CAPÍTULO

13 CONCEPTOS Y PRINCIPIOS DE DISEÑO

EL objetivo de los diseñadores es producir un modelo o representación de una entidad que se será construida a posteriori. Belady describe el proceso mediante el cual se desarrolla el modelo de diseño [BEL81]:

> En cualquier proceso de diseño existen dos fases importantes: la diversificación y la convergencia. La diversificación es la ***adquisición*** de un repertorio de alternativas, de un material primitivo de diseño: componentes, soluciones de componentes y conocimiento, todo dentro de catálogos, de libros de texto y en la mente. Durante la convergencia, el diseñador elige y combina los elementos adecuados y extraídos de este repertorio para satisfacer los objetivos del diseño, de la misma manera a como se establece en el documento de los requisitos, y de la manera en que se acordó con el cliente. La segunda fase es la ***eliminación*** gradual de cualquier configuración de componentes excepto de una en particular, y de aquí la creación del producto final.

La diversificación y la convergencia combinan intuición y juicio en función de la experiencia en construir entidades similares; un conjunto de principios y/o heurística que proporcionan la forma de guiar la evolución del modelo; un conjunto de criterios que posibilitan la calidad que se va **a** juzgar, y un proceso de iteración que por Último conduce a una representación final de diseño.

VISTAZO RÁPIDO

¿Qué es? El diseño es una representación significativa de ingeniería de algo que se va a construir. Se puede hacer el seguimiento basándose en los requisitos del cliente, y al mismo tiempo la calidad se puede evaluar y cotejar con el conjunto de criterios predefinidos para obtener un diseño «bueno». En el contexto de la ingeniería del software, el diseño se centra en cuatro áreas importantes de interés: datos, arquitectura, interfaces y componentes. En estas cuatro áreas se aplican los principios y conceptos que se abarcan en este capítulo

¿Quién lo hace? El ingeniero del software es quien diseña los sistemas basados en computadora, pero los conocimientos que se requieren en cada nivel de diseño funcionan de diferentes maneras. En el nivel de datos y de arquitectura, el diseño se centra en los patrones de la misma manera a como se aplican en la aplicación que se va a construir. En el nivel de la interfaz, es la ergonómica humana la que dicta nuestro enfoque de diseño. Y en el nivel de componentes, un «enfoque de programación» conduce a diseños de datos y procedimentales eficaces.

¿Por qué es importante? Si se construye una casa, ¿se hace sin un plano? Se correrían riesgos, se cometerían errores, habría un plano de casa sin sentido, con ventanas y puertas en sitios equivocados... un desastre. El software de computadora es considerablemente más complejo que una casa, de aquí que necesitemos un plano —el diseño—.

¿Cuáles son los pasos? El diseño comienza con el modelo de los requisitos. Se trabaja por transformar este modelo y obtener cuatro niveles de detalles de diseño: la estructura de datos, la arquitectura del sistema, la representación de la interfaz y los detalles a nivel de componentes. Durante cada una de las actividades del diseño, se aplican los conceptos y principios básicos que llevan a obtener una alta calidad.

¿Cuál es el producto obtenido? Por último se produce una especificación del diseño. La especificación se compone de los modelos del diseño que describen los datos, arquitectura, interfaces y componentes. Cada una de estas partes es lo que forma el producto obtenido del proceso de diseño.

¿Cómo puedo estar seguro de que lo he hecho correctamente? En cada etapa se revisan los productos del diseño del software en cuanto a claridad, corrección, finalización y consistencia, y se comparan con los requisitos y unos con otros.

El diseño del software, al igual que los enfoques de diseño de ingeniería en otras disciplinas, va cambiando continuamente a medida que se desarrollan métodos nuevos, análisis mejores y se amplia el conocimiento. Las metodologías de diseño del software carecen de la profundidad, flexibilidad y naturaleza cuantitativa que se asocian normalmente a las disciplinas de diseño de ingeniería más clásicas. Sin embargo, **sí** existen métodos para el diseño del software; también se dispone de calidad de diseño y se pueden aplicar notaciones de diseño. En este capítulo se explorarán los conceptos y principios fundamentales que se pueden aplicar a todo diseño de software. En los Capítulos **14,** 15, 16 y 22 se examinan diversos métodos de diseño de software en cuanto a la manera en que **se** aplican al diseño arquitectónico, de interfaz y a nivel de componentes.

13.1 DISEÑO DEL SOFTWARE E INGENIERÍA DEL SOFTWARE

El diseño del software se encuentra en el núcleo técnico de la ingeniería del software y se aplica independientemente del modelo de diseño de software que se utilice. Una vez que se analizan y especifican los requisitos del software, el diseño del software es la primera de las tres actividades técnicas – diseño, generación de código y pruebas — que se requieren para construir y verificar el software. Cada actividad transforma la información de manera que dé lugar por último a un software de computadora validado.

> **Cita:**
> Los milagros más comunes de la ingeniería del software son las transiciones desde el análisis hasta el diseño, y desde el diseño al código.
> **Richard Due**

Cada uno de los elementos del modelo de análisis (Capítulo 12) proporciona la información necesaria para crear los cuatro modelos de diseño que se requieren para una especificación completa de diseño. El flujo de información durante el diseño del software se muestra en la Figura **13.1.** Los requisitos del software, manifestados por los modelos de datos funcionales y de comportamiento, alimentan la tarea del diseño. Mediante uno de los muchos métodos de diseño (que se abarcarán en capítulos posteriores) la tarea de diseño produce un diseño de datos, un diseño arquitectónico, un diseño de interfaz y un diseño de componentes.

El *diseño de datos* transforma el modelo del dominio de información que se crea durante el análisis en las estructuras de datos que se necesitarán para implementar el software. Los objetos de datos y las relaciones definidas en el diagrama relación entidad y el contenido de datos detallado que se representa en el diccionario de datos proporcionan la base de la actividad del diseño de datos. Es posible que parte del diseño de datos tenga lugar junto con el diseño de la arquitectura del software. **A** medida que se van diseñando cada uno de los componentes del software, van apareciendo más detalles de diseño.

El *diseño arquitectónico* define la relación entre los elementos estructurales' principales del software, los patrones de diseño que se pueden utilizar para lograr los requisitos que se han definido para el sistema, y las restricciones que afectan a la manera en que se pueden aplicar los patrones de diseño arquitectónicos [SHA96]. La representación del diseño arquitectónico —el marco de trabajo de un sistema basado en computadora — puede derivarse de la especificación del sistema, del modelo de análisis y de la interacción del subsistema definido dentro del modelo de análisis.

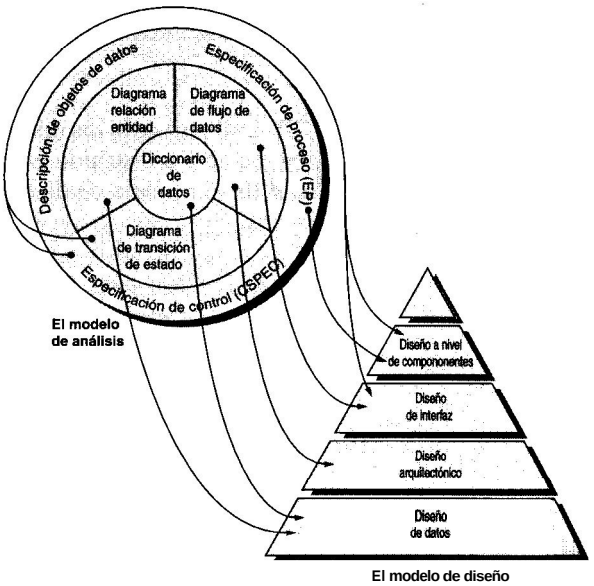


FIGURA 13.1. Conversión del modelo de análisis en un diseño de software.

El *diseño de la interfaz* describe la manera de comunicarse el software dentro de sí mismo, con sistemas que interoperan dentro de él y con las personas que lo utilizan. Una interfaz implica un flujo de información (por ejemplo, datos y/o control) y un tipo específico de comportamiento. Por tanto, los diagramas de flujo de control y de datos proporcionan gran parte de la información que se requiere para el diseño de la interfaz.

El *diseno a nivel de componentes* transforma los elementos estructurales de la arquitectura del software en una descripción procedimental de los componentes del software. La información que se obtiene de EP, EC y de DTE sirve como base para el diseño de los componentes.

La importancia del diseño del software se puede describir con una sola palabra —*calidad*—. El diseño es el lugar en donde se fomentará la calidad en la ingeniería del software. El diseño proporciona las representaciones del software que se pueden evaluar en cuanto a calidad. El diseño es la única forma de convertir exactamente los requisitos de un cliente en un producto o sistema de software finalizado. El diseño del software sirve como fundamento para todos los pasos siguientes del soporte del software y de la ingeniería del software. Sin un diseño, corremos el riesgo de construir un sistema inestable —un sistema que fallará cuando se lleven a cabo cambios; un sistema que puede resultar difícil de comprobar; y un sistema cuya calidad no puede evaluarse hasta muy avanzado el proceso, sin tiempo suficiente y con mucho dinero gastado en él—.

13.2 EL PROCESO DE DISEÑO

El diseño del software es un proceso iterativo mediante el cual los requisitos se traducen en un «plano» para construir el software. Inicialmente, el plano representa una visión holística del software. Esto es, el diseño se representa a un nivel alto de abstracción —un nivel que puede rastrearse directamente hasta conseguir el objetivo del sistema específico y según unos requisitos más detallados de comportamiento, funcionales y de datos—. A medida que ocurren las iteraciones del diseño, el refinamiento subsiguiente conduce a representaciones de diseño a niveles de abstracción mucho más bajos. Estos niveles se podrán rastrear aún según los requisitos, pero la conexión es más sutil.

13.2.1. Diseño y calidad del software

A lo largo de todo el proceso del diseño, la calidad de la evolución del diseño se evalúa con una serie de revisiones técnicas formales o con las revisiones de diseño abordadas en el Capítulo 8. McGlaughlin [MCG91] sugiere tres características que sirven como guía para la evaluación de un buen diseño:

- el diseño deberá implementar todos los requisitos explícitos del modelo de análisis, y deberán ajustarse a todos los requisitos implícitos que desea el cliente;
- el diseño deberá ser una guía legible y comprensible para aquellos que generan código y para aquellos que comprueban y consecuentemente, dan soporte al software;
- el diseño deberá proporcionar una imagen completa del software, enfrentándose a los dominios de comportamiento, funcionales y de datos desde una perspectiva de implementación.

