CAPÍTULO

8

CONCEPTOS DE DISEÑO

CONCEPTOS CLAVE

El diseño de software agrupa el conjunto de principios, conceptos y prácticas que llevan al desarrollo de un sistema o producto de alta calidad. Los principios de diseño establecen una filosofía general que guía el trabajo de diseño que debe ejecutarse. Deben entenderse los conceptos de diseño antes de aplicar la mecánica de éste, y la práctica del diseño en sí lleva a la creación de distintas representaciones del software que sirve como guía para la actividad de construcción que siga.

El diseño es crucial para el éxito de la ingeniería de software. A principios de la década de 1990, Mitch Kapor, creador de Lotus 1-2-3, publicó en *Dr. Dobbs Journal* un "manifiesto del diseño de software". Decía lo siguiente:

> ¿Qué es el diseño? Es donde se está con un pie en dos mundos —el de la tecnología y el de las personas y los propósitos humanos— que tratan de unificarse...
>
> Vitruvio, romano crítico de arquitectura, afirmaba que los edificios bien diseñados eran aquellos que tenían resistencia, funcionalidad y belleza. Lo mismo se aplica al buen software. *Resistencia:* un programa no debe tener ningún error que impida su funcionamiento. *Funcionalidad:* un programa debe se apropiado para los fines que persigue. *Belleza:* la experiencia de usar el programa debe ser placentera. Éstos son los comienzos de una teoría del diseño de software.

El objetivo del diseño es producir un modelo o representación que tenga resistencia, funcionalidad y belleza. Para lograrlo, debe practicarse la diversificación y luego la convergencia. Belady [Bel81] afirma que "la diversificación es la adquisición de un repertorio de alternativas, materia prima del diseño: componentes, soluciones con los componentes y conocimiento, todo lo cual

UNA MIRADA RÁPIDA

¿Qué es? El diseño es lo que casi todo ingeniero quiere hacer. Es el lugar en el que las reglas de la creatividad —los requerimientos de los participantes, las necesidades del negocio y las consideraciones técnicas— se unen para formular un producto o sistema. El diseño crea una representación o modelo del software, pero, a diferencia del modelo de los requerimientos (que se centra en describir los datos que se necesitan, la función y el comportamiento), el modelo de diseño proporciona detalles sobre arquitectura del software, estructuras de datos, interfaces y componentes que se necesitan para implementar el sistema.

¿Quién lo hace? Ingenieros de software llevan a cabo cada una de las tareas del diseño.

¿Por qué es importante? El diseño permite modelar el sistema o producto que se va a construir. Este modelo se evalúa respecto de la calidad y su mejora antes de generar código; después, se efectúan pruebas y se involucra a muchos usuarios finales. El diseño es el lugar en el que se establece la calidad del software.

¿Cuáles son los pasos? El diseño representa al software de varias maneras. En primer lugar, debe representarse la arquitectura del sistema o producto. Después se modelan las interfaces que conectan al software con los usuarios finales, con otros sistemas y dispositivos, y con sus propios componentes constitutivos. Por último, se diseñan los componentes del software que se utilizan para construir el sistema. Cada una de estas perspectivas representa una acción de diseño distinta, pero todas deben apegarse a un conjunto básico de conceptos de diseño que guíe el trabajo de producción de software.

¿Cuál es el producto final? El trabajo principal que se produce durante el diseño del software es un modelo de diseño que agrupa las representaciones arquitectónicas, interfaces en el nivel de componente y despliegue.

¿Cómo me aseguro de que lo hice bien? El modelo de diseño es evaluado por el equipo de software en un esfuerzo por determinar si contiene errores, inconsistencias u omisiones, si existen mejores alternativas y si es posible implementar el modelo dentro de las restricciones, plazo y costo que se hayan establecido.

está contenido en catálogos, libros de texto y en la mente". Una vez que se reúne este conjunto diversificado de información, deben escogerse aquellos elementos del repertorio que cumplan los requerimientos definidos por la ingeniería y por el modelo de análisis (capítulos 5 a 7). A medida que esto ocurre, se evalúan las alternativas, algunas se rechazan, se converge en "una configuración particular de componentes y, con ello, en la creación del producto final" [Bel81].

