

Diseño físico de datos y funciones.
Criterios de diseño. Documentación.

TEMA 55

ABACUS NT

Índice

- 1. *Introducción.***
- 2. *Diseño físico de datos. Criterios. Documentación.***
 - 2.1. Dependencia del producto comercial específico.**
 - 2.2. Consideraciones generales sobre el diseño físico de datos.**
- 3. *Diseño físico de funciones y su documentación. Criterios.***
 - 3.1. Modularización**
 - 3.1.1. Módulos**
 - 3.1.2. Estructuras de control.**
- 4. *Diseño arriba-abajo (top-down) y la codificación abajo-arriba (bottom-UP).***
 - 4.1. El diseño TOP-DOWN.**
 - 4.2. Codificación BOTTOM.UP**
- 5. *Estilo de codificación del software.***
 - 5.1. Documentación dentro del código.**
 - 5.2. Las pruebas y la depuración de errores.**
 - 5.2.1. La depuración de errores en la codificación.**
 - 5.2.2. Las pruebas del software.**
 - 5.2.3. Documentación de las pruebas.**
- 6. *Elección del lenguaje de programación final***
 - 6.1. Lenguajes más utilizados en el sector productivo**
- 7. *Conclusión.***
 - 7.1. Relación con el sistema educativo**
- 8. *Bibliografía***

1. Introducción.

Antes de nada, conviene diferenciar claramente el diseño físico de los datos y funciones del diseño lógico, que es el que se trata en temas anteriores del temario. El diseño físico de los datos es la implementación efectiva del diseño lógico, con una herramienta específica comercial.

Tomando como partida el diseño lógico desarrollado durante el análisis, nuestro cometido ahora será adaptar al contexto físico real dicho diseño, buscando la eficiencia al adaptarlo al entorno final.

Por ejemplo, un diseño lógico concreto variará considerablemente si se adapta a una implementación física con ficheros de tipo **secuencial indexado** o de **acceso directo** dependiendo de si utilizamos un **SSD** o un **HDD**. Igual ocurrirá si utilizamos un determinado **software comercial** para su implementación final.

En este tema se tratan en conjunto datos y funciones, ya que para ambos se pueden seguir procedimientos similares de adaptación al entorno físico.

2. Diseño físico de datos. Criterios. Documentación.

En la implementación física de los datos hay tres actores distintos:

- El esquema lógico general.
- El esquema lógico específico.
- DDL (Lenguaje de definición de datos) de la base de datos concreta o lenguaje comercial que se vaya a utilizar.

2.1. Dependencia del producto comercial específico.

La última etapa del proceso de diseño de datos es la que ataña a la organización física de los mismos, en la cual, teniendo presentes requisitos de los procesos del SGBD, del Sistema operativo y del hardware, se pretenden entre otros los siguientes objetivos:

- Minimizar los tiempos de respuesta.
- Minimizar el espacio de almacenamiento.
- Evitar las reorganizaciones costosas en tiempo y recursos.
- Proporcionar la máxima seguridad y fiabilidad.
- Optimizar el consumo de recursos.

En definitiva, cumplir los objetivos del sistema y conseguir la optimización de la ratio coste/beneficio.

Sin embargo, con frecuencia, una base de datos específica (MySQL, MariaDB, PostgreSQL, etc) presenta cierta diferencias de implementación respecto al modelo lógico SQL (Standard Query Languaje).

Nos podemos encontrar con alguno de los siguientes casos:

- El SGDB Soporta **todos** los conceptos del modelo lógico estándar, añadiendo o no, algunos específicos.
- El SGDB Soporta **algunos** de los conceptos del modelo lógico estándar, o presenta ciertas restricciones en casos determinados.

En ambos casos puede ser necesaria una programación complementaria:

- En el primer caso para la optimización del código, si bien esto dependerá de consideraciones internas de la empresa (a veces es mejor decidir compatibilidad que velocidad)
- En el segundo caso sin embargo, esta programación extra resulta **imprescindible**: es necesario solventar las restricciones no soportadas para que el modelo funcione.

El diseño físico resulta, por tanto, muy dependiente del producto comercial concreto que se escoja para la implementación de la aplicación.

2.2. Consideraciones generales sobre el diseño físico de datos.

El problema del diseño físico para el administrador de la Base de Datos consiste en proveer un conjunto eficiente de estructuras de acceso de modo que el optimizador pueda tomar las mejores decisiones.

Entre los instrumentos más importantes y generales del diseño físico se encuentra la selección de los **índices secundarios**, que es uno de los problemas clave en la instrumentación física de una base de datos.

