Introducción al Diseño de Sistemas de Información

Unidad Nº II: Diseño de Sistemas

Facultad Regional Santa Fe Universidad Tecnológica Nacional

Diseño de Sistemas de Información

Introducción

Veamos ahora con más detenimiento de qué estamos hablando concretamente cuando hablamos de 'Diseño de Software'.

Historia – La evolución del software

El contexto en el que se ha desarrollado el software está fuertemente ligado a la evolución de los sistemas informáticos. Un mejor rendimiento del hardware, una reducción del tamaño y un costo más bajo, han dado lugar a sistemas informáticos más sofisticados.
La evolución del software dentro del contexto de las áreas de aplicación de los sistemas basados en computadoras, puede verse de la siguiente manera :

Los primeros años 1950 - 1965[1]	La segunda era 1965 - 1975	La tercera era 1975 - 1985	La cuarta era 1985 -
• Orientación por lotes • Distribución limitada • Software "a medida"	• Multiusuario • Tiempo real • Bases de Datos • Software como producto	• Sistemas distribuidos • Incorporación de "inteligencia" • Hardware de bajo costo • Impacto en el consumo	• Potentes sistemas de escritorio • Tecnología orientada a objetos • Sistemas expertos • Redes neuronales artificiales • Computación paralela

En los **primeros años** de desarrollo de las computadoras, el hardware sufrió continuos cambios, mientras que el software se contemplaba simplemente como un añadido. La programación de computadoras era un arte para el cual existían pocos métodos sistemáticos. El desarrollo de software se realizaba virtualmente sin ninguna planificación (hasta que los planes comenzaron a desfasarse y los costos a crecer). Durante este período se utilizaba en la mayoría de los sistemas una orientación por lotes. Algunas excepciones fueron sistemas interactivos (Sistema de reservas de América Airlines) y sistemas de tiempo real para la defensa (SAGE). No obstante esto, la mayor parte del hardware se dedicaba a la ejecución de un único programa que, a su vez, se dedicaba a una aplicación específica.
Lo normal era que el hardware fuera de propósito general. Por otra parte, el software se diseñaba a medida para cada aplicación y tenía una distribución relativamente pequeña. La mayoría del software se desarrollaba y era utilizado por la misma persona u organización. La misma persona lo escribía, lo ejecutaba y, si fallaba, lo depuraba.

Debido a este entorno personalizado del software, el **diseño** era un proceso implícito, realizado en la mente de alguien, y la documentación normalmente no existía.
La **segunda era** en la evolución de los sistemas de computadoras se extiende desde la mitad de la década de los 60 hasta finales de los setenta. La multiprogramación y los sistemas multiusuarios introdujeron nuevos conceptos de interacción hombre máquina. Las técnicas interactivas abrieron un nuevo mundo de aplicaciones y nuevos niveles de sofisticación del hardware y del software. Los sistemas de tiempo real podían recoger, analizar y transformar datos de múltiples fuentes, controlando así los procesos y produciendo salidas en milisegundos en lugar de en minutos. Los avances en los dispositivos de almacenamiento en línea condujeron a la primera generación de sistemas de gestión de bases de datos.
Otra característica fue el establecimiento del *software como producto* y la llegada de las "casas de software".
Conforme crecía el número de sistemas, comenzaron a extenderse las bibliotecas de software de computadora. Se desarrollaban proyectos en los que se producían programas de decenas

[1] Los años indicados sirven solamente de referencia. No indican un año exacto de comienzo y fin de una era.

de miles de sentencias fuente. Todos esos programas (todas esas sentencias fuentes) tenían que ser corregidos cuando se detectaban fallos, modificados cuando cambiaban los requisitos de los usuarios o adaptados a nuevos dispositivos que se hubieran incorporado. Estas actividades se llamaron *mantenimiento del software*. El esfuerzo gastado en el mantenimiento comenzó a absorber recursos en una medida alarmante. Había comenzado una "crisis del software".

La **tercera era** en la evolución de los sistemas de computadoras comenzó a mediado de los setenta y llega hasta el momento actual. El procesamiento distribuido (múltiples computadoras, cada una ejecutando funciones concurrentemente y comunicándose con alguna otra) incrementó notablemente la complejidad de los sistemas informáticos. Las redes de área local y de área global, las comunicaciones digitales de alto ancho de banda y la creciente demanda de acceso "instantáneo" a los datos, pusieron una fuerte presión sobre los desarrolladores del software.

En esta era se produce la llegada de los microprocesadores y las computadoras personales.

Las computadoras personales han sido el catalizador del gran crecimiento de muchas compañías de software. Estas compañías venden decenas y centenares de copias de sus productos de software.

La **cuarta era** del software de computadora está evolucionando ahora. Las tecnologías orientadas a objetos están comenzando a ser utilizadas en muchas áreas de aplicación. Las técnicas de cuarta generación para el desarrollo de software ya están cambiando la forma en que algunos segmentos de la comunidad informática construyen los programas. Los sistemas expertos y el software de inteligencia artificial se han trasladado del laboratorio a las aplicaciones prácticas. El software de redes neuronales artificiales ha abierto excitantes posibilidades para el reconocimiento de formas y habilidades de procesamiento de información al estilo de como lo hacen los humanos.

