

Diseño Estructurado de Sistemas

El diseño estructurado de sistemas se ocupa de la identificación, selección y organización de los módulos y sus relaciones. Se comienza con la especificación resultante del proceso de análisis, se realiza una descomposición del sistema en módulos estructurados en jerarquías, con características tales que permitan la implementación de un sistema que no requiera elevados costos de mantenimiento.

La idea original del diseño estructurado fue presentada en la década de los '70, por Larry Constantine, y continuada posteriormente por otros autores: Myers, Yourdon y Stevens.

1. Introducción

El diseño estructurado es un enfoque disciplinado de la transformación de *qué* es necesario para el desarrollo de un sistema, a *cómo* deberá ser hecha la implementación.

La definición anterior implica que: el análisis de requerimientos del usuario (determinación del *qué*) debe preceder al diseño y que, al finalizar el diseño se tendrá medios para la implementación de las necesidades del usuario (el *cómo*), pero no se tendrá implementada la solución al problema. Cinco aspectos básicos pueden ser reconocidos:

1. Permitir que *la forma del problema guíe a la forma de la solución*. Un concepto básico del diseño de arquitecturas es: *las formas siempre siguen funciones*.
2. Intentar *resolver la complejidad de los grandes sistemas* a través de la segmentación de un sistema en *cajas negras*, y su organización en una jerarquía conveniente para la implementación.
3. Utilizar *herramientas*, especialmente *gráficas*, para realizar diseños de fácil comprensión. Un diseño estructurado usa diagramas de estructura (DE) en el diseño de la arquitectura de módulos del sistema y adiciona especificaciones de los módulos y cuplas (entradas y salidas de los módulos), en un Diccionario de Datos (DD).
4. Ofrecer un conjunto de *estrategias para derivar el diseño* de la solución, basándose en los resultados del proceso de *análisis*.
5. Ofrecer un conjunto de criterios para evaluar la calidad de un diseño con respecto al problema a ser resuelto, y las posibles alternativas de solución, en la búsqueda de la mejor de ellas.

El diseño estructurado produce sistemas fáciles de entender y mantener, confiables, fácilmente desarrollados, eficientes y que funcionan.

2. Diagrama de Estructura

Los *diagramas de estructura* (DE) sirven para el modelamiento *top-down* de la estructura de control de un programa descrito a través de un árbol de *invocación de módulos*. Fueron presentados en la década de los 70 como la principal herramienta utilizada en diseños estructurados, por autores como Constantine, Myers, Stevens e Yourdon. La Fig. 1 muestra un ejemplo:

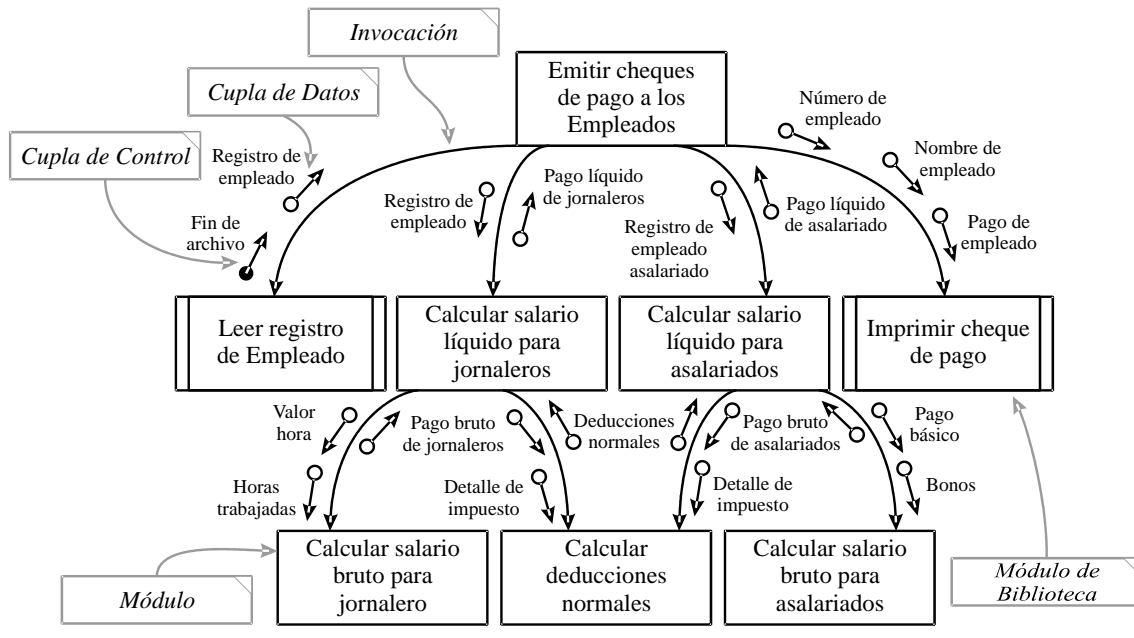


Fig. 1: Ejemplo de Diagrama de Estructura

Un diagrama de estructura permite modelar un programa como una jerarquía de módulos. Cada nivel de la jerarquía representa una descomposición más detallada del módulo del nivel superior. La notación usada se compone básicamente de tres símbolos:

