comparativo entre Diagramas de Procesamiento y de Estructura

Diagrama de Flujo:

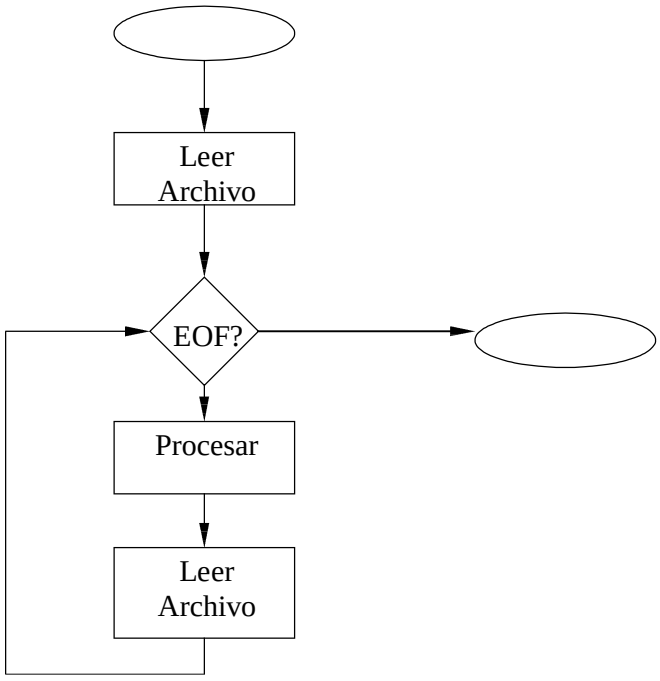
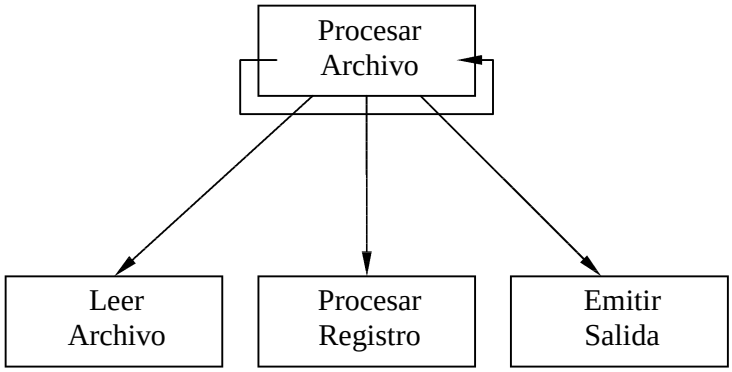


Diagrama de Estructura:



PRINCIPIOS FUNDAMENTALES

Unidad 4: Acoplamiento

4.0 Introducción

Muchos aspectos de la modularización pueden ser comprendidos solo si se examinan módulos en relación con otros. En principio veremos el concepto de *independencia*: diremos que dos módulos son totalmente independientes si ambos pueden funcionar completamente sin la presencia del otro. Esto implica que no existen interconexiones entre los módulos, y que se tiene un valor cero en la escala de "dependencia".
En general veremos que a mayor número de interconexiones entre dos módulos, se tiene una menor independencia.
El concepto de independencia funcional es una derivación directa del de modularidad y de los conceptos de abstracción y ocultamiento de la información.
La cuestión aquí es: cuanto debe conocerse acerca de un módulo para poder comprender otro módulo?. Cuanto más debamos conocer acerca del módulo B para poder comprender el módulo A, menos independientes serán A de B.
La simple cantidad de conexiones entre módulos, no es una medida completa de la independencia funcional. La dependencia funcional se mide con dos criterios cualitativos: *acoplamiento* y *cohesión.* Estudiaremos en principio el primero de ellos.
Módulos altamente "acoplados" estarán unidos por fuertes interconexiones, módulos débilmente acoplados tendrán pocas y débiles interconexiones, en tanto que los módulos "desacoplados" no tendrán interconexiones entre ellos y serán independientes.

El *acoplamiento* es un concepto abstracto que nos indica el grado de interdependencia entre módulos.

En la práctica podemos materializarlo como la probabilidad de que en la codificación, depuración, o modificación de un determinado módulo, el programador necesite tomar conocimiento acerca de partes de otro módulo. Si dos módulos están fuertemente acoplados, existe una alta probabilidad de que el programador necesite conocer uno de ellos en orden de intentar realizar modificaciones al otro.
Claramente, el costo total del sistema se verá fuertemente influenciado por el grado de acoplamiento entre los módulos.

4.1 Factores que influencian el Acoplamiento

Los cuatro factores principales que influyen en el acoplamiento entre módulos son:

- *Tipo de conexión entre módulos:* los sistemas normalmente conectados, tienen menor acoplamiento que aquellos que tienen conexiones patológicas.

- *Complejidad de la interface:* está determinada por la cantidad, accesibilidad, y estructura de la información que define la interface.

- *Tipo de flujo de información en la conexión:* los sistemas con acoplamiento de datos tienen menor acoplamiento que los sistemas con acoplamiento de control, y estos a su vez menos que los que tienen acoplamiento híbrido.

- *Momento en que se produce el ligado de la Conexión:* Conexiones ligadas a referentes fijos en tiempo de ejecución, resultan con menor acoplamiento que cuando el ligado tiene lugar en tiempo de carga, el cual tiene a su vez menor acoplamiento que cuando el ligado se realiza en tiempo de ligado (link-edición), el cual tiene menos acoplamiento que el que se realiza en tiempo de compilación, todos los que a su vez tiene menos acoplamiento que cuando el ligado se realiza en tiempo de codificación.

4.1.1 Tipos de conexiones entre módulos

Una conexión en un programa, es una referencia de un elemento, por nombre, dirección, o identificador de otro elemento.
Una conexión intermodular ocurre cuando el elemento referenciado está en un módulo diferente al del elemento referenciante.
Las referencias intermodulares deben realizarse a través de sus interfaces.

Si todas las conexiones de un sistema se restringen a implementarse por medio de interfaces completamente parametrizadas (con respecto a sus entradas y salidas), y cada módulo implementa una y solo una interfaz, diremos que el sistema está *mínimamente conectado*.

Diremos que un sistema está *normalmente conectado* cuando cumple con las condiciones de mínimamente conectado, excepto por alguna de las siguientes consideraciones:
- existe más de un punto de entrada para un mismo módulo (más de una interfaz)
- el módulo activador o llamador puede especificar como parte del proceso de activación un punto de retorno que no sea la próxima sentencia en el orden de ejecución.
- el control es transferido a un punto de entrada de un módulo por algún mecanismo distinto a una llamada explícita (ej. perform thru del Cobol).

El uso de múltiples puntos de entrada (múltiples interfaces) garantiza que existirán más que el número mínimo de interconexiones para el sistema. La presencia de múltiples puntos de entrada a un mismo módulo, puede ser un indicativo de que el módulo está llevando a cabo más de una función específica. Además, puede suceder que el programador superponga parcialmente el código de las funciones comprendidas dentro del mismo módulo, quedando dichas funciones *acopladas por contenido.*
De manera similar, *los puntos de retorno alternativo* son frecuentemente útiles dentro del espíritu de los sistemas normalmente conectados. Esto se da cuando un módulo continuará su ejecución en un punto que depende del valor resultante de una decisión realizada por un módulo subordinado invocado previamente. En un caso de mínima conexión, el módulo subordinado retornará el valor como un parámetro, el cual deberá ser testeado nuevamente en el módulo superior. Sin embargo, el módulo superior puede indicar por algún medio directamente el punto donde debe continuarse la ejecución del

programa, (un valor relativo + o - direcciones a partir de la instrucción llamadora, o un parámetro con una dirección explícita).

Si un sistema no está mínima o normalmente conectados, entonces algunos de sus módulos presentarán conexiones patológicas. Esto significa que al menos un módulo tendrá referencias explícitas a identificadores definidos dentro de los límites de otro módulo.

4.1.2 Complejidad de la interface

La segunda dimensión del acoplamiento es la *complejidad de la interface intermodular*. Cuanto más compleja es una conexión, mayor acoplamiento se tiene.
Tal lo visto en el capítulo dos, cuando se analizó qué es lo complejo para las personas, se determinó que existen tres factores que influyen en dicha complejidad:
- cantidad de información
- accesibilidad de la información
- estructura de la información

Analizaremos como influyen cada uno de estos factores en la especificación de las interfaces intermodulares.

Cantidad
Por *cantidad* de información, entenderemos el número de bits de datos, en el sentido que le asigna la teoría de la información este término, que el programador debe manejar para comprender la interface.
En términos simples, esto se relaciona con el número de argumentos o parámetros que son pasados en la llamada al módulo. Por ejemplo, una llamada a una subrutina que involucra 100 parámetros, será más compleja que una que involucra solo 3.
Cuando un programador ve una referencia a un módulo en el medio de otro, el *debe* conocer como se resolverá la referencia, y que tipo de transformación se transmitirá.

Consideremos el siguiente ejemplo: una llamada al procedimiento SQRT
Si la llamada es: SQRT(X)
el programador inferirá que X funciona como parámetro de entrada y como valor de retorno del procedimiento. Y es muy probable que esto sea así.

Ahora bien, si la llamada es: SQRT(X,Y)
el programador inferirá que X funciona como parámetro de entrada y que el resultado es retornado en Y. Es muy probable que esto sea así, aunque podría ser al revés.

Ahora supongamos que tenemos: SQRT(X,Y,Z)
el programador podrá inferir que X funciona como parámetro de entrada, que el resultado es retornado del procedimiento, y que en Z se retorna algún código de error. Esto podría ser cierto, pero es alta la probabilidad de que el orden de los parámetros sea diferente.

Vemos que a medida que crece la cantidad de parámetros, la posibilidad de error es mayor. Puede argumentarse que esto se soluciona con una adecuada documentación,

pero la realidad demuestra que en la mayoría de los casos los programas no están bien documentados.

Notaremos además que un módulo con demasiados parámetros, posiblemente esté realizando más de una función específica, y por lo tanto podría descomponerse en dos módulos las sencillos y funcionales con una menor cantidad de argumentos.

Accesibilidad

Quizá más importante que la cantidad de información es su *accesibilidad*. Cierta información acerca del uso de la interface debe ser comprendida por el programador para escribir o interpretar el código correctamente.
Consideraremos los siguientes puntos en esto:

- La interface es menos compleja si la información puede ser accedida (por el programador, no por la computadora) *directamente*; es más compleja si la información referencia *indirectamente* otros elementos de datos.

- La interface es menos compleja si la información es presentada *localmente* dentro de la misma sentencia de llamada. La interface es más compleja si la información necesaria es *remota* a la sentencia.

- La interface es menos compleja si la información es presentada en forma *standard* que si se presenta de forma *imprevista*.

- La interface es menos compleja si su naturaleza es *obvia* es menos compleja que si su naturaleza es *obscura*.

Observaremos el siguiente ejemplo: supongamos que tenemos la función DIST que calcula la distancia existente entre dos puntos. La fórmula matemática para realizar dicho cálculo es:

$$DIST = SQRT ((y1 - y0)^2 + (x1 - x0)^2)$$

Consideraremos las siguientes interfaces:

Opción 1. CALL DIST (X0, Y0, X1, Y1, DISTANCIA)
Opción 2. CALL DIST (ORIGEN, FIN, DISTANCIA)
Opción 3. CALL DIST (XCOORDS, YCOORDS, DISTANCIA)
Opción 4. CALL DIST (LINEA, DISTANCIA)
Opción 5. CALL DIST (LINETABLA)
Opción 6. CALL DIST

Trataremos de determinar cual de las interfaces es la menos compleja.
A primera vista podemos pensar que la opción 1 es la más compleja ya que involucra el mayor número de parámetros. Sin embargo la opción 1 presenta los parámetros en forma *directa.*

En contraste, la opción 2 presenta la información de manera ***indirecta.*** En orden de comprender la interface, deberemos ir a otra parte del programa y verificar que ORIGEN se define en términos de subelementos X0 e Y0, FIN como X1 e Y1.

La opción 3, además de presentar la información en forma *indirecta* además la presenta en forma *no estándar* lo cual complica más la interface.

La opción 4 presenta la misma desventaja que 2 y 3, presentando los valores en forma remota.

La opción 5 es aún más compleja. El identificador LINETABLE es *obscuro* .

La opción 6 a diferencia de las anteriores no representa parámetros localmente sino en forma *remota*.

Lamentablemente, algunos lenguajes como COBOL no permiten la llamada a módulos con parámetros dentro de un mismo programa. Además, existe cierta aversión a utilizar llamadas parametrizadas por algunos personas fundamentadas principalmente en:

- parametrizar una interface requiere más trabajo
- el proceso de parametrización mismo puede introducir errores
- en general la velocidad del programa es menor que cuando se usan variables globales

Estructura

Finalmente observaremos que la estructura de la información puede ser un punto clave en la complejidad.
La primera observación es que la información es menos compleja si se presenta en forma *lineal* y más compleja si se presenta en forma *anidada.*
La segunda observación es que la información se menos compleja se presenta en modo *afirmativo* o *positivo*, y es más compleja si se presenta en modo *negativo*.

Ambos conceptos tienen aplicación primaria en la escritura de código de programas. Por ejemplo, ciertas construcciones de sentencias IF anidadas son más complejas de entender que un secuencia de varias sentencias IF simples.
Similarmente, las expresiones lógicas que involucran operadores de negación (NOT) son más difíciles de comprender que aquellas que no lo presentan.

Estas filosofías de pensamiento linear y positivo también son importantes en las referencias intermodulares. Supongamos la siguiente instrucción:

DISTANCIA = SQRT (sum(square (dif (Y1,Y0), square (dif (X1, X0))))

Normalmente esta expresión para el común de los programadores resultará complicada de leer. Si por el contrario descomponemos la expresión en otras menores tendremos una mayor cantidad de elementos lineares, y una reducción en el anidamiento. La secuencia de expresiones resultantes resulta más sencilla de leer:

A = dif (Y1, Y0)
B = dif (X1, X0)
A2 = square (A)
B2 = square (B)
DISTANCIA = SQRT (sum (A2, B2))

4.1.3 Flujo de Información

Otro aspecto importante del acoplamiento tiene que ver con el *tipo* de información que se transmite entre el módulo superior y subordinado. Distinguiremos tres tipos de flujo de información:

- *datos*
- *control*
- *híbrido*

Los datos son información sobre la cual una pieza de programa opera, manipula, o modifica.
La información de control (aún cuando está representada por variables de dato) es aquella que gobierna como se realizarán las operaciones o manipulaciones sobre los datos.

Diremos que una conexión presenta *acoplamiento por datos* si la salida de datos del módulo superior es usada como entrada de datos del subordinado. Este tipo de acoplamiento también es conocido como de entrada-salida.

Diremos que una conexión presenta *acoplamiento de control* si el módulo superior comunica al subordinado información que controlará la ejecución del mismo. Esta información puede pasarse como datos utilizados como señales o "banderas" (flags) o bien como direcciones de memoria para instrucciones de salto condicional (branch-adress). Estos son elementos de control "disfrazados" como datos.

El acoplamiento de datos es mínimo, y ningún sistema puede funcionar sin él.
La comunicación de datos es *necesaria* para el funcionamiento del sistema, sin embargo, la comunicación de control es una característica no deseable y *prescindible*, que sin embargo aparece muy frecuentemente en los programas.
Se puede minimizar el acoplamiento si solo se transmiten datos a través de las interfaces del sistema.

El acoplamiento de control abarca todas las formas de conexión que comuniquen elementos de control. Esto no solo involucra transferencia de control (direcciones o banderas), si no que puede involucrar el pasaje de datos que cambia, regula, o sincroniza la ejecución de otro módulo.
Esta forma de acoplamiento de control indirecto o secundario se conoce como *coordinación.* La coordinación involucra a un módulo en el contexto procedural de otro. Esto puede comprenderse con el siguiente ejemplo: supongamos que el módulo A llama al módulo B suministrándole elementos de datos discretos. La función del módulo B es la de agrupar estos elemento de datos en un ítem compuesto y retornárselo al módulo A

(superior). El módulo B enviará al módulo A, señales o banderas indicando que necesita que se le suministre otro ítem elemental, o para indicarle que le está devolviendo el ítem compuesto. Estas banderas serán utilizadas dentro del módulo A para coordinar su funcionamiento y suministrar a B lo requerido.

Cuando un módulo modifica el contenido procedural de otro módulo, decimos que existe *acoplamiento híbrido*. El acoplamiento híbrido es una modificación de sentencias intermodular. En este caso, para el módulo destino o modificado, el acoplamiento es visto como de control en tanto que para el módulo llamador o modificador es considerado como de datos.
El grado de interdependencia entre dos módulos vinculados con acoplamiento híbrido es muy fuerte. Afortunadamente es una práctica en decadencia y reservada casi con exclusividad a los programadores en assembler.

4.1.4 Tiempo de ligado de conexiones intermodulares

"Ligado" o "Binding" es un término comúnmente usado en el campo del procesamiento de datos para referirse a un proceso que resuelve o fija los valores de identificadores dentro de un sistema.
El ligado de variables a valores, o más genéricamente, de identificadores a referentes específicos, puede tener lugar en diferentes estadios o períodos en la evolución del sistema. La historia de tiempo de un sistema puede pensarse como una línea extendiéndose desde el momento de la escritura del código fuente hasta el momento de su ejecución. Dicha línea puede subdividirse en diferentes niveles de refinamiento según distintas combinaciones de computador/lenguaje/compilador/sistema operativo.
De esta forma, el ligado puede tener lugar cuando el programador escribe una sentencia en el editor de código fuente, cuando un módulo es compilado o ensamblado, cuando el código objeto (compilado o ensamblado) es procesado por el "link-editor" o el "link-loader" (generalmente este proceso es el conocido como ligado en la mayoría de los sistemas), cuando el código "imagen-de-memoria" es cargado en la memoria principal, y finalmente cuando el sistema es ejecutado.
La importancia del tiempo de ligado radica en que *cuando los valor de variables dentro de una pieza de código son fijados más tarde, el sistema es más fácilmente modificable y adaptable al cambio de requerimientos.*

Veamos un ejemplo: supongamos que se nos encomienda la escritura de una serie de programas listadores siendo la impresora a utilizar en principio una del tipo matricial de 80 columnas que funciona con papel continuo de 12" de largo de página.

Alternativas:
1) Escribimos el literal "72" en todas las rutinas de impresión de todos los programas. (ligado en tiempo de escritura)
2) Reemplazamos el literal por la constante manifiesta LONG_PAG a la que asignamos el valor "72" en todos los programas (ligado en tiempo de compilación)
3) Ponemos la constante LONG_PAG en un archivo de inclusión externo a los programas (ligado en tiempo de compilación)

4) Nuestro lenguaje no permite la declaración de constantes por lo cual definimos una variable global LONG_PAG a la que le asignamos el valor de inicialización “72” (ligado en tiempo de link-edición)
5) Definimos un archivo de parámetros del sistema con un campo LONG_PAG al cual se le asigna el valor “72”. Este valor es leído junto con otros parámetros cuando el sistema se inicia. (ligado en tiempo de ejecución)
6) Definimos en el archivo de parámetros un registro para cada terminal del sistema y personalizamos el valor del campo LONG_PAG según la impresora que tenga vinculada cada terminal. De esta forma las terminales que tienen impresoras de 12” imprimen 72 líneas por página, y las que tienen una impresora de inyección de tinta que usan papel oficio, imprimen 80. (ligado en tiempo de ejecución)

Examinaremos ahora la relación existente entre el tiempo de ligado y las conexiones intermodulares, y como el mismo afecta el grado de acoplamiento entre módulos.
Nuevamente, una referencia intermodular fijada a un referente u objeto específico en tiempo de definición, tendrá un acoplamiento mayor a una referencia fijada en tiempo de traslación o posterior aún.
La posibilidad de compilación independiente de un módulo de otros facilitará el mantenimiento y modificación del sistema, que si debiera compilarse todos los módulos juntos. Igualmente, si la link-edición de los módulos es diferida hasta el instante previo a su ejecución, la implementación de cambios se verá simplificada.