La diversificación y la convergencia combinan la intuición y el criterio con base en la experiencia en la construcción de entidades similares, un conjunto de principios heurísticos que guían la forma en la que evoluciona el modelo, un conjunto de criterios que permiten evaluar la calidad y un proceso iterativo que finalmente conduce a una representación del diseño definitivo.

El diseño del software cambia continuamente conforme evolucionan los nuevos métodos, surgen mejores análisis y se obtiene una comprensión más amplia.[1] Incluso hoy, la mayor parte de las metodologías de diseño de software carece de profundidad, flexibilidad y naturaleza cuantitativa, que normalmente se asocian con las disciplinas de diseño de ingeniería más clásicas. No obstante, sí existen métodos para diseñar software, se dispone de criterios para el diseño con calidad y se aplica la notación del diseño. En este capítulo, se estudian los conceptos y principios fundamentales aplicables a todo el diseño de software, los elementos del modelo del diseño y el efecto que tienen los patrones en el proceso de diseño. En los capítulos 9 a 13 se presentarán varias metodologías de diseño de software, según se aplican en la obtención de arquitecturas e interfaces en el nivel de componente, así como a enfoques de diseño basados en patrones y orientados a web.

8.1 Diseño en el contexto de la ingeniería de software

Cita:

"El milagro más común de la ingeniería de software es la transición del análisis al diseño y de éste al código."

Richard Due'

El diseño de software se ubica en el área técnica de la ingeniería de software y se aplica sin importar el modelo del proceso que se utilice. El diseño del software comienza una vez que se han analizado y modelado los requerimientos, es la última acción de la ingeniería de software dentro de la actividad de modelado y prepara la etapa de **construcción** (generación y prueba de código).

Cada uno de los elementos del modelo de requerimientos (capítulos 6 y 7) proporciona información necesaria para crear los cuatro modelos de diseño necesarios para la especificación completa del diseño. En la figura 8.1 se ilustra el flujo de la información durante el diseño del software. El trabajo de diseño es alimentado por el modelo de requerimientos, manifestado por elementos basados en el escenario, en la clase, orientados al flujo, y del comportamiento. El empleo de la notación y de los métodos de diseño estudiados en los últimos capítulos produce diseños de los datos o clases, de la arquitectura, de la interfaz y de los componentes.

CONSEJO

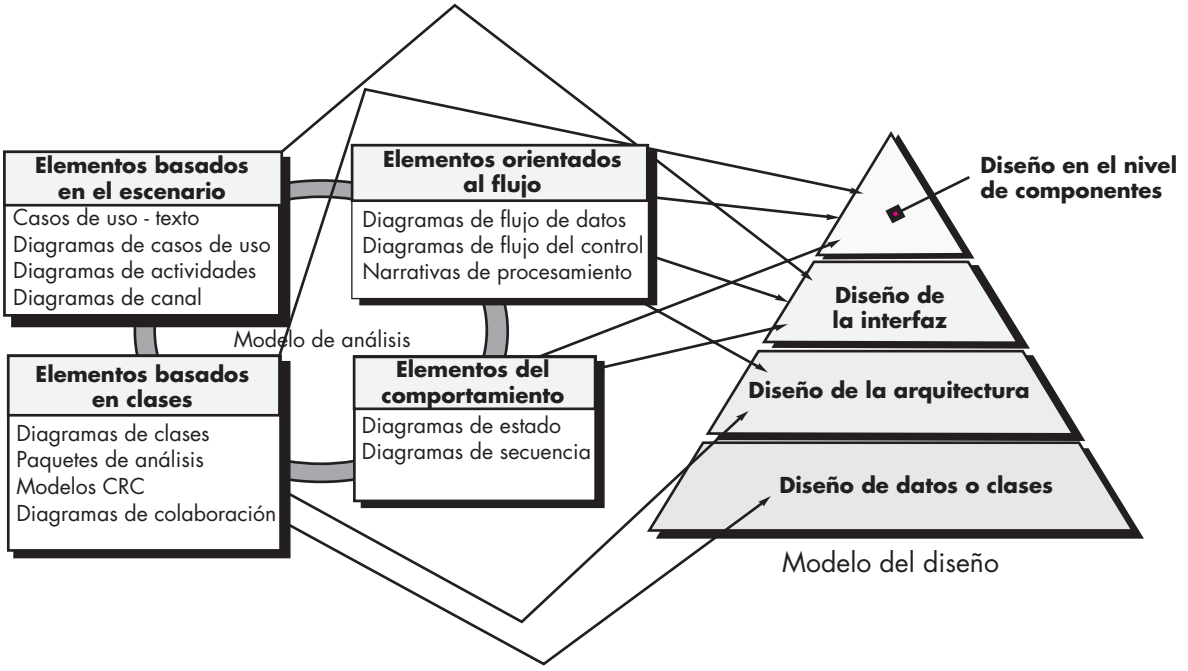
El diseño del software siempre debe comenzar con el análisis de los datos, pues son el fundamento de todos los demás elementos del diseño. Una vez obtenido el fundamento, se obtiene la arquitectura. Sólo entonces deben realizarse otros trabajos del diseño.