- Módulos
- Invocaciones
- Cuplas

2.1 Módulos

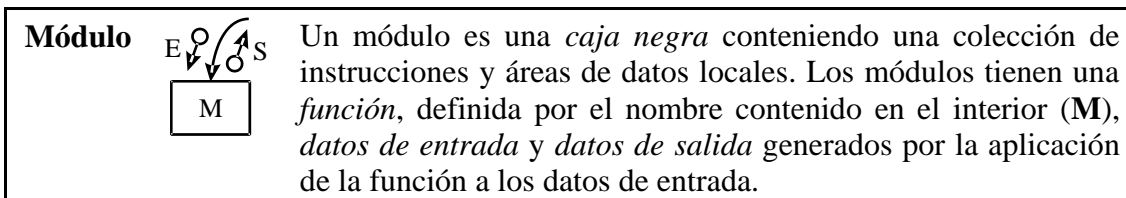
Un módulo es un conjunto de instrucciones que ejecutan alguna actividad, un procedimiento o función en PASCAL, una función en C o un párrafo en COBOL. Tal vez, la definición más precisa es que un módulo es una *caja negra*, pero como será mostrado a continuación son *cajas "casi" negras* o *grises*.

Desde un punto de vista práctico, un módulo es una colección de instrucciones de un programa con cuatro características básicas:

1. *Entradas y Salidas*: lo que un módulo recibe en una invocación y lo que retorna como resultado.
2. *Función*: las actividades que un módulo hace con la entrada para producir la salida.
3. *Lógica Interna*: por la cual se ejecuta la función.
4. *Estado Interno*: su área de datos privada, datos para los cuales sólo el módulo hace referencia.

Las entradas y salidas son, respectivamente, datos que un módulo necesita y produce. Una función es la actividad que hace un módulo para producir la salida. Entradas, salidas y funciones proveen una visión externa del módulo. La lógica interna son los algoritmos que ejecutan una función, esto es, junto con su estado interno representan la visión interna del módulo.

Un módulo es diseñado como una caja, su función es representada por un nombre en el interior y las entradas y salidas de un módulo son representadas por pequeñas flechas que entran y salen del módulo.



2.2 Relaciones entre Módulos (Invocaciones)

En la realidad, los módulos no son realmente cajas negras. Los diagramas de estructura muestran las invocaciones que un módulo hace a otros módulos. Estas invocaciones son diseñadas como una flecha que sale del módulo llamador y apunta al módulo llamado. La Fig. 2 muestra un ejemplo:

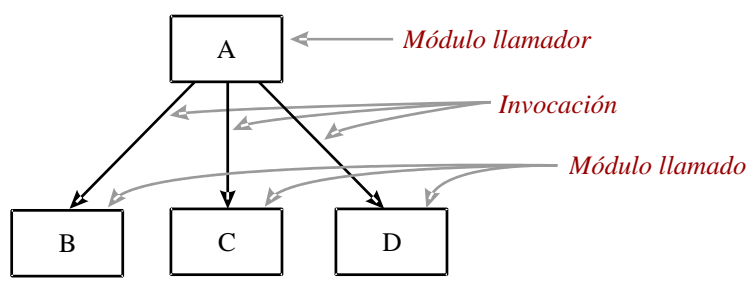
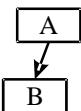


Fig. 2: Ejemplo de Invocación

En el ejemplo de la Fig. 2, el módulo **A** invoca (o llama) a los módulos **B**, **C** y **D**. La interpretación de las invocaciones provee información de la estructura interna del módulo llamador, que no concuerda con la idea de caja negra. Una caja negra no permite que se observe su interior y, las invocaciones que un módulo hace son componentes de su estructura interna. De todas formas, se dice que un módulo es una *caja casi negra* o *caja gris* porque ella permite que se observe solo las invocaciones.

