Introducción al Diseño de Sistemas de Información

Unidad Nº III: Diseño Estructurado

Facultad Regional Santa Fe Universidad Tecnológica Nacional

Diseño Estructurado

Como vimos, el Diseño de Software es :

> "*Proceso mediante el que se traducen los requisitos en una representación del software*" (Robert Pressman)

El objetivo más importante es :

- entregar las funciones requeridas por el usuario en una implementación de software (satisfaga una especificación funcional dada).

El diseño es un proceso mediante el cual se traducen los requisitos del sistema en una representación de software. A través de refinamientos sucesivos se genera una representación de diseño que se acerca mucho al código fuente.
El diseño se realiza, generalmente, en dos pasos :

- *Diseño preliminar*, el cual se centra en la transformación de los requisitos en los datos y la arquitectura del software.
- *Diseño detallado*, el que se ocupa del refinamiento de la representación arquitectónica que lleva a una estructura de datos detallada y a las representaciones algorítmicas del software.

El Diseño Estructurado se fundamenta en una *descomposición funcional sistematizada*, la que se lleva a cabo de acuerdo a las siguientes actividades/fases:

1. Representar el sistema como un Diagrama de Flujos de Datos (DFD).
2. Estructurar el sistema como *jerarquías de procesos* (utilizando Diagramas Estructurados).
   Análisis transformacional.
   Análisis transaccional.
3. Verificar proyecto y reestructurar.
   La verificación se realiza analizado si están correctamente aplicados los conceptos de:
   Cohesión
   Acoplamiento
   Alcance de efectos/control
   Tamaño de la interfaces
   Balance del diseño
   Etc.
4. Descomposición de procesos
   Refinamiento sucesivo
   Factorización
5. Preparar para la implementación.

El Diseño Estructurado (DE) se fundamenta en una *descomposición funcional* sistematizada, la que se lleva a cabo convirtiendo sistemáticamente DFDs en *Diagramas Estructurados* (DE, 'Structure Charts')
Para realizar estar transformación podemos aplicar estrategias ascendentes o descendentes, siendo sin embargo el diseño estructurado una técnica estrictamente 'top-down' (en el sentido de que el diseño se va realizando gradualmente por refinamiento sucesivo).

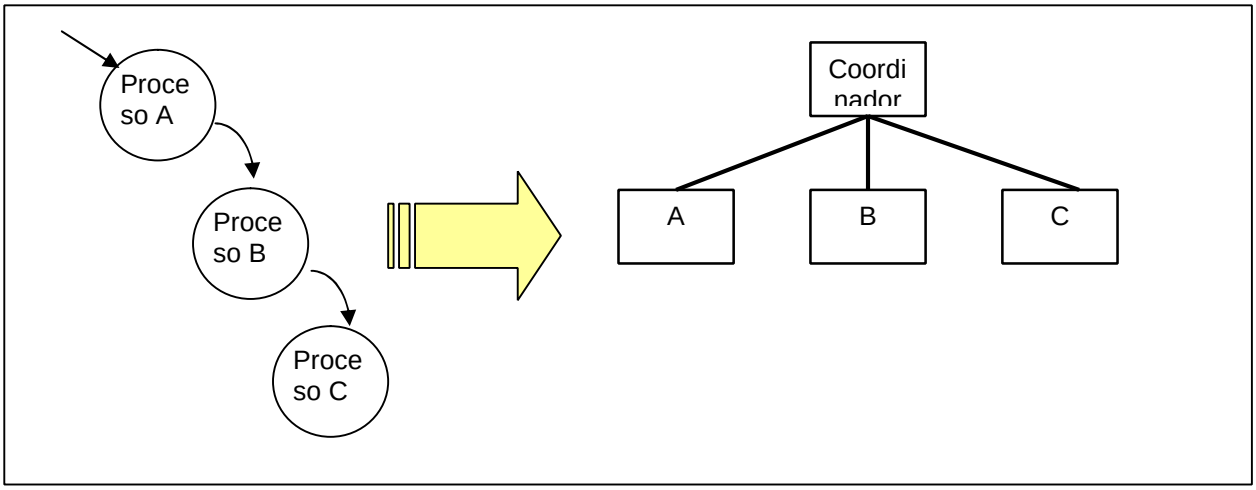

Figura III. 1 – Transformación de un DFD en un DE

Estrategias Ascendente (Bottom-Up) y Descendente (Top-Down)

Diseño ascendente (bottom-up)

El *diseño ascendente* se refiere a que la identificación de los procesos que se automatizarán se realiza partiendo del nivel más bajo hasta llegar al nivel superior.
Desde el punto de vista de una organización, se hace referencia a que los problemas que requieren una automatización inmediata se encuentran en los niveles inferiores, y por otra parte son los de menor costo de implementación.
El problema que presenta este enfoque, es que es muy difícil integrar los objetivos de los distintos subsistemas construidos en pos de un objetivo global. Esto es así debido a que cada uno fue concebido persiguiendo un objetivo particular, y en consecuencia no se previeron los mecanismos de interacción adecuados con otros subsistemas.
El principal problema de este enfoque es que no se consideran los objetivos globales de la organización, y en consecuencia no se satisfacen. Pueden presentarse además duplicación de esfuerzos para introducir y acceder a los datos; y también pueden introducirse al sistema datos carentes de valor.

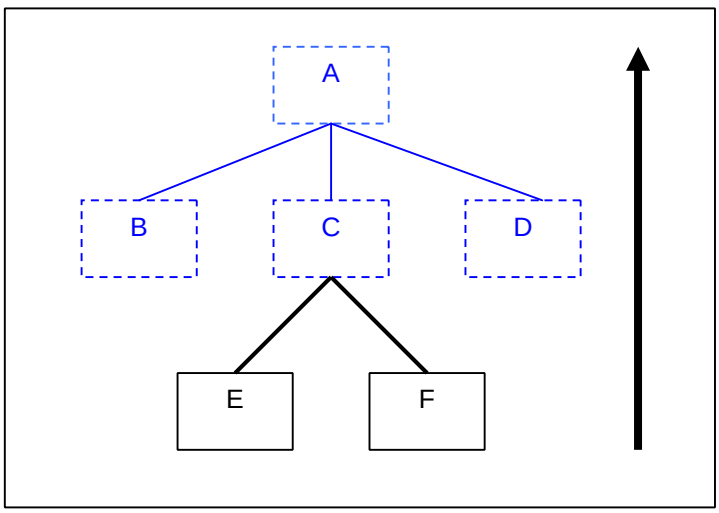

Figura III. 2 – Diseño Ascendente

Diseño descendente (top-down)