Existe un caso particular de acoplamiento de módulos derivado de la *estructura lexicográfica* del programa. Hablamos en este caso de *acoplamiento por contenido.*
Dos formas de acoplamiento por contenido pueden distinguirse:

- *Inclusión lexicográfica:* se da cuando un módulo está incluido lexicográficamente en otro, y es una forma menor de acoplamiento. Los módulos por lo general no pueden ejecutarse separadamente. Este es el caso en el que el módulo subordinado es activado *en línea* dentro del contexto del módulo superior.
- *Solapamiento parcial:* es un caso extremo de acoplamiento por contenido. Parte del código de un módulo está en intersección con el otro. Afortunadamente la mayoría de los lenguajes modernos de alto nivel no permiten este tipo de estructuras.

```
A     BEGIN A
      ______
B     BEGIN B
      ______
      ______
      END B
      ______
      END A
```

Inclusión Lexicográfica

```
A     BEGIN A
      ______
      ______
B     BEGIN B
      ______
      END A
      ______
      END B
```

Solapamiento parcial

En términos de uso, mantenimiento, y modificación, las consecuencias del acoplamiento por contenido son peores que las del acoplamiento de control. El acoplamiento por contenido hace que los módulos no puedan funcionar uno sin el otro. No ocurre lo mismo en el acoplamiento de control, en el cual un módulo, aunque reciba información de control, puede ser invocado desde diferentes puntos del sistema.

4.2 Acoplamiento de Entorno Común (common-environment coupling)

Siempre que dos o más módulos interactúan con un entorno de datos común, se dice que dichos módulos están en *acoplamiento por entorno común.*
Ejemplos de entorno común pueden ser áreas de datos globales como la DATA división del Cobol, un archivo en disco.
El acoplamiento de entorno común es una forma de acoplamiento de segundo orden, distinto de los tratados anteriormente. La severidad del acoplamiento dependerá de la cantidad de módulos que acceden simultáneamente al entorno común. En el caso extremo de solo dos módulos donde uno utiliza como entrada los datos generados por el otro hablaremos de un acoplamiento de *entrada-salida*.
El punto es que el acoplamiento por entorno común no es necesariamente malo y deba ser evitado a toda costa. Por el contrario existen ciertas circunstancias en que es una opción válida.

4.3 Desacoplamiento

El concepto de acoplamiento invita a un concepto recíproco: *desacoplamiento.* Desacoplamiento es cualquier método sistemático o técnica para hacer más independientes a los módulos de un programa.
Cada tipo de acoplamiento generalmente sugiere un método de desacoplamiento. Por ejemplo, el acoplamiento causado por ligado, puede desacoplarse cambiando los parámetros apropiados tal lo visto en el ejemplo de el contador de líneas de los programas impresores.
El desacoplamiento desde el punto de vista funcional, rara vez puede realizarse, excepto en los comienzos de la fase del diseño.
Como regla general, una disciplina de diseño que favorezca el acoplamiento de entrada-salida y el acoplamiento de control por sobre el acoplamiento por contenido y el acoplamiento híbrido, y que busque limitar el alcance del acoplamiento por entorno común es el enfoque más efectivo.

Otras técnicas para reducir el acoplamiento son:

- Convertir las referencias implícitas en explícitas. Lo que puede verse con mayor facilidad es más fácil de comprender.
- Estandarización de las conexiones.
- Uso de “buffers” para los elementos comunicados en una conexión. Si un módulo puede ser diseñado desde el comienzo asumiendo que un buffer mediará cada corriente de comunicación, las cuestiones temporización, velocidad, frecuencia, etc., dentro de un módulo no afectarán el diseño de otros.

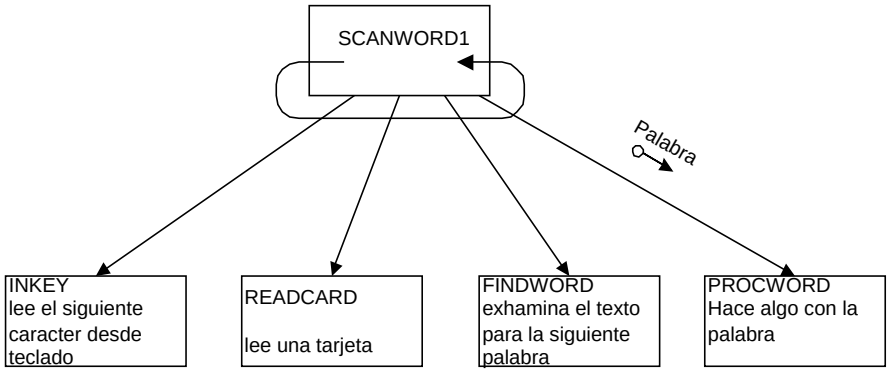
- Localización. Utilizado para reducir el acoplamiento por entorno común. Consiste en dividir el área común en regiones para que los módulos solo tengan acceso a aquellos datos que les son de su estricta incumbencia.

4.4 Una Aplicación

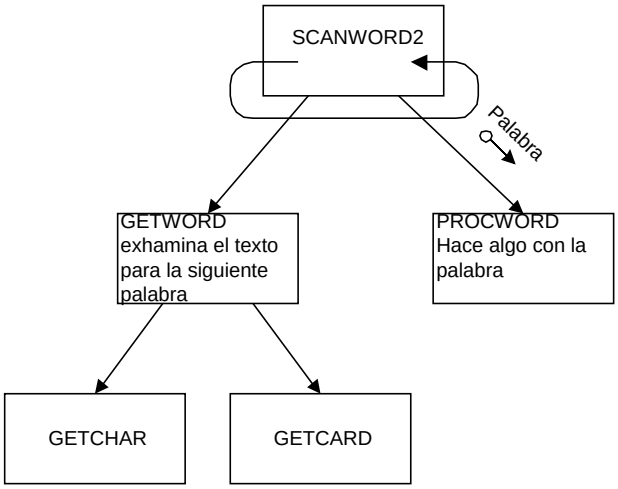
Para clarificar el concepto de acoplamiento veremos una aplicación. Debe escribirse un programa que realizará lo siguiente:

- el programa tendrá dos corrientes de entrada: una carácter a carácter desde teclado de una terminal, y la otra registro a registro desde una archivo en disco.
- se comienza leyendo los caracteres provenientes de tecla hasta que se recibe el carácter "RETURN", entonces se pasa a leer el archivo registro a registro hasta recibir un registro con "//" en su encabezado, lo cual indica que se vuelve a leer desde teclado.
- el paso anterior se realiza iterativamente, hasta que se recibe una señal de fin-de-transmisión desde la terminal (EOT). Entonces se continúa leyendo el archivo hasta el final (EOF).
- las corrientes de datos de ambas entradas se analizarán y separarán en *palabras* las que se pasarán al módulo existente ProcWord, el que realizará algo con ellas.

Se comisiona en primer lugar al programador Carlitos para que confeccione el programa, quién realiza el siguiente diagrama de estructura:



Cuando se presenta el problema a la programadora Nadine, ella realiza la siguiente solución:



Ambas estructuras presentan las siguientes características en común:

- Ambas son normalmente conectadas
- Cada una consiste de 5 módulos y 4 conexiones
- La lógica de encontrar palabras ha sido aislada en un módulo específico en ambos casos

Sin embargo analizaremos si ambas estructuras presentan el mismo grado de acoplamiento.
Para evaluar esto, necesitaremos mirar el tipo de información comunicada entre los módulos. Es importante notar que determinados flujos pueden comportarse como de datos y de control. Por ejemplo carácter de salida del módulo INKEY normalmente será un flujo de datos, pero en el caso especial de que el carácter será RETURN este funcionará como un flujo de control. En tales circunstancias es conveniente a efectos del estudio considerarlo como distintos flujos.

MODULO	**ENTRADAS**	**SALIDAS**
INKEY		char, EOT, Return
READCARD		Registro, EOF, //
FINDWORD	char, EOT, Return Registro, EOF, //, fuente	palabra, fin-palabras, deme-un-caracter, deme-un-registro, tome-una-palabra
PROCWORD	palabra	
GETCHAR		char, EOT, Return
GETCARD		Registro, EOF, //
GETWORD	char, EOT, Return Registro, EOF, //	palabra, fin-palabras
PROCWORD	palabra	

Como se aprecia en la tabla precedente podemos establecer las siguientes comparaciones: el diagrama de Carlitos tiene 13 flujos de control y 6 de datos, en tanto que el diagrama de Nadine tiene 9 flujos de control y 6 de datos. Obviamente el diagrama la estructura de Carlitos presenta un mayor grado de acoplamiento.
Por otro lado la interfaz del módulo FINDWORD de Carlitos será bastante más compleja que la de GETWORD de Nadine debido a la cantidad de parámetros que implica.

Unidad 5: Cohesión

5.0 Introducción: Relación Funcional

Hemos visto que la determinación de módulos en un sistema no es arbitraria. La manera en la cual dividimos físicamente un sistema en piezas (particularmente en relación con la estructura del problema) puede afectar significativamente la complejidad estructural del sistema resultante, así como el número total de referencias intermodulares.
Adaptar el diseño del sistema a la estructura del problema (o estructura de la aplicación, o dominio del problema) es una filosofía de diseño sumamente importante. A menudo encontramos que elementos de procesamiento del dominio de problema altamente relacionados, son trasladados en código altamente interconectado. Las estructuras que agrupan elementos del problema altamente interrelacionados, tienden a ser modularmente efectivas.
Imaginemos que tengamos una magnitud para medir el grado de relación funcional existente entre pares de módulos. En términos de tal medida, diremos que sistema más modularmente efectivo será aquel cuya suma de relación funcional entre pares de elementos que pertenezcan a diferentes módulos sea mínima. Entre otras cosas, esto tiende a minimizar el número de conexiones intermodulares requeridas y el acoplamiento intermodular.

Esta relación funcional intramodular se conoce como *cohesión.* La cohesión es la medida cualitativa de cuan estrechamente relacionados están los elementos internos de un módulo.

Otros términos utilizados frecuentemente son “fuerza modular”, “ligazón”, y “funcionalidad”.

En la práctica un elemento de procesamiento simple aislado, puede estar funcionalmente relacionado en diferentes grados a otros elementos. Como consecuencia, diferentes diseñadores, con diferentes “visiones” o interpretaciones de un mismo problema, pueden obtener diferentes estructuras modulares con diferentes niveles de cohesión y acoplamiento. A esto se suma el inconveniente de que muchas veces es difícil evaluar el grado de relación funcional de un elemento respecto de otro.

La cohesión modular puede verse como el cemento que amalgama juntos a los elementos de procesamiento dentro de un mismo módulo. Es el factor más crucial en el diseño estructurado, y el de mayor importancia en un diseño modular efectivo.
Este concepto representa la técnica principal que posee un diseñador para mantener su diseño lo más semánticamente próximo al problema real, o dominio de problema.

Claramente los conceptos de cohesión y acoplamiento están íntimamente relacionados. Un mayor grado de cohesión implica uno menor de acoplamiento. Maximizar el nivel de cohesión intramodular en todo el sistema resulta en una minimización del acoplamiento intermodular.
Matemáticamente el cálculo de la relación funcional intramodular (cohesión), involucra menos pares de elementos a los cuales debe aplicarse la medida, en comparación con el cálculo de la relación funcional intermodular (acoplamiento).

Ambas medidas son excelentes herramientas para el diseño modular efectivo, pero de las dos la más importante y extensiva es la cohesión.

Una cuestión importante a determinar es *como* reconocer la relación funcional.
El principio de cohesión puede ponerse en práctica con la introducción de la idea de un *principio asociativo*.
En la decisión de poner ciertos elementos de procesamiento en un mismo módulo, el diseñador, utiliza el principio de que ciertas ***propiedades*** o ***características*** relacionan a los elementos que las poseen. Esto es, el diseñador pondrá el objeto Z en el mismo módulo que X e Y, porque X, Y, y Z poseen un misma propiedad. De esta manera, el principio asociativo es *relacional*, y es usualmente verificable en tales términos (p.e. "es correcto poner Z junto a X e Y, porque tiene la misma propiedad que ellos") o en términos de miembro de un conjunto (p.e."es correcto poner Z junto a X e Y, pues todos pertenecen al mismo conjunto").
Debe tenerse en mente que la cohesión se aplica sobre todo el módulo, es decir sobre todos los pares de elementos. Así, si Z está relacionado a X e Y, pero no a A, B, y C, los cuales pertenecen al mismo módulo, la inclusión de Z en el módulo, redundará en baja cohesión del mismo.

Intencionalmente se ha usado el término "elemento de procesamiento" en esta discusión, en lugar de términos más comunes como instrucción o sentencia. Porqué:
Primero, un elemento de procesamiento puede ser algo que debe ser realizado en un módulo pero que aún no ha sido reducido a código. En orden de diseñar sistemas altamente modulares, debemos poder determinar la cohesión de módulos que todavía no existen.
Segundo, *elementos de procesamiento* incluyen *todas* las sentencias que aparecen en un módulo, no solo el procesamiento realizado por las instrucciones ejecutadas dentro de dicho módulo, si no también las que resultan de la invocación de subrutinas.
Por ejemplo, las sentencias individuales encontradas en el módulo B, el cual es invocado desde el módulo A, *NO* figuran dentro de la cohesión del módulo A. Sin embargo el procesamiento global (función) realizado por la llamada al módulo B, es claramente un elemento de procesamiento en el módulo llamador A, y por lo tanto participa en la cohesión del módulo A.

5.1 Niveles de Cohesión

Diferentes principios asociativos fueron desenvolviéndose a través de los años por medio de la experimentación, argumentos teóricos, y la experiencia práctica de muchos diseñadores.
Existen siete niveles de cohesión distinguibles por siete principios asociativos. Estos se listan a continuación en orden creciente del grado de cohesión, de menor a mayor relación funcional:

- Cohesión Casual (la peor)
- Cohesión Lógica (sigue a la peor)
- Cohesión Temporal (de moderada a pobre)
- Cohesión de Procedimiento (moderada)
- Cohesión de Comunicación (moderada a buena)

- Cohesión Secuencial
- Cohesión Funcional (la mejor)

Podemos visualizar el grado de cohesión como un espectro que va desde un máximo a un mínimo.

5.1.1 Cohesión Casual (la peor)

La ***cohesión casual*** ocurre cuando existe poca o ninguna relación entre los elementos de un módulo.
La cohesión casual establece un punto cero en la escala de cohesión.
Es muy difícil encontrar módulos puramente casuales. Puede aparecer como resultado de la modularización de un programa ya escrito, en el cual el programador encuentra un determinada secuencia de instrucciones que se repiten de forma aleatoria, y decide por lo tanto agruparlas en una rutina.
Otro factor que influenció muchas veces la confección de módulos casualmente cohesivos, fue la mala práctica de la programación estructurada, cuando los programadores mal entendían que modularizar consistía en cambiar las sentencias GOTO por llamadas a subrutinas
Finalmente diremos que si bien en la práctica es difícil encontrar módulos casualmente cohesivos en su totalidad, es común que tengan elementos casualmente cohesivos. Tal es el caso de operaciones de inicialización y terminación que son puestas juntas en un módulo superior.
Debemos notar que si bien la cohesión casual no es necesariamente perjudicial (de hecho es preferible un programa casualmente cohesivo a uno lineal), dificulta las modificaciones y mantenimiento del código.

5.1.2 Cohesión Lógica (sigue a la peor)

Los elementos de un módulo están ***lógicamente*** asociados si puede pensarse en ellos como pertenecientes a la misma clase lógica de funciones, es decir aquellas que pueden pensarse como juntas lógicamente.
Por ejemplo, se puede combinar en un módulo simple todos los elementos de procesamiento que caen en la clase de “entradas”, que abarca todas las operaciones de entrada.
Podemos tener un módulo que lea desde consola una tarjeta con parámetros de control, registros con transacciones erróneas de un archivo en cinta, registros con transacciones válidas de otro archivo en cinta, y los registros maestros anterior de un archivo en disco. Este módulo que podría llamarse “Lecturas”, y que agrupa todas las operaciones de entrada, es lógicamente cohesivo.
La cohesión lógica es más fuerte que la casual, debido a que representa un mínimo de asociación entre el problema y los elementos del módulo. Sin embargo podemos ver que un módulo lógicamente cohesivo no realiza una función específica, sino que abarca una serie de funciones.

5.1.3 Cohesión Temporal (de moderada a pobre)

Temporal cohesión significa que todos los elementos de procesamiento de una colección ocurren en el mismo período de tiempo durante la ejecución del sistema. Debido a que dicho procesamiento debe o puede realizarse en el mismo período de tiempo, los elementos asociados temporalmente pueden combinarse en un único módulo que los ejecute a la misma vez.
Existe una relación entre cohesión lógica y la temporal, sin embargo, la primera no implica una relación de tiempo entre los elementos de procesamiento. La cohesión temporal es más fuerte que la cohesión lógica, ya que implica un nivel de relación más: el factor tiempo. Si embargo la cohesión temporal aún es pobre en nivel de cohesión y acarrea inconvenientes en el mantenimiento y modificación del sistema.
Un ejemplo común de cohesión temporal son las rutinas de inicialización (start-up) comúnmente encontradas en la mayoría de los programas, donde se leen parámetros de control, se abren archivos, se inicializan variables contadores y acumuladores, etc.

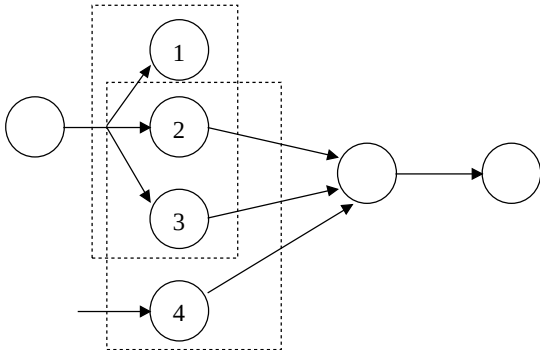
5.1.4 Cohesión de Procedimiento (moderada)

Elementos de procesamiento relacionados *proceduralmente* son elementos de una unidad procedural común. Estos se combinan en un módulo de cohesión procedural. Una unidad procedural común puede ser un proceso de iteración (loop) y de decisión, o una secuencia linear de pasos. En este último caso la cohesión es baja y es similar a la cohesión temporal, con la diferencia que la cohesión temporal no implica una determinada secuencia de ejecución de los pasos.
Al igual que en los casos anteriores, para decir que un módulo tiene *solo* cohesión procedural, los elementos de procesamiento deben ser elementos de alguna iteración, decisión, o secuencia, pero no deben estar vinculados con ningún principio asociativo de orden superior.
La cohesión procedural asocia elementos de procesamiento sobre la base de sus relaciones algorítmicas o procedurales.
Este nivel de cohesión comúnmente se tiene como resultado de derivar una estructura modular a partir de modelos de procedimiento como ser diagramas de flujo, o diagramas Nassi-Shneiderman.