Los diagramas de estructura no tienen especificado el orden de invocación de los módulos invocados. El orden de dibujo de los módulos **B**, **C**, y **D** (de izquierda a derecha) no debe ser interpretado como el orden de invocaciones ejecutado por el módulo **A**. Ese orden puede ser cambiado, al dibujar, para evitar que se crucen flechas o se dupliquen módulos, como ocurre con el módulo *Calcular Deducciones Normales* en la Fig. 1. A pesar que el orden de invocación de los módulos del mismo nivel en un diagrama de estructura, no es especificado por el formalismo, se recomienda que siempre que fuese posible, se siga un orden de izquierda a derecha (si esto no produce que se crucen flechas) que se corresponde con el orden de invocación, y permitiendo un orden de lectura que es patrón en la mayoría de los idiomas.

Una invocación, representa la idea de llamada a funciones o procedimientos en los lenguajes de programación convencionales. A continuación se describe una invocación estándar:

Invocación Estándar		El módulo A invoca al módulo B con la semántica de invocación de procedimientos o funciones en los lenguajes de programación convencionales (C, Pascal, etc.).
----------------------------	---	--

2.3 Comunicación entre Módulos (Cuplas)

Cuando una función o un procedimiento, en un lenguaje convencional, es invocado, comúnmente un conjunto de argumentos es comunicado y, en el caso de las funciones, también se espera que retorne un resultado. Estos datos comunicados en una invocación son modelados por medio de flechas, sobre el símbolo de invocación, llamadas *cuplas*.

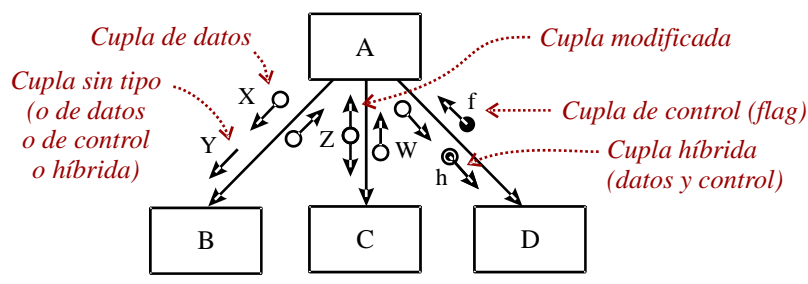





Fig. 3: Ejemplo de invocación con cuplas

Como se muestra en la Fig. 3, existen varios tipos de cuplas, basado en lo que ellas pueden producir en el módulo receptor, las cuales se describen a continuación:

Tipos de Cuplas		
Cupla de Datos		Una <i>cupla de datos</i> transporta datos “puros” a un módulo. No es necesario conocer la lógica interna del módulo receptor, para determinar los valores válidos de la cupla (ej.: número de cuenta, saldo, tabla de movimientos).
Cupla Modificada		Con una flecha doble (apuntando al modulo llamador y al módulo llamado) se especifica un argumento enviado a un módulo que deberá modificar su valor, fijando un nuevo valor disponible para el módulo llamador (en la implementación, se precisará que el lenguaje posea un mecanismo de pasaje de parámetros por referencia) (ej.: el buffer enviado a un módulo de lectura de un archivo).
Cupla de Resultados		Existen módulos que retornan valores sin la necesidad de que estén inicializados en el momento que se invocan. Estos casos son dos: <ol style="list-style-type: none"> 1. Cuplas similares al tipo Modificada cuyos valores previos a la invocación del módulo NO se utilizan para calcular el valor de retorno. Si bien en Pascal se implementa como las “Cuplas Modificadas”, es conveniente documentarlas en el DE como “de Resultados” por cuestiones de claridad. 2. Si el Módulo en cuestión es una función (retorna un valor), se debe documentar este valor de retorno como “Cupla de Resultado” cuyo nombre se corresponderá con el nombre de la función.

3. Criterios de Validación de Calidad

Los diagramas de estructura son simplemente una herramienta para modelar los módulos de un sistema y sus relaciones y, junto con las especificaciones de funcionalidad de los módulos y las estructuras de datos de las cuplas, componen un diseño inicial que deberá ser analizado y mejorado.

Uno de los principios fundamentales del diseño estructurado es que un sistema grande debería particionarse en módulos mas simples. Sin embargo, es vital que esa partición sea hecha de tal manera que los módulos sean tan independientes como sea posible y que cada módulo ejecute una única función. Para que los diseños tengan esas cualidades, son necesarios algunos criterios de medición que permitan clasificar y mejorar los diagramas de estructura. A continuación se describen los criterios utilizados para mejorar un diseño.