El enfoque *descendente* implica observar la gran imagen del sistema y luego, explosionarlo o desglosarlo en partes más pequeñas o subsistemas.
El *diseño descendente* obliga a enterarse primero de los objetivos globales de la organización, así como el establecimiento de la mejor manera de satisfacerlos dentro de un sistema integral.
Cuando se aplica un enfoque descendente se emplean las interrelaciones e interdependencias de los subsistemas, para apegarse lo mejor posible a las necesidades de la organización.
En este enfoque se da la importancia debida a las interfaces requeridas entre el sistema y sus subsistemas, lo cual no existe en el enfoque ascendente.
La aplicación de este enfoque posibilita contar con distintos grupos de desarrollo trabajando en forma simultanea en subsistemas independientes.
Un inconveniente que presenta este enfoque es que se divida el sistema en subsistemas 'incorrectos'. Se debe prestar atención para que la partición en subsistemas tenga sentido en el esquema global del sistema, y que se integren en forma correcta al sistema.
Debe tenerse presente que una vez que se realizan las divisiones en subsistemas, sus interfaces pueden descuidarse o simplemente ignorarse.
Debe tenerse siempre presente la retroalimentación entre los distintos subsistemas y grupos de desarrollo para mantener unificados los criterios.

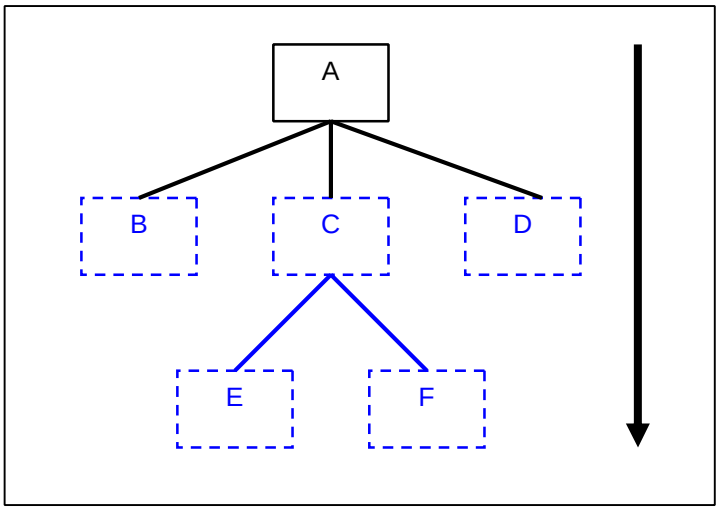

Figura III. 3 – Diseño Descendente

Diagramas Estructurados

Los Diagramas Estructurados (DE) describen una *arquitectura de programas* a través de una jerarquía de llamadas a *módulos*.
Estos DE son elaborados a partir de la descomposición funcional pura de los DFDs, y presentan una estructura de control limitada.
Como desventaja puede decirse que se tornan difíciles de leer cuando hay muchos detalles de los flujos de datos y control entre los distintos módulos.

Existe un acuerdo general en el sentido de que los sistemas más fáciles de cambiar son aquellos que están constituidos por *módulos manejablemente pequeños*, cada uno de los cuales es independiente, hasta donde es posible, de cualquier otro, de manera que pueden sacarse del sistema, cambiarse, y reponerse sin afectar el resto del sistema.

Un diseño modular reduce la complejidad, facilita los cambios, y produce como resultado una implementación más sencilla, permitiendo el desarrollo paralelo de las diferentes partes de un sistema.

Módulo lógico: es una *función* definida con un nombre que expresa el propósito de la función. Los procesos definidos en los Diagramas de Flujos de Datos pueden considerarse módulos lógicos.

Módulo físico: *implementación* particular de un módulo lógico. Generalmente está dado en términos de un grupo de sentencias en un lenguaje de programación, al cual puede hacerse referencia por un nombre.

Los módulos a menudo invocan a otra módulos, los cuales invocan a otros módulos, etc.. Esto permite formar una jerarquía jefe-subjefe, hasta alcanzar los módulos de trabajo finales.

Módulo manejablemente pequeño: una persona competente será capaz de tomar un listado del módulo, leerlo y hacerse mentalmente un cuadro de su función interna mientras decide cómo cambiarlo.

Independencia: aislar la función en la menor cantidad de módulos posibles.
Idealmente las funciones estén contenidas en cajas negras, donde la función producirá siempre resultados predecibles a partir de un conjunto de datos que pasen a través de la misma.

Los módulos que componen un sistema no pueden ser completamente *independientes* unos de otros o no existiría un sistema sino solamente un montón de módulos aislados.

Los *nombres de los módulos* siguen las mismas normas que para proceso: verbo activo y un objeto único.

Comunicación entre módulos

La estructura es de "llamada" o invocación de módulos. El módulo de nivel superior invoca a uno de nivel inferior (representado por una flecha que une al módulo llamador y el llamado).

Los módulos pueden intercambiar:

- Datos: representado por una flecha con un círculo vacío en un extremo.
- Señales: representado por una flecha con un círculo lleno en un extremo.

La tarea del diseñador es formar los módulos y diseñar sus interconexiones para minimizar la posibilidad del *efecto onda* (un cambio en un módulo repercute en otros módulos, y así).

Derivación del modelo físico a partir de un DFD

Para realizar la transición desde el flujo de información (DFD) a la estructura (DE) se debe proceder de la siguiente manera:

1. Establecer el tipo de flujo de información;
2. Determinar los límites del flujo;
3. Convertir el DFD en la estructura del programa;
4. Definir la jerarquía de control mediante factorización;
5. Refinar la estructura resultante usando medidas y heurísticas de diseño.

Similar a una estructura militar. Cada módulo tiene su propia tarea, que desempeña cuando recibe órdenes superiores; se comunica sólo con su jefe superior y con sus subordinados.
La no comunicación directa entre los módulos de trabajo de un mismo nivel o en general no respetando la jerarquía estipulada, es una forma de simplificar el acoplamiento intermodular, y facilita la comprensión del comportamiento del sistema.
El *tipo de flujo de información* es lo que determina el método de conversión requerido en el paso 3, y existen dos tipos: *flujos de transformación* y *flujos de transacción*.