5.1.5 Cohesión de Comunicación (moderada a buena)

Ninguno de los niveles de cohesión discutidos previamente están fuertemente vinculados a una estructura de problema en particular. *Cohesión de Comunicación* es el menor nivel en el cual encontramos una relación entre los elementos de procesamiento que es intrínsecamente *dependiente del problema*.

Decir que un conjunto de elementos de procesamiento están vinculados por comunicación significa que *todo los elementos operan sobre el mismo conjunto de datos* de entrada o de salida.



En el diagrama de la figura podemos observar que los elementos de procesamiento 1, 2, y 3, están asociados por comunicación sobre la corriente de datos de entrada, en tanto que 2, 3, y 4 se vinculan por los datos de salida.
Los diagramas de flujo de datos (DFD) son un medio objetivo para determinar si los elementos en un módulo están asociados por comunicación.

Las relaciones por comunicación presentan un grado de cohesión aceptable.
La cohesión por comunicación es común en aplicaciones comerciales. Ejemplos típicos pueden ser

- un módulo que imprima o grabe un archivo de transacciones
- un módulo que reciba datos de diferentes fuentes, y los transforme y ensamble en una línea de impresión.

5.1.6 Cohesión Secuencial

El siguiente nivel de cohesión en la escala es la asociación ***secuencial.*** En ella, los datos de salida (resultados) de un elemento de procesamiento sirven como datos de entrada al siguiente elemento de procesamiento.
En términos de un diagrama de flujo de datos de un problema, la cohesión secuencial combina una cadena linear de sucesivas transformaciones de datos.
Este es claramente un principio asociativo relacionado con el dominio del problema.

5.1.7 Cohesión Funcional (la mejor)

En el límite superior del espectro de relación funcional encontramos la ***cohesión funcional***. En un módulo completamente funcional, cada elemento de procesamiento, es parte integral de, y esencial para, la realización de una función simple.

En términos prácticos podemos decir que cohesión funcional es aquella que no es secuencial, por comunicación, por procedimiento, temporal, lógica, o casual.

Los ejemplos más claros y comprensibles provienen del campo de las matemáticas. Un módulo para realizar el cálculo de *raíz cuadrada* ciertamente será altamente cohesivo, y probablemente, completamente funcional. Es improbable que haya elementos superfluos más allá de los absolutamente esenciales para realizar la función matemática, y es improbable que elementos de procesamiento puedan ser agregados sin alterar el cálculo de alguna forma.
En contraste un módulo que calcule raíz cuadrada y coseno, es improbable que sea enteramente funcional (deben realizarse dos funciones ambiguas).
En adición a estos ejemplos matemáticos obvios, usualmente podemos reconocer módulos funcionales que son elementales en naturaleza. Un módulo llamado LEER-REGISTRO-MAESTRO, o TRATAR-TRANS-TIPO3, presumiblemente serán funcionalmente cohesivos, en cambio TRATAR-TODAS-TRANS presumiblemente realizará más de una función y será lógicamente cohesivo.

5.2 Criterios para establecer el grado de cohesión

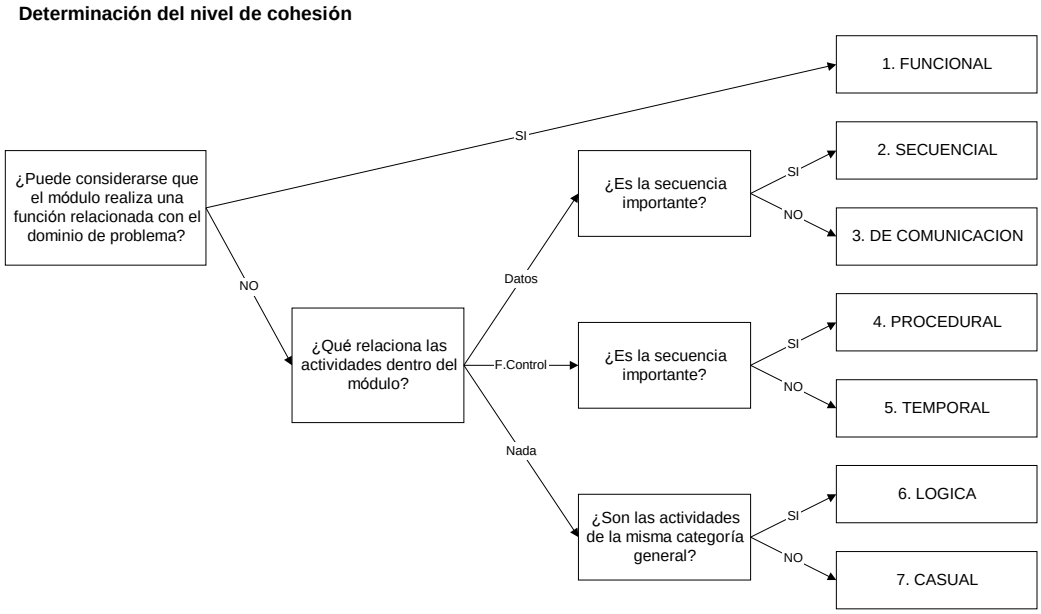
Una técnica útil para determinar si un módulo está acotado funcionalmente es escribir una frase que describa la función (propósito) del módulo y luego examinar dicha frase. Puede hacerse la siguiente prueba:

1) Si la frase resulta ser una sentencia compuesta, contiene una coma, o contiene más de un verbo, probablemente el módulo realiza más de una función; por tanto, probablemente tiene vinculación secuencial o de comunicación.
2) Si la frase contiene palabras relativas al tiempo, tales como "primero", "a continuación", "entonces", "después", "cuando", "al comienzo", etc., entonces probablemente el módulo tiene una vinculación secuencial o temporal.
3) Si el predicado de la frase no contiene un objeto específico sencillo a continuación del verbo, probablemente el módulo esté acotado lógicamente. Por ejemplo *editar todos los datos* tiene una vinculación lógica; *editar sentencia fuente* puede tener vinculación funcional.
4) Palabras tales como "inicializar", "limpiar", etc., implican vinculación temporal.

Los módulos acotados funcionalmente siempre se pueden describir en función de sus elementos usando una sentencia compuesta. Pero si no se puede evitar el lenguaje anterior, siendo aún una descripción completa de la función del módulo, entonces probablemente el módulo no esté acotado funcionalmente.

Es importante notar que no es necesario determinar el nivel preciso de cohesión. En su lugar, lo importante es intentar conseguir una cohesión alta y saber reconocer la cohesión baja, de forma que se pueda modificar el diseño del software para que disponga de una mayor independencia funcional.

Arbol de evaluación



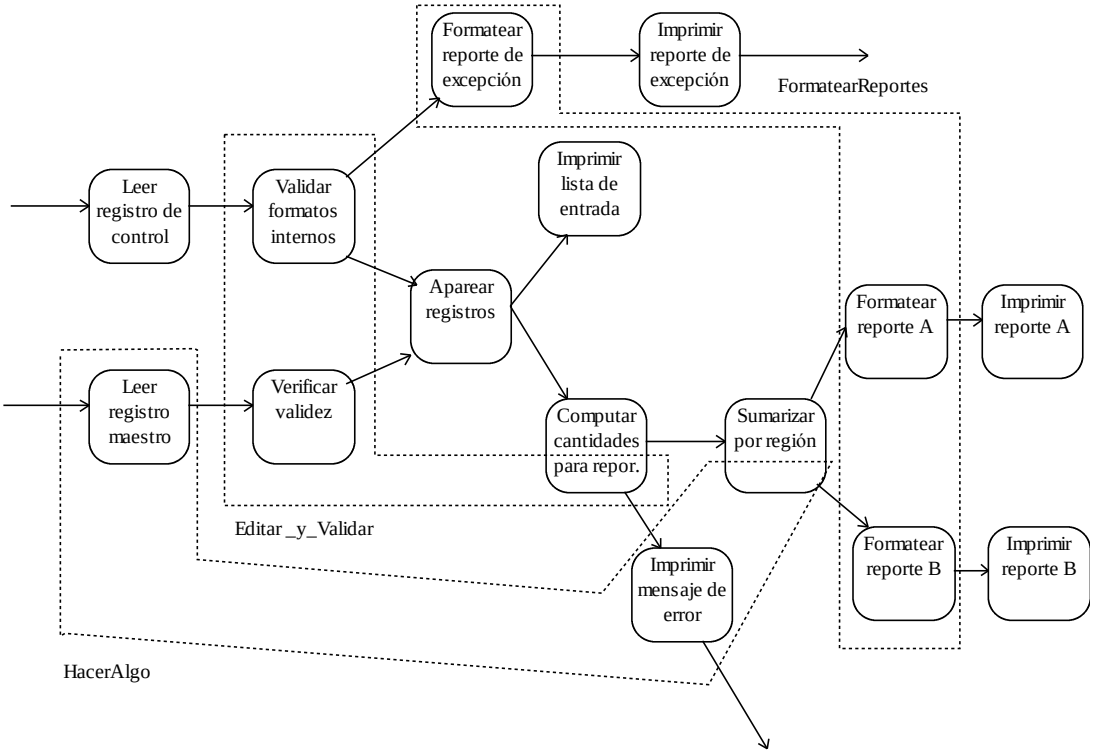
5.3 Comparación de Niveles de Cohesión

Utilizaremos el problema representado en la siguiente figura para ilustrar una variedad de particionamientos del mismo problema, correspondiente a diferentes niveles de cohesión.
Es fácil presentar ejemplos de cohesión casual y lógica particionando el diagrama de flujo de datos. El módulo "HacerAlgo" es un ejemplo de cohesión casual. Los módulos "FormatearReportes" y "Editar_y_Validar" son ejemplos de cohesión lógica.

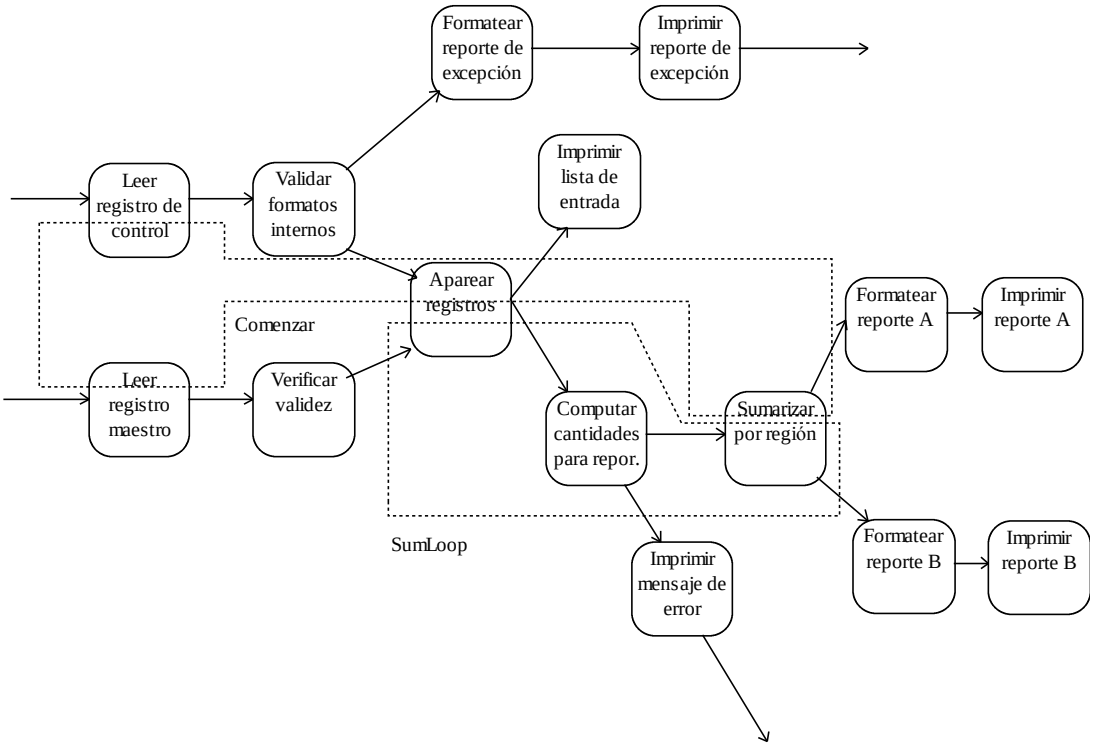
Debido a que el DFD es inherentemente no procedural, es un tanto difícil visualizar en él relaciones temporales y procedurales. Dos posibilidades serían los módulos "Comenzar" (temporal) y "SumLoop" (procedural).

Cohesión secuencial y de comunicación es fácilmente visible en un DFD. Los módulos "DoCombo" y "ObtenerMaestroVálido" son ejemplos de cohesión de comunicación y secuencial respectivamente.

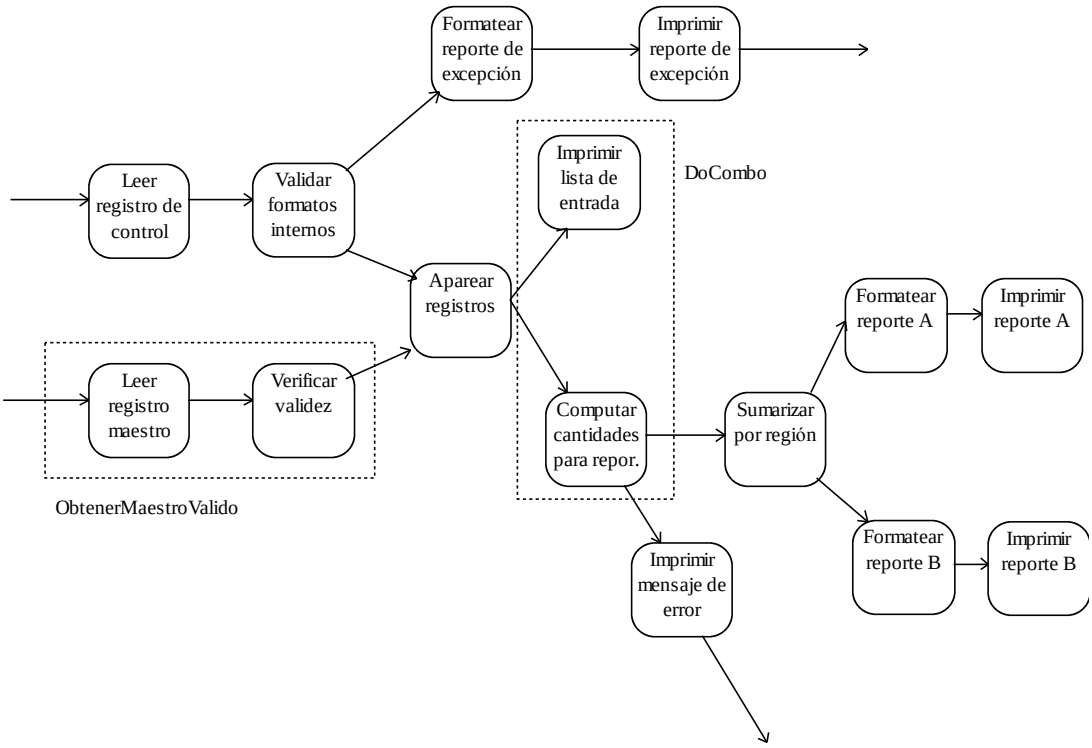
La identificación de cohesión funcional presenta una vez más dificultades. En un nivel superficial, la cohesión funcional sería equivalente a que cada transformación del DFD se corresponda con un módulo, pero un particular arreglo de estos en una jerarquía influencia la cohesión actual de módulos. Estos problemas pueden comprenderse mejor con los conceptos estratégicos introducidos en el capítulo dedicado a morfologías y metodologías.



HacerAlgo : cohesión casual
Editar_y_Validar: cohesión lógica
FormatearReportes: Cohesión lógica



Comenzar: cohesión temporal
SumLoop: cohesión procedural



ObtenerMaestroVálido: cohesión secuencial
DoCombo: cohesión de comunicación

5.4 Medición de Cohesión

Cualquier módulo, rara vez verifica un solo principio asociativo. Sus elementos pueden estar relacionados por una mezcla de los siete niveles de cohesión. Esto lleva a tener una escala continua en el grado de cohesividad más que una escala con siete puntos discretos.

Donde existe más de una relación entre un par de elementos de procesamiento, se aplica el máximo nivel que alcanzan. Por esto, si un módulo presenta cohesión lógica entre *todos* sus pares de elementos de procesamiento, y a su vez presenta cohesión de comunicación también entre *todos* dichos pares, entonces dicho módulo es considerado como de cohesión por comunicación.

Ahora, cuál sería la cohesión de dicho módulo si también contiene algún par de elementos completamente no relacionados? En teoría, debería tener algún tipo de promedio entre la cohesión de comunicación y la casual. Para propósitos de depuración, mantenimiento, y modificación, un módulo se comporta como si fuera "solo tan fuerte como sus vínculos más débiles".

El efecto sobre los costos de programación es próximo al menor nivel de cohesión aplicable dentro del módulo en vez del mayor nivel de cohesión.

Esto es: *la cohesión de un módulo es aproximada al nivel más alto de cohesión que es aplicable a todos los elementos de procesamiento dentro del módulo.*

Un módulo puede consistir de varias funciones *completas* relacionadas lógicamente. Esto es definitivamente más cohesivo que un módulo que liga lógicamente fragmentos de varias funciones.

La decisión de que nivel de cohesión es aplicable a un módulo dado requiere de cierto juicio humano. Algunos criterios establecidos son:
- la cohesión secuencial es más próxima al óptimo funcional que a su antecesor de comunicación.
- similarmente existe un salto mayor entre la cohesión lógica y la temporal que entre casual y lógica.

Podemos asignar la siguiente escala de valores para ayudar al diseñador en la calificación de niveles:

- 0: casual
- 1: lógica
- 3: temporal
- 5: procedural
- 7: de comunicación
- 9: secuencial
- 10: funcional

De cualquier modo, esta escala está basada en la experiencia de los autores, y no es una regla fija, si no una conclusión.

La obligación del diseñador es conocer los efectos producidos por la variación en la cohesión, especialmente en términos de modularidad, en orden de realizar soluciones de *compromiso* beneficiando un aspecto en contra de otro.

EL METODO

En esta sección se estudiarán métodos que harán uso de los conceptos estudiados en los capítulos precedentes, con el objeto de realizar el diseño de sistemas complejos.

Unidad 6: Morfología de Sistemas Simples

6.0 Introducción: Organización y Morfología

En el contexto del diseño de programas y de sistemas, utilizamos la palabra "organización" para describir la *forma* en que la estructura es utilizada para realizar una función deseada. Dicho de otra manera, la organización es la relación entre función y estructura.

El término "Morfología" se refiere a la *forma* de un sistema con respecto a su estructura. Por ejemplo, la profundidad de una estructura (el número de niveles de módulos subordinados) es una característica morfológica visible. El ancho de una estructura modular, es otra característica morfológica.

Nuestro propósito es doble. Primero, examinaremos organizaciones *comunes* de sistemas modulares, y morfologías de sistemas *comunes*. Segundo, realizaremos comparaciones entre organizaciones comunes y buenos sistemas, es decir aquellos con bajo acoplamiento y alta cohesión.

6.1 Organización de Sistemas Modulares

En que se basa el diseñador para decidir sobre una división particular de un programa o sistema en módulos? Cómo decide qué porciones del procesamiento total pertenecerán a un módulo determinado?