Conforme transitamos por la cuarta era, continúan intensificándose los problemas asociados con el software de computadoras :

1. La sofisticación del hardware ha dejado desfasada nuestra capacidad de construir software que pueda explotar el potencial del hardware.
2. Nuestra capacidad de construir nuevos programas no puede dar abasto a la demanda de nuevos programas.
3. Nuestra capacidad de mantener los programas existentes está amenazada por el *mal diseño* y el uso de recursos inadecuados.

La complejidad de los problemas

No todos los sistemas de software son complejos. En particular las aplicaciones que son especificadas, construidas, mantenidas, y usadas por la misma persona, no pueden considerarse problemas complejos. Estos sistemas tienden a tener propósitos limitados y un ciclo de vida muy corto. Estas aplicaciones generalmente son más tediosas que difícil de desarrollar, y en consecuencia aprender cómo se diseñan no es de nuestro interés.

Las aplicaciones de mayor envergadura (las que presentan comportamiento variado, mantienen la integridad de cientos de transacciones de registro de información, control de tráfico, etc.), son esencialmente complejas.

Esta complejidad es, en general, imposible de ser comprendida en su totalidad por una sola persona. Por esto se deben considerar formas disciplinadas para manejar la complejidad.

"*La complejidad del software es una propiedad esencial, no una cuestión accidental*" (Brooks).

En general la complejidad deriva de cuatro elementos :

- La complejidad del dominio del problema,
- La dificultad para administrar el proceso de desarrollo,
- La posible flexibilidad del software, y
- Los problemas de caracterizar el comportamiento de sistemas discretos.

La complejidad del dominio del problema

Los problemas que se intenta resolver en software frecuentemente involucran elementos de complejidad, tales como requerimientos que compiten entre sí, o contradictorios. A la complejidad propia del problema hay que agregarle las que surgen de *requerimientos no*

funcionales o de 'calidad', tales como usabilidad, performance, costo, continuidad en el tiempo, y confiabilidad.
Otro aspecto de la complejidad surge por el problema de comunicación entre los usuarios y los desarrolladores: los usuarios generalmente encuentran muy difícil dar en forma precisa sus necesidades de manera que los desarrolladores puedan entenderlas. En algunos casos extremos los usuarios pueden tener sólo una vaga idea de lo que desean. Los usuarios y desarrolladores tienen diferentes perspectivas de la naturaleza de la solución. En realidad, aún si los usuarios tienen perfecto conocimiento de sus necesidades, los desarrolladores tienen pocos instrumentos para capturar en forma precisa estos requerimientos. La manera común de expresar los requerimientos es con texto, acompañados por algunos diagramas. Estos documentos son difíciles de comprender, son abiertos a interpretación variada, y frecuentemente contienen elementos que son de diseño en vez de requerimientos esenciales.
Otra complicación adicional es que los requerimientos de un sistema frecuentemente cambian durante su desarrollo.

La dificultad de administrar el proceso de desarrollo

Hoy en día es habitual encontrar sistemas cuyo tamaño se mide en cientos de miles, y aún millones, de líneas de código. Ninguna persona puede entender tales sistemas en forma completa. Si estos sistemas son descompuestos de alguna manera significativa, nos podemos encontrar con cientos o miles de módulos separados. Esto hace que se utilicen grupos de desarrollo, e idealmente tan pequeño como sea posible. Más desarrolladores significa más complejidad de comunicación y de aquí más dificultad de coordinación, y en particular si el grupo está geográficamente disperso. Con un grupo de desarrollo, la clave del desafío de administración es siempre mantener una unidad e integridad de diseño.

La posible flexibilidad del software

En la industria del software existen muy pocos estándares para la construcción de código. Cada desarrollador puede construir de manera diferente una solución para un mismo problema. El desarrollo de software continua como un negocio de labor intensiva.

Los problemas de caracterizar el comportamiento de sistemas discretos

En aplicaciones grandes, puede haber cientos y aún miles de variables así como también más de un hilo de control. El conjunto completo de estas variables, sus valores actuales, las direcciones actuales y las pilas de llamada de cada proceso dentro del sistema constituyen el estado presente de la aplicación. El software es ejecutado sobre computadoras digitales, y por lo tanto constituyen un sistema con estados discretos. Los sistemas discretos tienen una cantidad finita de estados posibles, en grandes sistemas estos tienen una explosión combinatoria haciendo este número muy grande.
Los sistemas tienden a ser diseñados con una separación de intereses, para que el comportamiento de una parte del sistema tenga un impacto mínimo sobre el comportamiento de otra. Sin embargo cada evento externo al sistema tiene el potencial de colocar a este en un nuevo estado, y de esta manera el pasaje de estado a estado no es siempre determinístico. En algunas circunstancias, un evento externo puede corromper el estado de un sistema, porque sus diseñadores fallaron al tomar en cuenta ciertas interacciones entre eventos. En sistemas discretos todos los eventos externos pueden afectar cualquier parte del estado interno del sistema. Esta es la motivación primaria para un *testeo vigoroso* del sistema, pero excepto para sistemas triviales, el testeo exhaustivo es imposible. Debido a que no se cuenta con herramientas matemáticas o la capacidad intelectual para modelar el comportamiento completo de grandes sistemas, deben establecerse niveles aceptables para asegurar su validez.

Los cinco atributos de un sistema Complejo

Hay cinco atributos comunes a todo sistema complejo:

1. "Frecuentemente, la complejidad toma la forma de una jerarquía, por lo cual un sistema complejo está compuesto de subsistemas interrelacionados que tienen a la vez sus propios subsistemas, hasta que algún nivel menor de componentes elementales es alcanzado."
   La estructura jerárquica es un factor importante que permite entender, describir, y ver un sistema y sus partes.