Sistemas centrados en flujos de transformación

Si tenemos en mente el diagrama de flujo de datos de nivel 0 de un sistema, la información entra y sale en una forma del 'mundo exterior'. Por ejemplo, algunas formas de información del mundo exterior son las teclas pulsadas sobre un teclado de una terminal, los tonos sobre una línea telefónica y los dibujos de un monitor gráfico de computadora. Tales datos externos deben ser convertidos a una forma interna adecuada para el procesamiento. La Figura III.4

ilustra la evolución del flujo de datos a lo largo del tiempo. La información entra al sistema mediante caminos que transforman los datos externos a una forma interna y se identifican como *flujo entrante*. En el interior del software se produce una *transición*. Los datos entrantes pasan a través de un *centro de transformación*, moviéndose a lo largo de caminos que conducen ahora hacia la *salida* del software. Los datos que se mueven por estos caminos se llaman *flujo saliente*. El flujo de datos global ocurre de forma secuencial y sigue uno o unos pocos caminos directos. Cuando un segmento de un diagrama de flujo de datos exhibe estas características, lo que tenemos es un *flujo de transformación*.

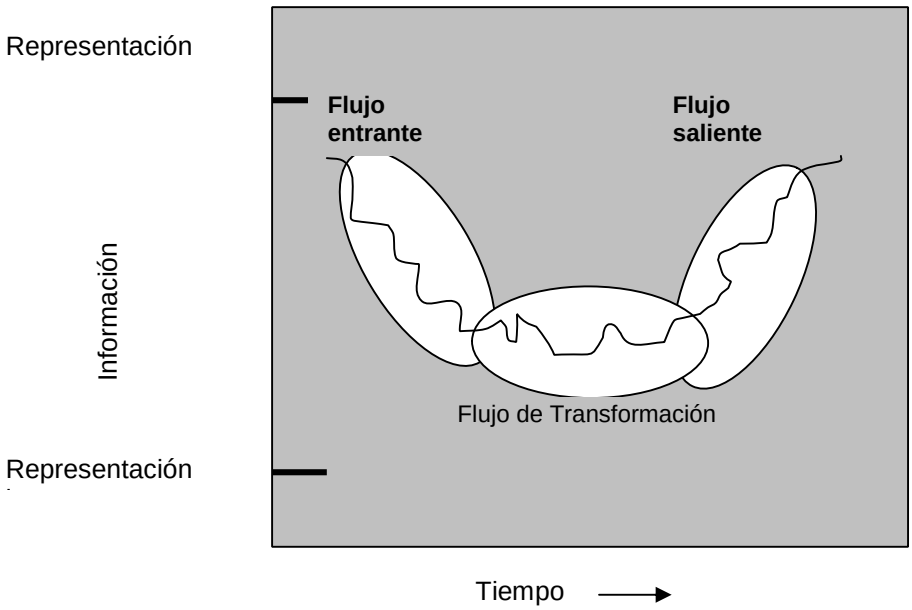

Figura III. 4 – Flujo de Información

- Un tramo de entrada maneja todas las funciones de entrada,
- Un tramo de transformación toma una entrada correcta y produce un resultado, y
- Un tramo de salida maneja toda la salida correspondiente a ese resultado.

Se derivan comúnmente de un DFD donde todas las transacciones siguen caminos iguales o similares.

Sistemas centrados en flujo de transacción

El modelo fundamental del sistema implica un flujo de transformación; por lo tanto, todo flujo de datos se puede clasificar en esa categoría. Sin embargo, a menudo, el flujo de información está caracterizado por un único elemento de datos, denominado *transacción*, que desencadena otro flujo de datos a través de uno de entre varios caminos. Cuando un DFD toma la forma que muestra la Figura III.5, lo que tenemos es un *flujo de transacción*.

El flujo de transacción se caracteriza por el movimiento de datos a través de un camino de llegada (también denominado *camino de recepción*) que convierte la información del mundo exterior en una transacción. Se evalúa la transacción y, de acuerdo con su valor, el flujo sigue por uno de los muchos *caminos de acción*. El centro de flujo de información desde el que emanan los caminos de acción se denomina *centro de transacción*.

Hay que tener en cuenta que en el DFD de un gran sistema, pueden estar presentes los dos tipos de flujos, de transformación y de transacción. Por ejemplo, dentro de un flujo de transacción, el flujo de información a través de un camino de acción puede tener características de flujo de transformación.

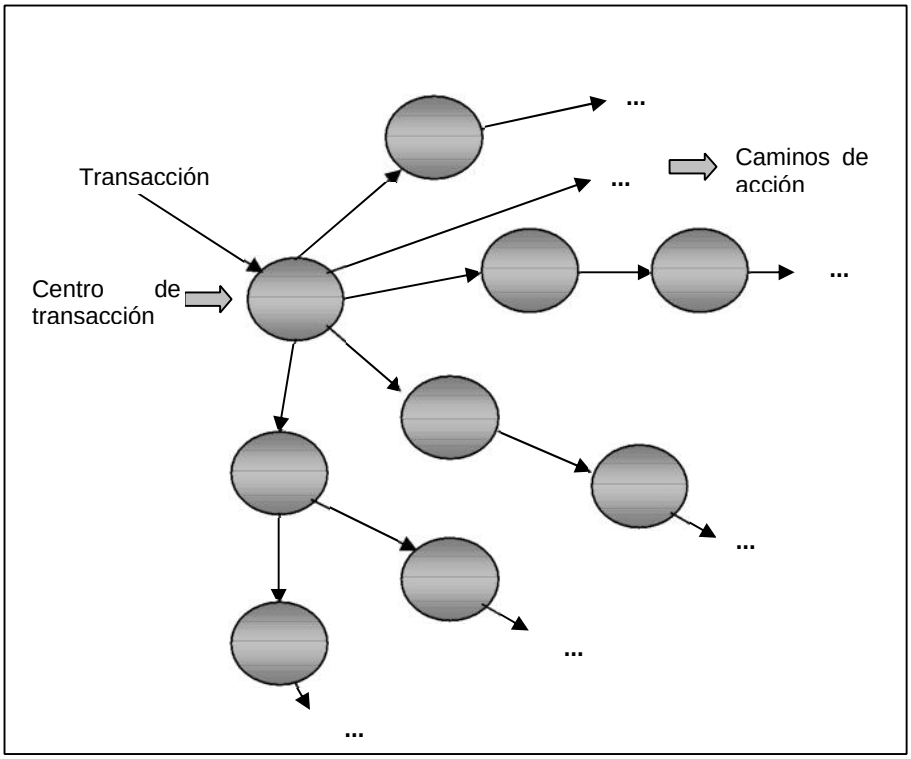

Figura III. 5 – Flujo de transacción

Pasaje de DFD a un Diagrama de Estructura (DE) jerárquica centrada en transformación