> **Cita:**
> Para lograr un buen diseño, hay que pensar en la manera correcta de llevar a cabo la actividad de diseño.
> **Katharine Whitehead**

Con el fin de evaluar la calidad de una representación de diseño, deberán establecerse los criterios técnicos para un buen diseño. Más adelante en este mismo Capítulo, se abordarán más detalladamente los criterios de calidad del diseño. Por ahora se presentarán las siguientes directrices:

1. Un diseño deberá presentar una estructura arquitectónica que (1) se haya creado mediante patrones de diseño reconocibles, (2) que esté formada por componentes que exhiban características de buen diseño (aquellas que se abordarán más adelante en este mismo capítulo), y (3) que se puedan implementar de manera evolutiva, facilitando así la implementación y la comprobación.
2. Un diseño deberá ser modular; esto es, el software deberá dividirse lógicamente en elementos que realicen funciones y subfunciones específicas.

¿Existen directrices generales que lleven a un buen diseño?

3. Un diseño deberá contener distintas representaciones de datos, arquitectura, interfaces y componentes (módulos).
4. Un diseño deberá conducir a estructuras de datos adecuadas para los objetos que se van a implementar y que procedan de patrones de datos reconocibles.
5. Un diseño deberá conducir a componentes que presenten características funcionales independientes.
6. Un diseño deberá conducir a interfaces que reduzcan la complejidad de las conexiones entre los módulos y con el entorno externo.
7. Un diseño deberá derivarse mediante un método repetitivo y controlado por la información obtenida durante el análisis de los requisitos del software.

Estos criterios no se consiguen por casualidad. El proceso de diseño del software fomenta el buen diseño a través de la aplicación de principios de diseño fundamentales, de metodología sistemática y de una revisión cuidadosa.

> **Cita:**
> Hay dos formas de construir un diseño de software: una forma es hacer un diseño tan simple que no existan obviamente deficiencias, y la otra es hacer un diseño tan complicado que no existan deficiencias obvias. La primera forma es mucho más difícil.
> **C.A.R. Hoare**

13.2.2. La evolución del diseño del software

La evolución del diseño del software es un proceso continuo que ha abarcado las últimas cuatro décadas. El primer trabajo de diseño se concentraba en criterios para el desarrollo de programas modulares [DEN73] y métodos para refinar las estructuras del software de manera descendente [WIR71]. Los aspectos procedimentales de la definición de diseño evolucionaron en una filosofía denominada *programación estructurada* [DAH71, MIL72]. Un trabajo posterior propuso métodos para la conversión del flujo de datos [STE74] o estructura de datos [JAC75, WAR74] en una definición de diseño. Enfoques de diseño más recientes hacia la derivación de diseño proponen un método orientado a objetos. Hoy en día, se ha hecho hincapié en un diseño de software basado en la arquitectura del software [GAM95, BUS96, BRO98].

Independientemente del modelo de diseño que se utilice, un ingeniero del software deberá aplicar un conjunto de principios fundamentales y conceptos básicos para el diseño a nivel de componentes, de interfaz, arquitectónico y de datos. Estos principios y conceptos se estudian en la sección siguiente.

13.1 PRINCIPIOS DEL DISEÑO

El diseño de software es tanto un proceso como un modelo. El *proceso* de diseño es una secuencia de pasos que hacen posible que el diseñador describa todas los aspectos del software que se va construir. Sin embargo, es importante destacar que el proceso de diseño simplemente no es un recetario. Un conocimiento creativo, experiencia en el tema, un sentido de lo que hace que un software sea bueno, y un compromiso general con la calidad son factores críticos de éxito para un diseño competente.

El *modelo* de diseño es el equivalente a los planes de un arquitecto para una casa. Comienza representando la totalidad de todo lo que se va a construir (por ejemplo, una representación en tres dimensiones de la casa) y refina lentamente lo que va a proporcionar la guía para construir cada detalle (por ejemplo, el diseño de fontanería). De manera similar, el modelo de diseño que se crea para el software proporciona diversas visiones diferentes de software de computadora.

Los principios básicos de diseño hacen posible que el ingeniero del software navegue por el proceso de diseño. Davis [DAV95] sugiere un conjunto' de principios para el diseño del software, los cuales han sido adaptados y ampliados en la lista siguiente:

- *En el proceso de diseño no deberá utilizarse «orejeras».* Un buen diseñador deberá tener en cuenta enfoques alternativos, juzgando todos los que se basan en los requisitos del problema, los recursos disponibles para realizar el trabajo y los conceptos de diseño presentados en la Sección **13.4.**
- *El diseño deberá poderse rastrear hasta el modelo de análisis.* Dado que un solo elemento del modelo de diseño suele hacer un seguimiento de los múltiples requisitos, es necesario tener un medio de rastrear cómo se han satisfecho los requisitos por el modelo de diseño.
- *El diseño no deberá inventar nada que ya esté inventado.* Los sistemas se construyen utilizando un conjunto de patrones de diseño, muchos de los cuales probablemente ya se han encontrado antes. Estos patrones deberán elegirse siempre como una alternativa para reinventar. Hay poco tiempo y los recursos son limitados. El tiempo de diseño se deberá invertir en la representación verdadera de ideas nuevas y en la integración de esos patrones que ya existen.
- *El diseño deberá «minimizar la distancia intelectual» [DAV95] entre el software y el problema como si de la misma vida real se tratara.* Es decir, la estructura del diseño del software (siempre que sea posible) imita la estructura del dominio del problema.
- *El diseño deberá presentar uniformidad e integración.* Un diseño es uniforme si parece que fue una persona la que lo desarrolló por completo. Las reglas de estilo y de formato deberán definirse para un equipo de diseño antes de comenzar el trabajo sobre el diseño. Un diseño se integra si se tiene cuidado a la hora de definir interfaces entre los componentes del diseño.
- *El diseño deberá estructurarse para admitir cambios.* Los conceptos de diseño estudiados en la sección siguiente hacen posible un diseño que logra este principio.
- *El diseño deberá estructurarse para degradarse poco a poco, incluso cuando se enfrenta con datos, sucesos o condiciones de operación aberrantes.* Un software bien diseñado no deberá nunca explotar como una «bomba». Deberá diseñarse para adaptarse a circunstancias inusuales, y si debe terminar de funcionar, que lo haga de forma suave.

PUNTO CLAVE

La consistencia del diseño y la uniformidad es crucial cuando se van a construir sistemas grandes. Se deber6 establecer un conjunto de reglas de diseño para el equipo del software antes de comenzar a trabajar.

- *El diseño no es escribir código y escribir código no es diseñar.* Incluso cuando se crean diseños procedimentales para componentes de programas, el nivel de abstracción del modelo de diseño es mayor que el código fuente. Las únicas decisiones de diseño realizadas a nivel de codificación se enfrentan con pequeños datos de implementación que posibilitan codificar el diseño procedimental.
- *El diseño deberá evaluarse en función de la calidad mientras se va creando, no después de terminarlo.* Para ayudar al diseñador en la evaluación de la calidad se dispone de conceptos de diseño (Sección **13.4)** y de medidas de diseño (Capítulos 19 y 24).
- *El diseño deberá revisarse para minimizar los errores conceptuales (semánticos).* A veces existe la tendencia de centrarse en minucias cuando se revisa el diseño, olvidándose del bosque por culpa de los árboles. Un equipo de diseñadores deberá asegurarse de haber afrontado los elementos conceptuales principales antes de preocuparse por la sintaxis del modelo del diseño.

Referencia cruzada

En el Capítulo 8 se presentan las directrices para llevar a cabo revisiones de diseño efectivas.

[1] Aquí solo se destaca un pequeño subconjunto de los principios de diseño **de** Davis. Para mas información, véase [DAV95].

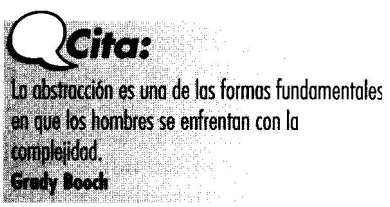
Cuando los principios de diseño descritos anteriormente se aplican adecuadamente, el ingeniero del software crea un diseño que muestra los factores de calidad tanto internos como externos [MEY88]. *Los factores de calidad externos* son esas propiedades del software que pueden ser observadas fácilmente por los usuarios (por ejemplo, velocidad, fiabilidad, grado de corrección, usabilidad)[2]. Los *factores de calidad internos* tienen importancia para los ingenieros del software. Desde una perspectiva técnica conducen a un diseño de calidad alta. Para lograr los factores de calidad internos, el diseñador deberá comprender los conceptos de diseño básicos.

13.4 CONCEPTOS DEL DISEÑO

Durante las últimas cuatro décadas se ha experimentado la evolución de un conjunto de conceptos fundamentales de diseño de software. Aunque el grado de interés en cada concepto ha variado con los años, todos han experimentado el paso del tiempo. Cada uno de ellos proporcionarán la base de donde el diseñador podrá aplicar los métodos de diseño más sofisticados. Cada uno ayudará al ingeniero del software a responder las preguntas siguientes:

- ¿Qué criterios se podrán utilizar para la partición del software en componentes individuales?
- ¿Cómo se puede separar la función y la estructura de datos de una representación conceptual del software?
- ¿Existen criterios uniformes que definen la calidad técnica de un diseño de software?

M.A. Jackson una vez dijo: «El comienzo de la sabiduría para un ingeniero del software es reconocer la diferencia entre hacer que un programa funcione y conseguir que lo haga correctamente».[JAC875] *Los* conceptos de diseño de software fundamentales proporcionan el marco de trabajo necesario para conseguir q u e lo haga correctamente».