El diseño de datos o clases transforma los modelos de clases (capítulo 6) en realizaciones de clases de diseño y en las estructuras de datos que se requieren para implementar el software. Los objetos y relaciones definidos en el diagrama CRC y el contenido detallado de los datos ilustrado por los atributos de clase y otros tipos de notación dan la base para el diseño de los datos. Parte del diseño de clase puede llevarse a cabo junto con el diseño de la arquitectura del software. Un diseño más detallado de las clases tiene lugar cuando se diseña cada componente del software.

El diseño de la arquitectura define la relación entre los elementos principales de la estructura del software, los estilos y patrones de diseño de la arquitectura que pueden usarse para alcanzar

1 Aquellos lectores interesados en la filosofía del diseño de software pueden consultar el inquietante análisis de Philippe Kruchen sobre el diseño "posmoderno" [Kru05a].

FIGURA 8.1 Traducción del modelo de requerimientos al modelo de diseño



Cita:

"Hay dos formas de construir un diseño del software. Una es hacerlo tan simple que sea obvio que no hay deficiencias y la otra es hacerlo tan complicado que no haya deficiencias obvias. El primer método es mucho más difícil."

C. A. R. Hoare

los requerimientos definidos por el sistema y las restricciones que afectan la forma en la que se implementa la arquitectura [Sha96]. La representación del diseño de la arquitectura —el marco de un sistema basado en computadora— se obtiene del modelo de los requerimientos.

El diseño de la interfaz describe la forma en la que el software se comunica con los sistemas que interactúan con él y con los humanos que lo utilizan. Una interfaz implica un flujo de información (por ejemplo, datos o control) y un tipo específico de comportamiento. Entonces, los modelos de escenarios de uso y de comportamiento dan mucha de la información requerida para diseñar la interfaz.

El diseño en el nivel de componente transforma los elementos estructurales de la arquitectura del software en una descripción de sus componentes en cuanto a procedimiento. La información obtenida a partir de los modelos basados en clase, flujo y comportamiento sirve como la base para diseñar los componentes.

Durante el diseño se toman decisiones que en última instancia afectarán al éxito de la construcción del software y, de igual importancia, a la facilidad con la que puede darse mantenimiento al software. Pero, ¿por qué es tan importante el diseño?

La importancia del diseño del software se resume en una palabra: *calidad*. El diseño es el sitio en el que se introduce calidad en la ingeniería de software. Da representaciones del software que pueden evaluarse en su calidad. Es la única manera de traducir con exactitud a un producto o sistema terminado los requerimientos de los participantes. Es el fundamento de toda la ingeniería de software y de las actividades que dan el apoyo que sigue. Sin diseño se corre el riesgo de obtener un sistema inestable, que falle cuando se hagan cambios pequeños, o uno que sea difícil de someter a prueba, o en el que no sea posible evaluar la calidad hasta que sea demasiado tarde en el proceso de software, cuando no queda mucho tiempo y ya se ha gastado mucho dinero.

CasaSegura

Diseño versus codificación

La escena: El cubículo de Jamie, cuando el equipo se prepara para traducir a diseño los requerimientos.