Encontraremos que un sistema modular usualmente está *centrado* alrededor de varios aspectos específicos de su función. Más allá de que el diseñador explícitamente reconozca una estructura modular en particular, usualmente podremos identificar un concepto o criterio de organización implícito.
En muchos casos, la estructura literalmente se centra alrededor de un módulo con un propósito o función muy distintivo. Entonces, podemos hablar de *diseño Centrado en la Transacción.* Tales sistemas se desarrollan en torno a módulos que realizan varias acciones asociadas con transacciones. Generalmente, existe un módulo (o pequeño grupo de módulos) que pasan todas las transacciones a sus módulos subordinados para procesar las transacciones.

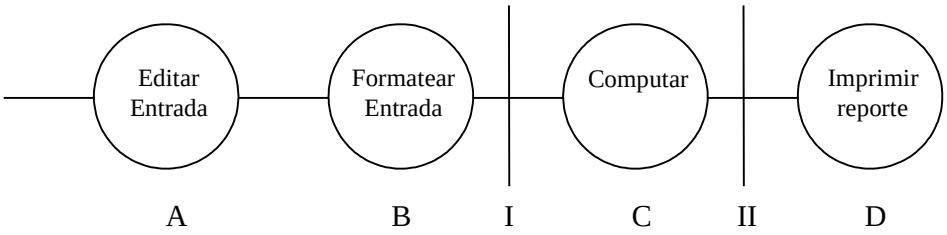
Algunos tipos de organizaciones modulares han sido reflejadas en **estrategias**: *procedimientos formales sistemáticos para desarrollar, desde la descripción de un problema, la estructura modular de sistemas del tipo deseado*.
Así existe un esquema conocido como *Análisis de Transacción.*

El diseño *centrado en el procedimiento* es derivado de representaciones procedurales (diagramas de flujo de control). Usualmente los sistemas centrados en el procesamiento alcanzan solo cohesión temporal o procedural.

El diseño *centrado en dispositivos*, el cual es común en porciones de sistemas operativos pero raro en otros casos, se enfoca en dispositivos físicos de entrada-salida y sus interfaces. Aunque en los niveles más bajos de un sistema se invocará algún módulo orientado a dispositivo, este enfoque usualmente no se pasa hacia los niveles superiores de la estructura.

Todo sistema puede pensarse involucrando una o más *transformaciones centrales*: principales funciones del sistema que "digieren" los datos provenientes de las corrientes de entrada y producen las mayores corrientes de salida del sistema. Debido a esto, podemos tener sistemas *Centrados en la Transformación.*
Una estrategia formal de diseño conocida como *Análisis de Transformación* se utiliza para derivar este tipo de sistemas.
En la práctica, el diseño centrado en la transformación no comienza con identificar las transformaciones como los módulos centrales del sistema. Es más sencillo identificar primero todo lo demás, y luego llamar al resto, transformaciones "esenciales" o "mayores" del sistema.

Consideremos el siguiente ejemplo:



Las funciones A y B, básicamente son operaciones que realizadas en secuencia, obtienen la principal corriente de datos del sistema. Hasta la primera línea vertical marcada como "I", los datos están ingresando al sistema. Luego de la línea II, los datos pueden pensarse como fluyendo fuera del sistema. Las partes remanentes del proceso no son ni entrada ni salida, por lo tanto C es la transformación central del sistema.

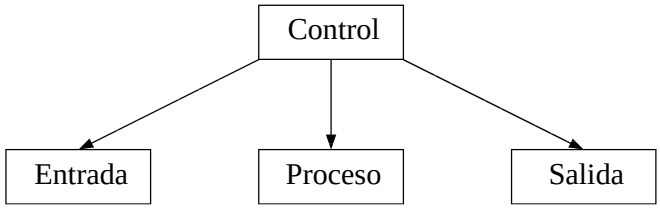
6.2 Modelos Específicos de Organización de Sistemas

Ocasionalmente, los diseñadores hacen uso de un modelo funcional o estructural específico como una guía para el diseño estructural. En un sentido, tal uso representa una preconcepción técnica acerca de como debe parecerse el sistema. Esto debería simplificar y generalizar la tarea del diseño, sin embargo muchas veces es perjudicial debido a la aplicación *literal* de los modelos.

Un modelo específico de organización de sistema es mostrado a continuación en dos variantes:

Versión IPO

Versión CIPO



Uno puede considerar la versión CIPO como la estructura literal de un sistema. En este caso, solo cuatro módulos serán implementados, a pesar del tamaño del problema. Nótese que el módulo "Entrada", implementado literalmente, probablemente alcance a ser solo lógicamente cohesivo.

En general diremos que, mientras que modelos estructurales específicos pueden desarrollarse, su simple y literal aplicación al diseño estructurado no es recomendable.

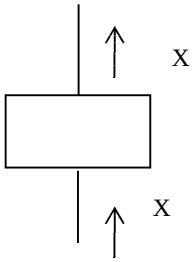
6.3 Factorización

La *factorización* implica la descomposición de una determinada tarea en una jerarquía de módulos partiendo de un módulo raíz o ejecutivo y llegando a los módulos atómicos que realiza las funciones *operativas.*
Análogamente a lo que sucede en las organizaciones humanas, un módulo ejecutivo de máximo nivel, no realiza ninguna de las tareas del sistema por si mismo, si no a través de sus subordinados. En el caso límite, un módulo ejecutivo solo contiene llamadas a otros módulos.
Diremos que el sistema estará *completamente factorizado* si todo el proceso (computación y manipulación de datos) es realizado en módulos atómicos de bajo nivel, y si todos los módulos no atómicos de nivel superior consisten solo de *control y coordinación.* En un sistema completamente factorizado, cada módulo no atómico es un ejecutivo con respecto a sus subordinados.

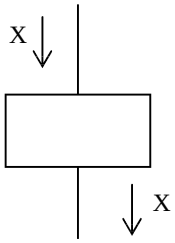
Basado en una regla de pensamiento de diseño y en estrategias sistemáticas, sistemas bien diseñados tienden a mostrar una distribución del proceso de decisión característico. A medida que se desciende en la estructura, la proporción de elementos de decisión decrece. El carácter de las decisiones también cambia. Los módulos de alto nivel normalmente tratan decisiones globales, en tanto que los de bajo nivel tratan aspectos específicos de las decisiones de nivel superior.

6.4 Flujo Aferente, Eferente, Transformación y Coordinación

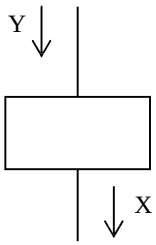
En el estudio de la estructura modular de un sistema, usualmente encontramos un pequeño número de *categorías* de módulos.
Notamos por ejemplo, que algunos módulos obtienen información de sus subordinados y luego la transmites hacia a arriba a sus superiores. Esto se conoce como *flujo aferente* de datos. Un módulo con este tipo de flujo se conoce como *módulo aferente*.
Normalmente este tipo de módulos están involucrados en procesos de *entrada*.



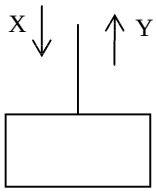
Otros módulos en cambio reciben datos de sus superiores y lo transmiten a sus subordinados. En tal caso hablamos de *flujo eferente* y de *módulos eferentes.*
Normalmente este tipo de módulos están involucrados en procesos de *salida*.



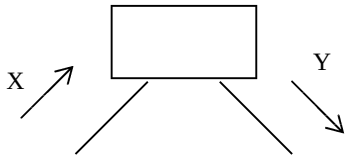
Normalmente, tanto los módulo aferentes como los eferentes, no transmitirán los datos tal como los reciben, si no que usualmente realizarán alguna transformación sobre ellos. Más allá de esto, lo importante es que estos módulos pasan información hacia arriba y hacia abajo.



Existe otro tipo de módulos que solamente realizan una transformación sobre los datos que reciben y los devuelven en otra forma. En este caso hablamos de *flujo de transformación* y de *módulos de transformación.*
Un caso típico de este tipo de módulo son los aquellos que realizan cálculos, por ejemplo los matemáticos (raíz cuadrada, etc.).



Finalmente, observamos que existen otros módulos cuya función es la de coordinar y administrar los tareas de otros. En tal caso hablamos de *flujo de coordinación* y de *módulos de coordinación.*
Este tipo de módulos pueden encontrarse en las porciones de entrada, transformaciones centrales, y salidas de un determinado sistema.
En un sistema bien diseñado, usualmente encontraremos este tipo de módulos en un nivel alto en la jerarquía.



Además de estos tipos específicos, es común encontrar módulos que combinen más de un tipo de flujo. Así encontramos módulos de coordinación y eferentes, o aferentes, etc.

6.5 Morfología de Sistemas

A lo largo de este capítulo hemos examinado las estructuras modulares desde diferentes puntos de vista. Hemos visto que el diseñador puede verse motivado a desarrollar una estructura centrada en la transacción, en el procedimiento, u verse influenciado por otro tipo de organizaciones modulares, las cuales se toman como *modelos*. También hemos visto que consideraciones de *factorización* influenciarán la distribución de módulos y la cantidad de "inteligencia" dentro de cada módulo. Finalmente, hemos visto que otra manera de caracterizar a los módulos es según los flujos de información entrantes y salientes, en aferentes, eferentes, de coordinación, o de transformación.

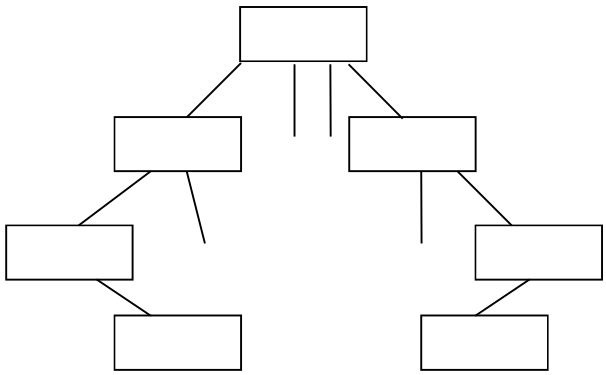
Ahora podemos poner todas las piezas juntas, y examinar la *morfología* o forma de una estructura completa.
Veremos que ciertas características como el *ancho* o amplitud y la *profundidad*, se encuentran en todos los sistemas. Comparando estas características de un determinado diseño contra las de otro, podremos determinar que dicho diseño es bueno o malo.

Profundidad
Una de las características morfológicas obvias es la *profundidad*, esto es el número de *niveles* de la jerarquía.
La profundidad, por sí misma, no es una medida válida de la calidad de un diseño.
En la práctica se observa que un programa de simple (300 líneas) puede tener una profundidad de 5 o 6 niveles, y algunos de mediana complejidad pueden tener fácilmente una profundidad de 10 a 12 niveles.

Amplitud
Otro aspecto a considerar sobre la forma de un sistema es la *amplitud*, esto es la cantidad de módulos que tiene de ancho el diseño.
A primera vista podremos decir que obviamente, cuando mayor sea la amplitud del sistema, mayor será su complejidad.

Una medida relacionada con la amplitud en forma directa es lo que los diseñadores llaman el ancho de salida (*fan-out*). Esto es el número de módulos inmediatamente subordinados que tiene un determinado módulo.
El fan-out promedio de una estructura será el promedio de los fan-out de todos los módulos excepto los inferiores o terminales que tienen fan-out cero.
Cuando mayor sea el fan-out de un módulo, mayor será su complejidad, debido a que debe contener más lógica de control y coordinación.
En una estructura modular efectiva, el fan-out promedio tiende a ser bajo.

Morfologías generales
En lugar de manejar medidas primitivas como amplitud y profundidad, consideraremos la morfología general del sistema.
Basados en la observación y estudio de numerosos sistemas durante varios años, se determinó que los sistemas mejor diseñados tienden a tener una forma como la siguiente:



La estructura puede compararse con un *cigarro, un plato volador, o una mezquita*.
Note que la forma de mezquita tiene un alto fan-out en los módulos de alto nivel, y de bajo nivel. Debemos observar que la forma de mezquita es característica de los sistemas bien diseñados, pero es potencialmente peligroso si es usado como una herramienta de diseño. Veremos también que las metodologías de Análisis de Transacción, y Análisis de Transformación, que estudiaremos más adelante, tienden a producir diseños con esta forma.

Estructuras Balanceadas y Desbalanceadas
Veremos ahora dos características morfológicas conocidas como "balanceo" y "desbalanceo".
Veamos las siguientes estructuras:

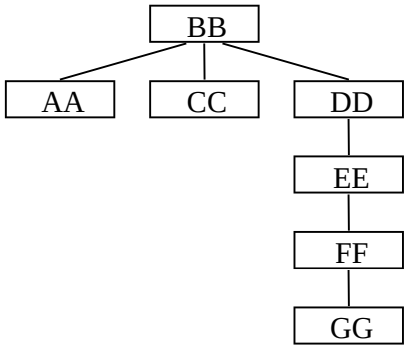


Figura A

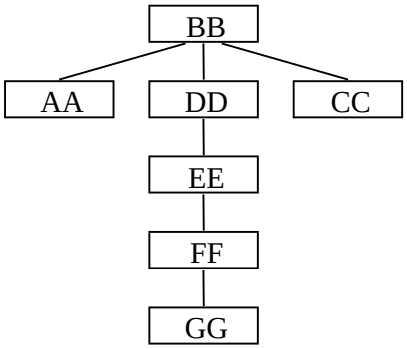


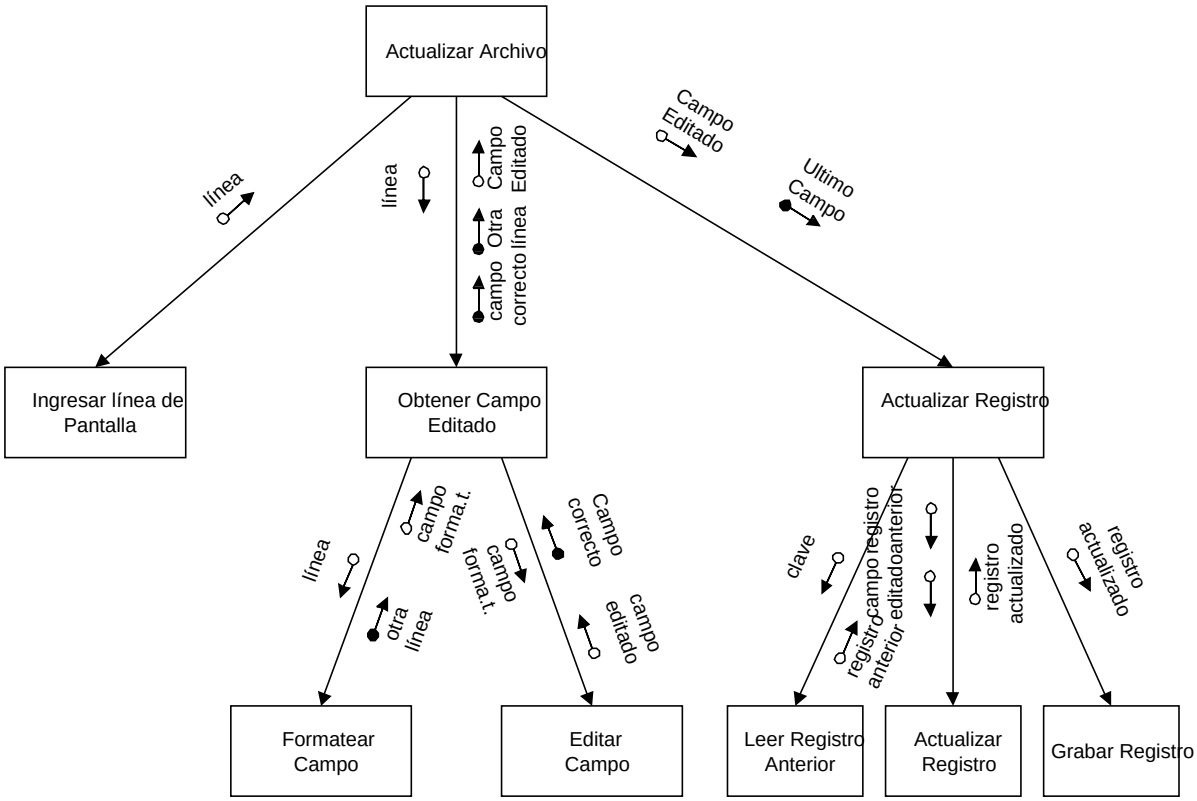
Figura B

A simple vista muchos dirán que la estructura de la figura A está desbalanceada, y que la figura B está balanceada. Sin embargo, debido a que solo las relaciones topológicas entre módulos son estructuralmente relevantes, la figura B es equivalente a la A.
El concepto de balanceo sin embargo puede ser de utilidad si podemos fijar un orden predeterminado en el dibujo de la estructura. El flujo de datos desde el origen de entradas físicas, atravesando transformaciones, y llegando finalmente a las salidas establece una determinada orientación predefinida.
En nuestro ejemplo de las figuras previas (A, B), AA es un módulo aferente cuya salida es procesada a través de BB y CC y eventualmente GG. Podemos decir que dichas estructuras están desbalanceadas respecto del flujo de datos.
Los sistemas pueden estar desbalanceados en la dirección de sus entradas, como sucede en el ejemplo previo (fig. A), o estar desbalanceado en la dirección de sus salidas.

Dependiendo del tipo de balanceo o desbalanceo que se observe, tendremos estructuras orientadas a las entradas, o *input-driven*, orientadas a las salidas u *output-driven*, y balanceadas.

Estructuras Input-Driven

Un sistema altamente desbalanceado en la dirección de sus entradas, obtiene todas sus entradas en forma física elemental (raw, crudo), cerca del **tope** de la jerarquía. Todo el procesamiento de las entradas tiene lugar en niveles menores de la jerarquía, y principalmente, en ramas de la jerarquía que son predominantemente eferentes. En verdad **el sistema entero es predominantemente eferente**. Esto es tradicionalmente llamado un sistema input-driven.
Un sistema input-driven puede comparase a un gerente sin recepcionista o secretaría que se ve forzado a atender personalmente los requerimientos operativos.
En el siguiente ejemplo se puede ver una estructura de un programa que recibe sus entradas desde una terminal CRT en forma de líneas elementales. Un mismo campo puede conformarse de una línea para lo cual se utiliza un carácter de continuación de línea. Estas líneas son recibidas directamente por el módulo ejecutivo, y de las mismas debe extraerse y editarse campos con los que se actualizarán un archivo maestro.