13.4.1. Abstracción

Cuando se tiene en consideración una solución modular a cualquier problema, se pueden exponer muchos *niveles de abstracción.* En el nivel más alto de abstracción, la solución se pone como una medida extensa empleando el lenguaje del entorno del problema. En niveles inferiores de abstracción, se toma una orientación más procedimental. La terminología orientada a problemas va emparejada con la terminología orientada a la implementación en un esfuerzo por solucionar el problema. Finalmente, en el nivel más bajo de abstracción, se establece la solución para poder implementarse directamente. Wasserman [WAS83] proporciona una definición útil:

> La noción psicológica de «abstracción» permite concentrarse en un problema a algún nivel de generalización **sin** tener en consideración los datos irrelevantes de bajo nivel; la utilización de la abstracción también permite trabajar con conceptos y términos que son familiares en el entorno del problema sin tener que transformarlos en una estructura no familiar.. .

Cada paso del proceso del software es un refinamiento en el nivel de abstracción de la solución del software. Durante la ingeniería del sistema, el software se asigna como un elemento de un sistema basado en computadora. Durante el análisis de los requisitos del software, la solución del software se establece en estos términos: «aquellos que son familiares en el entorno del problema». A medida que nos adentramos en el proceso de diseño, se reduce el nivel de abstracción. Finalmente el nivel de abstracción más bajo se alcanza cuando se genera el código fuente.

A medida que vamos entrando en diferentes niveles de abstracción, trabajamos para crear abstracciones procedimentales y de datos. Una *abstracción procedimental* es una secuencia nombrada de instrucciones que tiene una función específica y limitada. Un ejemplo de abstracción procedimental sería la palabra «abrir» para una puerta. «Abrir» implica una secuencia larga de pasos procedimentales (por ejemplo, llegar a la puerta; alcanzar **y** agarrar el pomo de la puerta; girar el pomo y tirar de la puerta; separarse al mover la puerta, etc.).

Como diseñador, trabaje mucho y duro poro derivar abstracciones tonto procedimentales como de datos que sirvan poro el problema que tengo en ese momento, pero que también se puedon volver a utilizar en otros situaciones.

Una *abstracción de datos* es una colección nombrada de datos que describe un objeto de datos (Capítulo 12). En el contexto de la abstracción procedimental *abrir,* podemos definir una abstracción de datos llamada **puerta.** Al igual que cualquier objeto de datos, la

[2] En el Capítulo 19 se presenta un estudio más detallado sobre los factores de calidad.

abstracción de datos para **puerta** acompañaría a un conjunto de atributos que describen esta puerta (por ejemplo, tipo de puerta, dirección de apertura, mecanismo de apertura, peso, dimensiones). Se puede seguir diciendo que la abstracción procedimental *abrir* hace uso de la información contenida en los atributos de la abstracción de datos **puerta.**

La *abstracción de control* es la tercera forma de abstracción que se utiliza en el diseño del software. Al igual que las abstracciones procedimentales y de datos, este tipo de abstracción implica un mecanismo de control de programa sin especificar los datos internos. Un ejemplo de abstracción de control es el semáforo de sincronización [KAI83] que se utiliza para coordinar las actividades en un sistema operativo. El concepto de abstracción de control se estudia brevemente en el Capítulo 14.

13.4.2. Refinamiento

El *refinamiento paso a paso* es una estrategia de diseño descendente propuesta originalmente por Niklaus Wirth [WIR71]. El desarrollo de un programa se realiza refinando sucesivamente los niveles de detalle procedimentales. Una jerarquía se desarrolla descomponiendo una sentencia macroscópica de función (una abstracción procedimental) paso a paso hasta alcanzar las sentencias del lenguaje de programación. Wirth [WIR71] proporciona una visión general de este concepto:

> En cada paso (del refinamiento), se descompone una o varias instrucciones del programa dado en instrucciones más detalladas. Esta descomposición sucesiva o refinamiento de especificaciones termina cuando todas las instrucciones se expresan en función de cualquier computadora subyacente o de cualquier lenguaje de programación.. . De **la** misma manera que se refinan las tareas, los datos también se tienen que refinar, descomponer o estructurar, y es natural refinar el programa y las especificaciones de los datos en paralelo.
>
> Todos **los** pasos del refinamiento implican decisiones de diseño. **Es** importante que.. . el programador conozca los criterios subyacentes (para decisiones de diseño) y **la** existencia de soluciones alternativas., .

El proceso de refinamiento de programas propuesto por Wirth es análogo al proceso de refinamiento y de partición que se utiliza durante el análisis de requisitos. La diferencia se encuentra en el nivel de detalle de implementación que se haya tomado en consideración, no en el enfoque.

Existe la tendencia de entrar en detalle inmediatamente, saltándose los pasos de refinamiento. Esto conduce o errores y omisiones y hace que el diseño seo más difícil de revisar. Realice el refinamiento paso a paso.

El refinamiento verdaderamente es un proceso de *elaboración.* Se comienza con una sentencia de función (o descripción de información) que se define a un nivel alto de abstracción. Esto es, la sentencia describe la función o información conceptualmente, pero no proporciona información sobre el funcionamiento interno de la información. El refinamiento hace que el diseñador trabaje sobre la sentencia original, proporcionando cada vez más detalles a medida que van teniendo lugar sucesivamente todos y cada uno de los refinamientos (elaboración).

13.4.3. Modularidad

El concepto de modularidad se ha ido exponiendo desde hace casi cinco décadas en el software de computadora. La arquitectura de computadora (descrita en la Sección 13.4.4) expresa la modularidad; es decir, el software se divide en componentes nombrados y abordados por separado, llamados frecuentemente *módulos,* que se integran para satisfacer los requisitos del problema.

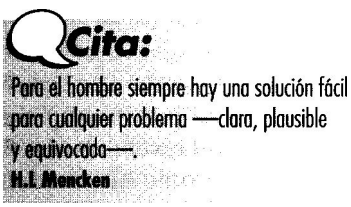
Se ha afirmado que «la modularidad es el Único atributo del software que permite gestionar un programa intelectualmente»[MYE78]. El software monolítico (es decir, un programa grande formado por un Único módulo) no puede ser entendido fácilmente por el lector. La cantidad de rutas de control, la amplitud de referencias, la cantidad de variables y la complejidad global hará que el entendimiento esté muy cerca de ser imposible. Para ilustrar este punto, tomemos en consideración el siguiente argumento basado en observaciones humanas sobre la resolución de problemas.

Pensemos que $C(x)$ es una función que define la complejidad percibida de un problema x, y que $E(x)$ es una función que define el esfuerzo (oportuno) que se requiere para resolver un problema x. Para dos problemas p_1 y p_2, si

$$C(p_1) > C(p_2) \qquad (13.1a)$$

implica que

$$E(p_1) > E(p_2) \qquad (13.1b)$$



En general, este resultado es por intuición obvio. Se tarda más en resolver un problema difícil.

Mediante la experimentación humana en la resolución de problemas se ha averiguado otra característica interesante. Esta es,

$$C(p_1 + p_2) > C(p_1) + C(p_2) \qquad (13.2)$$

La ecuación (13.2) implica que la complejidad percibida de un problema que combina p_1 y p_2 es mayor

que la complejidad percibida cuando se considera cada problema por separado. Teniendo en cuenta la ecuación (13.2) y la condición implicada por la ecuación (13.1), se establece que

$$E(p_1 + p_2) > E(p_1) + E(p_2) \qquad (13.3)$$

Esto lleva a una conclusión: «divide y vencerás» —es más fácil resolver un problema complejo cuando se rompe en piezas manejables—. El resultado expresado en la ecuación (13.3) tiene implicaciones importantes en lo que respecta a la modularidad y al software. Es, de hecho, un argumento para la modularidad.

No modularice de más. La simplicidad de cada módulo se eclipsará con la complejidad de la integración.

Es posible concluir de la ecuación (13.3) que si subdividimos el software indefinidamente, el esfuerzo que se requiere para desarrollarlo sería mínimo. Desgraciadamente, intervienen otras fuerzas, que hacen que esta conclusión sea (tristemente) falsa. Como muestra la Figura 13.2, el esfuerzo (coste) para desarrollar un módulo de software individual disminuye a medida que aumenta el número total de módulos. Dado el mismo conjunto de requisitos, tener más módulos conduce a un tamaño menor de módulo. Sin embargo, a medida que aumenta el número de módulos, también crece el esfuerzo (coste) asociado con la integración de módulos. Estas características conducen también a la curva total del coste o esfuerzo que se muestra en la figura. Existe un número M de módulos que daría como resultado un coste mínimo de desarrollo, aunque no tenemos la sofisticación necesaria para predecir M con seguridad.

Las curvas que se muestran en la Figura 13.2 proporcionan en efecto una guía útil cuando se tiene en consideración la modularidad. La modularidad deberá aplicarse, pero teniendo cuidado de estar próximo a M. Se deberá evitar modularizar de más o de menos. Pero, ¿cómo conocemos el entorno de M? ¿Cuánto se deberá modularizar el software? Para responder a estas preguntas se deberán comprender los conceptos de diseño que se estudiarán más adelante dentro de este capítulo.

Referencia cruzada

Los métodos de diseño se estudian en los Capítulos 14, 15, 16 y 22.

Cuando se tiene en consideración la modularidad surge otra pregunta importante. ¿Cómo se define un módulo con un tamaño adecuado? La respuesta se encuentra en los métodos utilizados para definir los módulos dentro de un sistema. Meyer [MEY88] define cinco criterios que nos permitirán evaluar un método de diseño en relación con la habilidad de definir un sistema modular efectivo:

¿Cómo se puede evaluar un método de diseño para determinar si va a conducir a una modularidad efectiva?

Capacidad de descomposición modular. Si un método de diseño proporciona un mecanismo sistemático para descomponer el problema en subproblemas, reducirá la complejidad de todo el problema, consiguiendo de esta manera una solución modular efectiva.

Capacidad de empleo de componentes **modulares**. Si un método de diseño permite ensamblar los componentes de diseño (reusables) existentes en un sistema nuevo, producirá una solución modular que no inventa nada ya inventado.

Capacidad de comprensión modular. Si un módulo se puede comprender como una unidad autónoma (sin referencias a otros módulos) será más fácil de construir y de cambiar.

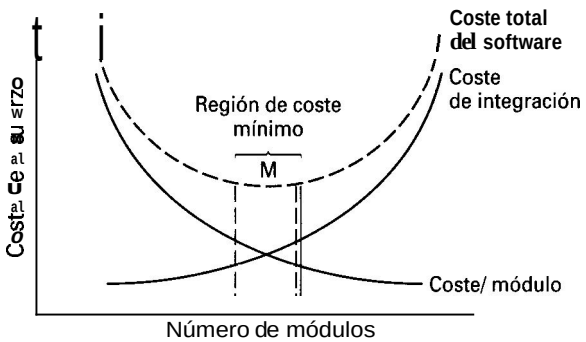


FIGURA 13.2. Modularidad y costes de software.