2. “La elección de qué componentes en un sistema son primitivas es relativamente arbitrario y depende del observador del sistema.”
3. “Las vinculaciones internas de las componentes son generalmente más fuertes que las vinculaciones entre los componentes. Esto tiende a separar la dinámica de alta frecuencia de las componentes - involucrando la estructura interna de las componentes - de la dinámica de baja frecuencia - involucrando la interacción entre las componentes.”
   Esta diferencia entre interacción interna y externa de las componentes proveen una clara diferenciación de intereses entre las diferentes partes del sistema, haciendo posible estudiar cada parte en forma aislada.
4. “Los sistemas jerárquicos están generalmente compuestos de solamente pocas clases de subsistemas diferentes en varias combinaciones y órdenes.”
   Los sistemas complejos tienen patrones comunes. Estos patrones pueden requerir el reuso de pequeñas o grandes componentes.
5. “Un sistema complejo que trabaja, invariablemente ha evolucionado de un sistema simple que trabajaba... Un sistema complejo diseñado de la nada nunca trabajará, y no puede ser ‘emparchado’ para hacer que trabaje. Se debe empezar nuevamente, comenzando con un sistema simple que trabaje.”

Diseño de software

La evolución del diseño de software, como parte del proceso de desarrollo de software, es un proceso continuo que se ha ido produciendo durante las últimas tres décadas. Los primeros trabajos sobre diseño se centraron sobre los criterios para el desarrollo de programas *modulares* y los métodos para mejorar la *arquitectura* del software de una manera descendente. Los aspectos procedimentales de la definición del diseño evolucionaron hacia una filosofía denominada *programación estructurada*. Posteriores trabajos propusieron métodos para la traducción del flujo de datos o de la estructura de los datos, en una definición de diseño. Nuevos enfoques para el diseño proponen un método *orientado a objetos* para la obtención del diseño.
Cada metodología de diseño de software introduce heurísticas y notaciones propias, así como una visión algo particular de lo que caracteriza a la *calidad* del diseño. Sin embargo, todas las metodologías tienen varias características comunes :

1. Un *mecanismo* para la traducción de la representación del campo de información en una representación de diseño;
2. Una *notación* para representar los componentes funcionales y sus interfaces;
3. *Heurísticas* para el refinamiento y la partición, y
4. *Criterios* para la valoración de la calidad.

Diseño puede definirse como :

> *"Proceso iterativo de tomar un modelo lógico de un sistema junto con un conjunto de objetivos fuertemente establecidos para este sistema y producir las especificaciones de un sistema físico que satisfaga estos objetivos*." (Gane - Sarson).
>
> "*...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. Taylor)
>
> "*Proceso mediante el que se traducen los requisitos en una representación del software*" (Robert Pressman)
>
> "*Actividad por la cual un relevamiento de datos y funciones de un sistema (modelo esencial) se traduce en un plan de implementación. El modelo es volcado en una tecnología determinada*".
>
> El diseño es una *solución de software* al problema estudiado (propósito del sistema).

Objetivos del Diseño

El objetivo más importante es :

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

Pero además para lograr esto deben considerarse, entre otros, los aspectos de :

- *Rendimiento* : cuán rápido permitirá el diseño realizar el trabajo dado un recurso particular de hardware. Es decir que contemple las limitaciones del medio donde será implementado el sistema, y alcance los requerimientos de performance y uso de recursos.
- *Control* : protección contra errores humanos, máquinas defectuosas, o daños intencionales.
- Cambiabilidad : facilidad con la cual el diseño permite modificar el sistema.

Generalmente estos factores trabajan unos contra otros: un sistema con muchos controles tenderá a degradar su rendimiento, un sistema diseñado para un alto rendimiento solo podrá ser cambiado con dificultad, etc..

Además deberá:

- Satisfacer criterios de diseño sobre la *forma interna* y *externa* del producto obtenido.

- Satisfacer *restricciones* sobre el proceso de diseño en sí mismo, tales como su tiempo o costo, o las herramientas disponibles para hacer el diseño.

Una vez establecidos los requisitos del sistema, el diseño 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 validado.
Un flujo de información posible durante el desarrollo del sistema puede ser el siguiente :