Participantes: Jamie, Vinod y Ed, miembros del equipo de ingeniería de software para *CasaSegura*.

La conversación:

Jamie: Ustedes saben, Doug [el gerente del equipo] está obsesionado con el diseño. Tengo que ser honesto, lo que realmente amo es codificar. Denme C++ o Java y soy feliz.

Ed: No… te gusta diseñar.

Jamie: No me estás escuchando; codificar es lo mío.

Vinod: Creo que Ed quiere decir que en realidad no es codificar lo que te gusta; te gusta diseñar y expresarlo en código. El código es el lenguaje que usas para representar el diseño.

Jamie: ¿Y qué tiene de malo?

Vinod: El nivel de abstracción.

Jamie: ¿Qué?

Ed: Un lenguaje de programación es bueno para representar detalles tales como estructuras de datos y algoritmos, pero no es tan bueno para representar la arquitectura o la colaboración entre componentes… algo así.

Vinod: Y una arquitectura complicada arruina al mejor código.

Jamie (piensa unos momentos): Entonces, dicen que no puede representarse la arquitectura con código... eso no es cierto.

Vinod: Claro que es posible implicar la arquitectura con el código, pero en la mayor parte de lenguajes de programación, es muy difícil lograr un panorama rápido y amplio de la arquitectura con el análisis del código.

Ed: Y eso es lo que queremos hacer antes de empezar a codificar.

Jamie: Está bien, tal vez diseñar y codificar sean cosas distintas, pero aún así me gusta más codificar.

8.2 El proceso de diseño

El diseño de software es un proceso iterativo por medio del cual se traducen los requerimientos en un "plano" para construir el software. Al principio, el plano ilustra una visión holística del software. Es decir, el diseño se representa en un nivel alto de abstracción, en el que se rastrea directamente el objetivo específico del sistema y los requerimientos más detallados de datos, funcionamiento y comportamiento. A medida que tienen lugar las iteraciones del diseño, las mejoras posteriores conducen a niveles menores de abstracción. Éstos también pueden rastrearse hasta los requerimientos, pero la conexión es más sutil.

8.2.1 Lineamientos y atributos de la calidad del software

Cita:

"…escribir un fragmento inteligente de código que funcione es una cosa; diseñar algo que dé apoyo a largo plazo a una empresa es otra muy diferente".

C. Ferguson

A través del proceso de diseño se evalúa la calidad de éste de acuerdo con la serie de revisiones técnicas que se estudia en el capítulo 15. McGlaughlin [McG91] sugiere tres características que funcionan como guía para evaluar un buen diseño:

- Debe implementar todos los requerimientos explícitos contenidos en el modelo de requerimientos y dar cabida a todos los requerimientos implícitos que desean los participantes.
- Debe ser una guía legible y comprensible para quienes generan el código y para los que lo prueban y dan el apoyo posterior.
- Debe proporcionar el panorama completo del software, y abordar los dominios de los datos, las funciones y el comportamiento desde el punto de vista de la implementación.

En realidad, cada una de estas características es una meta del proceso de diseño. Pero, ¿cómo se logran?

Lineamientos de la calidad. A fin de evaluar la calidad de una representación del diseño, usted y otros miembros del equipo de software deben establecer los criterios técnicos de un buen diseño. En la sección 8.3 se estudian conceptos de diseño que también sirven como crite-

rios de calidad del software. En este momento, considere los siguientes lineamientos para el diseño:

¿Cuáles son las características de un buen diseño?

1. Debe tener una arquitectura que 1) se haya creado con el empleo de estilos o patrones arquitectónicos reconocibles, 2) esté compuesta de componentes con buenas características de diseño (éstas se analizan más adelante, en este capítulo), y 3) se implementen en forma evolutiva,[2] de modo que faciliten la implementación y las pruebas.
2. Debe ser modular, es decir, el software debe estar dividido de manera lógica en elementos o subsistemas.
3. Debe contener distintas representaciones de datos, arquitectura, interfaces y componentes.
4. Debe conducir a estructuras de datos apropiadas para las clases que se van a implementar y que surjan de patrones reconocibles de datos.
5. Debe llevar a componentes que tengan características funcionales independientes.
6. Debe conducir a interfaces que reduzcan la complejidad de las conexiones entre los componentes y el ambiente externo.
7. Debe obtenerse con el empleo de un método repetible motivado por la información obtenida durante el análisis de los requerimientos del software.
8. Debe representarse con una notación que comunique con eficacia su significado.