Continuidad modular. Si pequeños cambios en los requisitos del sistema provocan cambios en los módulos individuales, en vez de cambios generalizados en el sistema, se minimizará el impacto de los efectos secundarios de los cambios.

Protección modular. Si dentro de un módulo se produce una condición aberrante y sus efectos se limitan a ese módulo, se minimizará el impacto de los efectos secundarios inducidos por los errores.

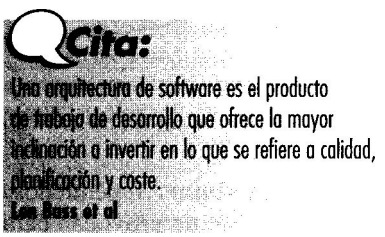
Finalmente, es importante destacar que un sistema se puede diseñar modularmente, incluso aunque su implementación deba ser «monolítica». Existen situaciones (por ejemplo, software en tiempo real, software empotrado) en donde no es admisible que los subprogramas introduzcan sobrecargas de memoria y de velocidad por mínimos que sean (por ejemplo, subrutinas, procedimientos). En tales situaciones el software podrá y deberá diseñarse con modularidad como filosofía predominante. El código se puede desarrollar «en línea». Aunque el código fuente del programa puede no tener un aspecto modular a primera vista, se ha mantenido la filosofía y el programa proporcionará los beneficios de un sistema modular.

13.4.4. Arquitectura del software

La *arquitectura del software* alude a la «estructura global del software y a las formas en que la estructura proporciona la integridad conceptual de un sistema» [SHA95a]. En su forma más simple, la arquitectura es la estructura jerárquica de los componentes del programa (módulos), la manera en que los componentes interactúan y la estructura de datos que van a utilizar los componentes. Sin embargo, en un sentido más amplio, los «componentes» se pueden generalizar para representar los elementos principales del sistema y sus interacciones[3].

La STARS Software Architecture Technology Guide proporciona mbs información y recursos en la dirección **www-ast.tds-gn.lmco.com/arch/guide.html**

Un objetivo del diseño del software es derivar una representación arquitectónica de un sistema. Esta representación sirve como marco de trabajo desde donde se llevan a cabo actividades de diseño más detalladas. Un conjunto de patrones arquitectónicos permiten que el ingeniero del software reutilice los conceptos a nivel de diseño.



Shaw **y** Garlan [SHA95a] describen un conjunto de propiedades que deberán especificarse como parte de un diseño arquitectónico:

Propiedades estructurales. Este aspecto de la representación del diseño arquitectónico define los componentes de un sistema (por ejemplo, módulos, objetos, filtros) y la manera en que esos componentes se empaquetan e interactúan unos con otros. Por ejemplo, **los** objetos se empaquetan para encapsular tanto los datos como el procesamiento que manipula **los** datos e interactúan mediante la invocación de métodos (Capítulo 20).

Propiedades extra-funcionales. La descripción del diseño arquitectónico deberá ocuparse de cómo la arquitectura de diseño consigue **los** requisitos para el rendimiento, capacidad, fiabilidad, seguridad, capacidad de adaptación y otras características del sistema.

Familias de sistemas relacionados. El diseño arquitectónico deberá dibujarse sobre patrones repetibles que se basen comúnmente en el diseño de familias de sistemas similares. En esencia, el diseño deberá tener la habilidad de volver a utilizar **los** bloques de construcción arquitectónicos.

Dada la especificación de estas propiedades, el diseño arquitectónico se puede representar mediante uno o más modelos diferentes [GAR95]. Los *modelos estructurales* representan la arquitectura como una colección organizada de componentes de programa. Los *modelos del marco de trabajo* aumentan el nivel de abstracción del diseño en un intento de identificar los marcos de trabajo (patrones) repetibles del diseño arquitectónico que se encuentran en tipos similares de aplicaciones. Los *modelos dinámicos* tratan los aspectos de comportamiento de la arquitectura del programa, indicando cómo puede cambiar la estructura o la configuración del sistema en función de los acontecimientos externos. **Los** *modelos de proceso* se centran en el diseño del proceso técnico de negocios que tiene que adaptar el sistema. Finalmente los *modelos funcionales* se pueden utilizar para representar la jerarquía funcional de un sistema.

Para representar el diseño arquitectónico se utilizan cinco tipos diferentes de modelos.

Se ha desarrollado un conjunto de *lenguajes de descripción arquitectónica* (LDAs) para representar los modelos destacados anteriormente [SHA95b]. Aunque se han propuesto muchos LDAs diferentes, la mayoría proporcionan mecanismos para describir los componentes del sistema y la manera en que se conectan unos con otros.

13.4.5. Jerarquía de control

La *jerarquía de control,* denominada también estructura de programa, representa la organización de los componentes de programa (módulos) e implica una jerarquía de control. No representa los aspectos procedimentales del software, ni se puede aplicar necesariamente a todos los estilos arquitectónicos.

Referencia cruzada

En el Capítulo **14** se presenta un estudio detallado de estilos **y patrones** arquitectónicos.

[3] Por ejemplo, los componentes arquitectónicos de un sistema cliente/servidor se representan en un nivel de abstracción diferente. Para más detalles véase el Capítulo 28.

Para representar la jerarquía control de aquellos estilos arquitectónicos que se avienen a la representación se utiliza un conjunto de notaciones diferentes. El diagrama más común es el de forma de árbol (Fig. 13.3) que representa el control jerárquico para las arquitecturas de llamada y de retorno[4]. Sin embargo, otras notaciones, tales como los diagramas de Warnier-Orr [ORR77] y Jackson [JAC83] también se pueden utilizar con igual efectividad. Con objeto de facilitar estudios posteriores de estructura, definiremos una serie de medidas y términos simples. Según la Figura 13.3, la *profundidad* y la *anchura* proporcionan una indicación de la cantidad de niveles de control y el *ámbito de control* global, respectivamente. El *grado de salida* es una medida del número de módulos que se controlan directamente con otro módulo. El *grado de entrada* indica la cantidad de módulos que controlan directamente un módulo dado.

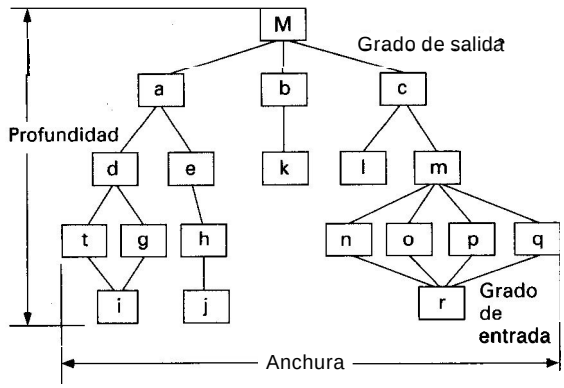


FIGURA 13.3. Terminologías de estructura para un estilo arquitectónico de llamada y retorno.

La relación de control entre los módulos se expresa de la manera siguiente: se dice que un módulo que controla otro módulo es *superior* a él, e inversamente, se dice que un módulo controlado por otro módulo es *subordinado* del controlador [YOU79]. Por ejemplo, en la Figura 13.3 el módulo ***M*** es superior a los módulos *a, b* y *c*. El módulo *h* está subordinado al módulo ***e*** y por último está subordinado al módulo ***M***. Las relaciones en anchura (por ejemplo, entre los módulos *d* y ***e),*** aunque en la práctica se puedan expresar, no tienen que definirse con terminología explícita.

Si desarrollamos un software orientado o objetos, no se aplicarán los medidos estructurales destacadas aquí. Sin embargo, se aplicarán otros (las que se abordan en lo Porte Cuarta).

La jerarquía de control también representa dos características sutiles diferentes de la arquitectura del software: visibilidad y conectividad. La *visibilidad* indica el conjunto de componentes de programa que un componente dado puede invocar o utilizar como datos, incluso cuando se lleva a cabo indirectamente. Por ejemplo, un módulo en un sistema orientado a objetos puede acceder al amplio abanico de objetos de datos que haya heredado, ahora bien solo utiliza una pequeña cantidad de estos objetos de datos. Todos los objetos son visibles para el módulo. La *conectividad* indica el conjunto de componentes que un componente dado invoca o utiliza directamente como datos. Por ejemplo, un módulo que hace directamente que otro módulo empiece la ejecución está conectado a él[5].

13.4.6. División estructural

Si el estilo arquitectónico de un sistema es jerárquico, la estructura del programa se puede dividir tanto horizontal como verticalmente. En la Figura 13.4.a la partición horizontal define ramas separadas de la jerarquía modular para cada función principal del programa. Lo *módulos de control*, representados con un sombreado más oscuro se utilizan para coordinar la comunicación entre ellos y la ejecución de las funciones. El enfoque más simple de la división horizontal define tres particiones -entrada, transformación de datos (frecuentemente llamado procesamiento) y salida—. La división horizontal de la arquitectura proporciona diferentes ventajas:

- proporciona software más fácil de probar
- conduce a un software más fácil de mantener
- propaga menos efectos secundarios
- proporciona software más fácil de ampliar

¿Cuáles son las ventajas de la partición horizontal?

Como las funciones principales se desacoplan las unas de las otras, el cambio tiende a ser menos complejo y las extensiones del sistema (algo muy común) tienden a ser más fáciles de llevar a cabo sin efectos secundarios. En la parte negativa la división horizontal suele hacer que los datos pasen a través de interfaces de módulos y que puedan complicar el control global del flujo del programa (si se requiere un movimiento rápido de una función a otra).

[4] Una arquitectura de llamada y de retorno (Capítulo 14) es una estructura de programa clásica que descompone la función en una jerarquía de control en donde el programa «principal» invoca un número de componentes de programa que a su vez pueden invocar aún a otros componentes.

[5] En el Capítulo 20, exploraremos el concepto de herencia para el software orientado a objetos. Un componente de programa puede heredar una lógica de control y/o datos de otro componente sin referencia explícita en el código fuente. Los componentes de este tipo serán visibles, pero no estarán conectados directamente. Un diagrama de estructuras (Capítulo 14) indica la conectividad.