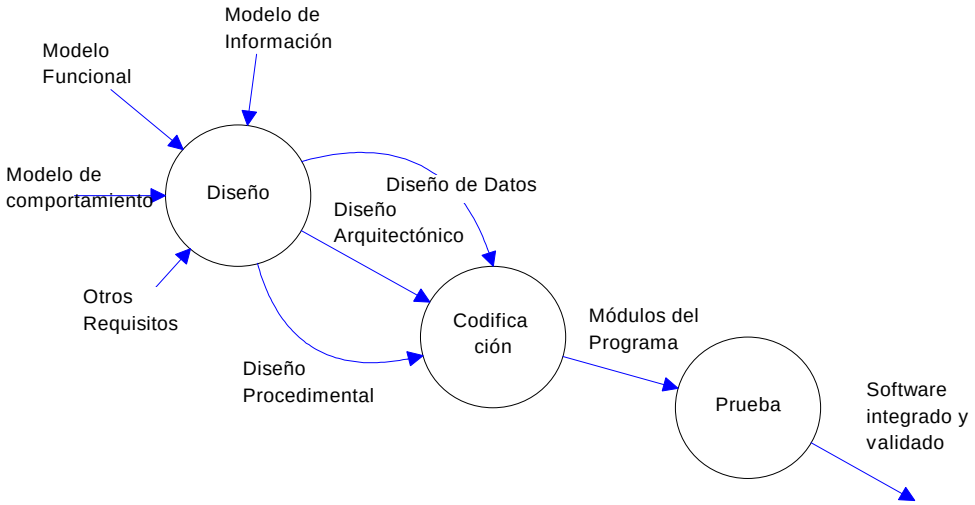


Figura II.1 – Entradas y Salidas a la fase de Diseño

Los requisitos del sistema, establecidos mediante los modelos de *información*, *funcional* y de *comportamiento*, alimentan a la fase de diseño. En esta fase se realiza el *diseño de datos*, el *diseño arquitectónico, y* el *diseño procedimental*.

- El *diseño de datos* transforma el modelo de información creado durante el análisis, en las estructuras de datos que se requerirán para implementar el sistema.
- El *diseño arquitectónico* define las relaciones entre los principales elementos estructurales del sistema, como así también las cuestiones de forma del producto.
- El *diseño procedimental* transforma los elementos estructurales en una descripción procedimental del software, especificando cada uno de los procesos a ser construidos y el sistema en su totalidad. Se genera el código fuente, y se llevan a cabo las pruebas para integrar y validar el software.

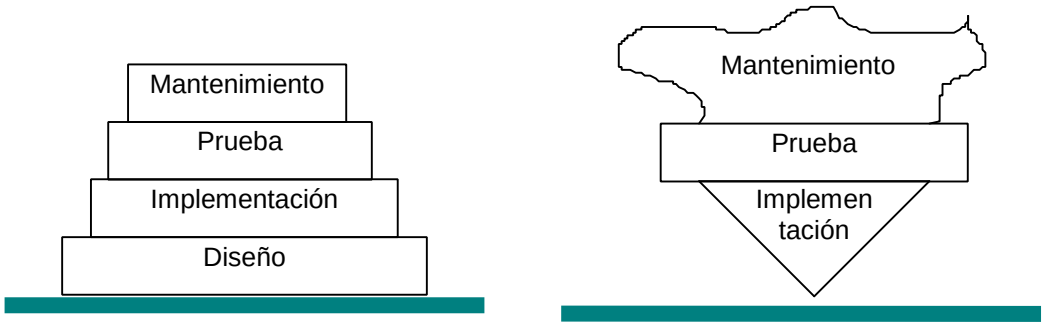
Estas fases absorben aproximadamente el 75% del costo del proceso de desarrollo de software (excluyendo el mantenimiento).

Es en el diseño donde se toman las decisiones que afectarán finalmente el éxito de la implementación del sistema, y la facilidad de mantenimiento que tendrá el software.

Y debemos tener en cuenta que:

- ✓ El diseño implica fundamentalmente *calidad*.
- ✓ El diseño es el proceso en el que se asienta la calidad del desarrollo del software.
- ✓ El diseño es la única forma mediante la cual podemos traducir con precisión los requisitos del cliente en un producto o sistema de software acabado.
- ✓ El diseño sirve como base para las etapas posteriores de desarrollo y mantenimiento.

Sin diseño, el riesgo es construir un sistema inestable, que falle cuando se realicen pequeños cambios, que pueda ser difícil de probar, un sistema cuya calidad no pueda ser evaluada hasta avanzado el proceso de desarrollo, cuando quede poco tiempo y se haya gastado ya mucho dinero.



Con diseño

Sin diseño

El proceso de diseño

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.
Desde el punto de vista metodológico, el diseño se realiza 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.

Dentro de este contexto se llevan a cabo varias actividades de diseño diferentes: diseño de datos, diseño arquitectónico, diseño de interfaz de usuario (entradas y salidas), y diseño procedimental.

En este proceso deben tenerse en cuenta los siguientes criterios de calidad de un diseño:

1. Debe exhibir una *organización jerárquica* que haga un uso inteligente del control entre los componentes del software.
2. Debe ser *modular*; esto es, el software debe estar dividido de forma lógica en elementos que realicen funciones y subfunciones específicas.
3. Debe contener representaciones distintas y separadas de los datos y de los procedimientos.
4. Debe llevar a módulos que exhiban características funcionales independientes.
5. Debe llevar a interfaces que reduzcan la complejidad de las conexiones entre módulos y el entorno exterior.
6. Debe obtenerse mediante un método que sea reproducible y que esté conducido por la información obtenida durante el análisis de los requisitos del sistema.

Estas características no se consiguen fácilmente. Aplicando principios fundamentales de diseño, una metodología sistemática y una concienzuda revisión, puede obtenerse un buen diseño.

El diseño como problema

El Diseño de software puede considerarse como un *problema* de muy difícil solución, ya que en el proceso de desarrollo de software se encuentra que:

- No puede enunciarse el problema de manera precisa y definitiva.
- No existen reglas que establezcan si se alcanzó la solución.
- Las soluciones no son “correctas” o “incorrectas”, sino que las hay “mejores” y “peores”.
- No hay criterios de evaluación objetivos.
- No se puede experimentar con las soluciones.
- Cada problema es esencialmente distinto.
- No hay límite para las soluciones ni los métodos (siempre hay nuevas herramientas y tecnologías).