Estos lineamientos de diseño no se logran por azar. Se consiguen con la aplicación de los principios de diseño fundamentales, una metodología sistemática y con revisión.

INFORMACIÓN

Evaluación de la calidad del diseño. La revisión técnica

El diseño es importante porque permite que un equipo de software evalúe la calidad[3] de éste antes de que se implemente, momento en el que es fácil y barato corregir errores, omisiones o inconsistencias. Pero, ¿cómo se evalúa la calidad durante el diseño? El software no puede someterse a prueba porque no hay nada ejecutable. ¿Qué hacer?

Durante el diseño, la calidad se evalúa por medio de la realización de una serie de revisiones técnicas (RT). Las RT se estudian con detalle en el capítulo 15,[4] pero es útil hacer un resumen de dicha técnica en este momento. Una revisión técnica es una reunión celebrada por miembros del equipo de software. Por lo general, participan dos, tres o cuatro personas, en función del alcance de la información del diseño que se revisará. Cada persona tiene un papel: el *líder de la revisión* planea la reunión, establece la agenda y coordina la junta; el *secretario* toma notas para que no se pierda nada; el *productor* es la persona cuyo trabajo (por ejemplo, el diseño de un componente del software) se revisa. Antes de la reunión, se entrega a cada persona del equipo una copia del producto del trabajo de diseño y se le pide que la lea y que busque errores, omisiones o ambigüedades. El objetivo al comenzar la reunión es detectar todos los problemas del producto, de modo que puedan corregirse antes de que comience la implementación. Es común que la RT dure entre 90 minutos y 2 horas. Al final de ella, el equipo de revisión determina si se requiere de otras acciones por parte del productor a fin de que se apruebe el producto como porción del modelo del diseño final.

Cita:

"La calidad no es algo que se deje arriba de los sujetos y objetos como si fuera el remate de un árbol de navidad."

Robert Pirsig

Atributos de la calidad. Hewlett-Packard [Gra87] desarrolló un conjunto de atributos de la calidad del software a los que se dio el acrónimo FURPS: funcionalidad, usabilidad, confiabilidad, rendimiento y mantenibilidad. Los atributos de calidad FURPS representan el objetivo de todo diseño de software:

2 Para sistemas pequeños, en ocasiones el diseño puede desarrollarse en forma lineal.
3 Los factores de calidad que se estudian en el capítulo 23 ayudan al equipo de revisión cuando evalúa aquélla.
4 Tal vez el lector considere oportuno revisar el capítulo 15 en este momento. Las revisiones técnicas son una parte crítica del proceso de diseño y un mecanismo importante para lograr su calidad.

CONSEJO

Los diseñadores del software tienden a centrarse en el problema que se va a resolver. No olvide que los atributos FURPS siempre forman parte del problema. Deben tomarse en cuenta.

- La *funcionalidad* se califica de acuerdo con el conjunto de características y capacidades del programa, la generalidad de las funciones que se entregan y la seguridad general del sistema.
- La *usabilidad* se evalúa tomando en cuenta factores humanos (véase el capítulo 11), la estética general, la consistencia y la documentación.
- La *confiabilidad* se evalúa con la medición de la frecuencia y gravedad de las fallas, la exactitud de los resultados que salen, el tiempo medio para que ocurra una falla (TMPF), la capacidad de recuperación ante ésta y lo predecible del programa.
- El *rendimiento* se mide con base en la velocidad de procesamiento, el tiempo de respuesta, el uso de recursos, el conjunto y la eficiencia.
- La *mantenibilidad* combina la capacidad del programa para ser ampliable (extensibilidad), adaptable y servicial (estos tres atributos se denotan con un término más común: *mantenibilidad*), y además que pueda probarse, ser compatible y configurable (capacidad de organizar y controlar los elementos de la configuración del software, véase el capítulo 22) y que cuente con la facilidad para instalarse en el sistema y para que se detecten los problemas.