La división vertical (Fig. 13.4.b), frecuentemente llamada *factorización (factoring)* sugiere que dentro de la estructura de programa el control (toma de decisiones) y el trabajo se distribuyan de manera descendente. Los **módulos del** nivel superior deberán llevar a cabo funciones de control y no realizarán mucho trabajo de procesamiento. Los módulos que residen en la parte inferior de la estructura deberán ser los trabajadores, aquellos que realizan todas las tareas de entrada, proceso y salida.

La naturaleza del cambio en las estructuras de programas justifica la necesidad de la división vertical. En la Figura 13.4b se puede observar que un cambio en un módulo de control (parte superior de la estructura) tendrá una probabilidad mayor de propagar efectos secundarios a los módulos subordinados a él. Un cambio en el módulo de trabajador, dado su nivel bajo en la estructura, es menos probable que propague efectos secundanos. En general, los cambios en los programas de computadora giran alrededor del programa (es decir, su comportamiento básico es menos probable que cambie). Por esta razón las estructuras con división vertical son menos susceptibles a los efectos secundarios cuando se producen cambios y por tanto se podrán mantener mejor —un factor de calidad clave—.

PUNTO CLAVE

Los módulos de «trabajador» tienden a cambiar de forma más frecuente que los módulos de control. Si se colocan en la parte inferior de la estructura, se reducen los efectos secundarios (originados por el cambio).

13.4.7. Estructura de datos

La *estructura de datos* es una representación de la relación lógica entre elementos individuales de datos. Como la estructura de la información afectará invariablemente al diseño procedimental final, la estructura de datos es tan importante como la estructura de programa para la representación de la arquitectura del software.

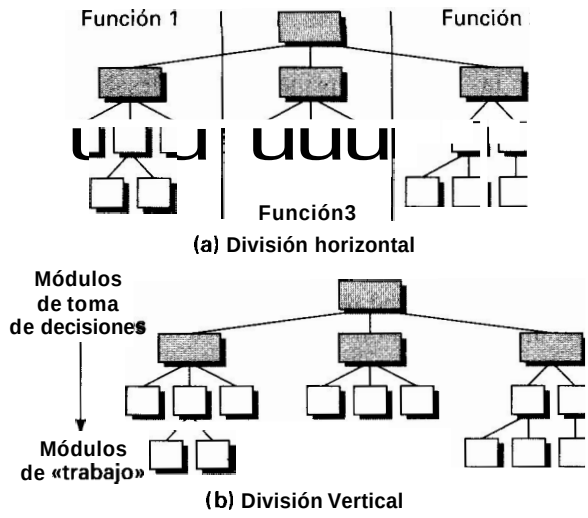
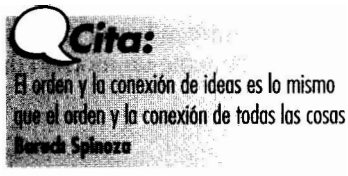


FIGURA 13.4. Partición estructural.

La estructura dicta las alternativas de organización, métodos de acceso, grado de capacidad de asociación y procesamiento de la información. Se han dedicado libros enteros (por ejemplo, [AHO83], [KRU84], [GAN89]) a estos temas, y un estudio más amplio sobre este tema queda fuera del ámbito de este libro. Sin embargo, es importante entender la disponibilidad de métodos clásicos para organizar la información y los conceptos que subyacen a las jerarquías de información.

La organización y complejidad de una estructura de datos están limitados Únicamente por la ingenuidad del diseñador. Sin embargo, existe un número limitado de estructuras de datos clásicas que componen los bloques de construcción para estructuras más sofisticadas.



Un *elemento escalar* es la estructura de datos más simple. Como su nombre indica, un elemento escalar representa un solo elemento de información que puede ser tratado por un identificador; es decir, se puede lograr acceso especificando un sola dirección en memoria. El tamaño y formato de un elemento escalar puede variar dentro de los límites que dicta el lenguaje de programación. Por ejemplo, un elemento escalar puede ser: una entidad lógica de un bit de tamaño; un entero o número de coma flotante con un tamaño de 8 a 64 bits; una cadena de caracteres de cientos o miles de bytes.

Cuando los elementos escalares se organizan como una lista o grupo contiguo, se forma un *vector secuencial*. Los vectores son las estructuras de datos más comunes y abren la puerta a la indexación variable de la información.

Cuando el vector secuencia1 se amplía a dos, tres y por último a un número arbitrario de dimensiones, se crea un *espacio n-dimensional*. El espacio n-dimensional más común es la matriz bidimensional. En muchos lenguajes de programación, un espacio n-dimensional se llama array.

Los elementos, vectores y espacios pueden estar organizados en diversos formatos. Una *lista enlazada* es una estructura de datos que organiza elementos escalares no contiguos, vectores o espacios de manera (llamados *nodos*) que les permita ser procesados como una lista. Cada nodo contiene la organización de datos adecuada (por ejemplo, un vector) o un puntero o más que indican la dirección de almacenamiento del siguiente nodo de la lista. Se pueden añadir nodos en cualquier punto de la lista para adaptar una entrada nueva en la lista.

Otras estructuras de datos incorporan o se construyen mediante las estructuras de datos fundamentales descritas anteriormente. Por ejemplo, una *estructura de datos jerárquica* se implementa mediante listas mul-

tienlazadas que contienen elementos escalares, vectores y posiblemente espacios n-dimensionales. Una estructura jerárquica se encuentra comúnmente en aplicaciones que requieren categorización y capacidad de asociación.

Invierta por lo menos todo el tiempo que necesite diseñando estructuras de datos, el mismo que pretende invertir diseñando algoritmos poro manipularlos. Si es así, se ahorrará mucha tiempo.

Es importante destacar que las estructuras de datos, al igual que las estructuras de programas, se pueden representar a diferentes niveles de abstracción. Por ejemplo, una pila es un modelo conceptual de una estructura de datos que se puede implementar como un vector o una lista enlazada. Dependiendo del nivel de detalle del diseño, los procesos internos de la **pila** pueden especificarse o no.

13.4.8. Procedimiento de software

La *estructura de programa* define la jerarquía de control sin tener en consideración la secuencia de proceso y de decisiones. El procedimiento de software se centra en el procesamiento de cada módulo individualmente. El procedimiento debe proporcionar una especificación precisa de procesamiento, incluyendo la secuencia de sucesos, los puntos de decisión exactos, las operaciones repetitivas e incluso la estructura/organización de datos.

Existe, por supuesto, una relación entre la estructura y el procedimiento. El procesamiento indicado para cada módulo debe incluir una referencia a todos los módulos subordinados al módulo que se está describiendo. Es decir, una representación procedimental del software se distribuye en capas como muestra la Figura 13.5[6].

13.4.9. Ocultación de información

El concepto de modularidad conduce a todos los diseñadores de software a formularse una pregunta importante: «¿Cómo se puede descomponer una solución de software para obtener el mejor conjunto de módulos?» El principio de *ocultación de información* [PAR72] sugiere que los módulos se caracterizan por las decisiones de diseño que (cada uno) oculta al otro. En otras palabras, los módulos deberán especificarse y diseñarse de manera que la información (procedimiento y datos) que está dentro de un módulo sea inaccesible a otros módulos que no necesiten esa información.

Ocultación significa que se puede conseguir una modularidad efectiva definiendo un conjunto de módulos independientes que se comunican entre sí intercambiando sólo la información necesaria para lograr la función del software. La abstracción ayuda a definir las entidades (o información).

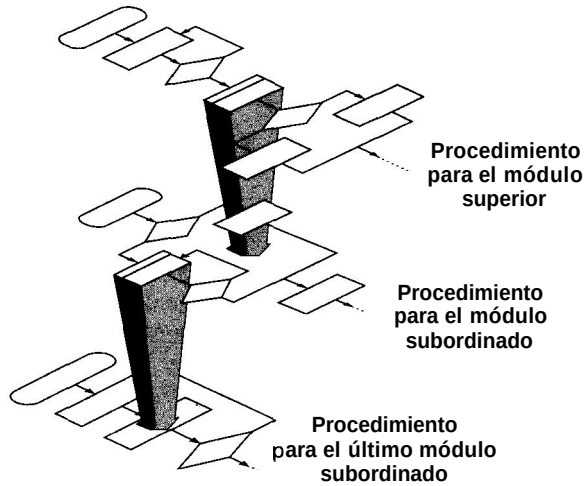


FIGURA 13.5. El **procedimiento se distribuye en capas.**

13.5. DISEÑO MODULAR EFECTIVO

Los conceptos fundamentales de diseño descritos en la sección anterior sirven para incentivar 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 (véase la Sección 13.4.3), facilita los cambios (un aspecto crítico de la capacidad de mantenimiento del software), y da como resultado una implementación más fácil al fomentar el desarrollo paralelo de las diferentes partes de un sistema.

13.5.1. Independencia funcional

El concepto de *independenciafuncional* es la suma de la modularidad y de los conceptos de abstracción y ocultación de información. En referencias obligadas sobre el diseño del software, Pamas [PAR72] y Wirth [WIR71] aluden a las técnicas de refinamiento que mejoran la independencia de módulos. Trabajos posteriores de Stevens, Wyers y Constantine [STE74] consolidaron el concepto.

Un código es «determinante» si se describe con una sola oración —sujeto, verbo y predicada —.

La independencia funcional se consigue desarrollando módulos con una función «determinante» y una «aver-

[6] Esto no es verdad para todas las estructuras arquitectónicas, Por ejemplo, la estratificación jerárquica de procedimientos no se encuentra en arquitecturas orientadas a objetos.

sión» a una interacción excesiva con otros módulos. Dicho de otra manera, queremos diseñar el software de manera que cada módulo trate una subfunción de requisitos y tenga una interfaz sencilla cuando se observa desde otras partes de la estructura del programa. **Es** justo preguntarse por qué es importante la independencia. El software con una modularidad efectiva, es decir, módulos independientes, es más fácil de desarrollar porque la función se puede compartimentar**y** las interfaces se simplifican (tengamos en consideración las ramificaciones cuando el desarrollo se hace en equipo). Los módulos independientes son más fáciles de mantener (y probar) porque se limitan **los** efectos secundarios originados por modificaciones de diseño/código; porque se reduce la propagación de errores; y porque es posible utilizar módulos usables. En resumen, la independencia funcional es la clave para un buen diseño y el diseño es la clave para la calidad del software.