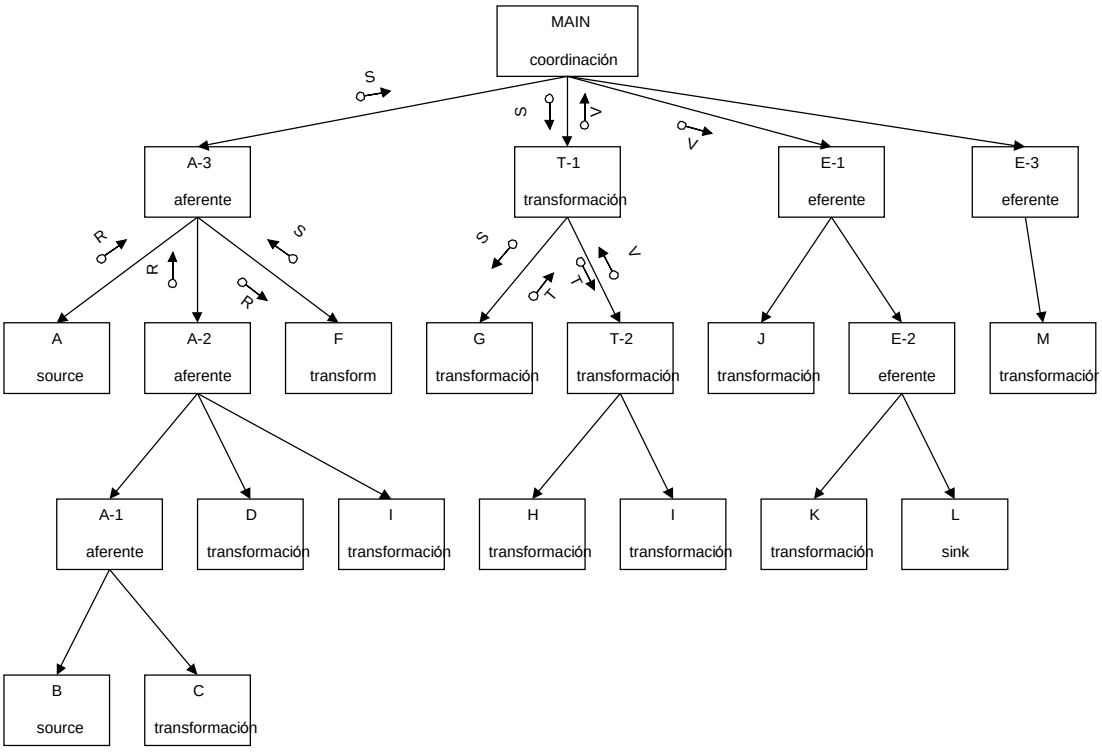
Estructuras Output-Driven
Un sistema en el cual el módulo **tope** produce las salidas del sistema en forma elemental o cruda (raw).
El código *Output-driven* puede verse como filosóficamente diferente. Con código input-driven, las entradas determinan que ocurre en el proceso: los ítems son leídos primero, luego el código decide que hacer con ellos. De otro modo, inicialmente podemos

identificar un ítem que será producido como *salida* del sistema, y en base a esto, desarrollar el proceso que será necesario para obtener dicho ítem.
En este caso, **el sistema es predominantemente aferente**. Un sistema estrictamente output-driven activará todos sus módulos aferentes, en los niveles bajos de la estructura, antes de poder obtener una primera entrada.
Por supuesto, los eventos pueden sucederse en el mismo orden que en un sistema input driven, pero la estructura del sistema será muy diferente.
Se observa como regla general que los sistemas output-driven son menos eficientes que un equivalente input-driven.

Estructuras Balanceadas
Los sistemas balanceados no tendrán ítems elementales en el tope de la jerarquía, en ninguna de las ramas de entrada ni salida. Generalmente implica que el nivel tope tendrá control inmediato sobre las funciones más importantes del sistema.
Los sistemas balanceados en sus ramas aferentes y eferentes también se conocen como *centrados en la transformación*. Este es el tipo de estructura que buscaremos lograr.
El balance también puede alcanzarse teniendo entradas y salidas elementales en el tope de la estructura, pero esto es un reflejo de una violación a la regla de factorización, provocando que el módulo ejecutivo se deba inmiscuir en los detalles.

6.6 Morfología Centrada en la Transformación
De los factores morfológicos relacionados con la simplicidad de un sistema, la morfología conocida como organización *centrada en la transformación* es la más importante.
La siguiente figura refleja un modelo centrado en la transformación:

fig.8.18



Este modelo combina varias características morfológicas deseables. Es altamente **factorizado**. Las ramas aferentes y eferentes están **balanceadas**. El modelo no es input-driven ni output-driven.
Las ramas aferentes y eferentes tienen una estructura característica. En la forma totalmente factorizada, esta estructura involucra en cada nivel una transformación simple, o conjunto de transformaciones alternativas realizadas por módulos subordinados de **transformación**, cuyas entradas son suministradas por el último módulo subordinado aferente (en la rama aferente), o cuyas salidas alimentan al próximo módulo subordinado eferente (en la rama eferente).

Este tipo de modelo fue derivado empíricamente de una cuidadosa revisión de morfologías de sistemas, comparando sistemas simples y económicos de implementar, mantener y modificar, con otros complicados y costosos.

Unidad 7: Heurísticas del Diseño

7.0 Introducción

En este capítulo, desarrollaremos algunas heurísticas con las cuales las estructuras de sistemas pueden mejorarse. Entenderemos por ***heurísticas***, ciertos "trucos", los cuales nos permitirán incrementar la modularidad del sistema. Las heurísticas no son reglas rígidas. Son útiles porque sirven como indicadores para examinar y verificar las estructuras, para potenciales mejoras.

7.1 Tamaño de Módulo

El tamaño del módulo está relacionado con la modularidad, más no solo de manera simple de "cortar en programa en más piezas". Esto es, la modularidad técnica no se incrementa simplemente cuando el tamaño de módulo decrece, mientras otros aspectos permanecen igual.

Para la mayoría de los propósitos, módulos de mucho más de cien sentencias, están fuera del tamaño óptimo con respecto a la economía de detección y corrección de errores. En el otro extremo de la escala el límite es menos obvio. Excepto para ocasionales programadores "descarrilados", que piensan que modularidad es equivalente a partir un programa en módulos de una sentencia, encontraremos que los módulos muy pequeños, han sido diseñados concisa y deliberadamente, y usualmente por razones funcionales. Sin embargo, normalmente, módulos menores a cinco a nueve líneas, deben considerarse para un replanteo. Esto es especialmente relevante cuando existen muchos de estos módulos muy pequeños en el sistema.

Sugerencias para el tamaño óptimo de un módulo han provenido de diferentes fuentes en el transcurso de los años. Una de las sugerencias conocidas es la de Baker quién recomienda que un módulo debe consistir de aproximadamente cincuenta líneas, lo que coincide con el número de líneas que se pueden poner en una hoja de listado común.

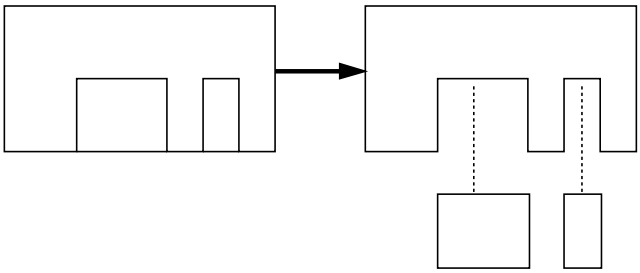
Otra recomendación proviene de Weinberg, cuyos estudios muestran que la comprensión de un programador se disipa fácilmente si el módulo tiene más de treinta líneas.

No siempre módulo muy grandes o muy pequeños son necesariamente malos. La cuestión importante es que los módulos reflejen la estructura del problema lo más fielmente posible. Descomponer un módulo simplemente para llevarlo a un tamaño óptimo, aún perdiendo su significado con respecto a la estructura del problema, no es una buena estrategia.

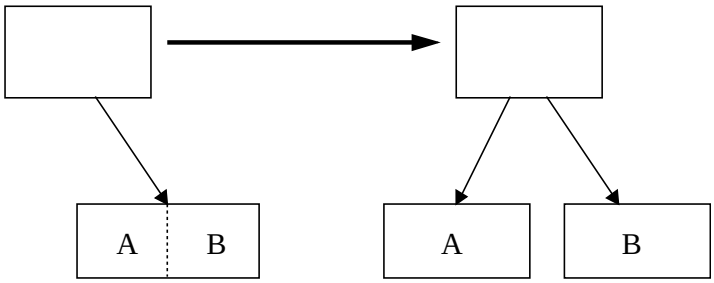
Debemos examinar separadamente cada caso de módulos muy grandes o muy pequeños. Un módulo demasiado grande a menudo puede deberse a dos razones principales:

- Una factorización incompleta en módulos subordinados apropiados
- Dos o más funciones han sido combinadas (frecuentemente con cohesión lógica) en un mismo módulo.

En el primer caso, se puede realizar una reducción a través de la identificación y extracción de subfunciones.



En el segundo caso, podemos descomponer el módulo en sus funciones constitutivas.



En cualquier caso, los diagramas de estructura pueden usarse como herramienta, y las modificaciones estructurales deben probarse sobre papel.
Recalcamos que lo importe es dar sentido a la nueva estructura dentro del contexto del problema. Si no es posible dar una interpretación razonable a la nueva estructura, debe mantenerse la original.

Cuando tratamos con módulos muy pequeños, debemos distinguir dos casos:
- el módulo atómico (nivel más bajo)
- módulo no atómico

En el caso de *módulos atómicos*, las cuestiones a considerar son: la cantidad de llamadas desde módulos superiores que recibe el módulo (fan-in), y la sobrecarga (overhead) producido por el proceso de llamada a subrutina.
Un módulo muy pequeño puede ser comprimido en sus módulos superiores. Cuando el módulo tiene un uso simple (fan-in = 1) esta es una alternativa a considerar. Pero si el módulo tiene usos múltiples (alto fan-in) puede ser muy peligroso absorverlo en sus módulos superiores, porque se duplicará el esfuerzo de implementación y mantenimiento del mismo.
En el caso de que la sobrecarga producida por el mecanismo de llamada a subrutina sea intolerable, se puede testear y depurar el módulo independientemente y luego utilizar la inclusión *en línea* del módulo subordinado en sus superiores (ligado en tiempo de compilación). La mayoría de los lenguajes dan facilidades para esto como ser el *COPY* del Cobol, *#include* del C, *%INCLUDE* del PL/I, las *macros* de la mayoría de los assembler., etc., o utilizar facilidades de los preprocesadores de compilación.

Cuando el módulo es *no atómico*, el análisis es más complicado. Las opciones son comprimir el módulo hacia arriba en su superior, o hacia abajo en un subordinado. Ambas opciones deben ser consideradas.
Un caso especial es el llamado *módulo fantasma*, un módulo que lo único que contiene son llamadas a otros módulos subordinados. Obviamente estos módulos son el caso límite de módulos muy pequeños.

A través de esta discusión, se asumió que el diseñador conoce previo a la implementación el tamaño que tendrán los módulos. Normalmente esto no requiere un proceso de estimación sustancial. La experiencia ha demostrado que el tamaño comparativamente pequeño de módulos en sistemas razonablemente modulares simplifica el proceso de estimación. En adición, el proceso de estimación se torna más exacto cuando el diseñador tiene experiencia con la función que realizarán varios módulos.

7.2 Amplitud del Control (Fan-out)

La *amplitud del control* o ancho de salida (fan-out), es el número de subordinados inmediatos de un módulo. Al igual que en con el tamaño de módulo, amplitudes de control muy altas o muy bajas, son indicadores de un posible diseño pobre.
Ya se ha visto anteriormente que la función de un administrador se torna muy compleja si el número de subordinados excede 7±2 subordinados inmediatos. En general deben verificarse anchos de salida mayores a 10 y aquellos de solamente 1 o 2. También observaremos que una amplitud de control muy alta es más peligroso que una baja.

Una amplitud de control demasiado baja puede incrementarse en la mayoría de los casos descomponiendo el módulo en subfunciones subordinadas adicionales, o comprimiendo el módulo en sus superiores. Como se dijo en el punto anterior, esto es válido si se puede dar a los nuevos módulos un sentido dentro de la estructura del problema.

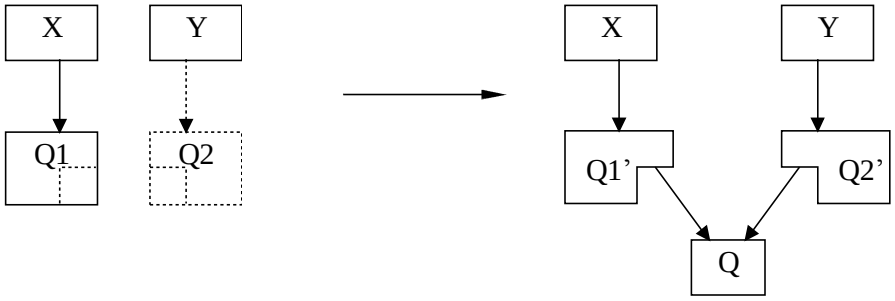
Una amplitud de control demasiado alta puede ser un indicativo de una reticencia por parte del diseñador a delegar responsabilidades en módulos subordinados. Sin embargo en la mayoría de los casos, esto proviene de estructuras "panqueque" o fallas en la definición de niveles intermedios. Para solucionar este problema, debe considerarse la posibilidad de agruparse varios subordinados en una función combinada. El principio de cohesión debe guiar este proceso para evitar módulos de baja cohesión.

7.3 Ancho de Entrada (Fan-In)

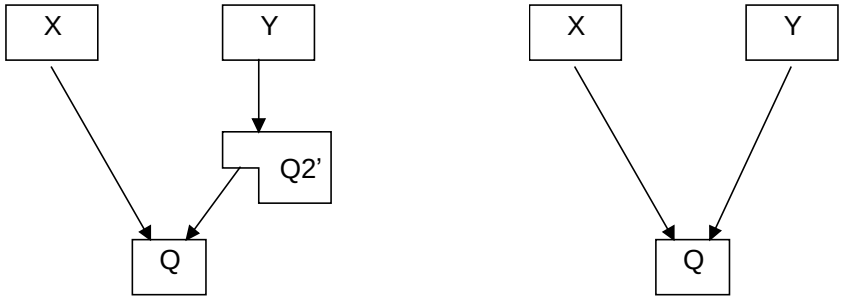
Siempre que sea posible, desearemos maximizar el *ancho de entrada* de un módulo (fan-in) durante el proceso de diseño. Cada instancia de múltiples entradas de un módulo, representa que ha podido evitarse duplicidad de código.
Sin embargo, un gran ancho de entrada no es algo que deba buscarse a cualquier costo. Por ejemplo, es ridículo combinar muchas funciones "no relacionadas" en un módulo bajamente cohesivo, con el propósito de incrementar el fan-in.

Un gran ancho de entrada es alcanzado a través de un proceso analítico que acompaña los pasos del proceso de diseño estructurado. Cada vez que vamos a dibujar un nuevo módulo en el diagrama de estructura, primero debemos verificar si no existe ya algún otro módulo que realice la función requerida. Si es así, dibujaremos una flecha hacia la caja del módulo ya existente en el diagrama, en lugar de dibujar una nueva. En orden de evitar un "enriedo" de flechas en el diagrama, pueden utilizarse conectores dentro de la página (un círculo) y fuera de página (un polígono), similarmente a la que se hace en los diagramas de flujo.
Es importante notar que la especificación del fan-in es una tarea del diseñador, no del implementador (codificador).

Un problema puede producirse cuando el nuevo módulo es parecido pero no exactamente igual al módulo existente. Si el diseñador no distingue la diferencia subyacente entre ambos módulos, pueden generarse problemas durante la integración del sistema. Podrán fallar el nuevo módulo, el anterior uso del módulo existente.
La clave está en comprender cual es la parte común de ambos módulos, y aislarlo en un nuevo módulo. Por ejemplo, supongamos que tenemos una función Q1 que parece ser similar a Q2. Si Q es la parte común de ambas funciones, podemos reestructurar nuestro diseño de la siguiente manera:



Las porciones remanentes Q1' y Q2' pueden reestructurarse absorviéndolas en sus módulos superiores si fueran pequeñas. Podemos tener así una serie de alternativas diferentes.



7.4 Alcance de Efecto / Alcance de Control

Cada decisión o sentencia condicional (if-then-else) en un sistema tiene algunas consecuencias: ciertos procesos se realizarán o no como resultado de una decisión. Equivalentemente podemos decir que cierto procesamiento es condicional en base a la salida o resultado de una decisión. Es importante aprender donde se encuentran los resultados de una determinada decisión, dentro de una estructura modular. En orden de este estudio, introduciremos dos términos nuevos: *alcance de efecto* y *alcance de control*.

El *alcance de efecto* de una decisión es la colección de todos los módulos que contienen algún procesamiento que está condicionado por dicha decisión.

Aunque solo una pequeña parte del procesamiento de un módulo se vea afectada por la decisión. Si la activación de un módulo completo está condicionada por la decisión, el módulo superior (invocador) también se incluye dentro del alcance de efecto.

El *alcance de control* de un módulo es el módulo mismo y *todos* sus subordinados.

El alcance de control es un parámetro puramente estructural independiente de las funciones del módulo.

Estamos en condiciones ahora de plantear la siguiente heurística de diseño que involucra al alcance de efecto y al alcance de control:

Para una decisión dada, el alcance de efecto debe ser un subconjunto del alcance de control del módulo en el cual se encuentra la decisión.

En otras palabras, todos los módulos que son afectados o influenciados por una determinada decisión deben estar subordinados finalmente al módulo que toma la decisión. La toma de decisiones y la estructura modular se interrelacionan mejor cuando las decisiones se toman a un nivel no más alto que lo necesario dentro de estructura jerárquica, con el objetivo de ubicar el alcance de efecto dentro del alcance de control. En el caso ideal, el alcance de efecto debe estar limitado al módulo en el cual se realiza la decisión y a aquellos módulos *inmediatamente* subordinados al mismo.

Examinemos los siguientes cuatro casos:

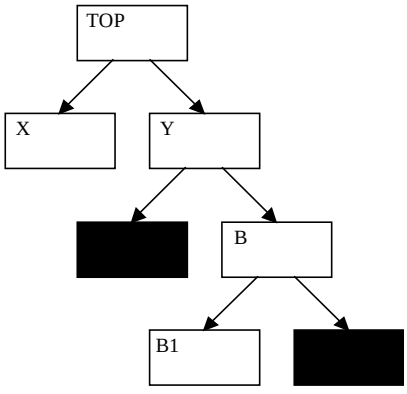


Fig. 1

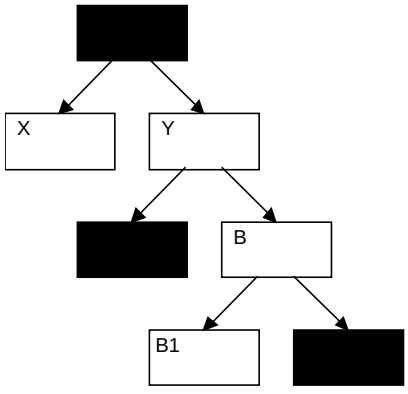


Fig. 2

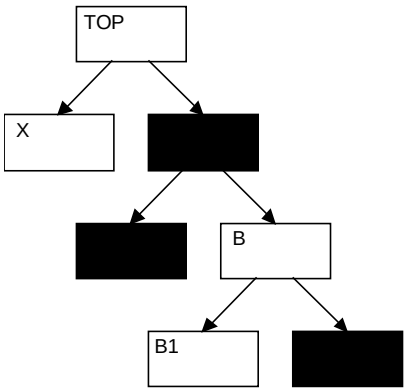


Fig. 3

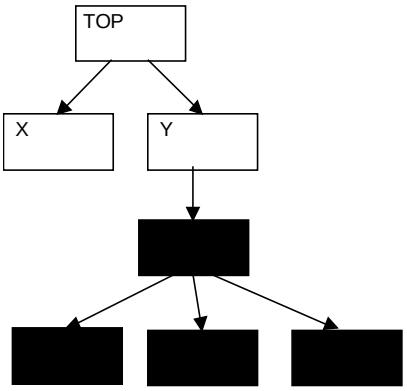


Fig. 4

El alcance de efecto de la decisión se representa por los módulos sombreados. La Fig.1 ilustra un caso en el cual el alcance de efecto no es un subconjunto del alcance del control.
La Fig.2 muestra una estructura en la cual el alcance de efecto está contenido dentro del alcance del control, pero puede discutirse que la decisión es realizada demasiado alta dentro de la jerarquía. En la Fig. 3 se ha llevado la decisión al nivel justo para incluir el alcance de efecto dentro del alcance de control. La Fig. 4 muestra el caso ideal en el los módulos a quienes afecta la decisión son subordinados inmediatos del módulo que toma la decisión.