- Se determina la forma más 'bruta' de entrada, y se rastrea a través del flujo de datos hasta alcanzar un punto donde no se puede decir que es una entrada. (Entrada)
- Se toma la salida final y se rastrea hacia atrás dentro del sistema hasta donde no se pueda concebir más como una salida. (Salida)
- La porción del DFD que se encuentra entre el límite de la entrada y el límite de la salida es el centro de transformación. (Transformación)

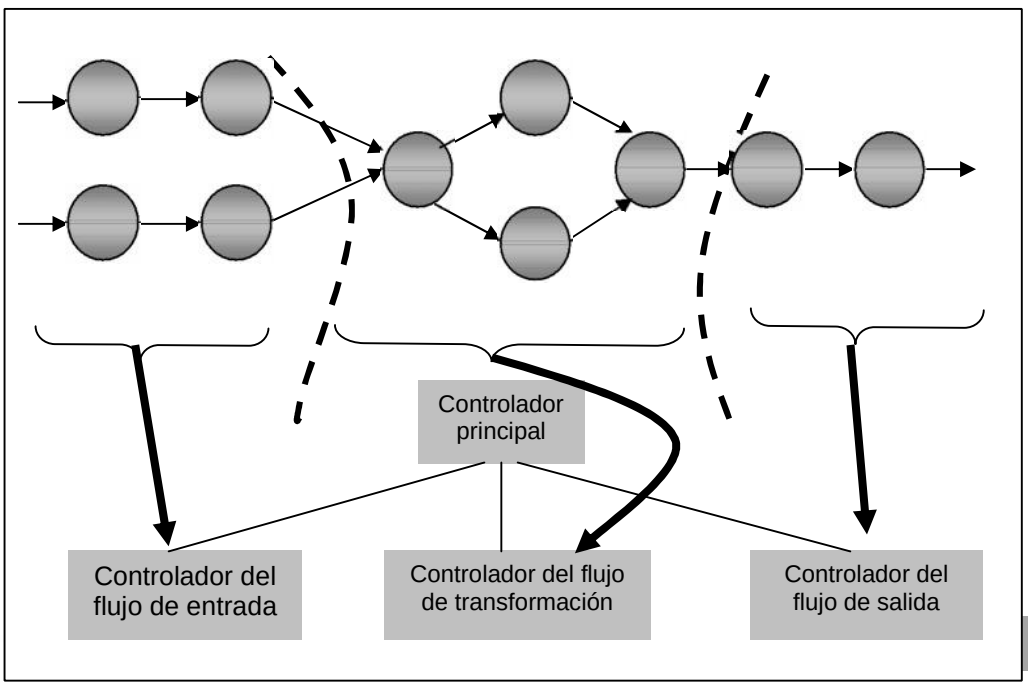

Figura III. 6 – DFD a DE centrado en transformación

Pasaje de DFD a un Diagrama de Estructura (DE) jerárquica centrada en transacción

- Se rastrea la forma más bruta de entrada y se rastrea a través del flujo de datos hasta alcanzar un punto donde ingresa a un "analizador" de la entrada.
- Se identifican los diversos caminos (transacciones) que puede tomar la entrada analizada.
- Se identifica el proceso "unificador" de las transacciones
- Se crean los módulos analizador (determina qué tipo de entrada), despachador (determina el camino a seguir por cada entrada), y unificador (realiza el proceso común para las transacciones).

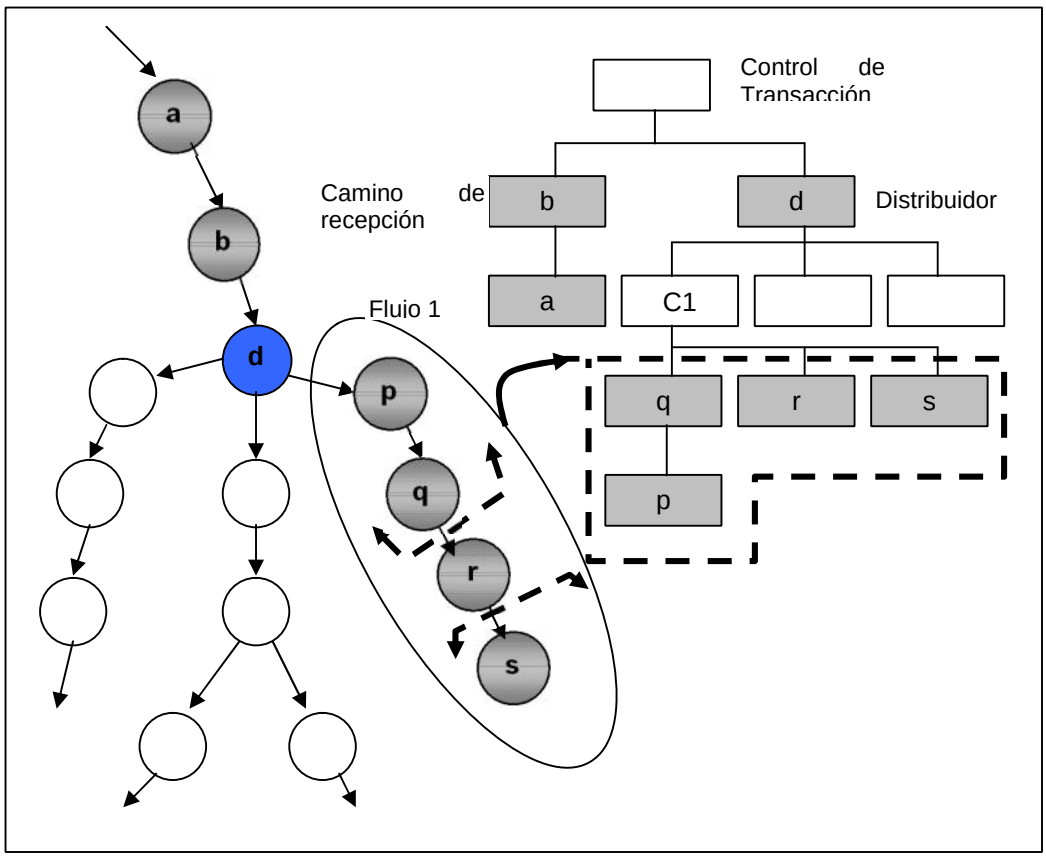

Figura III. 7 – DFD a DE centrado en transacción

La mayoría de los sistemas implican combinaciones de las dos estructuras.

DISEÑO MODULAR EFECTIVO

Los fundamentos de diseño descriptos anteriormente sirven todos para incentivar los diseños modulares. De hecho, la *modularidad* se ha convertido en un enfoque aceptado en todas las disciplinas de ingeniería. Un diseño modular reduce la complejidad, facilita los cambios (un aspecto crítico de la facilidad de mantenimiento del software) y produce como resultado una implementación más sencilla, permitiendo el desarrollo paralelo de las diferentes partes de un sistema.