La independencia se mide mediante dos criterios cualitativos: la cohesión y el acoplamiento. La *cohesión* es una medida de la fuerza relativa funcional de un módulo. El *acoplamiento* es una medida de la independencia relativa entre los módulos.

13.5.2. Cohesión

La cohesión es una extensión natural del concepto de ocultación de información descrito en la Sección **13.4.8.** Un módulo cohesivo lleva a cabo una sola tarea dentro de un procedimiento de software, lo cual requiere poca interacción con los procedimientos que se llevan a cabo en otras partes de un programa. Dicho de manera sencilla, un módulo cohesivo deberá (idealmente) hacer una sola cosa.

la cohesión es una indicación cualitativa del grado que tiene un módulo para centrarse en una sola cosa.

La cohesión se puede representar como un «espectro». Siempre debemos buscar la cohesión más alta, aunque la parte media del espectro suele ser aceptable. La escala de cohesión no es lineal. Es decir, la parte baja de la cohesión es mucho «peor» que el rango medio, que es casi tan «bueno» como la parte alta de la escala. En la práctica, un diseñador no tiene que preocuparse de categorizar la cohesión en un módulo específico. Más bien, se deberá entender el concepto global, y así se deberán evitar los niveles bajos de cohesión al diseñar los códigos.

En la parte inferior (y no deseable) del espectro, encontraremos un módulo que lleva a cabo un conjunto de tareas que se relacionan con otras débilmente, si es que tienen algo que ver. Tales módulos se denominan *coincidentalmente cohesivos.* Un módulo que realiza tareas relacionadas lógicamente (por ejemplo, un módulo que produce todas las salidas independientemente del tipo) es *lógicamente cohesivo.* Cuando un módulo contiene tareas que están relacionadas entre **sí** por el hecho de que todas deben ser ejecutadas en el mismo intervalo de tiempo, el módulo muestra *cohesión temporal.*

Como ejemplo de baja cohesión, tomemos en consideración un módulo que lleva a cabo un procesamiento de errores de un paquete de análisis de ingeniería. El módulo es invocado cuando los datos calculados exceden los límites preestablecidos. Se realizan las tareas siguientes: **(1)** calcula los datos complementarios basados en los datos calculados originalmente; (2) produce un informe de errores (con contenido gráfico) en la estación de trabajo del usuario; **(3)** realiza los cálculos de seguimiento que haya pedido el usuario; **(4)** actualiza una base de datos, y (5) activa un menú de selección para el siguiente procesamiento. Aunque las tareas anteriores están poco relacionadas, cada una es una entidad funcional independiente que podrá realizarse mejor como un módulo separado. La combinación de funciones en un solo módulo puede servir sólo para incrementar la probabilidad de propagación de errores cuando se hace una modificación a alguna de las tareas procedimentales anteriormente mencionadas.

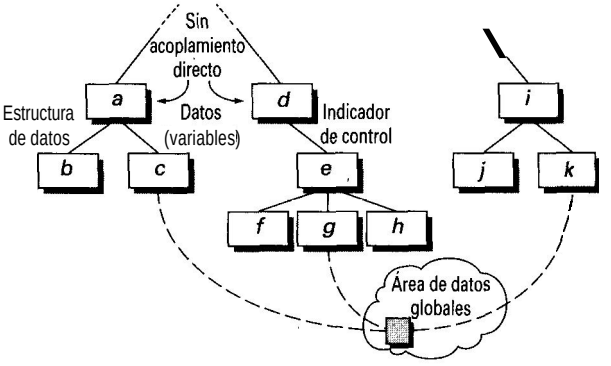


FIGURA 13.6. Tipos de acoplamiento.

Los niveles moderados de cohesión están relativamente cerca unos de otros en la escala de independencia modular. Cuando los elementos de procesamiento de un módulo están relacionados, y deben ejecutarse en un orden específico, existe *cohesión procedimental.* Cuando todos los elementos de procesamiento se centran en un área de una estructura de datos, tenemos presente una *cohesión de comunicación.* Una cohesión alta se caracteriza por un módulo que realiza una única tarea.

Si nos concentramos en uno sola coso durante el diseño a nivel de componentes, hoy que realizarlo con cohesión.

Como ya se ha mencionado anteriormente, no es necesario determinar el nivel preciso de cohesión. **Más** bien, es importante intentar conseguir una cohesión alta y reconocer cuando hay poca cohesión para modificar el diseño del software y conseguir una mayor independencia funcional.

13.5.3. Acoplamiento

El acoplamiento es una medida de interconexión entre módulos dentro de una estructura de software. El acoplamiento depende de la complejidad de interconexión entre los módulos, el punto donde se realiza una entrada o referencia a un módulo, y los datos que pasan a través de la interfaz.

En el diseño del software, intentamos conseguir el acoplamiento más bajo posible. Una conectividad sencilla entre los módulos da como resultado un software más fácil de entender y menos propenso a tener un «efecto ola» [STE75] causado cuando ocurren errores en un lugar y se propagan por el sistema.

CLAVE

El acoplamiento es una indicación cualitativa del grado de conexión de un módulo con otros y con el mundo exterior.

La Figura 13.6 proporciona ejemplos de diferentes tipos de acoplamiento de módulos. Los módulos *a* y *d* son subordinados a módulos diferentes. Ninguno está relacionado y por tanto no ocurre un acoplamiento directo. El módulo *c* es subordinado al módulo *a* y se accede a él mediante una lista de argumentos por la que pasan los datos. Siempre que tengamos una lista convencional simple de argumentos (es decir, el paso de datos; la existencia de correspondencia uno a uno entre elementos), se presenta un acoplamiento bajo (llamado *acoplumiento de datos)* en esta parte de la estructura. Una variación del acoplamiento de datos, llamado *acoplamiento de marca (stamp),* se da cuando una parte de la estructura de datos (en vez de argumentos simples) se pasa a través de la interfaz. Esto ocurre entre los módulos *b* y *a*.

Sistemas altamente *acoplados* conducen a depurar verdaderas pesadillas. Evítelos.

En niveles moderados el acoplamiento se caracteriza por el paso de control entre módulos. El *acoplamiento de control* es muy común en la mayoría de los diseños de software y se muestra en la Figura 13.6 en donde un «indicador de control» (una variable que controla las decisiones en un módulo superior o subordinado) se pasa entre los módulos *d* y *e*.

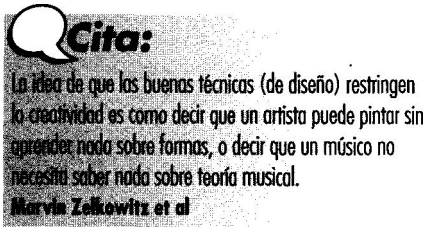
Cuando los módulos están atados a un entorno externo al software se dan niveles relativamente altos de acoplamiento. Por ejemplo, la E/S 'acoplaun módulo a dispositivos, formatos y protocolos de comunicación. El *acoplamiento externo* es esencial, pero deberá limitarse a unos pocos módulos en una estructura. También aparece un acoplamiento alto cuando varios módulos hacen referencia a un área global de datos. El *acoplamiento común,* tal y como se denomina este caso, se muestra en la Figura 13.8. Los módulos *c,* g y *k* acceden a elementos de datos en un área de datos global (por ejemplo, un archivo de disco o un área de memoria totalmente accesible). El módulo *c* inicializa el elemento. Más tarde el módulo g vuelve a calcular el elemento y lo actualiza. Supongamos que se produce un error y que g actualiza el elemento incorrectamente. Mucho más adelante en el procesamiento, el módulo *k* lee el elemento, intenta procesarlo y falla, haciendo que se interrumpa el sistema. El diagnóstico de problemas en estructuras con acoplamiento común es costoso en tiempo y es difícil. Sin embargo, esto no significa necesariamente que el uso de datos globales sea «malo». Significa que el diseñador del software deberá ser consciente de las consecuencias posibles del acoplamiento común y tener especial cuidado de prevenirse de ellos.

El grado más alto de acoplamiento, *acoplamiento de contenido,* se da cuando un módulo hace uso de datos o de información de control mantenidos dentro de los límites de otro módulo. En segundo lugar, el acoplamiento de contenido ocurre cuando se realizan bifurcaciones a mitad de módulo. Este modo de acoplamiento puede y deberá evitarse.

13.6 HEURÍSTICA DE DISEÑO PARA UNA MODULARIDAD EFECTIVA

Una vez que se ha desarrollado una estructura de programa, se puede conseguir una modularidad efectiva aplicando los conceptos de diseño que se introdujeron al principio de este capítulo. La estructura de programa se puede manipular de acuerdo con el siguiente conjunto de heurísticas:

1. *Evaluar la «primera iteración» de la estructura de programa para reducir al acoplamiento y mejorar la cohesión.* Una vez que se ha desarrollado la estructura del programa, se pueden explosionar o implosionar los módulos con vistas a mejorar la independencia del módulo. Un *módulo explosionado* se convierte en dos módulos o más en la estructura final de programa. *Un módulo implosionudo* es el resultado de combinar el proceso implicado en dos o más módulos.



Un módulo explosionado se suele dar cuando existe un proceso común en dos o más módulos y puede redefinirse como un módulo de cohesión separado. Cuando se espera un acoplamiento alto, algunas veces se pueden implosionar los módulos para reducir el paso de control, hacer referencia a los datos globales y a la complejidad de la interfaz.

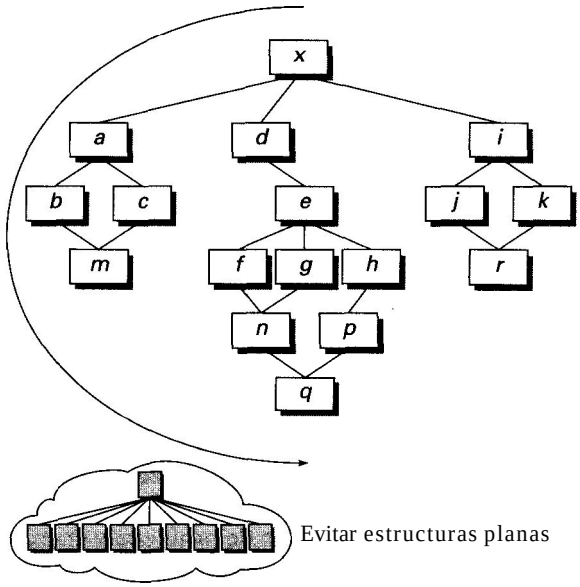


FIGURA 13.7. Estructuras de programa.