No todo atributo de la calidad del software se pondera por igual al diseñarlo. Una aplicación tal vez se aboque a lo funcional con énfasis en la seguridad. Otra quizá busque rendimiento con la mira puesta en la velocidad de procesamiento. En una tercera se persigue la confiabilidad. Sin importar la ponderación, es importante observar que estos atributos de la calidad deben tomarse en cuenta cuando comienza el diseño, *no* cuando haya terminado éste y la construcción se encuentre en marcha.

8.2.2 La evolución del diseño del software

Cita:

"Un diseñador sabe que alcanzó la perfección no cuando no hay nada por agregar, sino cuando no hay nada que quitar."

Antoine de Saint-Exupery

La evolución del diseño del software es un proceso continuo que ya ha cubierto casi seis décadas. Los primeros trabajos de diseño se concentraban en criterios para el desarrollo de programas modulares [Den73] y en métodos para mejorar estructuras de software con un enfoque de arriba abajo [Wir71]. Los aspectos de procedimiento del diseño evolucionaron hacia una filosofía llamada *programación estructurada* [Dah72], [Mil72]. Los trabajos posteriores propusieron métodos para traducir el flujo de datos [Ste74] o la estructura de éstos (por ejemplo, [Jac75], [War74]) a una definición de diseño. Los enfoques más nuevos (por ejemplo, [Jac92], [Gam95]) propusieron un enfoque orientado a objeto para diseñar derivaciones. En los últimos tiempos, el énfasis al desarrollar software se pone en la arquitectura de éste [Kru06] y en los patrones de diseño susceptibles de emplearse para implementar arquitecturas y niveles más bajos de abstracciones del diseño (por ejemplo, [Hol06], [Sha05]). Se da cada vez más importancia a los métodos orientados al aspecto (por ejemplo, [Cla05], [Jac04]), al desarrollo orientado al modelo [Sch06] y a las pruebas [Ast04], que se concentran en llegar a una modularidad eficaz y a la estructura arquitectónica de los diseños que se generan.

? ¿Qué características son comunes en todos los métodos de diseño?

En la industria del software se aplican varios métodos de diseño, aparte de los ya mencionados. Igual que los métodos de análisis presentados en los capítulos 6 y 7, cada método de diseño de software introduce heurística y notación únicas, así como un punto de vista sobre lo que caracteriza a la calidad en el diseño. No obstante, todos estos métodos tienen algunas características en común: 1) un mecanismo para traducir el modelo de requerimientos en una representación del diseño, 2) una notación para representar las componentes funcionales y sus interfaces, 3) una heurística para mejorar y hacer particiones y 4) lineamientos para evaluar la calidad.

Sin importar el método de diseño que se utilice, debe aplicarse un conjunto de conceptos básicos al diseño en el nivel de datos, arquitectura, interfaz y componente. En las secciones que siguen se estudian estos conceptos.

CONJUNTO DE TAREAS

Conjunto de tareas generales para el diseño

1. Estudiar el modelo del dominio de la información y diseñar las estructuras de datos apropiadas para los objetos de datos y sus atributos.
2. Seleccionar un estilo de arquitectura que sea adecuado para el software con el uso del modelo de análisis.
3. Hacer la partición del modelo de análisis en subsistemas de diseño y asignar éstos dentro de la arquitectura:
   - Asegúrese de que cada subsistema sea cohesivo en sus funciones.
   - Diseñe interfaces del subsistema.
   - Asigne clases de análisis o funciones a cada subsistema.
4. Crear un conjunto de clases de diseño o componentes:
   - Traduzca la descripción de clases de análisis a una clase de diseño.
   - Compare cada clase de diseño con los criterios de diseño; considere los aspectos hereditarios.
   - Defina métodos y mensajes asociados con cada clase de diseño.
   - Evalúe y seleccione patrones de diseño para una clase de diseño o subsistema.
   - Revise las clases de diseño y, si se requiere, modifíquelas.
5. Diseñar cualesquiera interfaces requeridas con sistemas o dispositivos externos.
6. Diseñar la interfaz de usuario.
   - Revise los resultados del análisis de tareas.
   - Especifique la secuencia de acciones con base en los escenarios de usuario.
   - Cree un modelo de comportamiento de la interfaz.
   - Defina los objetos de la interfaz y los mecanismos de control.
   - Revise el diseño de la interfaz y, si se requiere, modifíquelo.
7. Efectuar el diseño en el nivel de componente.
   - Especifique todos los algoritmos en un nivel de abstracción relativamente bajo.
   - Mejore la interfaz de cada componente.
   - Defina estructuras de datos en el nivel de componente.
   - Revise cada componente y corrija todos los errores que se detecten.
8. Desarrollar un modelo de despliegue.