3.1 Acoplamiento

El acoplamiento entre módulos clasifica el grado de independencia entre pares de módulos de un DE. El objetivo es minimizar el acoplamiento, es decir, maximizar la independencia entre módulos. A pesar de que el acoplamiento, es un criterio que clasifica características de *una invocación* (una relación existente entre dos módulos), será usado para clasificar un DE completo. Un DE se caracteriza por el *peor* acoplamiento existente entre pares de sus módulos, ya que ese es el problema que debe ser resuelto para mejorar la calidad del DE completo.

Un bajo acoplamiento indica un sistema bien particionado y puede obtenerse de tres maneras:

- *Eliminando relaciones innecesarias*: Por ejemplo, un módulo puede recibir algunos datos, innecesarios para él, porque debe enviarlos para un módulo subordinado.
- *Reduciendo el número de relaciones necesarias*: Cuanto menos conexiones existan entre módulos, menor será la posibilidad del efecto en cadena (un error en un módulo aparece como síntoma en otro).
- *Debilitando la dependencia de las relaciones necesarias*: Ningún módulo se tiene que preocupar por los detalles internos de implementación de cualquier otro. Lo único que tiene que conocer un módulo debe ser su función y las cuplas de entrada y salida (cajas negras).

3.2 Cohesión

Otro medio para evaluar la partición en módulos (además del acoplamiento) es observar como las actividades de un módulo están relacionadas unas con otras; este es el criterio de cohesión. Generalmente el tipo de cohesión de un módulo determina el nivel de acoplamiento que tendrá con otros módulos del sistema.

Cohesión es la medida de intensidad de asociación funcional de los elementos de un módulo. Por elemento debemos entender una instrucción, o un grupo de instrucciones o una llamada a otro módulo, o un conjunto de procedimientos o funciones empaquetados en el mismo módulo.

El objetivo del diseño estructurado es obtener módulos altamente cohesivos, cuyos elementos estén fuerte y genuinamente relacionados unos con otros. Por otro lado, los elementos de un módulo no deberían estar fuertemente relacionados con elementos de otros módulos, porque eso llevaría a un fuerte acoplamiento entre ellos.

3.3 Descomposición (Factoring)

La descomposición es la separación de una función contenida en un módulo, para un nuevo módulo. Puede ser hecha por cualquiera de las siguientes razones.

3.3.1 Reducir el tamaño del módulo

La descomposición es una manera eficiente de trabajar con módulos grandes. Un buen tamaño para un módulo es alrededor de media página (30 líneas). Ciertamente, toda codificación de un módulo debería ser visible en una página (60 líneas).

La cantidad de líneas no es un patrón rígido, otros criterios para determinar cuando es conveniente terminar de realizar la descomposición, son los siguientes:

- *Funcionalidad*: Terminar de realizar la descomposición cuando no se pueda encontrar una función bien definida. No empaquetar líneas de código dispersas, de otros módulos, porque probablemente juntas podrán formar módulos con mala cohesión.
- *Complejidad de Interfaz*: Terminar de realizar la descomposición cuando la interfaz de un módulo *es tan compleja* como el propio módulo. Un módulo de mil líneas es muy confuso, mas mil módulos de una línea son aún más confusos.

3.3.2 Hacer el sistema más claro

La descomposición no debería ser hecha de una manera arbitraria, los módulos resultantes de la descomposición de un módulo deben representar sub-funciones del módulo de mas alto nivel en el DE.

En una descomposición no se debe preocupar por conceptos de programación. Si una sub-función, presentada como un módulo separado permite una mejor comprensión del diseño, puede ser subdividida, aún cuando, en una implementación, el código del módulo sea programado dentro del módulo jefe.

3.3.3 Minimizar la duplicación de código

Cuando se reconoce una función que puede ser reutilizada en otras partes del DE, lo mas conveniente es convertirla en un módulo separado. Así, se puede localizar mas fácilmente las funciones ya identificadas y evitar la duplicación del mismo código en el interior de otro módulo. De esta manera, los problemas de inconsistencia en el mantenimiento (si esa función debe ser modificada) pueden ser evitados, y se reduce el costo de implementación.

3.3.4 Separar el trabajo de la ‘administración’

Un administrador o gerente de una compañía bien organizada debería coordinar el trabajo de los subordinados en lugar de *hacer* el trabajo. Si un gerente hace el trabajo de los subordinados no tendrá tiempo suficiente para coordinar y organizar el trabajo de los subordinados y, por otro lado, si hace el trabajo los subordinados no serían necesarios. Lo mismo se puede aplicar al diseño del DE, relacionado a los módulos de *Trabajo* (edición, cálculo, etc.) y a los módulos de *Gerencia* (decisiones y llamadas para otros módulos).