II. Intentar minimizar las estructuras con un alto grado de salida; esforzarse por la entrada a medida que aumenta la profundidad. La estructura que se muestra dentro de la nube en la Figura 13.7 no hace un uso eficaz de la factorización. Todos los módulos están «planos», al mismo nivel y por debajo de un solo módulo de control. En general, una distribución más razonable de control se muestra en la estructura de la derecha. La estructura toma una forma oval, indicando la cantidad de capas de control y módulos de alta utilidad a niveles inferiores.

III. Mantener el ámbito del efecto de un módulo dentro del ámbito de control de ese módulo. El *ámbito del efecto* de un módulo *e* se define como todos lo otros módulos que se ven afectados por la decisión tomada en el módulo ***e***. El ámbito de control del módulo *e* se compone de todos los módulos subordinados y superiores al módulo *e*. En la Figura 13.7, si el módulo *e* toma una decisión que afecta al módulo *r*, tenemos una violación de la heurística III, porque el módulo *r* se encuentra fuera del ámbito de control del módulo *e*.

IV. *Evaluar las interfaces de los módulos para reducir la complejidad y la redundancia,* y *mejorar la consistencia.* La complejidad de la interfaz de un módulo es la primera causa de los errores del software. Las interfaces deberán diseñarse para pasar información de manera sencilla y deberán ser consecuentes con la función de un módulo. La inconsistencia de interfaces (es decir, datos aparentemente sin relacionar pasados a través de una lista de argumentos u otra técnica) es una indicación de poca cohesión. El módulo en cuestión deberá volverse a evaluar.

V. *Definir módulos cuya función se pueda predecir, pero evitar módulos que sean demasiado restrictivos.* Un módulo es predecible cuando se puede tratar como una caja negra; es decir, los mismos datos externos se producirán independientemente de los datos internos de procesamiento[7]. Los módulos que pueden tener «memoria» interna no podrán predecirse a menos que se tenga mucho cuidado en su empleo.

Un módulo que restringe el procesamiento a una sola subfunción exhibe una gran cohesión y es bien visto por el diseñador. Sin embargo, un módulo que restringe arbitrariamente el tamaño de una estructura de datos local, las opciones dentro del flujo de control o los modos de interfaz externa requerirá invariablemente mantenimiento para quitar esas restricciones.

Referencia Web

En la dirección de internet **www.dacs.dtic.mil/techs/design/Design.ToC.html** se puede enconhor un informe detallado sobre los métodos de diseño de software entre los que se incluyen el estudio de todos los conceptos y principios de diseño que se abortan en este capítulo.

VI. Intentar conseguir módulos de «entrada controlada)), evitando «conexiones patológicas». Esta heurística de diseño advierte contra el acoplamiento de contenido. El software es más fácil de entender y por tanto más fácil de mantener cuando los módulos que se interaccionan están restringidos y controlados. Las *conexiones patológicas* hacen referencia a bifurcaciones o referencias en medio de un módulo.

[7] Un módulo de «caja negra. es una abstracción procedimental.

13.7 EL MODELO DEL DISEÑO

Los principios y conceptos de diseño abordados en este capítulo establecen las bases para la creación del modelo de diseño que comprende representaciones de datos, arquitectura, interfaces y componentes. Al igual que en el modelo de análisis anterior al modelo, cada una de estas representaciones de diseño se encuentran unidas unas a otras y podrán sufrir un seguimiento hasta los requisitos del software.

En la Figura 13.1, el modelo de diseño se representó como una pirámide. El simbolismo de esta forma es importante. Una pirámide es un objeto extremadamente estable con una base amplia y con un centro de gravedad bajo. Al igual que la pirámide, nosotros queremos crear un diseño de software que sea estable. Crearemos un modelo de diseño que se tambalee fácilmente con vientos de cambio al establecer una base amplia en el diseño de datos, mediante una región media estable en el diseño arquitectónico y de interfaz, y una parte superior aplicando el diseño a nivel de componentes.

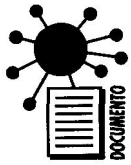
Los métodos que conducen a la creación del modelo de diseño se presentan en los Capítulos **14,** 15, 16 y 22 (para sistemas orientados a objetos). Cada método permite que el diseñador cree un diseño estable que se ajuste a los conceptos fundamentales que conducen a un software de alta calidad.

13.8 DOCUMENTACIÓN DEL DISEÑO

La *Especificación del diseño* aborda diferentes aspectos del modelo de diseño y se completa a medida que el diseñador refina su propia representación del software. En primer lugar, ***se*** describe el ámbito global del esfuerzo realizado en el diseño. La mayor parte de la información que se presenta aquí se deriva de la *Especificación del sistema* y del modelo de análisis *(Especificación de los requisitos del software).*

A continuación, se especifica el diseño de datos. Se definen también las estructuras de las bases de datos, cualquier estructura externa de archivos, estructuras internas de datos y una referencia cruzada que conecta objetos de datos con archivos específicos.

El diseño arquitectónico indica cómo se ha derivado la arquitectura del programa del modelo de análisis. Además, para representar la jerarquía del módulo se utilizan gráficos de estructuras.



Especificación del diseño del software

Se representa el diseño de interfaces internas y externas de programas y se describe un diseño detallado de la interfaz hombre-máquina. En algunos casos, se podrá representar un prototipo detallado del IGU.

Los componentes — elementos de software que se pueden tratar por separado tales como subrutinas, funciones o procedimientos — se describen inicialmente con una narrativa de procesamienfo en cualquier idioma (Castellano, Inglés). La narrativa de procesamiento explica la función procedimental de un componente (módulo). Posteriormente, se utiliza una herramienta de diseño procedimental para convertir esa estructura en una descripción estructural.

La *Especificación del diseño* contiene una referencia cruzada de requisitos. El propósito de esta referencia cruzada (normalmente representada como una matriz simple) es: (1) establecer que todos los requisitos se satisfagan mediante el diseño del software, y (2) indicar cuales son los componentes críticos para la implementación de requisitos específicos.

El primer paso en el desarrollo de la documentación de pruebas también se encuentra dentro del documento del diseño. Una vez que se han establecido las interfaces y la estructura de programa podremos desarrollar las líneas generales para comprobar los módulos individuales y la integración de todo el paquete. En algunos casos, esta sección se podrá borrar de la *Especificación del diseño.*

Las restricciones de diseño, tales como limitaciones físicas de memoria o la necesidad de una interfaz externa especializada, podrán dictar requisitos especiales para ensamblar o empaquetar el software. Consideraciones especiales originadas por la necesidad de superposición de programas, gestión de memoria virtual, procesamiento de alta velocidad u otros factores podrán originar modificaciones en diseño derivadas del flujo o estructura de la información. Además, esta sección describe el enfoque que se utilizará para transferir software al cliente.

La última sección de la *Especificación del diseño* contiene datos complementarios. También se presentan descripciones de algoritmos, procedimientos alternativos, datos tabulares, extractos de otros documentos y otro tipo de información relevante, todos mediante notas especiales o apéndices separados. Será aconsejable desarrollar un *Manual preliminar de Operaciones/Instalación* e incluirlo como apéndice para la documentación del diseño.

RESUMEN

El diseño es el núcleo técnico de la ingeniería del software. Durante el diseño se desarrollan, revisan y documentan los refinamientos progresivos de la estructura de datos, arquitectura, interfaces y datos procedimentales de los componentes del software. El diseño da como resultado representaciones del software para evaluar la calidad.

Durante las cuatro últimas décadas se han propuesto diferentes principios y conceptos fundamentales del diseño del software. Los principios del diseño sirven de guía al ingeniero del software a medida que avanza el proceso de diseño. Los conceptos de diseño proporcionan los criterios básicos para la calidad del diseño.

La modularidad (tanto en el programa como en los datos) y el concepto de abstracción permiten que el diseñador simplifique y reutilice los componentes del software. El refinamiento proporciona un mecanismo para representar sucesivas capas de datos funcionales. El programa y la estructura de datos contribuyen a tener una visión global de la arquitectura del software, mientras que el procedimiento proporciona el detalle necesario para la implementación de los algoritmos. La ocultación de información y la independencia funcional **pro**porcionan la heurística para conseguir una modularidad efectiva.

Concluiremos nuestro estudio de los fundamentos del diseño con las palabras de Glendord Myers [MYE78]:

> Intentamos resolver el problema dándonos prisa en el proceso de diseño de forma que quede el tiempo suficiente **hasta** el final del proyecto como para descubrir los errores que se cometieron por correr en el proceso de diseño...

La moraleja es: ¡No te precipites durante el diseño! Merece la pena esforzarse por un buen diseño.

No hemos terminado nuestro estudio sobre el diseño. En los capítulos siguientes, se abordan los métodos de diseño. Estos métodos combinados con los fundamentos de este capítulo, forman la base de una visión completa del diseño del software.

REFERENCIAS

[AHO83] Aho, A.V.J., Hopcroft, y J. Ullmann, *Data Structures and Algorithms,* Addison-Wesley, 1983.

[BAS89] Bass, L., P. Clements y R. Kazman, *Software Architecture in Practice,* Addison-Wesley, 1998.

[BEL81] Belady, L., Foreword to *Software Design: Methods and Techniques* (L.J. Peters, autor), Yourdon Press, 1981.

[BRO98] Brown, W.J. et al., *Anti-Patterns,* Wiley, 1998.

[BUS96] Buschmann, F. et al., *Pattern-Oriented Software Architecture,* Wiley, 1996.

[DAV95] Davis, A., *201 Principles of Software Development,* McGraw-Hill, 1995.

[DEN73] Dennis, J., «Modularity», in *Advanced Course on Software Engineering,* F.L. Bauer, ed., Springer-Verlag, Nueva York, 1973, pp. 128-182.