Tipos de módulos

Para definir módulos en una arquitectura de software, se utiliza la *abstracción* y el *ocultamiento de información*. Ambos atributos deben ser traducidos a características operativas del módulo, caracterizadas por el *historial de incorporación*, el *mecanismo de activación* y el *camino de control*.

El *historial de incorporación* se refiere al momento en el que se incluye el módulo en la descripción del software en lenguaje fuente. Por ejemplo, un módulo definido como *macro de tiempo de compilación* es incluido por el compilador, mediante la inserción de su código, al encontrar una referencia en el código creada por el desarrollador. Un subprograma convencional (p. ej.: una subrutina o un procedimiento) es incluido mediante la generación un código de bifurcación y enlace.

Existen dos *mecanismos de activación.* Convencionalmente, un módulo es invocado mediante *referencia* (p. ej.: una sentencia "de llamada", 'CALL', 'RUN'). Sin embargo, en las aplicaciones de tiempo real, un módulo puede ser invocado mediante una *interrupción;* esto es, un suceso exterior produce una discontinuidad en el procesamiento, que da como resultado el paso del control a otro módulo. Los mecanismos de activación son importantes porque pueden afectar a la estructura del programa.

El camino de control de un módulo describe la forma en la que se ejecuta internamente. Los módulos convencionales tienen una única entrada y una única salida y ejecutan secuencialmente una tarea. Algunas veces se necesitan caminos de control más sofisticados. Por ejemplo, un módulo puede ser *reentrante.* Esto es, un módulo se diseña de forma que de ninguna manera pueda modificarse a sí mismo o a las direcciones que referencia localmente. Así, el módulo puede ser usado para más de una tarea concurrentemente.

Dentro de una estructura de programa, un módulo puede ser clasificado como

- Un módulo *secuencial* que se referencia y se ejecuta sin interrupción aparente por parte del software de la aplicación.
- Un módulo *incremental* que puede ser interrumpido, antes de que termine, por el software de la aplicación y, posteriormente, restablecida su ejecución en el punto en que se interrumpió.
- Un módulo *paralelo* que se ejecuta a la vez que otro módulo, en entornos de multiprocesadores concurrentes.

Los módulos secuenciales son los que se encuentran más frecuentemente y están caracterizados como macros en tiempo de compilación y como subprogramas convencionales - subrutinas, funciones o procedimientos. Los módulos incrementales, denominados frecuentemente *corrutinas,* mantienen un puntero de entrada que permite volver a ejecutar el módulo desde el punto de interrupción. Dichos módulos son extremadamente útiles en sistemas conducidos por interrupciones. Los módulos paralelos, denominados algunas veces *conrutinas,* se encuentran aplicaciones de cálculo de alta velocidad (p. ej.: procesamiento distribuido) que necesitan dos o más CPUs trabajando en paralelo. Cuando se utilizan corrutinas o conrutinas, puede que la jerarquía de control no sea típica. Las estructuras no jerárquicas u *homólogas* requieren unos métodos de diseño especiales.

Independencia funcional

El concepto de *independencia funcional* es una derivación directa del de modularidad y de los conceptos de abstracción y ocultamiento de información. En conocidos artículos sobre diseño de software, Parnas y Wirth aluden a las técnicas de refinamiento que aumentan la independencia modular. Posteriores artículos de Stevens, Myers y Constantine asentaron el concepto. La *independencia funcional* se adquiere desarrollando módulos con "una clara" función y una "aversión" a una excesiva interacción con otros módulos.

Dicho de otra forma, se trata de diseñar software de forma que cada módulo se centre en una subfunción específica de los requisitos y tenga una interfaz sencilla, cuando se ve desde otras partes de la estructura del software.

Es justo preguntar por qué es importante la *independencia*. El software con *modularidad efectiva,* es decir, con módulos independientes, es fácil de desarrollar porque su función puede ser partida y se simplifican las interfaces (considérense las aplicaciones cuando el desarrollo es realizado por un equipo). Los módulos independientes son más fáciles de mantener (y de probar) debido a que se limitan los efectos secundarios producidos por las modificaciones en el

diseño del código, se reduce la propagación de errores y se fomenta la reutilización de los módulos. Resumiendo,

la independencia funcional es la clave de un buen diseño y el diseño es la clave de la calidad del software.

La independencia se mide con dos criterios cualitativos: la *cohesión* y el *acoplamiento*.

La *cohesión* es una medida de la fortaleza funcional relativa de un módulo.

El *acoplamiento* es una medida de la interdependencia relativa entre los módulos.

Cohesión

La *cohesión* es una extensión del concepto de ocultamiento de información. Un módulo cohesivo ejecuta una tarea sencilla de un procedimiento de software y requiere poca interacción con procedimientos que ejecutan otras partes de un programa. Dicho de forma sencilla, un módulo cohesivo sólo hace (idealmente) una cosa. La cohesión es la medida en la cual todas las partes de un módulo se corresponden entre sí.
La cohesión puede presentarse como un "espectro":

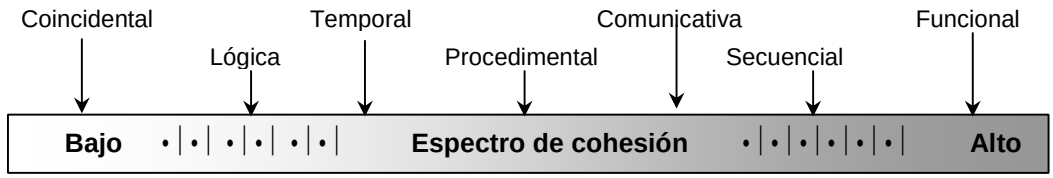

Figura III. 8 – "Espectro" de cohesión

Lo deseable siempre es conseguir una gran cohesión, aunque un punto medio del espectro normalmente es aceptable. La escala de cohesión no es lineal. Es decir, una cohesión baja es mucho "peor" que una de la mitad del espectro, que, a su vez, es casi tan "buena" como una gran cohesión.

En la práctica, un diseñador no tiene que preocuparse por el grado preciso de cohesión de un módulo específico. En vez de ello, debe comprender el concepto general y evitar los niveles bajos de cohesión en el diseño de los módulos.