Una vez diseñadas las aplicaciones se conocerá cuáles son las consultas más frecuentes y/o prioritarias a la base de datos, por lo que será conveniente crear un índice secundario que ayude a localizar las filas seleccionadas en dichas consultas y reducir así los accesos a disco.

Normalmente se realiza un estudio estadístico por el cual se determina que consultas son más frecuentes y por tanto indicarán qué índices se deben crear para proporcionar acceso eficiente a los datos que son reclamados con mayor frecuencia.

La determinación de estos índices debe ser siempre incluida en el diseño físico de los datos.

Existen otros elementos importantes en el diseño físico, aunque no todos los sistemas comerciales disponen de ellos. Algunos de estos elementos son:

- Características de los registros físicos.
- Asignación de dirección calculada (hashing y punteros)
- Agrupamiento de registros (clustering)
- Bloqueo y compresión de datos.
- Asignación de memoria intermedias (buffering)
- Asignación de conjuntos de datos a particiones y a disposiciones físicas.
- Organización física del fichero: contigua, encadenada, indexada, etc.

En general, existen tres estrategias de los fabricantes en cuanto al diseño:

- El SGB impone una estructura interna, dejándole al diseñador muy poca flexibilidad, lo que suele aumentar la **independencia física/lógica, pero disminuye la eficiencia.**
- El administrador diseña la estructura interna, lo que en general, supone una importante carga para el administrador y puede influir **negativamente en la independencia, aunque puede mejorar la eficiencia.**
- El SGBD proporciona una estructura interna **opcional** que el diseñador puede cambiar a fin de optimizar el rendimiento de la base de datos.

Actualmente (año 2020) las últimas versiones de los productos comerciales más utilizados suelen escoger la tercera opción, ya que es la que más maniobrabilidad proporciona: el administrador puede optar por un mayor control y eficiencia o dejar en manos del SGDB la gestión automática de las estructuras internas de datos, obteniendo menor eficiencia, pero una mayor celeridad en el desarrollo.

3. Diseño físico de funciones y su documentación. Criterios.

Partiendo de las especificaciones del sistema y de los diagramas de flujo de datos (DFD), en el diseño físico pasamos el esquema lógico a su implementación física.

Para implementar las funciones físicas, conviene partir de un seudocódigo o un **diseño algorítmico similar**, ya que este nos proporciona una implementación bastante aproximada a la forma física concreta del lenguaje de implementación que vayamos a utilizar para programar dicho código.

Se dispone de esta forma de un código sobre el que están definidas las estructuras de control, sin detallar la sintaxis del lenguaje final de programación.

Una vez efectuada una planificación de la fase de codificación, que se llevará a cabo teniendo en cuenta la complejidad de los problemas a resolver, la potencia de las herramientas disponibles y la cualificación del personal encargado, se procederá a la elaboración del código fuente.

Los criterios que se seguirán en esta fase de diseño incluyen:

- Modularización de las funcionalidades del sistema
- Estudio de la interfaz modular (para interactuar con los demás módulos)
- **Minimización del acoplamiento:** Grado de interdependencia entre los módulos. Un buen diseño minimiza el acoplamiento al máximo.
- **Maximización de la Cohesión:** Mide la relación que existe entre los elementos de un mismo módulo, es decir, las relaciones intramodulares. Por el contrario, mientras mayor es la cohesión de un módulo, mejor está diseñado. Los autores Stevens, Myers y Constantine desarrollaron una escala para medir la cohesión, cuyos valores ordenados de mayor a menor van desde la cohesión funcional hasta la coincidental.

3.1. Modularización

Es necesario indicar que los adjetivos modular y estructurado se suelen aplicar a los conocidos como lenguajes de tercera y cuarta generación y al código fuente creado por ellos.

3.1.1. Módulos

La programación modular surge **como mejora de la programación estructurada**, con el objetivo de evitar la creación de programas de un único bloque de tamaño desmesurado, y así mejorar el mantenimiento de los mismos.

Para ello, los programas se descomponen en fragmento denominados módulos, los cuales, según el autor Luis Joyanes Aguilar, en su libro *Fundamentos de programación. Algoritmos, estructuras de datos y objetos* (2008), son **unidades de código claramente definidas** y manejables, con **interfaces modulares**, creados para ejecutar una **tarea específica**, que son compilados (o interpretados) de forma **independiente**. Es por ello que son analizados y desarrollados por separado. Por lo que la estructura de un programa modular consistirá en un módulo principal el cual se encargará de llamar (o invocar) al resto de módulos o subprogramas.