Fundamentos del diseño

En las últimas tres décadas se ha establecido un conjunto de *conceptos fundamentales* para el diseño de software. Todos dan al diseñador de software una base sobre la que pueden aplicarse metodologías de diseño más o menos sofisticadas. Todos ayudan al ingeniero de software a responder las siguientes preguntas:

- ¿Qué criterios se pueden usar para partir el software en componentes individuales?
- ¿Cómo se separan los detalles de una función o de la estructura de datos de la representación conceptual del software?
- ¿Existen criterios uniformes que definen la calidad técnica de un diseño de programas?

Abstracción

Cuando se considera una solución modular para cualquier problema, pueden formularse muchos *niveles de abstracción*. En el nivel superior de abstracción, se establece una solución en términos amplios, usando el lenguaje del entorno del problema. En los niveles inferiores de abstracción se toma una orientación más procedimental. La terminología orientada al problema se acompaña con una terminología orientada a la implementación, en un esfuerzo para establecer una solución. Por último, en el nivel más bajo de abstracción, se establece la solución, de forma que pueda implementarse directamente.
Una definición que podemos considerar es:

".. 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 .."

Casa paso de un proceso de desarrollo de software es un refinamiento del nivel de abstracción de la solución de software. Durante la ingeniería del sistema, el software se considera como un elemento de un sistema basado en computadora. Durante el análisis de los requisitos del software, se establece la solución en términos de "lo que es familiar al entorno del problema". Conforme nos movemos desde lo preliminar al diseño detallado, se reduce el nivel de abstracción. Finalmente se alcanza el nivel más bajo de abstracción, cuando se genera el código fuente.
Las abstracciones que se crean son tanto de *datos* como de *procedimientos*.
Los conceptos de *refinamiento sucesivos* y *modularidad* están muy cerca del concepto de abstracción. Conforme evoluciona un diseño de software, cada nivel de módulos de la estructura del programa representa un refinamiento en el nivel de abstracción del software.

Refinamiento

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.
El proceso de refinamiento de programas propuesto por Wirth es análogo al proceso de refinamiento y de partición usado durante el análisis de requisitos. La diferencia está en el nivel de detalle que se considera y no en el método.
El refinamiento es realmente un proceso de elaboración. Comenzamos con una declaración de la función (o una descripción de la información) definida a un nivel superior de *abstracción*. Es decir, la declaración describe la función o la información conceptualmente, pero no proporciona información sobre el funcionamiento interno de la función o sobre la estructura interna de la información. El refinamiento hace que el diseñador amplíe la declaración original, dando cada vez más detalles conforme se produzcan los sucesivos refinamientos (elaboraciones).

Modularidad

El concepto de modularidad para el software de computadora se lleva teniendo en cuenta desde hace casi cuatro décadas. La arquitectura implica modularidad: esto es, 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.
Se considera que la "modularidad es el atributo individual del software que permite a un programa ser intelectualmente manejable". El software monolítico, (es decir, un gran programa compuesto de un único módulo) no puede ser fácilmente abarcado por un lector. El número de caminos de control, la expansión de las referencias, el número de variables y la complejidad global podrían hacer imposible su correcta comprensión. Para ilustrar este punto, consideremos la siguiente disquisición, basada en observaciones sobre la resolución humana de problemas.
Sea *C(x)* una función que define la complejidad de un problema x *y E(x)* una función que define el esfuerzo (en tiempo) requerido para resolver un problema x. Para dos problemas, p_1 *y* p_2, si

$$C(p_1) > C(p_2) \quad (1)$$

se deduce que

$$E(p_1) > E(p_2) \quad (2)$$

Para un caso general, este resultado es intuitivamente obvio. Se tarda más tiempo en resolver un problema difícil.
Se ha encontrado otra propiedad interesante, a partir de la experimentación sobre la resolución humana de problemas. Se trata de la siguiente:

$$C(p1 + p2) > C(p1) + C(p2) \quad (3)$$

Esta última ecuación indica que la complejidad de un problema compuesto por *p1 y p2* es mayor que la complejidad total cuando se considera cada problema por separado. Considerando la desigualdad (3) y la condición implicada por las desigualdades (1) y (2), se deduce que

$$E(p1 + P2) > E(p1) + E(p2) \quad (4)$$

Esto nos lleva a una conclusión del tipo "divide y vencerás" - es más fácil resolver un problema complejo cuando se divide en trozos más manejables. La desigualdad (4) tiene implicaciones importantes por lo que respecta a la modularidad y el software. Se trata, de hecho, de un argumento a favor de la modularidad.
Se podría concluir a partir de la desigualdad (4) que, si partiéramos el software indefinidamente, el esfuerzo requerido para desarrollarlo sería insignificantemente pequeño. Desgraciadamente, en el juego intervienen otros factores, haciendo que esa conclusión no sea (tristemente) válida. De acuerdo con la Figura II.2, el esfuerzo (costo) de desarrollo de un módulo individual disminuye conforme aumenta el número total de módulos. Para un mismo conjunto de requisitos, al haber más módulos, el tamaño de cada uno es más pequeño. Sin embargo, conforme crece el número de módulos, el esfuerzo (costo) asociado a las interfaces entre los módulos también crece. Esto nos lleva a una curva de costo o esfuerzo total como la que muestra la figura. Hay un número, M, de módulos para el que el costo de desarrollo es mínimo, pero no tenemos los medios suficientes para poder predecir M con seguridad.
Las curvas de la Figura II.2 son una guía útil cuando se considera la modularidad. Debemos modularizar, pero hemos de tener cuidado para situarnos cerca del valor M. Debe evitarse tanto una excesiva modularización como una pobre. Pero ¿cómo saber cuándo estamos "cerca del valor *M'?* ¿Cómo de modular debe hacerse el software? El tamaño de un módulo dependerá de su función y su aplicación.