Cohesión coincidente

Un módulo que realice un conjunto de tareas que están *débilmente relacionadas* entre sí, si es que lo están, es *coincidentalmente cohesivo.*
No puede apreciarse que los elementos del módulo lleven a cabo ninguna función definible. En general se trata de un conjunto de tareas débilmente relacionadas entre sí.
Ejemplos: código divido en base a una medida de cantidad de líneas.

Cohesión lógica

Un módulo que realiza tareas que están relacionadas de forma lógica es *lógicamente cohesivo.*
En este caso, varias funciones semejantes, pero ligeramente diferentes, están combinadas juntas. A menudo requiere una señal de control para indicarle cómo debe ejecutarse. Estos módulos son difíciles de modificar, debido a que los circuitos lógicos son complejos.
Para resolver esta situación, los módulos deben ser remplazados por módulos de propósitos especiales a razón de uno por función.
Ejemplos: Módulo que valida todos los tipos de transacciones empleando secciones comunes de código cuando corresponde, y salteando otras partes cuando no son requeridas; Un módulo que produzca toda la salida independientemente de su tipo.

Cohesión temporal

Cuando un módulo contiene tareas que están relacionadas por el hecho de que se ejecutan en el mismo momento, se dice que exhibe *cohesión temporal.*

El módulo presenta varias funciones cuyo único elemento común es el de ser ejecutados al *mismo tiempo*.
La *cambiabilidad* mejora aislando cada función en su propio módulo.
Ejemplos: Módulos de “inicialización”, “preparación previa”, “reiniciación automática”.

Como ejemplo de baja cohesión, consideremos un módulo que realice el procesamiento de errores de un paquete de análisis de ingeniería. El módulo se invoca cuando los datos calculados exceden unos límites previamente especificados. Realiza las siguientes tareas:

(1) cálculo de datos suplementarios basándose en los datos calculados originalmente;
(2) producción de un informe sobre los errores (con contenido gráfico) en la estación de trabajo del usuario;
(3) realización de todos los cálculos que siga solicitando el usuario;
(4) actualización de una base de datos;
(5) ayuda en la selección de un menú para el siguiente procesamiento.

Aunque las tareas anteriores están ‘un poco’ relacionadas, cada una es una entidad funcional independiente que puede ser realizada mejor por un módulo aparte. La combinación de funciones en un único módulo sólo sirve para que aumente la posibilidad de propagación de errores, cuando se hace una modificación en una de las tareas mencionadas anteriormente.

Los niveles moderados de cohesión están relativamente cercanos unos a otros en su grado de independencia modular.

Cohesión de procedimiento

Cuando los elementos de procesamiento de un módulo están relacionados y deben ejecutarse en un orden específico, existe *cohesión procedimental.*
Los módulos han sido derivados de un diagrama de flujo y cada procedimiento del diagrama ha sido armado en un módulo.
Dentro del módulo se ejecutan varias funciones, pero al menos están relacionadas a través del flujo de control entre ellas.

Cohesión de comunicación

Cuando todos los elementos de procesamiento se concentran sobre un área de una estructura de datos, se presenta una *cohesión de comunicación.*
Las funciones del módulo operan todas sobre la misma corriente de datos además de ser cohesivas de procedimiento.
Ejemplos: “Representar el resultado y grabar registro del diario”, “Leer sentencia fuente y eliminar blancos”, “Calcular solución e imprimir resultado”.

Cohesión funcional

Realiza una y solo una función identificable. Los módulos cohesivos funcionalmente pueden comúnmente describirse a través de una sola frase, con un verbo activo y un objeto único.
Una alta cohesión se caracteriza por que el módulo realiza una tarea procedimental determinada.
Ejemplos: “Calcular la mejor solución”, “Validar consulta”, “Representar Respuesta”.

El siguiente extracto proporciona un conjunto de criterios sencillos para establecer el grado de cohesión (denominada "vinculación" en esta referencia):

> 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", "cuándo, "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.

Una alternativa, para el mismo fin puede ser:

Escribir una sola oración comenzando con "El propósito total de este módulo es …", y luego examinar la oración.
Si la misma no puede ser completada → *cohesivo coincidentemente*.
Si tiene un objetivo plural e incluye la palabra *todas* → *cohesivo lógicamente*. ("Validar todas las transacciones")
Si emplea palabras que guardan relación con el tiempo o secuencia (primero, próximo, luego, después, de otra manera, comienzo) → *cohesivo temporal*, de *procedimiento* o de *comunicación*.
Si consiste de un solo verbo activo con un objeto no plural → *cohesivo funcionalmente*.

Como ya hemos dicho, 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.

Acoplamiento

El *acoplamiento* es una medida de la *interconexión entre los módulos* de una estructura de programa. Al igual que la cohesión, el acoplamiento puede representarse mediante un espectro:

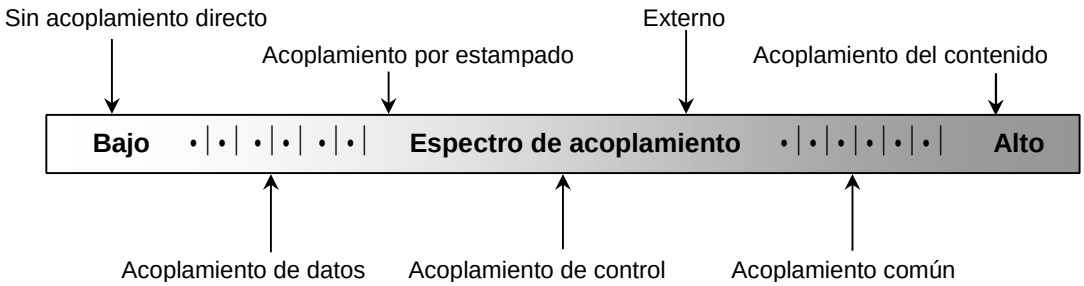

Figura III. 9 – "Espectro" de acoplamiento

El acoplamiento depende de la complejidad de las interfaces entre los módulos, del punto en el que se hace una entrada o referencia a un módulo y de los datos que pasan a través de la interfaz.
En el diseño de software buscamos el más bajo acoplamiento posible. La conectividad sencilla entre módulos da como resultado un software que es más fácil de comprender y menos propenso al "efecto onda" producido cuando los errores aparecen en una posición y se propagan a lo largo del sistema.
La Figura III.11 muestra ejemplos de módulos pertenecientes a una estructura con poco acoplamiento. Los módulos 1 y 2 están subordinados a diferentes módulos. No están relacionados y, por tanto, no existe acoplamiento directo. El módulo 3 es subordinado del módulo 2 y está accesible mediante una lista de argumentos convencionales, a través de la cual pasan los datos.