7.5 Sumario

Debemos destacar que lo visto en este capítulo son ***heurísticas*** y no reglas "religiosas". Heurísticas tales como tamaño del módulo, ancho de entrada, ancho de control, alcance de efecto/control, son sumamente valiosas si se utilizan apropiadamente, pero pueden llevar a resultados pobres si se interpretan demasiado *literalmente.*

Unidad 8: Análisis de Transformación

8.0 Introducción

En el capítulo anterior hemos visto que los sistemas cuya morfología o forma está centrada en la transformación, tienden a ser asociados con bajos costos de desarrollo, mantenimiento, y modificación. Hemos visto también que los sistemas altamente factorizados tienden a ser económicos. Los módulos de alto nivel toman la mayoría de las decisiones, y los módulos inferiores realizan el trabajo de detalle.

El *Análisis de Transformación*, o *diseño centrado en la transformación*, es una estrategia para derivar diseños estructurados que son bastante buenos (con respecto a su modularidad) y que necesitan solo una modesta reestructuración para arribar al diseño final.

Es una forma particular de la estrategia descendente (top-down), que toma ventaja de la perspectiva global. Aplicado rigurosamente, el análisis de transacción conduce a estructuras que son altamente factorizadas. Produce un número variable de módulos en los niveles intermedios de la jerarquía, los cuales representan composición de funciones básicas. Siempre se trata de evitar que los módulos intermedios realicen cualquier "trabajo" excepto el de control y coordinación de sus subordinados.

El propósito de la estrategia es el de identificar las funciones de procesamiento primarias del sistema, las entradas de alto nivel de dichas funciones, y las salidas de alto nivel. Se crean módulos de alto nivel dentro de la jerarquía que realizan cada una de estas tareas: creación de entradas de alto nivel, transformación de entradas en salidas de alto nivel, y procesamiento de dichas salidas.

El análisis de transformación es un modelo de *flujo de información* más que un modelo procedural.

La estrategia de análisis de transformación consiste de **cuatro pasos** principales:

1) plantear el problema como un diagrama de flujo de datos
2) identificar los elementos de datos aferentes y eferentes
3) factorización del primer nivel
4) factorización de las ramas aferente, eferente, y de transformación

8.1 El Primer Paso: obtención del Diagrama de Flujo de Datos del problema

En orden de llevar adelante la estrategia del análisis de transformación, es necesario estudiar el flujo de datos a través del sistema. Para esto debemos dibujar un diagrama de flujo de datos (DFD) del sistema que estamos diseñando.

Una cuestión de importancia es *como* debe realizarse el diagrama de flujo. Hay varios enfoques para este proceso analítico.
Ante todo debemos tener en cuenta que un DFD no es un modelo procedural.

Algunos diseñadores experimentados en el uso de DFD's, generalmente comienzan con las entradas físicas (p.e. un mensaje desde una terminal) y trabajan dichas entradas "corriente abajo" a través de las sucesivas transformaciones hasta llegar a las salidas físicas.
Otra estrategia, es partir de las salidas y seguir el flujo "hacia arriba", hasta llegar a las entradas físicas.
La adopción de uno de los dos enfoques es más cuestión de preferencia del diseñador que una cuestión técnica.
Desafortunadamente, muchos novatos en el uso de DFD's encuentran que estos dos enfoques tienden a "perderlos" en los detalles procedurales, los cuales deben dejarse de lado en este momento.
Una estrategia es comenzar con una burbuja de alto nivel, e ir explotándola en sucesivos refinamientos. Esto es un ejercicio útil de pensamiento descendente.
La cantidad de detalle mostrada dependerá del problema y del diseñador, aunque es aconsejable llegar a un considerable nivel de detalle en la etapa de diseño.

En el dibujo de un DFD es esencial no enmarañarse en aspectos de *procedimiento* o *toma de decisiones*. El diagrama no debe mostrar cosas como iteraciones, inicialización, terminación, decisiones, o procedimientos de recuperación.

Sugerencias para la realización de DFD's:

- Desarrolle consistentemente el problema, desde las corrientes de entrada hacia las salidas o viceversa, dependiendo de su preferencia. Utilice el pensamiento descendente para descomponer tareas no triviales.

- Nunca intente mostrar lógica de control. Si se encuentra pensando en términos de iteraciones o decisiones, está en el rumbo equivocado. Recordar siempre que las flechas representan flujos de datos, no de control.

- Ignore los procedimientos de inicialización y terminación en este momento.

- Omita el tratamiento de errores simples.

- Etiquete cuidadosamente los flujos que entran y salen de las burbujas.

- Asegúrese de que el uso de los operadores * y + es el correcto.

- Asegúrese de que el flujo de datos es el correcto para el nivel de detalle que está mostrando. En caso de duda es preferible mucho detalle a poco detalle.

8.2 El Segundo Paso: Identificar los Elementos de Datos Aferentes y Eferentes

En la discusión de morfología de sistemas, introducimos los conceptos de flujo de datos aferente y flujo de datos eferente. Vimos también que los módulos pueden categorizarse en función de sus tipos de flujos en módulos aferentes y módulos eferentes.

Definiremos ahora ***elementos aferentes de datos****: son aquellos elementos de datos de alto nivel que habiendo sido removidos de sus entradas físicas, todavía pueden considerarse entradas al sistema.*
Por lo tanto, los elementos aferentes de datos son el nivel más alto de abstracción para el término "entradas al sistema". Representan el más alto nivel de entradas.
En general, los elementos de dato aferentes deben tener la menor resemblanza posible con lo datos de entrada "crudos" obtenidos de sus entradas físicas. Esto es, el bloqueo físico, los caracteres de control, debe ser removido. Solo datos limpios para el proceso deben permanecer.
Identificaremos los elementos aferentes de dato comenzando en las entradas físicas al sistema, y siguiendo los flujos de datos en el diagrama hasta que identifiquemos que la corriente de datos no pueda ser considerada como entrante. Esto implica un juicio de valor por parte del diseñador; la idea es llegar lo más lejos posible de las entradas físicas.
Este proceso se realizará para cada corriente de entrada. A menudo encontraremos que varias corrientes físicas de entrada finalizarán en el mismo elemento de dato eferente.

Comenzando en el extremo opuesto con las salidas físicas, trataremos de identificar los ***elementos eferentes de datos***. Esto lo realizaremos siguiendo los elementos de datos desde sus salidas físicas a través de los flujos, hasta que no puedan seguir siendo considerados como datos de salida del sistema.
Estos elementos pueden ser considerados como "datos lógicos de salida", que han sido producido por el "proceso central" del sistema, y los cuales serán convertido en "datos físicos de salida". Realizaremos este proceso con cada corriente de salida.
Este proceso también implica un juicio de valor por parte del diseñador; la idea es llegar lo más lejos posible de las salidas físicas.

Este proceso de identificar elementos aferentes y eferentes dejará algunas transformaciones en el medio entre dichos elementos. Estas son denominadas ***transformaciones centrales***. Ellas son el trabajo principal del sistema. Ellas transforman las entradas principales del sistema, en las mayores salidas.
Ocasionalmente los elementos aferentes y eferentes de datos pudieran ser los mismos. En tal caso no existe transformaciones centrales.

Podría pensarse este paso es un intento de modelizar todos los sistemas con un flujo IPO trivial. En verdad, la mayoría de los sistemas pueden pensarse de esta manera, al fin todos tienen alguna entrada, realizan alguna transformación, y emiten alguna salida. Sin embargo la importancia de este enfoque radica en que muchos diseñadores tienden a concentrarse demasiado en las corrientes de entrada generar sistemas input-driven. Este enfoque permite obtener sistemas balanceados.

8.3 El Tercer Paso: Factorización del Primer Nivel

Habiendo identificado los elementos aferentes y eferentes de datos del sistema, especificaremos un *módulo principal*, el cual al ser activado, realizará todas la tareas del sistema invocando a sus subordinados.

Para *cada* elemento de datos aferente que alimente a la transformación central, se especificará un *módulo aferente* como subordinado inmediato del módulo principal. Su función final será la de suministrar al módulo principal el elemento aferente de datos.

Similarmente, para *cada* elemento eferente de datos, definiremos un *módulo eferente* subordinado al módulo principal. Dicho módulo transformará el elemento eferente de datos en las salidas físicas finales.

Finalmente, para *cada* transformación central, o composición funcionalmente cohesiva de transformaciones centrales, especificaremos un módulo de transformación subordinado al módulo principal, el cual aceptará desde el módulo principal los datos de entrada apropiados y los transformará en los datos de salida apropiados, los cuales serán devueltos al módulo principal.

Por lo general veremos que existe una correspondencia simple (usualmente uno a uno) entre el diagrama de flujo de datos y el nivel de factorización inicial.
El módulo principal en el controlador principal o ejecutivo del proceso. Su función es controlar y coordinar los módulos aferentes, de transformación, y eferentes, tratando con los datos de máximo nivel del sistema. Este módulo llamará a los módulos aferentes para obtener las principales entradas del sistema, pasará dichas entradas a módulos de transformación, entregará los resultados de estos a otros módulos de transformación , y finalmente a los módulos eferentes. Este proceso de invocaciones involucrará las mayores decisiones e iteraciones lógicas del proceso general del sistema.

8.4 El Cuarto Paso: Factorización de los Flujos Aferentes, Eferentes y de Transformación

Tres estrategias distintas son usadas para factorizar los tres tipos de módulos subordinados (aferente, eferente, y de transformación) en módulos subordinados de bajo nivel.
No existe una razón particular para comenzar con la porción aferente del sistema, pero muchos diseñadores lo encuentran una forma más natural de proceder.
No es necesario factorizar completamente una rama hasta el menor nivel de detalle antes de continuar con otra rama, pero si es importante identificar todos los módulos subordinados de un módulo dado antes de comenzar con otro módulo.

Factorización del módulo aferente: para factorizar un módulo aferente, debe considerarse que este provee a su módulo superior información elaborada a través de algún tipo de transformación. Por esto el módulo aferente tendrá como subordinado uno o más módulos de *transformación.* A su vez esta transformación recibirá sus datos de algún(os) módulo(s) aferentes subordinados al módulo aferente principal. Cada uno de estos nuevos módulos aferentes pueden ser descompuestos de igual manera a la indicada.

Factorización del módulo eferente: la factorización del módulo eferente principal, es esencialmente simétrica a la del módulo aferente. Para un módulo eferente dado, buscamos la *siguiente* transformación a ser aplicada que nos aproximará los datos a su

forma última de salida. Por lo tanto existirá un módulo de transformación subordinado al módulo eferente principal. La salida de este módulo de transformación alimentará la entrada de un nuevo módulo eferente. Naturalmente, puede haber más de una "siguiente transformación" y más de un módulo eferente siguiente.

Factorización de los módulos de transformación central: poco es conocido acerca del la factorización óptima de los módulos de transformación central. Obviamente, para cada transformación, buscaremos sub transformaciones que compondrán la transformación total. Buscaremos también *composiciones o grupos* de funciones mostradas como transformaciones centrales en el DFD original. Estas se insertarán como módulos intermedios en la jerarquía entre el nivel superior y las funciones que componen el grupo.

El juicio y la experiencia del diseñador, son guiados en este proceso, por consideraciones de *cohesión* y *acoplamiento* como así también por varias heurísticas que se estudiarán posteriormente.

8.5 El Quinto Paso: Desviaciones

La estrategia descripta asume una estructura ortodoxa en la cual el flujo de datos es entrante o saliente en cualquier rama, pero nunca los ambos a la vez. Consecuentemente, se espera que los módulos aferentes tengan solo subordinados aferentes y de transformación. Similarmente, los módulos eferentes tendrán solo subordinados eferentes y de transformación, y los módulos de transformación (más allá de donde se encuentren en la estructura) solo tendrán subordinados de transformación.
Sin embargo, los problemas del mundo real, frecuentemente requieren excepciones a estas reglas para evitar procesos torpes.
Por ejemplo, si un determinado módulo aferente produce un flujo que es utilizado solo en casos muy especiales, será normal ubicar este módulo más abajo en la estructura subordinado al módulo que requiera dicho flujo de excepción, ya que hacerlo depender directamente del módulo superior resultaría artificial.
Debe tenerse en mente que el objetivo, tal lo planteado anteriormente, es tratar que la estructura de programa refleje la estructura del problema lo más cercanamente posible. Un DFD detallado es una guía para la estructura del problema, y si los requerimientos del problema necesitan una desviación de la organización centrada en la transformación ortodoxa, esta debe aparecer en el diagrama.

8.6 Como finalizar la factorización

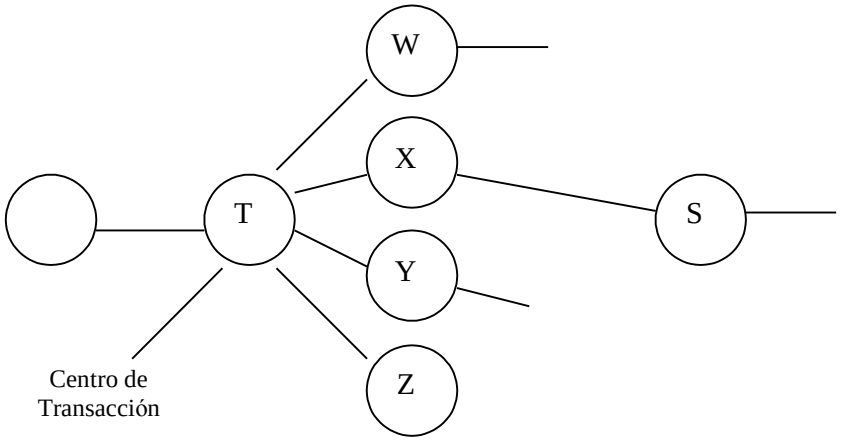
Varios criterios pueden utilizarse para determinar cuando la factorización funcional de módulos debe terminarse. El fin puede alcanzarse cuando para una transformación es difícil distinguir claramente subtareas.
Cuando un módulo de terceras partes (función de biblioteca) es alcanzado, no puede continuarse factorizando.
Similarmente cuando se alcanza un módulo con interfaces directas a señales físicas de entrada / salida, significa el fin de la factorización.

Finalmente, cuando identificamos módulos muy pequeños, puede ser un indicativo de que se ha alcanzado el nivel más bajo de la estructura.

Unidad 9: Análisis de Transacción

9.0 Introducción

En el último capítulo exploramos la estrategia del análisis de transformación como la estrategia principal para el diseño de programas y sistemas bien estructurados. En verdad, el análisis de transformación, servirá de guía en el diseño de la mayoría de los sistemas. Sin embargo hay numerosas situaciones en las cuales estrategias adicionales pueden utilizarse para suplementar, y aún reemplazar, el enfoque básico del análisis de transformación.
Una de estas estrategias suplementarias principales se conoce como *Análisis de Transacción.*
El análisis de transacción es sugerido por un DFD del siguiente tipo:



En este DFD existe una transformación que bifurca la corriente de datos de entrada en varias corrientes de salida discretas. En muchos sistemas tal transformación puede ocurrir dentro de la transformación *central.* En otros, podremos encontrarla tanto en las ramas aferentes como eferentes del diagrama de estructura.

La frase análisis de transacción sugiere que construiremos un sistema alrededor del concepto de "transacción", y para muchos la palabra transacción está asociada con sistemas administrativos. Esto si bien es cierto, es común encontrar centros de transacción en los sistemas administrativos, también pueden encontrarse en otro tipo de sistemas como ser los de tiempo real, aplicaciones de ingeniería, etc.
Un factor importante es como definimos el término transacción. En el sentido más general podemos decir:

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.

Acorde a esta definición, un gran número de situaciones encontradas en aplicaciones de procesamiento de datos comunes pueden ser consideradas transacciones. Por ejemplo cualquiera de los siguiente casos pueden considerarse transacciones:

- El operador presiona el botón de inicio de un dispositivo de entrada.
- Algún tipo de datos de entrada que designe un ingreso en el inventario.
- Un carácter de escape desde una terminal, indicando la necesidad de un procesamiento especial.
- Una interrupción de hardware ante un índice fuera de los rangos definidos dentro de un programa de aplicación.
- Un cuelgue o descuelgue de teléfono para un sistema de control de llamadas telefónicas.

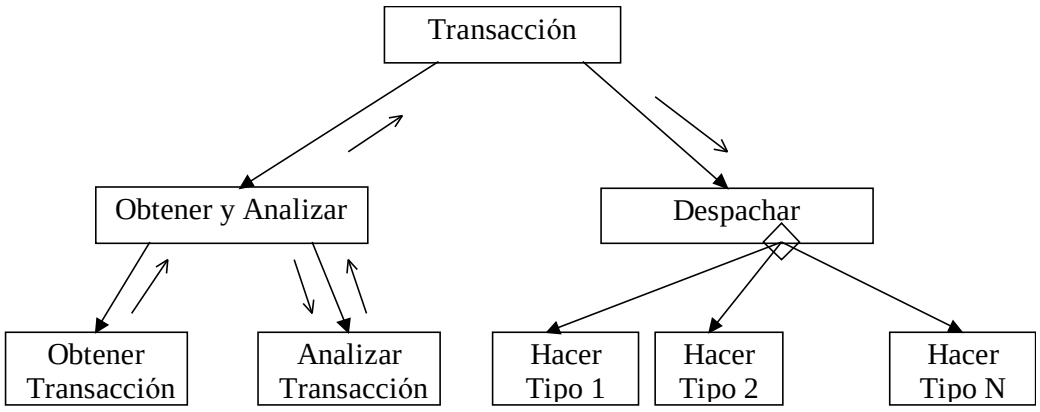
9.1 Estrategia del Análisis de Transacción

9.1.1 Centro de Transacción

La estrategia del Análisis de Transacción, simplemente reconoce que los DFD de la forma planteada previamente pueden mapearse en una estructura de programa particular.
Un centro de transacción de un sistema debe ser capaz de:

- obtener o responder a transacciones en forma "cruda"
- analizar cada transacción para determinar su tipo
- despachar acorde al tipo de transacción
- completar el procesamiento de cada transacción

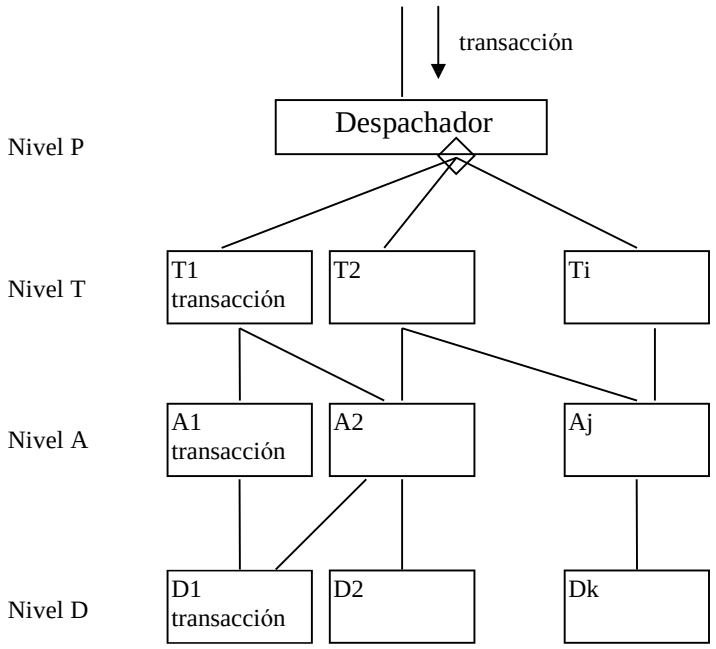
En la forma más factorizada, el centro de transacción puede ser modularizado de la siguiente manera:



La raíz de esta estructura (transacción), puede estar subordinada a cualquier parte de un sistema mayor. Cada un de los módulos “Obtener Trans.”, “Analizar Trans.”, “Hacer tipo 1”, “Hacer tipo n”, pueden ser en sí mismas raíces de subsistemas enteros.