Es importante tener en cuenta que un sistema se puede diseñar de forma modular, incluso aunque su implementación tenga que ser "monolítica". Hay situaciones (p. ej.: software de tiempo real, software de microprocesadores) en las que son inaceptables la menor velocidad relativa y el exceso de memoria derivados del uso de subprogramas (p. ej.: subrutinas, procedimientos). En tales situaciones, el software puede, y debe, ser diseñado considerando la

modularidad como filosofía principal. El código podrá desarrollarse sin subprogramas. Aunque el código fuente del programa pueda no parecer modular a primera vista, se habrá mantenido la filosofía y el programa tendrá las ventajas de los sistemas modulares.

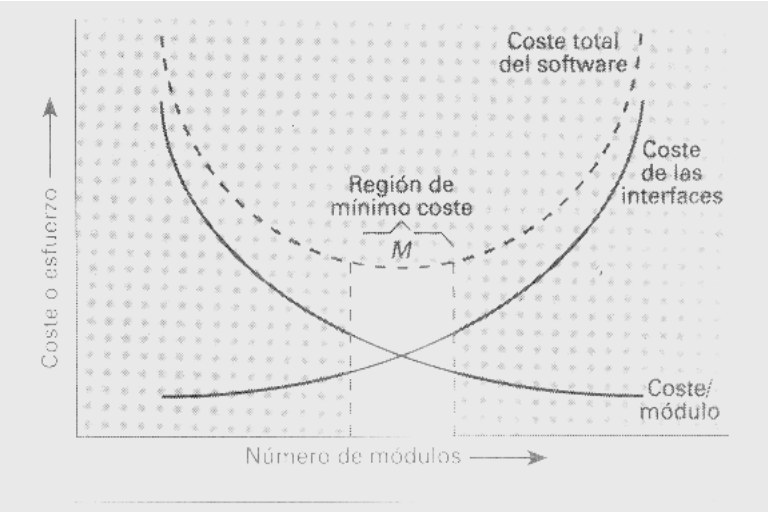


Figura II.2 – Relación Costo - Esfuerzo

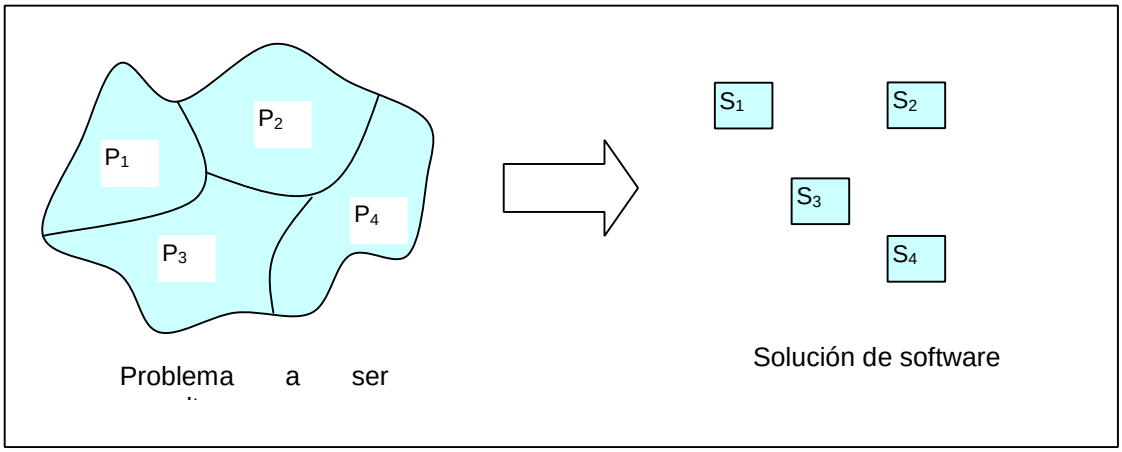
Arquitectura del software

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

(1) la estructura jerárquica de los componentes procedimentales (módulos) y
(2) la estructura de los datos.

La arquitectura del software se obtiene mediante un *proceso de partición*, que relaciona los elementos de una solución de software con partes de un problema del mundo real definido implícitamente durante el análisis de los requisitos. La evolución del software y de la estructura de los datos comienza con una definición del problema. La solución aparece cuando cada parte del problema está resuelta mediante uno o más elementos de software. Este proceso, representa una transición entre el análisis de requisitos del software y el diseño.

Un problema puede ser resuelto mediante diferentes estructuras. Se puede usar una metodología de diseño de software para obtener estructuras, pero debido a que cada una se basa en un concepto fundamental diferente de "buen diseño", cada método de diseño dará como resultado una estructura diferente, para el mismo conjunto de requisitos del software. No hay una respuesta fácil a la pregunta, "¿cuál es el mejor?". Aún no hemos llegado a esa etapa de la ciencia. Sin embargo, hay características de una estructura que se pueden examinar para determinar la calidad global.



Diseño arquitectónico

El objetivo principal del diseño arquitectónico es desarrollar una estructura de programa modular y representar las relaciones de control entre los módulos. Además, el diseño arquitectónico mezcla la estructura de programas y la estructura de datos y define las interfaces que facilitan el flujo de los datos a lo largo del programa.