Acoplamiento de datos

Mientras exista una lista de argumentos sencilla (es decir, se pasan datos simples; existe una correspondencia uno a uno entre elementos), se exhibe un bajo acoplamiento *(acoplamiento de datos* en el espectro) en esta porción de la estructura.

Un módulo le transfiere datos a otro como parte de la invocación o respuesta. Esta es la forma más deseada, ya que representa un acoplamiento mínimo. Se debe buscar la menor transferencia de datos.
El acoplamiento por pasaje de parámetros es mejor que datos globales.

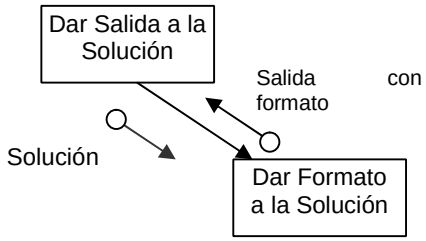

Figura III. 10 – Acoplamiento de datos

Una variación del acoplamiento de datos, denominado *acoplamiento por estampado,* se presenta cuando se pasa una porción de una estructura de datos (en vez de argumentos simples) a través de una interfaz de módulo.

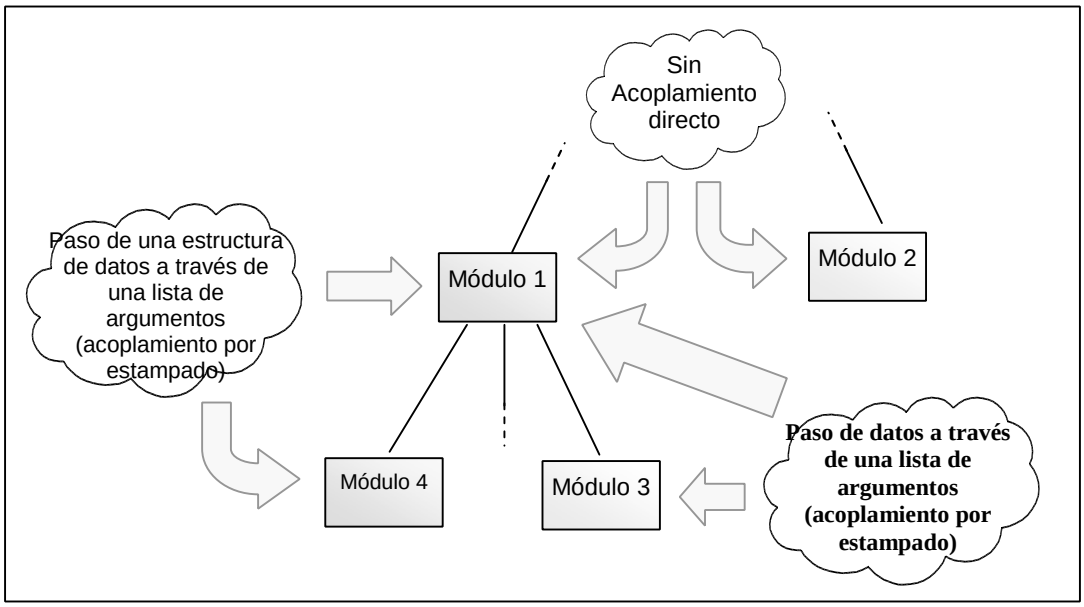

Figura III. 11 – Módulos con bajo acoplamiento

En niveles moderados, el acoplamiento se caracteriza por el paso de control entre módulos.

Acoplamiento de control

El *acoplamiento de control* es muy frecuente en la mayoría de los diseños de software y se ilustra en la Figura III.13. En su forma más sencilla, el control se pasa mediante un "indicador" sobre el que se toman las decisiones en un módulo subordinado o superior; se realiza el pasaje de señales o variables de control entre módulos.

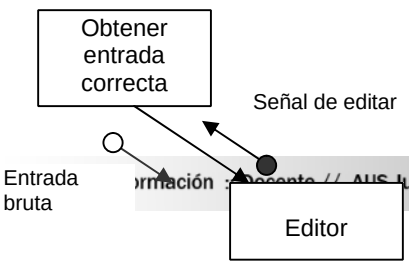

Figura III. 12 – Acoplamiento de Control

Se debe mantener al *mínimo absoluto* necesario el acoplamiento de control. A mayor cantidad de señales se hace más complejo el mantenimiento.
Las señales de control transferidas hacia abajo, en el momento de la invocación, son indicaciones de que el módulo invocado no es caja negra; será ejecutado en forma diferente dependiendo de la señal de control, esta es una situación no deseable.

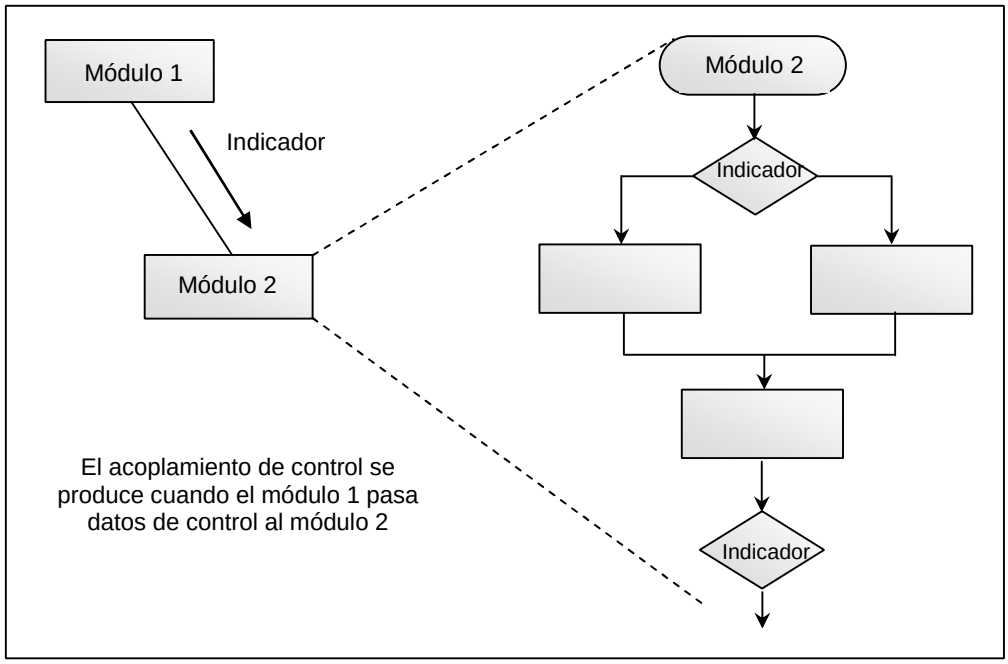

Figura III. 13 – Acoplamiento de Control entre módulos

Los niveles relativamente altos de acoplamiento se producen cuando los módulos están ligados a un entorno externo al software.