Las ventajas que ofrece este tipo de programación son:

- Facilita el desarrollo de programas en **equipo**
- Permite la **reutilización** de módulos, reduciendo el tiempo de desarrollo.
- Simplifica el diseño y ejecución de las **pruebas**
- Reducción de los costes de **mantenimiento**
- Mejora la **claridad del diseño** y de la **documentación**
- Se reduce el número de **bugs** en el programa al reutilizarse módulos ya probados anteriormente.

Esto nos podría llevar a pensar que la mejor estrategia es dividir el software en el mayor número de módulos posibles, no obstante, esto provoca, a su vez, que el esfuerzo necesario para desarrollar y mantener las interfaces por las que se comunican se incremente. Es por ello que es necesario buscar un equilibrio entre ambos esfuerzos.

3.1.2. Estructuras de control.

Un programa estructurado es aquel que evita código cruzado y utiliza exclusivamente las estructuras de control básicas: secuencial, condicional y repetitiva en cualquiera de sus variantes.

Existen otras estructuras de control, como la **recursividad**, que permite que un subprograma se active a sí mismos como parte de su propia ejecución.

La **concurrentia** que permite la ejecución paralela (por tiempo compartido, o por multiproceso) de múltiples tareas, su sincronización y comunicación entre sí.

El **manejo de excepciones** que permite atrapar los errores del sistema o ciertas condiciones definidas previamente, pasando el control a subprogramas manejadores de excepciones

4. Diseño arriba-abajo (top-down) y la codificación abajo-arriba (bottom-UP).

4.1. El diseño TOP-DOWN.

En la fase de análisis de un programa se suelen aplicar las técnicas de arriba-abajo (top-down) consistentes en ir estudiando el problema en sucesivos niveles de menor a mayor grado de detalle.

En el nivel de mínimo detalle toda la aplicación se presenta como una **caja negra** con unas entradas y unas salidas, de la cual sabemos qué hace algo, pero sin analizar el cómo. .

A continuación , se baja un escalón hacia un mayor nivel de detalle, cambia el punto de vista y, lo que antes aparecía como una caja negra (proceso único en un diagrama de contexto) se convierte en un proceso de nivel mayor que 0 en un DFD, en una primitiva funcional y, más tarde, en una unidad de programación o en un subprograma que a su vez puede contener otros subprogramas que se presentan como cajas negras y que dejarán de serlo en futuros descensos hacia mayores niveles de detalle.

El diseño Top-down también se conoce como técnica de refinamiento sucesivo y como hemos comentado se utiliza en la fase de diseño del software.

4.2. Codificación BOTTOM.UP

Una vez que se ha efectuado el estudio de una aplicación, es necesario pasar a su codificación. Para ello el programador irá traduciendo las especificaciones del análisis a código fuente. La cuestión es por dónde empezar.

Si la aplicación comienza a codificarse a partir del menor nivel de detalle, no será posible efectuar pruebas hasta su completa terminación, puesto que mientras que los módulos de máximo detalle no estén terminados no podrá ejecutarse.

5. Estilo de codificación del software.

Una vez que se ha generado el código fuente, lo que el módulo realiza ha de poder ser comprendido sin necesidad de consultar la documentación del análisis.

5.1. Documentación dentro del código.

La documentación dentro del código es de importancia capital pues es la mejor garantía de unos programas legibles.

Hay que cuidar la elección de los nombres de los identificadores de forma que sean suficientemente explicativos sin alcanzar tamaños enormes que dificulten su utilización.

Otro elemento de documentación es la posibilidad de incluir comentarios en lenguaje natural como parte del código fuente..

Podemos distinguir entre comentarios de prólogo y los comentarios de descriptivos . Los primeros se deben incluir en el inicio de cada subprograma y entre otros datos pueden incluir:

- el nombre del subprograma
- su propósito
- una descripción del interfaz que incluya
- un comentario sobre su funcionamiento, limitaciones, variables principales utilizadas etc.
- Un resumen de la historia de su desarrollo que incluya el autor, o autores y los modificadores posteriores junto con la fecha en que cada uno efectuó su trabajo.

Un último elemento que afecta a la documentación interna de un programa es la forma en que aparece el listado del mismo. La sangría del código puede realzar las estructuras de control utilizadas por el programador ayudando a su mejor comprensión.

5.2. Las pruebas y la depuración de errores.

5.2.1. La depuración de errores en la codificación.

El proceso de compilación constituye un primer filtro que va a detectar todos aquellos errores sintácticos que se hayan introducido durante la codificación de un módulo.