Para comprender la importancia del diseño arquitectónico, consideremos una breve historia de la vida diaria:

> Usted ha decidido construir la casa de sus sueños. No teniendo experiencia en tales asuntos, visita a un arquitecto y le explica sus deseos (por ejemplo: número y tamaño de las habitaciones, estilo arquitectónico, piscina, techos especiales, etc.). El arquitecto lo escucha atentamente, le hace algunas preguntas y luego le dice que tendrá un diseño dentro de unas cuantas semanas.
>
> Como está ansioso por recibir esa llamada, se imagina muchas imágenes diferentes (y excesivamente caras) de su nueva casa. ¿Qué es lo que le presentará?. Finalmente, suena el teléfono y usted acude presuroso a la oficina.
>
> Extendiendo un gran pergamino en papel de Manila, el arquitecto le muestra un diagrama de la fontanería del baño del segundo piso y procede a explicárselo con gran detalle.
>
> "¿Pero qué hay del diseño general?", dice usted.
>
> "No se preocupe", le dice el arquitecto, "lo verá más adelante".

¿Parece algo inusual el método del arquitecto?, ¿Nos quedaríamos satisfechos con la respuesta final del arquitecto?. Por supuesto que no. Todo el mundo quiere ver primero una vista general de la casa, un plano de los pisos y otra información que nos daría una *visión arquitectónica*. Pero muchos desarrolladores de software actúan como el constructor de nuestra historia. Ellos se centran en la "fontanería" (detalles y código de los procedimientos), excluyendo la arquitectura del software.

Jerarquía de control

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

Para representar la jerarquía de control se utilizan muchas notaciones diferentes. La más común es un diagrama en forma de árbol, como el que muestra la Figura II.3. Sin embargo, también se pueden utilizar otras notaciones igualmente efectivas, tales como los diagramas de Warnier-Orr y de Jackson.

Para facilitar posteriores tratamientos de la estructura, vamos a definir algunas medidas simples y algunos términos sencillos. Refiriéndonos a la Figura II.3, la *profundidad* y la *anchura* son una indicación del número de niveles de control y de la amplitud global del control, respectivamente. El *grado de salida* es una medida del número de módulos que están directamente controlados por otros módulos. El *grado de entrada* indica cuántos módulos controlan directamente a un módulo dado.

Las relaciones de control entre los módulos se expresan de la siguiente forma: un módulo que controla a otro módulo se dice que es *superior a* él, e inversamente, un módulo controlado por otro se dice que es un *subordinado* del controlador. Por ejemplo, refiriéndonos a la Figura II.3, el módulo M es superior a los módulos *a*, *b* y *c*. El módulo *h* es subordinado del módulo *e y* en última instancia es subordinado del módulo M. Las relaciones en función de la anchura (p. ej.: entre los módulos *d* y *e*), aunque se pueden expresar en la práctica, no es necesario definirlas con una terminología explícita.

La jerarquía de control también representa dos características, sutilmente diferentes, de la arquitectura del software: la *visibilidad* y la *conectividad*. La *visibilidad* indica el conjunto de componentes del programa que pueden ser invocados o utilizados sus datos por un componente dado, incluso cuando se haga indirectamente. La *conectividad* indica el conjunto de componentes a los que directamente se invoca o se utilizan sus datos en un determinado módulo. Por ejemplo, un módulo que directamente puede provocar la ejecución de otro módulo, está conectado a ése último.

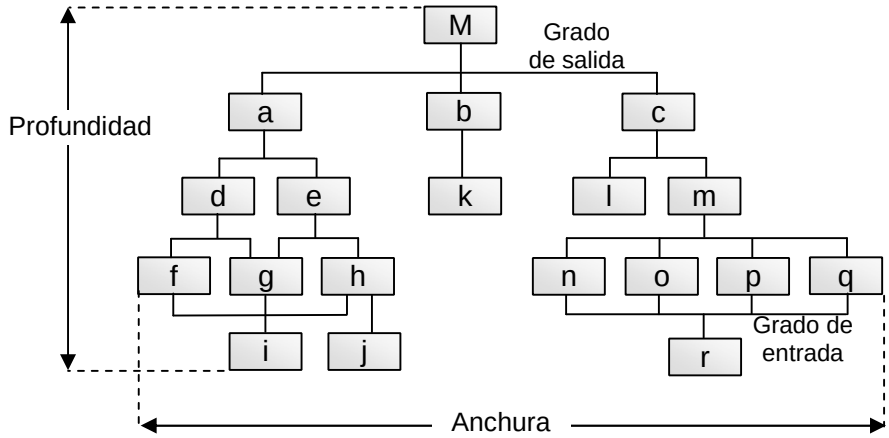


Figura II. 3 – Jerarquía de Control

Procedimientos del software

La estructura del programa define la jerarquía de control, independientemente de las decisiones y secuencias de procesamiento. El procedimiento del software, Figura II.4, 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.

Existe, por supuesto, una relación entre la estructura y el procedimiento. El procesamiento indicado para cada módulo debe incluir referencias a todos los módulos subordinados del módulo que se describe. Esto es, la representación procedimental del software se realiza por *capas*.