Acoplamiento patológico/externo/interno

Se presenta cuando un módulo apunta al interior de otro, ya sea para extraer datos, para transferir el control, o modificar la forma de ejecutar del otro módulo.
Este tipo de acoplamiento no es visible en los diagramas, y en consecuencia es difícil de detectar.
Debe evitarse a toda costa.
Por ejemplo, un módulo que se acopla con dispositivos, formatos y protocolos de comunicación específicos.
El acoplamiento externo muchas veces es necesario, pero debe limitarse a un pequeño número de módulos dentro de una estructura. También se presenta un alto acoplamiento cuando varios módulos referencian un área de datos global. Este tipo de acoplamiento también se denomina *acoplamiento común* y se muestra en la Figura III.14.

Los módulos *C, E y* N acceden todos a un elemento de datos de un área de datos global (p. ej.: un archivo de disco, el COMMON de FORTRAN o un tipo de datos externos en el lenguaje de programación C). El módulo C lee el elemento e invoca a E, quien recalcula y actualiza el elemento. Supongamos que se produce un error y E actualiza incorrectamente el elemento. Mucho más adelante en el procesamiento, el módulo N lee el elemento, intenta procesarlo y falla, haciendo que se interrumpa la ejecución del programa. La causa aparente de la terminación es el módulo N; la causa real, el módulo E.

El diagnóstico de los problemas que aparecen en estructuras con considerable acoplamiento *común* consume mucho tiempo y es difícil. Sin embargo, esto no significa que el uso de datos globales sea necesariamente "malo". Significa que un diseñador de software deber ser consciente de las posibles consecuencias del acoplamiento común y tener cuidado para protegerse de ellas.

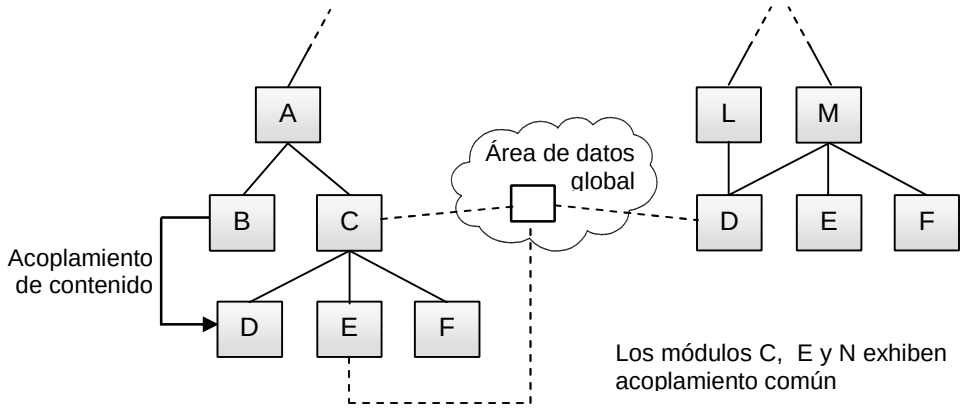

Figura III. 14 – Acoplamiento patológico

El mayor grado de acoplamiento, el *acoplamiento por contenido o patológico*, se produce cuando un módulo utiliza información de datos o de control contenida dentro de los límites de otro módulo. Secundariamente, el acoplamiento por contenido se produce cuando se realiza una bifurcación hacia la mitad de un módulo. Este modo de acoplamiento puede y debe evitarse.

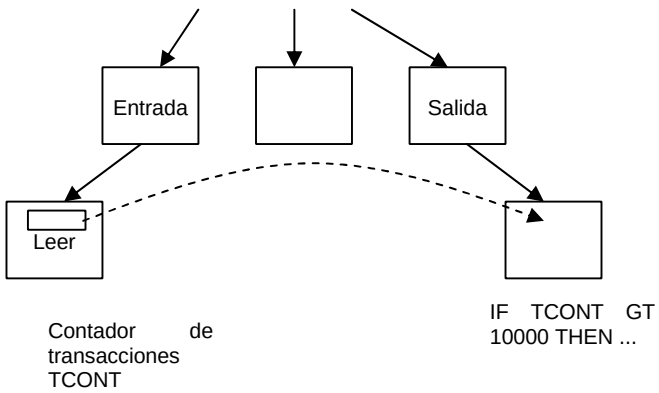

Figura III. 15 – Acoplamiento patológico

Los modos de acoplamiento anteriores surgen por decisiones de diseño que se toman en el desarrollo de la estructura. Sin embargo, durante la codificación se pueden introducir variantes

de acoplamiento externo. Por ejemplo, el acoplamiento con el compilador viene dado por su vinculación del código fuente con atributos específicos (y frecuentemente no estándar) del compilador; el acoplamiento con el sistema operativo se produce por la vinculación del diseño y del código resultantes con "servicios" del sistema operativo (SO), lo que puede causar estragos cuando cambia el SO.

El acoplamiento y cohesión de módulos están relacionados.

Un módulo *altamente cohesivo*, cuyas partes contribuyan todas a una sola función, probablemente será *débilmente acoplado* a otros módulos.

Un módulo *pobremente cohesivo*, combinación de partes no relacionadas, probablemente necesite un *alto acoplamiento* con otros módulos.

Resumen

Dado entonces, los modelos de procesos DFD realizados durante el "análisis", aplicando los criterios (transformación y transacción) para transformar estos en Diagramas Estructurados (DE) se comienza una de las actividades de "Diseño", y se comienza a pensar el problema en una solución de software.

Por otra parte, en esta transformación es *crucial*, tener permanentemente presente los conceptos de ***acoplamiento*** y ***cohesión***, los que ayudan a lograr un buen diseño de software. Recordando que

"el objetivo es lograr el menor nivel de acoplamiento posible y el mayor grado de cohesión posible".

Bibliografía:

(a) "Ingeniería del Software – Un Enfoque Práctico" – Roger S. Pressman - Ed. McGraw-Hill (1993)
(b) "Structured Techniques - The Basis for CASE" - James Martin – Carma McClure – Ed. Prentice Hall – (1985)
(c) "Análisis y Diseño de Sistemas" – Kenneth E. Kendall y Julie E. Kendall – Ed. Prentice-Hall Hispanoamericana (1991)
(d) "Análisis Estructurado de Sistemas" – Chris Gane – Trish Sarson – Ed. El Ateneo – (1988)