Cuanto más avanzado es un compilador, mejor es su capacidad de detectar errores de sintaxis y más explicativos son los mensajes de aviso que en estos casos se producen.

Además de errores es muy posible que surjan anomalías en tiempo de ejecución. Para depurarlas son muy útiles herramientas tales como trazadores y analizadores de ejecución, que permiten ejecutar el programa paso a paso mientras se va viendo el código fuente visualizar y cambiar valores a las variables, conocer el estatus de los canales de comunicación abiertos .. situar puntos de interrupción fijos y condicionales.. etc., En resumen, suponen una gran ayuda en el proceso de depuración de anomalías del código fuente.

5.2.2. Las pruebas del software.

La prueba del software es un elemento crítico para obtener la garantía de que las aplicaciones cumplen con los niveles de calidad requeridos.

Esta fase supone una última oportunidad de verificar las especificaciones del diseño y las codificaciones de un programa.

5.2.3. Documentación de las pruebas.

Como resultado de la realización de las actividades y tareas de este módulo, se obtienen los siguientes documentos:

- Documento de Diseño Técnico, DDT.
- Documento de Operación, DOP.
- Plan de Pruebas del equipo lógico del proyecto, PPRB.

Documento de diseño técnico (DDT)

- Entorno Físico del sistema.
- Diseño Técnico.
- Diseño Detallado de Componentes del Sistema.
- Diseño de Bases de Datos o Ficheros.

El índice de dicho documento será el siguiente:

1. Diseño de la Arquitectura del Sistema.
 - 1.1. Diseño de la Arquitectura Modular.
 - 1.1.1. Interfases entre Componentes.
 - 1.2. Interfases con otros Sistemas.
 - 1.3. Diseño Detallado.
 - 1.3.1. Diseño Detallado de los Componentes.
 - 1.3.2. Pantallas e Informes asociados.
 - 1.3.3. Estructura de datos asociados.
2. Diseño Físico de Datos.
 - 2.1. Estructura Física de la Base de Datos o de los Ficheros.
 - 2.2. Estructuras de Tablas y Vistas.
 - 2.3. Ficheros Auxiliares.
 - 2.4. Descripción de Atributos.
3. Entorno Tecnológico del Sistema.
 - 3.1. Especificaciones del Entorno Tecnológico.
 - 3.1.1. Equipo Físico.
 - 3.1.2. Equipo Lógico.
 - 3.1.3. Comunicaciones.
 - 3.2. Restricciones Técnicas.
4. Especificación del Plan de Desarrollo e Implementación.
5. Control de Calidad del Diseño Técnico.
 - 5.1. Revisión del Diseño Técnico.
 - 5.2. Validación del Diseño Técnico.

Documento de operación (DOP)

Que contiene:

1. Procedimientos de operación de los componentes del sistema.
2. Procedimientos de seguridad y control.

Plan de pruebas del equipo lógico del proyecto (PPRB)

Que se ampliará con:

1. Diseño del Plan de pruebas del Sistema.
2. Definición del entorno de pruebas.

6. Elección del lenguaje de programación final

6.1. Lenguajes más utilizados en el sector productivo

Según la publicación “**2019 The top Programming Languages**” del IEEE Spectrum (el espectro del Instituto de ingeniería eléctrica y electrónica), de los diez lenguajes con más importancia en el sector productivo, 9 son multiparadigma y/o orientados a objetos, lo que muestra la gran importancia de la P.O.O actualmente. Estos lenguajes son:

1. **Python**: Desarrollado por Guido Van Rossum que liberó su código en 1991 bajo licencia Python License originalmente y GNU GPL en la actualidad, es un lenguaje de programación interpretado, multiplataforma, que soporta P.O.O, programación imperativa y funcional. Hace hincapié en la legibilidad del código, utilizando un tipado dinámico y conteo para el administrador de memoria. Debe su nombre a la aficción de su desarrollador por los Monthly Python.
2. **Java**: Fue desarrollado en 1985 y está enfocado hacia la red, debido al auge de Internet en los años 90. Es independiente del ordenador donde se ejecute y para conseguir esta independencia utiliza una máquina virtual java (JVM) que nos permite utilizar código compilado en otra computadora distinta. Entre sus características destacamos:
 - Sintaxis similar a C++
 - Dispone de un recolector de basura
 - Elimina el uso de punteros, utilizando referencias
 - Soporta programación multihilo
 - Dispone de mecanismos de detección de errores
 - Tiene amplias librerías de clases

3. **C:** Desarrollado en 1972 a partir de otro lenguaje llamado B, con la idea de conseguir un lenguaje de alto nivel, pero con posibilidad de acceso a bajo nivel. Esta propiedad hace que este lenguaje sea muy adecuado para la programación de sistemas.