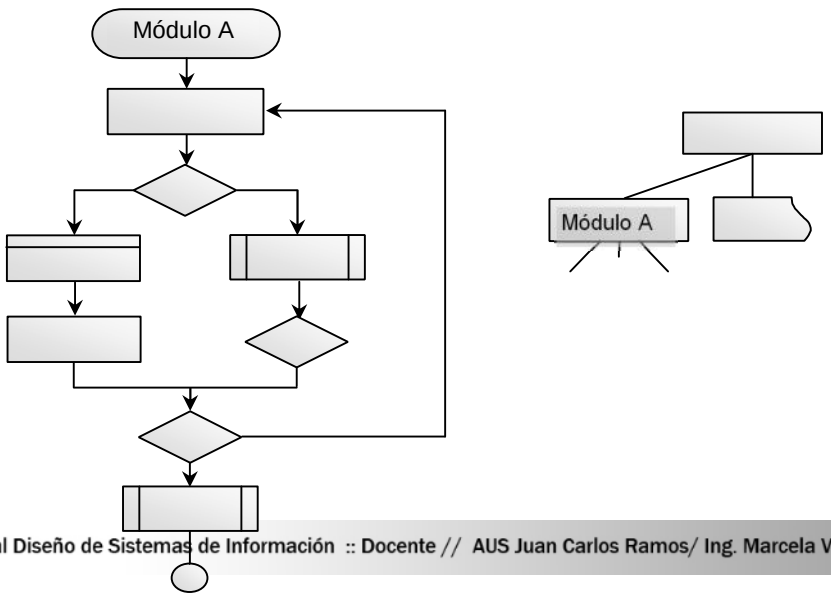


Figura II.4 – Procedimiento de Software

Ocultamiento de información

El concepto de modularidad lleva a todo diseñador de software a plantearse una pregunta fundamental: "¿cómo descomponer una solución de software obteniendo el mejor conjunto de módulos?".
El principio de *ocultamiento de infor*mación sugiere que los módulos se han de "caracterizar por decisiones de diseño que los oculten unos a otros". En otras palabras, los módulos deben especificarse y diseñarse de forma que la información (procedimientos y datos) contenida dentro de un módulo sea inaccesible a otros módulos que no necesiten tal información.
El ocultamiento implica que para conseguir una modularidad efectiva hay que definir un conjunto de módulos independientes, que se comuniquen con los otros sólo mediante la información que sea necesaria para realizar la función del software.
La *abstracción* ayuda a definir las entidades procedimentales (o de información) que componen el software. El ocultamiento establece y refuerza las restricciones de acceso a los detalles procedimentales internas de un módulo y a cualquier estructura de datos localmente utilizada en el módulo.
El uso del ocultamiento de información como criterio de diseño para los sistemas modulares, revela sus mayores beneficios cuando se hace necesario realizar *modificaciones*, durante la prueba y, más adelante, el *mantenimiento* del software. Debido a que la mayoría de los datos y de los procedimientos estarán ocultos a otras partes del software, será menos probable que los errores introducidos inadvertidamente durante la modificación se propaguen a otros lugares del software.

Eficacia y Eficiencia

Eficacia se entiende la medida en que se alcanzan los objetivos prefijados. En tanto *eficiencia* es la relación insumo/producto involucrado en la ejecución de determinado sistema.
De esta manera se dice que un sistema es ***efectivo*** si logra los resultados previstos como meta. Por ejemplo, diremos que es efectivo si proporciona la información necesaria en la oportunidad fijada.
Por otra parte, un sistema es ***eficiente*** si los resultados obtenidos son más valiosos que los recursos empleados en su obtención. Así, el grado de eficiencia estará en relación con los insumos: horas hombre, horas máquina, elementos materiales, etc., utilizados. Un sistema *eficaz* (efectivo) puede ser eficiente o ineficiente en la medida del aprovechamiento de los insumos. No tiene sentido hablar de eficiencia en un sistema que no es eficaz.
La importancia de estos términos radica en que la labor fundamental en el proceso de desarrollo de sistemas es determinar la *eficacia* de un sistema. El estudio de la eficacia debe preceder al estudio de la eficiencia. Ambos conceptos estarán presentes a lo largo de todo el proceso de desarrollo.
Aunque la eficiencia es un fin recomendable, se debe tener en cuenta que:

1. La eficiencia es un *requisito de rendimiento*, y como tal se debe establecer durante el análisis de requisitos del sistema. El sistema debe ser tan eficiente como se quiera, no tan eficiente como sea humanamente posible.
2. La eficiencia se incrementa con un buen diseño.

La eficiencia del código y la simplicidad del código van de la mano. En general no hay que sacrificar la claridad, la legibilidad o la corrección en aras de unas mejoras en eficiencia que no sean esenciales.

Vimos hasta aquí los principales aspectos relacionados fundamentalmente con lo que denominamos *diseño arquitectónico* y *diseño procedimental*, que podríamos denominar “diseño interno”, para diferenciarlo de lo que podemos llamar “diseño externo”, y que involucra el aspecto visible del software, lo que los usuarios ven, y que es tan importante como el diseño interno, y para el que deben respetarse y seguirse lineamientos establecidos para que el software sea aceptado (uno de los aspectos más conocidos es el de “amigabilidad”). El diseño

externo cubrirá los aspectos de interfaces de usuarios y salidas del sistema. Estos aspectos abordaremos más adelante.

Referencias - Bibliografía

- "Ingeniería del Software – Un enfoque práctico", Roger. S. Pressman, Mc Graw-Hill, 1993
- "Análisis Estructurado Moderno", Edward Yourdon, Prentice Hall, 1989
- "Object-oriented analysis and design with applications", Grady Booch, Benjamin-Cummings, 1994.