El resultado de este tipo de organización es un sistema en el cual los módulos en los niveles medio y alto de un DE son fáciles de implementar, porque ellos obtienen el trabajo hecho por la manipulación de los módulos de los niveles inferiores.

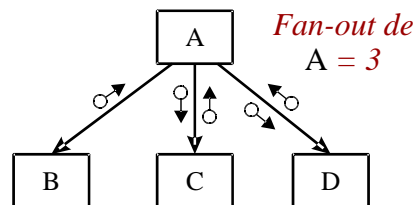
La separación del trabajo de la *administración* mejora la mantenibilidad del diseño. Una alteración en un sistema es: un cambio de control o un cambio de trabajo, pero raramente ambos.

3.3.5 Crear módulos más generales

Otra ventaja de la descomposición es que, frecuentemente, se pueden reconocer módulos más generales y, así, más útiles y reutilizables en el mismo sistema y, además, pueden ser generadas bibliotecas de módulos reutilizables en otros sistemas.

3.4 Fan-Out

El *fan-out* de un módulo es usado como una medida de *complejidad*. Es el número de subordinados inmediatos de un módulo (cantidad de módulos invocados).

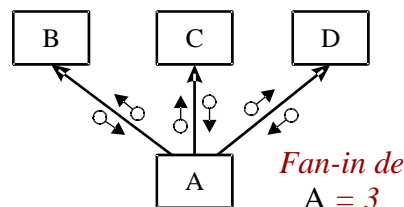


Si un módulo tiene un fan-out muy grande, entonces compone el trabajo de muchos módulos subordinados y, casi con certeza, tiene toda la funcionalidad no trivial representada por ese subárbol en el DE.

Para tener acotada la complejidad de los módulos se debe limitar el *fan-out* a no más de siete más o menos dos (7 ± 2). Un módulo con muchos subordinados puede fácilmente ser mejorado por descomposición.

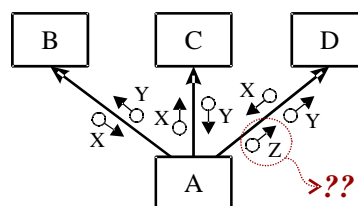
3.5 Fan-In

El *fan-in* de un módulo es usado como una medida de *reusabilidad*, es el número de superiores inmediatos de un módulo (la cantidad de módulos que lo invocan).



Un alto fan-in es el resultado de una descomposición inteligente. Durante la programación, tener una función llamada por muchos superiores evita la necesidad de codificar la misma función varias veces. Existen dos características fundamentales que deben ser garantizadas en módulos con un alto fan-in:

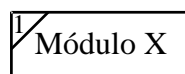
- **Buena Cohesión:** Los módulos con mucho fan-in deben tener alta cohesión, con lo cual es muy probable que tengan buen acoplamiento con sus llamadores.
- **Interfaz Consistente:** Cada invocación para el mismo módulo debe tener el mismo número y tipo de parámetros. En el ejemplo que sigue, hay un error.



4. Algunas consideraciones adicionales

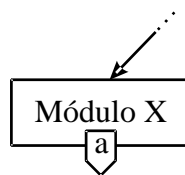
1. En primer término es importante destacar que bajo ningún aspecto es aceptable graficar una invocación de un módulo hacia otro que esté dibujado más arriba, es decir que no se puede dibujar ninguna flecha con sentido ascendente, a lo sumo se puede realizar una invocación hacia un módulo que está a la misma altura (hermano), en cuyo caso la flecha queda horizontal.
2. En los casos que exista una importante reutilización puede ser necesario dibujar el mismo módulo en distintos lugares del gráfico a fin de que las invocaciones existentes en el diagrama no se crucen. Cuando esto ocurre el módulo debe ser dibujado en cada ocurrencia con la incorporación de una línea diagonal en el extremo superior izquierdo y un número que lo identifica (todos los cuadros que se refieren al mismo módulo tienen el mismo nombre y el mismo número).

Estos módulos se grafican entonces de la siguiente manera:



3. Adicionalmente, si algunos de estos módulos u otros dentro de un diagrama de gran tamaño deben continuar la cadena de invocación se puede utilizar la siguiente sintaxis para desdoblar el diagrama de estructuras.

En el diagrama original se incorpora un pentágono rotulado (en este caso "a")



Se comienza un diagrama aparte indicando en el inicio el mismo pentágono con idéntico rótulo y continuando la gráfica de los módulos invocados por el módulo original.