Es adecuado para programadores expertos porque hace programas muy eficientes, por el contrario, su depuración puede ser bastante compleja. Tiene las siguientes características:

- Es débilmente tipado
- Impone pocas restricciones al programador
- Dispone de operadores a nivel de bit
- Soporta el uso de punteros
- Utiliza librerías

4. **C++:** Fue inventado en los laboratorios de AT&T en 1980. Inicialmente fue una gran mejora de C. Actualmente tiene un estándar, denominado ISO C++.

C++ soporta programación orientada a objetos incluyendo características como:

- Implementación de clases
- Herencia múltiple
- Acceso a los atributos y métodos

5. **R:**

Es un entorno y lenguaje de programación con un enfoque al análisis estadístico.

Se trata de uno de los lenguajes de programación más utilizados en investigación científica, junto con Matlab, siendo además muy popular en los campos de aprendizaje automático (machine learning), minería de datos, investigación biomédica, bioinformática y matemáticas financieras. R es orientado a objetos, se puede integrar con bases de datos, tiene numerosas funciones de cálculo y funciones gráficas compatibles con LaTeX.

6. **Javascript:**

Es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

Se utiliza principalmente en su forma del lado del cliente (client-side), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web

JavaScript se diseñó con una sintaxis similar a C, aunque adopta nombres y convenciones del lenguaje de programación Java. Sin embargo, Java y JavaScript tienen semánticas y propósitos diferentes.

Todos los navegadores modernos interpretan el código JavaScript integrado en las páginas web. Para interactuar con una página web se provee al lenguaje JavaScript de una implementación del Document Object Model (DOM).

Actualmente es ampliamente utilizado para enviar y recibir información del servidor junto con ayuda de otras tecnologías como AJAX.

7. C#:

Pronunciado C sharp, es un lenguaje de programación multiparadigma desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, de los lenguajes de programación diseñados para la infraestructura de lenguaje común (CLI).

Su sintaxis básica deriva de C/C++, utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.

8. Matlab:

Matrix Laboratory es un sistema de cómputo numérico que ofrece un IDE y lenguaje de programación llamado M que es interpretado y orientado a objetos. Este lenguaje permite operaciones de vectores y matrices, funciones, cálculo lambda y representación de gráficos 2D y 3D.

9. Swift

Swift es un lenguaje de programación **multiparadigma** creado por **Apple** enfocado en el desarrollo de **aplicaciones para iOS y macOS**. Fue presentado en la WWDC 2014 y puede usar cualquier biblioteca programada en Objective-C y llamar a funciones de C.

10. Go

Desarrollado por Google en 2009, la concurrencia, la sencillez, tipado estático, y el no utilizar excepciones son las principales características de Go. Es el lenguaje de programación de sistemas multiproceso más eficaz existente en la actualidad, inspirado en la sintaxis de C. Soporta orientación a objetos parcialmente, ya que no implementa mecanismos de herencia y las clases son declaradas como componentes separados de Interfaces y estructuras.

7. Conclusión.

La fase de diseño constituye la última fase previa a la implementación física de la aplicación en un sistema computacional concreto.

Recoge pues todas las orientaciones y trabajos previos de las fases de análisis y diseño lógico, suponiendo la confirmación o refutación de su validez y dividiéndose también en diseño físico funcional y diseño físico de la estructura externa de datos.

7.1. Relación con el sistema educativo

- GS – DAW - DAM –Entornos de Desarrollo (PES)
- GS – DAW – DAM – Programación (PES)

8. Bibliografía

- Pressman, R. S. **Ingeniería del Software. Un enfoque práctico**, 3^a edición. Ed. McGraw-Hill, 2000.
- Sommerville, I.: **Ingeniería de Software**. 6^a Edición. Addison-Wesley Iberoamericana, 2002.
- Piattini, M .y otros: **Análisis y diseño detallado de Aplicaciones Informáticas de Gestión**. RA-MA, 2007.
- Cerrada, J. A.: **Introducción a la Ingeniería del Software**. 1^a Edición de 2000. Editorial Centro de Estudios Ramón Areces, S.A.
- <https://www.infor.uva.es/~jvalvarez/docencia/metrica>