[GAM95] Gamma, E., et al, *Design Patterns,* Addison-Wesley, 1995.

[GAN89] Gannet, G., *Handbook of Algorithms and Data Structures,* 2.ª ed, Addison-Wesley, 1989.

[GAR95] Garlan, D., y M. Shaw, «An Introduction to Software Architecture», *Advances in Software Engineering and Knowledge Engineering,* vol. 1, V. Ambriola y G. Tortora (eds.), World Scientific Publishing Company, 1995.

[JAC75] Jackson, M.A., *Principles of Program Design,* Academic Press, 1975.

[JAC83] Jackson, M.A., *System Development,* Prentice-Hall, 1983.

[KAI83] Kaiser, S.H., *The Design of Operating Systems for Small Computer Systems,* Wiley-Inerscience, 1983, pp. 594 ss.

[JAC92] Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.

[KRU84] Kruse R.L., *Data Structures and Program Design,* Prentice-Hall, 1984.

[MCG91] McGlaughlin, R., «Some Notes on Program Design», *Software Engineering Notes,* vol. 16, n.º 4, Octubre 1991, pp. 53-54.

[MYE78] Myers, G., *Composite Structured Design,* Van Nostrand, 1978.

[MEY88] Meyer, B., *Object-Oriented Software Construction,* Prentice-Hall, 1988.

[MIL72] Mills, H.D., «Mathematical Foundations for Structured Programming», Technical Report FSC71-6012, IBM Corp., Federal Systems Division, Gaithersburg, Maryland, 1972.

[NAS73] Nassi, I., y B. Shneiderman, «Flowchart Techniques for Structured Programming», *SIGPLAN Notices,* ACM, Agosto 1973.

[ORR77] Orr, K.T., *Structured Systems Development,* Yourdon Press, Nueva York, 1977.

[PAR72] Parnas, D.L.,«On Criteria to be used in Decomposing Systems into Modules», *CACM*, vol. 14, n.º 1, Abril 1972, pp. 221-227.

[ROS75] Ross, D., J. Goodenough y C. Irvine, «Software Engineering: Process, Principies and Goals», *IEEE Computer*, vol. 8, n.º 5, Mayo 1975.

[SHA95a] Shaw, M., y D. Garlan, «Formulations and Formalisms in Software Architecture», *Volume 1000-Lecture Notes in Computer Science,* Springer Verlag, 1995.

[SHA95b] Shaw, M. et al., «Abstractions for Software Architecture and Tools to Support Them», *IEEE Truns. Software Engineering,* vol. 21, n.º 4, Abril 1995, pp. 314-335.

[SHA96] Shaw, M., y D. Garlan, *Software Architecture,* Prentice-Hall, 1996.

[SOM96] Sommerville, I., *Software Engineering,* 3.ª ed., Addison-Wesley, 1989.

[STE74] Stevens, W., G. Myers y L. Constantine, «Structured Design», *IBM Systems Journal,* vol. 13, n.º 2, 1974, pp. 115-139.

[WAR74] Warnier, J., *Logical Constructionof Programs,* Van Nostrand-Reinhold, 1974.

[WAS83] Wasserman, A., «Information System Design Methodology», in *Software Design Techniques* (P. Freeman y A. Wasserman, eds.), 4.ª ed., IEEE Computer Society Press, 1983, p. 43.

[WIR71] Wirth, N., «Program Development by Stepwise Refinement», *CACM,* vol. 14, n.º 4, 1971, pp. 221-227.

[YOU79] Yourdon, E., y L. Constantine, *Structured Design,* Prentice-Hall, 1979.

PROBLEMAS Y PUNTOS A CONSIDERAR

13.1. Cuando se «escribe» un programa, ¿se está diseñando software? ¿En qué se diferencia el diseño del software de la codificación?

13.2. Desarrolle tres principios de diseño adicionales que añadir a los destacados en la Sección 13.3.

13.3. Proporcione ejemplos de tres abstracciones de datos y las abstracciones procedimentales que se pueden utilizar para manipularlos.

13.4. Aplique un «enfoque de refinamiento paso a paso» para desarrollar tres niveles diferentes de abstracción procedimental para uno o más de los programas que se muestran a continuación:

a. Desarrolle un dispositivo para rellenar los cheques que, dada la cantidad numérica en pesetas, imprima la cantidad en letra tal y como se requiere normalmente.

b. Resuelva iterativamente las raíces de una ecuación transcendental.

c. Desarrolle un simple algoritmo de planificación *round-robin* para un sistema operativo

13.5. ¿Es posible que haya algún caso en que la ecuación **(13.2)** no sea verdad? ¿Cómo podría afectar este caso a la modularidad?

13.6. ¿Cuándo deberá implementarse un diseño modular como un software monolítico? ¿Cómo se puede llevar a cabo? ¿Es el rendimiento la única justificación para la implementación de software monolítico?

13.7. Desarrolle al menos cinco niveles de abstracción para uno de los problemas de software siguientes:

a. un vídeo-juego de su elección.

b. un software de transformación en tres dimensiones para aplicaciones gráficas de computador.

c. un intérprete de lenguajes de programación.

d. un controlador de un robot con dos grados de libertad.

e. cualquier problema que hayan acordado usted y su profesor.

Conforme disminuye el grado de abstracción, su foco de atención puede disminuir de manera **que** en la última abstracción (código fuente) solo se necesite describir una única tarea.

13.8. Obtenga el trabajo original de Parnas [PAR72] y resuma el ejemplo de software que utiliza el autor para ilustrar la descomposición en módulos de un sistema. ¿Cómo se utiliza la ocultación de información para conseguir la descomposición?

13.9. Estudie la relación entre el concepto de ocultación de información como atributo de modularidad eficaz y el concepto de independencia del módulo.

13.10. Revise algunos de los esfuerzos que haya realizado recientemente en desarrollar un software y realice un análisis de **los** grados de cada módulo (con una escala ascendente del 1 al 7). Obtenga los mejores y los peores ejemplos.

13.11. Un grupo de lenguajes de programación de alto nivel soporta el procedimiento interno como construcción modular. ¿Cómo afecta esta construcción al acoplamiento y a la ocultación de información?

13.12. ¿Qué relación tienen los conceptos de acoplamiento y de movilidad del software? Proporcione ejemplos que apoyen esta relación.

13.13. Haga un estudio de la manera en que ayuda la partición estructural para ayudar a mantener el software.

13.14. ¿Cuál es el propósito de desarrollar una estructura de programa descompuesta en factores?

13.15. Describa el concepto de ocultación de información con sus propias palabras.

13.16. ¿Por qué es una buena idea mantener el efecto de alcance de un módulo dentro de su alcance de control?

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Donald Norman ha escrito dos libros *(The Design of Everyday Things,* Doubleday, 1990, y *The Psychology of Everyday Things,* Harpercollins, 1988) que se han convertido en clásicos de literatura de diseño y es una lectura «obligada» para todos los que diseñan cualquier cosa que el ser humano va utilizar. Adams *(Conceptual Blockbusting,* 3." ed. Addison-Wesley, 1986) ha escrito un libro que es una lectura esencial para los diseñadores que quieren ampliar **su** forma de pensar. Finalmente un texto clásico de Polya *(How to Solve It, 2."* ed., Princeton University Press, 1988) proporciona un proceso genérico de resolver problemas que puede servir de ayuda a los diseñadores cuando se enfrentan con problemas complejos.

Siguiendo la misma tradición, Winograd et al. *(Bringing to Software,* Addison-Wesley, 1996) aborda los diseños del software que funcionan, los que no funcionan y las razones. Un libro fascinante editado por Wixon y Ramsey *(Field Methods Casebookfor Software Design,* Wiley, 1996) sugiere los «métodos de investigación de campos» (muy similares a los que utilizan los antropólogos) para entender cómo realizan los usuarios el trabajo que realizan y entonces diseñar el software que cubra sus necesidades. Beyer y Holtzblatt *(Contextual Design: A Customer-Centrered Approach to Systems,* Academic Press, 1997) ofrecen otra visión del diseño de software que integra al cliente/usuario en todos los aspectos del proceso de diseño del software.

McConnell *(Code Complete,* Microsoft Press, 1993) presenta **un** estudio excelente de los aspectos prácticos para el diseño de software de computadora de alta calidad. Robertson *(Simple Program Design,* 3.ª ed., Boyd & Fraser Publishing) presenta un estudio introductorio de diseño del software que es útil para aquellos que empiezan a estudiar sobre el tema.

Dentro de una antología editada por Freeman y Wasserman *(Software Design Techniques,* 4.ª ed., IEEE, 1983) se encuentra un estudio histórico excelente de trabajos importantes sobre diseño del software. Este trabajo de enseñanza reimprime muchos de los trabajos clásicos que han formado la base de las tendencias actuales en el diseño del software. Buenos estudios sobre los fundamentos del diseño de software se pueden encontrar en los libros de Myers [MYE78], Peters *(Software Design: Methods and Techniques,* Yourdon Press, Nueva York, 1981), Macro *(Software Engineering: Concepts and Management,* Prentice-Hall, 1990) y Sommerville *(Software Engineering,* Addison-Wesley, 5.ª ed., 1995).

Tratamientos matemáticamente rigurosos del software de computadora se pueden encontrar en los libros de Jones *(Software Development: A Rigourous Approach,* Prentice-Hall, 1980), Wulf *(Fundamental Structures of Computer Science,* Addison-Wesley, 1981) y Brassard y Bratley *(Fundamental of Algorithmics,* Prentice-Hall, 1995).

Todos estos libros ayudan a proporcionar el fundamento teórico necesario para comprender el software de computadora.

Kruse *(Data Structures and Program Design,* Prentice-Hall, 1994) y Tucker et al. *(Fundamental of Computing II: Abstraction, Data Structures, and Large Software Systems,* McGraw-Hill, 1995) presentan una información valiosa sobre estructuras de datos. Las medidas de calidad de diseño presentadas desde una perspectiva técnica y de gestión son abordadas por Card y Glass *(Measuring Design Quality,* Prentice-Hall, 1990).

Una amplia variedad de fuentes de información sobre el diseño del software y otros temas relacionados están disponibles en Intemet. Una lista actualizada de referencias relevantes para los conceptos y métodos de diseño se puede encontrar en **http://www.pressman5.com**