En forma ortodoxa, la subestructura debajo del módulo de despacho puede ser modelada en un sistema de cuatro niveles. Estos niveles son los siguientes:

- Procesador de transacción (o nivel P)
- Nivel de Transacción (o nivel T)
- Nivel de Acción (o nivel A)
- Nivel de Detalle (o nivel D)



En la forma ortodoxa, el procesador de transacciones (Despachador) esperará recibir una transacción desde su subordinado cuando sea activado. Un sistema puede incluir cualquier número de centros de transacción. Un centro de transacción puede estar ubicado en una rama aferente, de transformación, o eferente.
Las salidas del centro de transacción pueden consistir de:

- Una versión convertida o formateada de la transacción entrante, las cuales pueden ser pasadas hacia arriba para alimentar niveles superiores de procesos aferentes.

- Una simple bandera indicando cuando la transacción entrante fue válida. Podemos esperar encontrar este tipo de forma de validación frecuentemente en las ramas aferentes.

- Resultados computados basados en el procesamiento de la transacción entrante. Los resultados pueden pasarse hacia arriba al módulo superior para ser utilizados en otras transformaciones centrales , o hacia abajo hacia los niveles inferiores de procesos eferentes.

- Una forma actualizada (modificada) de un elemento o elementos de datos base, internos o externos.

9.1.2 La Estrategia

Utilizando las figuras previas como modelos, podemos resumir los pasos de la estrategia del análisis de transacción como sigue:

1) *Identificar las fuentes de transacción.* En muchos casos, las transacciones serán mencionadas explícitamente en la definición del problema, en cuyo caso será usual asumir que las transacciones provienen de un medio físico de entrada. En otros casos, el diseñador puede tener que reconocer módulos aferentes, eferentes, o de transformación que generen transacciones. Esto puede hacerse más obvio luego de los primeros pasos de factorización de un diseño centrado en la transformación. Más de una corriente de transacción puede alimentar el módulo de nivel P, las cuales pueden mezclarse desde diferentes direcciones (p.e. aferentes y eferentes). Las corrientes de transacción pueden también mezclarse con corrientes de datos no transaccionales.

2) *Especificar la organización centrada en la transacción apropiada*. La figura planteada puede ser un buen modelo, pero el diseñador deber sentirse libre de alterarla como considere necesario, basándose en la teoría y las heurísticas vistas.

3) *Identificar las transacciones y sus acciones definidas.* De nuevo, a menudo encontraremos que todos los requisitos de información serán provistos por la definición del problema mismo. Si las transacciones son generadas internamente por el sistema, el diseñador deberá definir cuidadosamente el proceso a realizar para cada transacción.

4) *Identificar situaciones potenciales en que puedan combinarse módulos*. Como en el caso del análisis de transformación, podemos encontrar situaciones en las que un módulo de nivel intermedio pueda ser creado desde un grupo funcionalmente cohesivo de módulos de bajo nivel. Esta combinación puede ser apropiada en situaciones en las que la sintaxis o semántica de varias transacciones es similar.

5) *Para cada transacción, o colección cohesiva de transacciones, especificar un* ***módulo de transacción*** *para procesarlo completamente*. Debido a que las transacciones en un sistema son a menudo similares, existe una tentación a agrupar el procesamiento de varias transacciones en un módulo. Esto deberá evitarse si el

módulo resultante tiene baja cohesión. Trataremos evitar módulos con solo cohesión de comunicación, o lógica.

6) *Para cada acción en una transacción, especificar un **módulo de acción** subordinado al módulo de transacción correspondiente.* En esencia esto es un paso de factorización tal como se vio anteriormente. Debe notarse que pueden existir muchas oportunidades para que módulos de transacción compartan módulos de acción comunes.

7) *Para cada paso detallado en un módulo de acción, especificar un **módulo de detalle** subordinado a algún módulo de acción que lo necesite*. Claramente esto es una continuación del proceso de factorización. Para un sistema grande con transacciones complejas, podremos tener varios niveles de módulos de detalle. En adición, debe tenerse en mente que módulos de acción similares pueden compartir módulos de detalle siempre que sea posible.

A través de este proceso, el diseñador será guiado por los principios de cohesión, acoplamiento, y las heurísticas de diseño. Además el diseñador deberá recordar el principio fundamental mencionado repetidamente: La estructura del sistema deberá reflejar lo más cercanamente posible la estructura del problema.

9.2 Ejemplo de Análisis de Transacción

Programa liquidación de sueldos.
Se debe realizar liquidación de haberes de sueldo del personal. Para ello se dispone de tres archivos que contienen los datos necesarios, Empleados, Escalafones, y DatosBases. La estructura de estos almacenes es la siguiente:

EMPLEADOS = {Empleado}
Empleado = n_legajo + cod_escalafon + datos_personales

ESCALAFONES = {Escalafón}
Escalafón = @cod_escalafon + descripción + {cod_cpto}

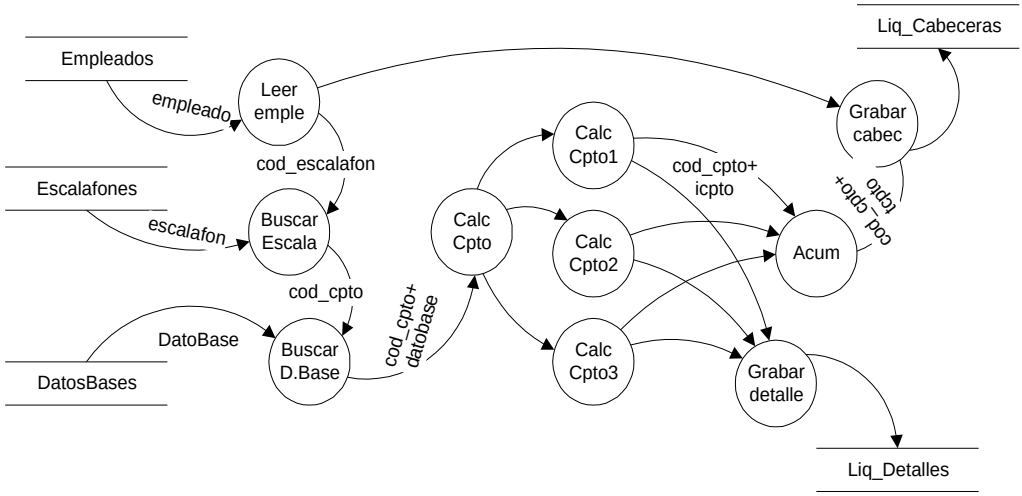
DATOSBASES = {DatoBase}
DatoBase = @cod_cpto + datos_básicos

El proceso debe generar la liquidación en dos archivos: Liq_Cabeceras y Liq_Detalles, cuyas estructuras son las siguientes:

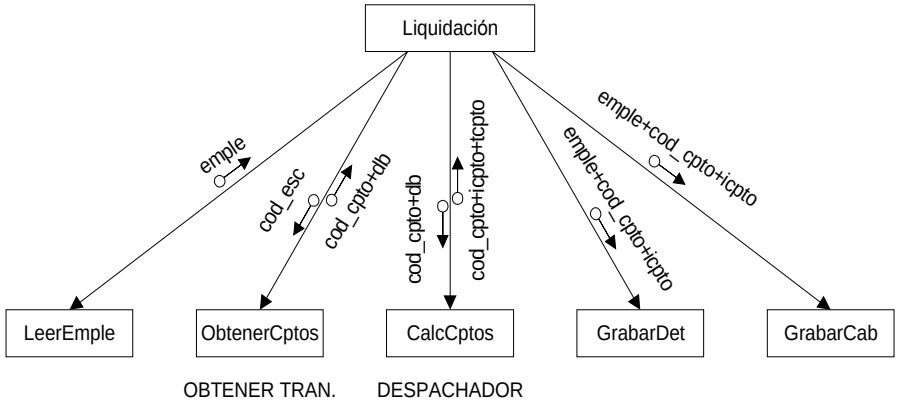
LIQ_CABECERAS = {Liq_Cabecera}
Liq_Cabecera = n_legajo + tot_cptos

LIQ_DETALLES = {Liq_Detalle}
Liq_Detalle = n_legajo + cod_cpto + imp_cpto

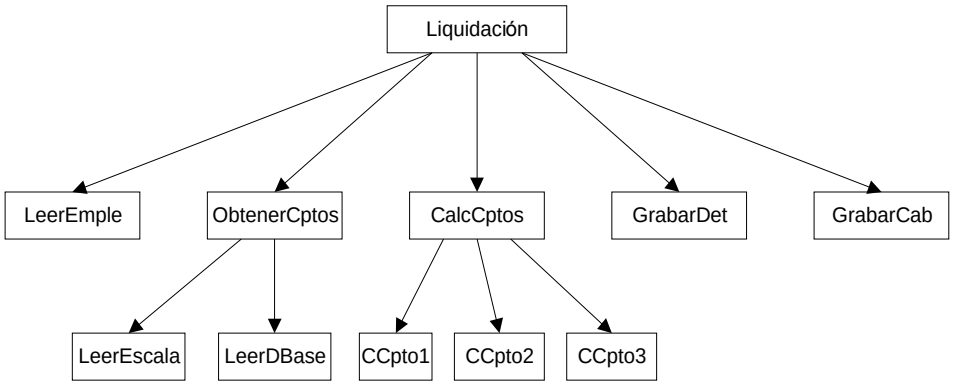
Diagrama de Flujo de Datos



Primer nivel de Factorización



Final



9.3 Consideraciones especiales en el Procesamiento de Transacciones

Las ideas de la estrategia del análisis de transacción son muy familiares a la mayoría de profesionales de EDP. Muchos han intentado utilizar estos principios para organizar sistemas enteros. La experiencia ha establecido que dichos sistemas son más sencillos de organizar en principio, pero son más difíciles de modificar y mantener posteriormente.

En el caso límite, el módulo ejecutivo de mayor nivel puede ser el centro de transacción. Por lo tanto, todas las transacciones serán consolidadas en el módulo ejecutivo en una organización input-driven. Esto puede mezclar también transacciones de diferentes tipos y niveles de importancia, para el sistema. Esto llevará a que el módulo ejecutivo sea poco cohesivo.

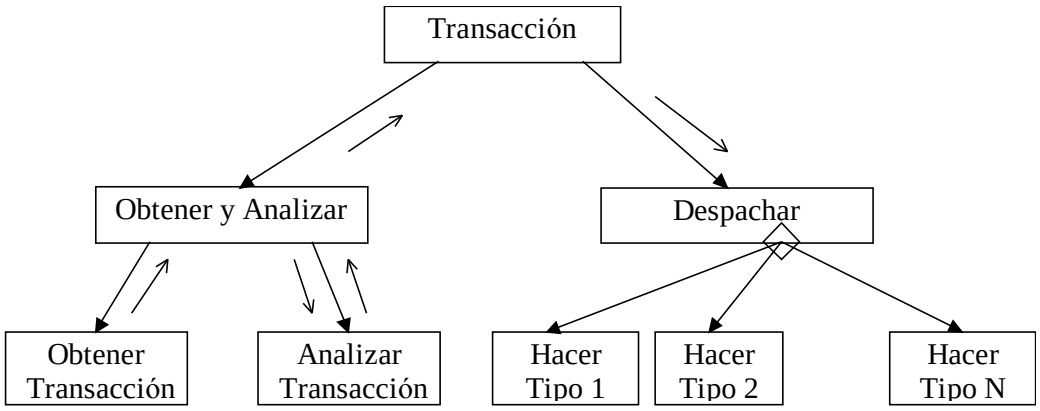
Un ejecutivo, el cual es el centro de transacción, no controla el flujo general del proceso, sino que está involucrado en detalles procedurales para realizar alguna parte de la tarea general.

11.3.1 Estado de dependencia en Procesadores de Transacción

Un caso especial se tiene cuando el procesamiento de transacción incorpora un proceso de decisión estado-dependiente, esto ocurre cuando la salida de una determinada decisión depende no solo de los datos de entrada, si no además de los valores ingresados anteriormente. Por ejemplo una transacción tipo X podrá ser procesada como tipo X o tipo Y, dependiendo de que una transacción del tipo A haya sido procesada correctamente antes.

11.3.2 Procesamiento de Transacciones Sintáctico y Semántico

Entenderemos por elementos ***sintácticos***, aquellos aspectos de procesamiento relacionados a la ***forma*** que toma la transacción. Entenderemos por ***semántico,*** las ***acciones*** resultantes: el "que" y el "como".



Validando el formato de la transacción proveniente de la rama aferente (Obtener y Analizar), y convirtiéndola a un código interno, el resto del sistema Transacción, Despachar, y los superiores a Transacción, pueden ser escritos para que operen independientemente de la forma que la transacción tome. De esta manera es sencillo cambiar la apariencia y el procesamiento de la transacción de modo independiente uno de otro. En adición, los módulos de procesamiento de transacción, pueden ser usados correctamente con transacciones obtenidas en otros formatos desde fuentes completamente diferentes.

11.3.3 Efecto de colocar centros de transacción a diferentes niveles en la jerarquía

En algunos casos, el diseñador se encuentra diseñando un sistema con solo un centro de transacción, pero con varias opciones concernientes a la ubicación del centro de transacción dentro de la jerarquía. Es posible, y a veces tentador, ubicar el centro de transacción en el módulo ejecutivo de máximo nivel. Esto es generalmente una idea pobre, por razones de acoplamiento y cohesión. Entonces, dónde debe ubicarse el centro de transacción? Finalmente, acoplamiento y cohesión son los mejores criterios para decidir donde ubicarlo.
De cualquier modo, existe un aspecto filosófico de esta decisión que el diseñador debe tener en mente: ubicar el centro de transacción alto en la jerarquía, refleja la decisión del diseñador de permitir que el entorno (aquello que existe fuera del sistema de computadora) controle al sistema. Contrariamente, ubicar el centro de transacción en los niveles bajos de la jerarquía, refleja el deseo del diseñador de que el sistema control el entorno.
Por qué esto es así? Debe recordarse que el centro de transacción es un punto a partir del cual distintos tipos de procesamientos tendrán lugar, dependiendo de la naturaleza precisa de un elemento de datos. Por lo tanto, es el centro de transacción (el módulo de nivel P) está en el tope de la jerarquía, es análogo a que el presidente de una compañía

diga: "No conozco a qué tipo de situaciones se enfrentará la corporación en los próximos milisegundos, pero yo quiero responder apropiadamente".
Si el centro de transacción se ubica cerca del nivel inferior de la jerarquía, entonces el tope de la jerarquía se organizará de modo acorde a las recomendaciones del método de Análisis de Transformación. En este caso, el módulo ejecutivo es análogo a un administrador que conoce precisamente que datos espera de sus subordinados, y exactamente qué hacer con dichos datos. Esto es más aproximado al caso en que el administrador controla el entorno, más que el caso en que el entorno lo controle a él.
Algunos diseñadores piensan que la cuestión del control sobre el entorno es meramente un reflejo de opción entre un sistema batch o un sistema en-línea o en tiempo real. Esto NO es así. Un sistema en línea puede tener sus centros de transacción cerca del tope y en el nivel inferior de la jerarquía. Si es ubicado cerca del tope de la jerarquía, refleja el deseo del diseñador de hacerlo tan interactivo como sea posible. Esto es como decir en un sistema en línea: "no tengo idea de qué es lo que el usuario va a tipear en la terminal, pero voy a realizar sus comandos". Un sistema en línea que tenga su centro de transacción cerca del nivel bajo de la jerarquía, refleja el deseo del diseñador de guiar al usuario de terminal a través de un dialogo ordenado para que realice lo que el ***sistema*** quiere que realice. Los módulos de nivel superior de este tipo de sistema forzarán al usuario a que suministre las entradas que el sistema desea.
De modo similar los sistemas batch (en lote) pueden tener el centro de transacción alto o bajo en la jerarquía según la filosofía del diseñador.

Si emanar juicios de valoración sobre que filosofía utilice el diseñador, diremos que finalmente los principios de acoplamiento y cohesión serán los árbitros finales de lo bueno o malo que sea el sistema.

Un buen ejemplo de un sistema en línea en el cual el entorno controla al sistema son los monitores de teleproceso en los sistemas mainframe, o los shell de comandos en Unix.

Unidad 10: Estrategias de diseño Alternativas

10.0 Introducción

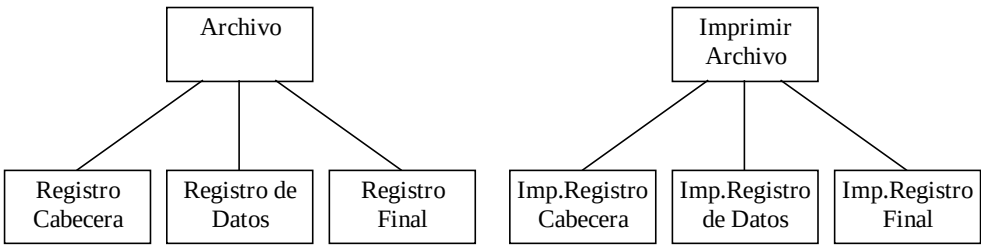
Como se ha visto, el diseño de sistemas puede derivarse de una manera metodológica, analizando el diagrama de flujo de datos asociado con el problema. Dependiendo de la naturaleza de la aplicación, el diseño centrado en la transformación, o el diseño centrado en la transacción nos llevarán a una estructura modular altamente cohesiva y con bajo acoplamiento.
Sin embargo, estas dos estrategias no son la únicas maneras de derivar buenos diseños de modo sistemático. Otros investigadores tales como Michel Jackson, Jean Dominique Warnier, y Davis Parnas, han desarrollado otras estrategias que analizaremos brevemente a continuación.

10.1 Método de Diseño basado en la Estructura de Datos

Una estrategia de diseño popular se basa en el análisis de las estructuras de datos, en lugar del análisis de los flujos de datos. Esta ha sido desarrollada por Michel Jackson y J.D.Warnier.
Esta estrategia se sumariza en las siguientes actividades:

1) Definir las estructuras de datos a ser procesadas
2) Derivar la estructura del programa basado en la estructura de datos
3) Definir la tarea a ser realizada en términos de operaciones elementales, y ubicar cada una de dichas operaciones en componentes de la estructura del programa.

Implícitamente en el enfoque de la estructura de datos, está presente el hecho de que la mayoría de las aplicaciones de EDP trabajan con jerarquías de datos, como por ejemplo: campos dentro de registros de archivos. Por lo tanto, este enfoque desarrolla una jerarquía de módulos, que en un sentido, es una imagen espejo de la jerarquía de datos asociados con el problema. La siguiente figura representa la estructura de un archivo secuencial simple y la estructura de un programa que imprime dicho archivo.



Estructura del archivo *Estructura del programa*

Es común que las aplicaciones EDP (Electronic Data Processing) involucren más de un conjunto de datos. Desafortunadamente, los conjuntos de datos, muchas veces tienen diferentes estructuras. Jackson enfatiza que si una aplicación se implementará con un solo programa, es decir una estructura jerárquica simple, debe haber un mapeo directo

uno-a-uno, *en todos los niveles de la jerarquía*, entre los elementos de datos de *cada* conjunto de datos y los módulos que son responsables del procesamiento de dichos elementos.
Por ejemplo, la figura (a) muestra las estructuras de datos para una aplicación que mezcla datos financieros y personales de empleados. La figura (b) muestra la estructura del archivo de salida con los datos compuestos de cada empleado. La figura (c) muestra la jerarquía de módulos que realizarán la operación deseada. Debe notarse la correspondencia uno-a-uno entre los módulos del diagrama de estructura y los elementos de la estructura de datos.

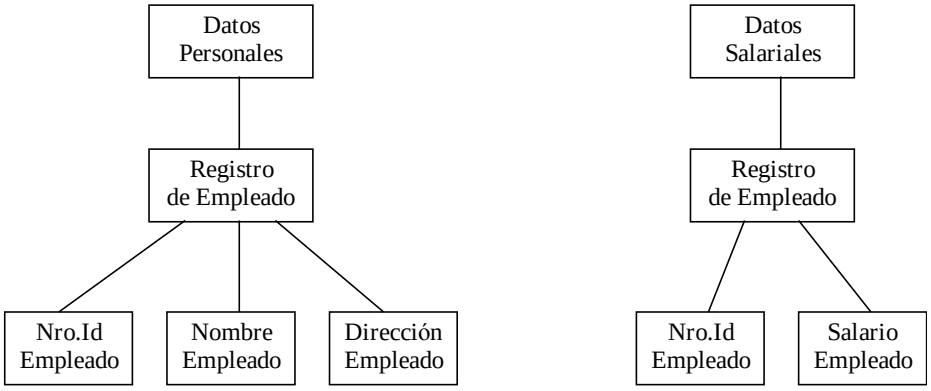


Fig. (a)

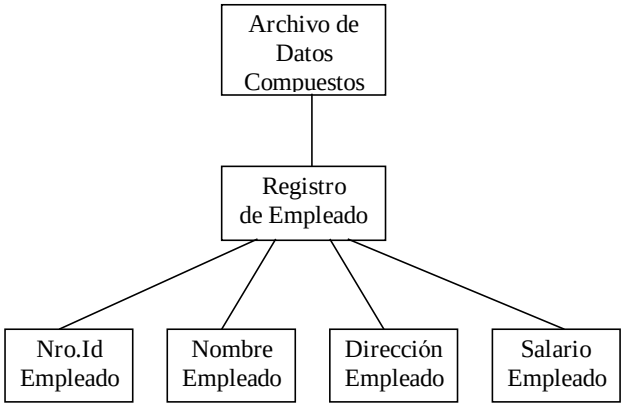


Fig. (b)

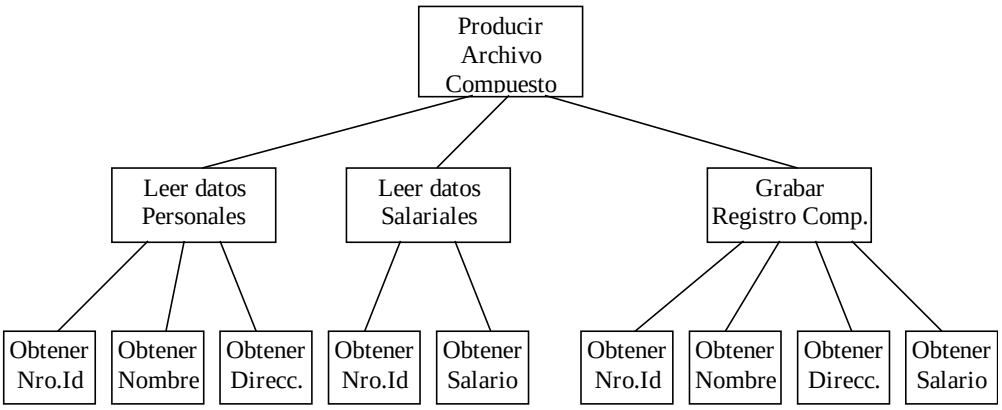


Fig. (c)

Ahora bien, si no puede realizarse la correspondencia uno-a-uno entre los elementos de la estructura de datos, entonces se produce un *choque de estructuras*. Este fenómeno es una parte importante en el enfoque de estructura de datos. En términos prácticos significa que una aplicación no puede ser implementada de forma natural con una jerarquía simple de módulos. En su lugar, Jackson propone un enfoque (conocido como *inversión de programa*) que involucra múltiples programas (o más precisamente, múltiples jerarquías de módulos). Como ejemplo supongamos que debemos diseñar un programa que emita un reporte. El mismo tiene como entrada un solo archivo y como salida un único reporte, pero la estructura del archivo es totalmente incompatible con la estructura del reporte. La solución, según Jackson, involucrará dos programas. Uno que descompone el archivo de entrada en piezas elementales de datos, y otro que recombina dichas piezas elementales en una forma compatible con la estructura requerida por el reporte de salida.

Un ejemplo común de choque de estructura puede ser llamado *"choque de ordenamientos"*. Esto sucede por ejemplo cuando deben procesarse dos archivos donde existe correspondencia entre los valores de sus registros pero el ordenamiento de ambos no coincide. La solución obvia es clasificar uno de los archivos según el orden del otro. Necesitamos entonces dos programas para implementar el programa en forma natural.

Otro tipo común de choque de estructuras se conoce como *"choque de límites"*, y usualmente es causado por las características de bloqueo de los dispositivos físicos de entrada-salida. Supongamos que tenemos que actualizar un archivo maestro, y las transacciones de actualización son leídas en bloques de 100 caracteres, los cuales no se corresponden con la estructura del archivo. La solución pasará por tener un programa que descomponga los bloques de caracteres en las transacciones lógicas necesarias para el programa de actualización.

Como comentario final, podemos decir que el enfoque basado en la estructura de datos puede aplicarse con éxito en problemas pequeños, pero puede presentar inconvenientes en problemas grandes donde deben tratarse múltiples estructuras de datos con varios choques de estructuras. Sin embargo es un método que puede utilizarse en ocasiones combinado con las estrategias del análisis de transformación y de transacción, y siguiendo los principios fundamentales del diseño como cohesión y acoplamiento.

10.2 Criterio de Descomposición de Parnas

Otro enfoque de diseño modular interesante es descrito por D.Parnas como un conjunto de reglas para la descomposición de sistemas en subsistemas.
Parnas propuso una serie de recomendaciones para la descomposición de un problema total en *unidades de diseño*, o porciones del problema a diseñar. Las unidades de diseño se relacionan mediante *interfaces de diseño* (Parnas utiliza el término conexiones) las cuales son *cualquier* tipo de interrelaciones o interdependencias. Las unidades de diseño y sus interfaces de diseño pueden tener o no relaciones con los actuales módulos y conexiones.
Parnas sugiere que "una subrutina o programa sea una colección ensamblada de código de varias unidades de diseño".
El criterio de descomposición de Parnas puede sintetizarse de la siguiente manera:

1) La descomposición NO debe basarse en procedimientos o diagramas de flujos.
2) Cada interface de diseño debe tener la mínima y suficiente información para especificarla correctamente.
3) Cada unidad de diseño "esconde" una solución.
4) Cada unidad de diseño debe ser especificada a otras unidades de diseño (o programadores de otras unidades de diseño) solo con el nivel de detalle necesario y suficiente (no demasiado excesivo, ni poco).

El primer criterio se relaciona con el principio de cohesión, mientras los restantes con el de acoplamiento.

El criterio de descomposición de Parnas no constituye en sí una metodología general para el diseño estructural de sistemas, sin embargo, su idea subyacente puede ser de gran utilidad si se lo utiliza en conjunción con las metodologías de diseño estructurado.

PRAGMATISMOS

Unidad 11: Empaquetado

11.0 Introducción

Trataremos ahora dos pasos prácticos en el diseño de un sistema modular funcional. Finalmente, debemos lograr que el sistema "entre" en el espacio de memoria física disponible, y por otro lado deben implementarse los procesos de entrada-salida en los dispositivos físicos actuales. Estos dos pasos conciernen a la implementación física del sistema en el recurso computacional que se dispone.

El *empaquetado* se refiere a la asignación de módulos del sistema en secciones manejadas como distintas unidades físicas de ejecución sobre la máquina.
Cada una de estas unidades se llaman *unidades de carga*, y serán consideradas como una porción del sistema procesadas como una unidad por el sistema operativo.

Dependiendo del sistema computacional, las unidades de carga toman diferentes denominaciones: "programas", "overlays", "módulo de carga", "paso de trabajo" (job step) , etc.
Históricamente las limitaciones de memoria y velocidad de ejecución, condicionaron el diseño. Esto producía como consecuencia una disminución de la modularidad del sistema.
Como regla general, no podemos minimizar simultáneamente tamaño de memoria y tiempo de ejecución. El objetivo es obtener un arreglo de empaquetados que minimice el tiempo de ejecución del programa satisfaciendo las limitaciones de espacio.
El empaquetado de esta manera podrá ser realizado luego que la estructura modular completa haya sido diseñada.
Difiriendo el empaquetado hasta el final del proceso de diseño, podemos mejorar la eficiencia y la modularidad técnica del sistema.

Utilizaremos una estrategia conocida como *análisis procedural*, para estudiar el problema de organización del sistema en unidades de carga eficientes.

11.1 Análisis Procedural

El *análisis procedural* consiste de un conjunto de criterios para determinar que módulos deben pertenecer a la misma unidad de carga en por de maximizar la eficiencia.
No existen reglas fijas para realizar el empaquetado de módulos en unidades de carga, y será el criterio del diseñador el que deberá determinar como se compongan las unidades de carga.
El análisis procedural involucra tres pasos fundamentales:

- Determinar el tamaño aproximado de cada módulo en la estructura.
- Aplicar alguno de los criterios que se discuten más abajo para determinar agrupamientos preferidos y prioridades entre preferencias.
- Encontrar agrupamientos de módulos tales que quepan dentro del tamaño de la unidad de carga, minimizando la cantidad de agrupamientos que deban partirse en más de una unidad de carga.

11.1.1 Criterios de Agrupamiento

El concepto general es muy simple: deben incluirse en una misma unidad de carga aquellos módulos que se conectarán muchas veces durante la ejecución del sistema.

Iteraciones
Deben incluirse en la misma unidad de carga módulos conectados por referencias iterativas.
Este criterio se aplica a las estructuras iterativas o loops. La regla de pensamiento es que, siempre que sea posible, debe incluirse en la misma unidad de carga al módulo referenciado y al referenciante.

Volumen
Deben incluirse en la misma unidad de carga módulos con alto volumen de referencias de conexión entre ellos.
Para esto debe estimarse la cantidad de veces que un módulo invocará a otro durante la ejecución. Si el volumen de veces que esta invocación se producirá es alto, es conveniente entonces que los módulos pertenezcan a la misma unidad de carga.

Frecuencia
Cuando la frecuencia de referencia es un criterio observable, la regla de pensamiento es que los módulos relacionados por referencias altamente frecuentes deben ubicarse en al misma unidad de carga.
Debe observarse que las invocaciones condicionales reducen la frecuencia de invocación.

Intervalo
Deben incluirse en la misma unidad de carga los módulos con intervalos breves de tiempo entre activaciones.

11.1.2 Criterios de Aislación

Podemos observar además que existen situaciones en las que será deseable que dos módulos no pertenezcan a la misma unidad de carga. Los siguientes criterios se verifican en tales circunstancias:

Funciones Opcionales
Debe ubicarse en una unidad de carga por separado cualquier función opcional.

Funciones por Unica Vez
Deben ubicarse en una unidad de carga por separado las funciones que se utilicen una única vez en el sistema.

Sorts
Deben ubicarse los módulos aplicados a la entrada y a la salida de un proceso de sort en unidades de carga separadas.

Unidad 12: Optimización de Sistemas Modulares

12.0 Introducción

Entenderemos por optimización las modificaciones que se realicen al sistema en pos de maximizar la eficiencia del mismo en cuanto el uso de recursos computacionales.

12.1 Criterios de Optimización

Los criterios o creencias más comunes que encontramos acerca de la optimización de programas son:

- La eficiencia de un sistema depende de la Competencia del Diseñador
- En muchos casos, el camino más simple es el más eficiente
- Solo una pequeña parte de un sistema típico tiene impacto sobre la eficiencia general
- Sistemas Modulares Simples pueden ser optimizados fácilmente
- El sobre énfasis en la optimización va en detrimento de otros objetivos del diseño
- La optimización es irrelevante si el programa no funciona, entiéndase por esto no hacer lo que debería hacer.

12.2 Un Enfoque para la Optimización de Módulos

Existen dos enfoques para optimizar un sistema:

- optimizar el código dentro de los módulos
- cambiar la estructura del sistema para mejorar su performance

Las técnicas para optimizar código son técnicas específicas de programación fuera del alcance de esta discusión, y dependen en la mayoría de los casos de los lenguajes, compiladores, y entornos de desarrollo que se utilicen.
Sin embargo puede plantearse un esquema general de ataque para optimizar módulos. En primer lugar debe tenerse en cuenta que por lo general, no *todos* los módulos de un sistema deberán ser optimizados. Por el contrario, se conoce que mejorando un 5% de módulos críticos que insumen la mayor parte del tiempo de un sistema, se puede mejorar la performance global del mismo. El esquema general de optimización implica los siguientes pasos:

1) *Determinar el tiempo de ejecución para cada módulo o unidad de carga.* Para esto se utilizan monitores de hardware y paquetes para la medición de performance. Por simplicidad denotaremos como **i** al tiempo de ejecución del módulo **Ti**.

2) *Examinar cada módulo para estimar su potencial mejora*. Este paso implica un proceso de estimación y depende del conocimiento y la experiencia del programador en el lenguaje, sistema operativo y hardware. Denotaremos este potencial de mejora como **Ii**.

3) *Estimar el costo que involucra cada mejora.* Por costo entenderemos, salario del programador o analista, tiempo de computador, y otros costos involucrados en la producción de una versión optimizada del sistema. Denotaremos este costo como **Ci**.

4) *Establecer prioridades para realizar las mejoras*. Podremos establecer esta prioridad Pi con la siguiente expresión: ***Pi = A * Ii * Ti - B * Ci,*** donde A y B son factores de peso apropiado.

5) *Optimizar los módulos con mayor prioridad.*

12.3 Cambios estructurales por Eficiencia

En un determinado número de casos, la optimización del código dentro de los límites de un módulo puede no ser suficiente para alcanzar el nivel de eficiencia deseado. Será necesario entonces, modificar la estructura del sistema. Antes de realizar este tipo de optimización, es importante que el diseñador identifique las fuentes o causas de la ineficiencia. Es importante que el diseñador obtenga estadísticas concernientes a los tiempos de transición intermodulares, como así de los tiempos de ejecución intramodulares.
Existe solo un pequeño número de modificaciones estructurales que pueden mejorar notablemente la velocidad de ejecución y quizá los requerimientos de memoria. Discutiremos estas en los párrafos siguientes.

12.3.1 Macros o código incluido lexicográficamente

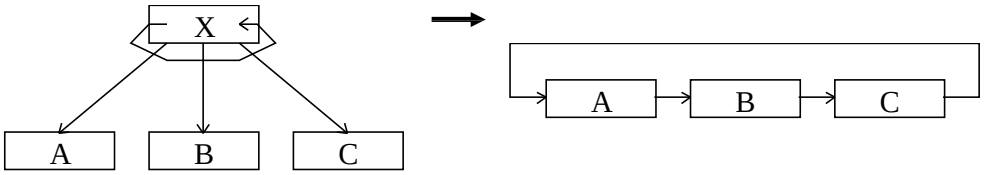
Antes de realizar modificaciones en la estructura actual, el diseñador de considerar la posibilidad de cambiar el *tipo* de módulo preservando la estructura.
En particular, el diseñador debe recordar que las macros (rutinas lexicográficamente incluidas en el cuerpo de su módulo superior) y subrutinas representan *compromisos* entre tiempo de ejecución y requerimientos de memoria. La "transmisión" de parámetros a una macro es realizada durante el tiempo de compilación o ensamble, y el cambio de contexto (conocido también como prólogo/epílogo, o salvado y restaurado del estado de la máquina) puede ser optimizado por el compilador/ensamblador. También, debido a que el cuerpo de la macro se incluye lexicográficamente en la secuencia de código de su superior, cualquier optimización de los registros de hardware, realizada por el compilador/ensamblador, se aplicará a través de los límites de la macro.
Finalmente, si el módulo es referenciado solo una vez, o un pequeño número de veces dentro de la estructura, o bien si el tamaño del cuerpo del módulo es pequeño comparado con el tamaño de prólogo/epílogo para la carga del mismo, el uso de macros, probablemente ahorrará tiempo de ejecución y espacio de memoria. La transmisión de parámetros, el cambio de contexto, y otros tipos de sobrecargas producidas por las invocaciones intermodulares, consumen memoria y tiempo de CPU.

12.3.2 Estructuras panqueque

Como regla general se tiene que las estructuras muy profundas (muchos niveles) tienen una gran sobrecarga debido a los procedimientos de invocación intermodulares. Sin embargo existen numerosas excepciones a esta regla que no siempre puede ser aplicada con seguridad. Solo si luego de un análisis pertinente se detecta que "aplastando" la

estructura se puede obtener una mejora substancial, se deben aplicar técnicas en pos de lograrlo.

Consideremos una estructura de dos niveles consistentes de un módulo de control y sus subordinados. Cada invocación a dichos subordinados, representan una sobrecarga. Si dichas invocaciones se encuentra dentro de un proceso iterativo, la sobrecarga puede llegar a ser considerable. Si la lógica de control del módulo superior no es compleja, una estructura de este tipo puede convertirse en un nivel homólogo vinculado por transferencias de control incondicionales.
La estructura "panqueque" resultante siempre tendrá menor cohesión y mayor acoplamiento. Esta técnica generalmente funciona con lenguajes "antiguos", debido a que las transferencias de control incondicionales se implementan en ellos con gran eficiencia. Sin embargo la mayoría de los lenguajes de programación "modernos" no permiten transferencias incondicionales, por lo cual esta técnica es inaplicable en dichos casos.



12.3.3 Compresión

La más ubicua de todas las manipulaciones estructurales para mejorar la eficiencia puede ser llamada "demodularización", ya que esta consiste en comprimir todo o parte de un módulo en otro. En el caso más simple, esto se realiza a través de la inclusión lexicográfica de un módulo en otro. La ganancia en eficiencia en esta maniobra simple es equivalente a la diferencia de sobrecarga entre la llamada a un subordinado lexicográficamente incluido y una llamada a un subordinado externo.
El código del subordinado puede embeberse en el código del superior quitándole los elementos de límite y los elementos de ligado, o simulándolos. Si el cuerpo del subordinado es copiado en-línea en cada llamada dentro del superior, entonces se incrementarán los requerimientos de memoria, y el análisis es similar la realizado con las macros.
El efecto de realizar la compresión antes o después que el sistema se implemente es importante. Por lo general si la compresión se realiza previo a la implementación, es posible que el módulo resultante de la compresión oculte para siempre la funcionalidad independiente del módulo que se comprimió, y cada vez que se necesite un función similar, esta sea reescrita. Por el contrario cuando se pospone la compresión hasta el momento de la implementación, se mantiene la estructura intacta durante el diseño, lo cual permite la reutilización de los módulos involucrados.

12.3.4 Otras

Otras técnicas a considerar para maximizar la eficiencia de ejecución de un programa incluyen:

- Cambio técnicas de comunicación (globalizar variables locales)
- Recodificación