8.3 CONCEPTOS DE DISEÑO

Durante la historia de la ingeniería de software, ha evolucionado un conjunto de conceptos fundamentales sobre su diseño. Aunque con el paso de los años ha variado el grado de interés en cada concepto, todos han soportado la prueba del tiempo. Cada uno da al diseñador del software el fundamento desde el que pueden aplicarse métodos de diseño sofisticados. Todos ayudan a responder las preguntas siguientes:

- ¿Qué criterios se usan para dividir el software en sus componentes individuales?
- ¿Cómo se extraen los detalles de la función o la estructura de datos de la representación conceptual del software?
- ¿Cuáles son los criterios uniformes que definen la calidad técnica de un diseño de software?

M. A. Jackson [Jac75] dijo: "El principio de la sabiduría [para un ingeniero de software] es reconocer la diferencia que hay entre hacer que un programa funcione y lograr que lo haga bien". Los conceptos fundamentales del diseño del software proveen la estructura necesaria para "hacerlo bien".

En las secciones que siguen, se da un panorama breve de los conceptos importantes del diseño de software, tanto del desarrollo tradicional como del orientado a objeto.

Cita:

"La abstracción es uno de los modos fundamentales con los que los humanos luchamos con la complejidad."

Grady Booch

8.3.1 Abstracción

Cuando se considera una solución modular para cualquier problema, es posible plantear muchos niveles de abstracción. En el más elevado se enuncia una solución en términos gruesos con el uso del lenguaje del ambiente del problema. En niveles más bajos de abstracción se da la descripción más detallada de la solución. La terminología orientada al problema se acopla con la que se orienta a la implementación, en un esfuerzo por enunciar la solución. Por último,

en el nivel de abstracción más bajo se plantea la solución, de modo que pueda implementarse directamente.

CONSEJO

Como diseñador, trabaje mucho para obtener abstracciones tanto de procedimiento como de datos que sirvan para el problema en cuestión. Será aún mejor si sirvieran para un dominio completo de problemas.

Cuando se desarrollan niveles de abstracción distintos, se trabaja para crear abstracciones tanto de procedimiento como de datos. Una *abstracción de procedimiento* es una secuencia de instrucciones que tienen una función específica y limitada. El nombre de la abstracción de procedimiento implica estas funciones, pero se omiten detalles específicos. Un ejemplo de esto sería la palabra *abrir*, en el caso de una puerta. *Abrir* implica una secuencia larga de pasos del procedimiento (caminar hacia la puerta, llegar y tomar el picaporte, girar éste y jalar la puerta, retroceder para que la puerta se abra, etcétera).[5]

Una *abstracción de datos* es un conjunto de éstos con nombre que describe a un objeto de datos. En el contexto de la abstracción de procedimiento *abrir*, puede definirse una abstracción de datos llamada **puerta**. Como cualquier objeto de datos, la abstracción de datos para **puerta** agruparía un conjunto de atributos que describirían la puerta (tipo, dirección del abatimiento, mecanismo de apertura, peso, dimensiones, etc.). Se concluye que la abstracción de procedimiento *abrir* usaría información contenida en los atributos de la abstracción de datos **puerta**.

8.3.2 Arquitectura

WebRef

En la dirección **www.sei.cmu.edu/ata/ata_init.html** hay un análisis profundo de la arquitectura del software.

La *arquitectura del software* alude a "la estructura general de éste y a las formas en las que ésta da integridad conceptual a un sistema" [Sha95a]. En su forma más sencilla, la arquitectura es la estructura de organización de los componentes de un programa (módulos), la forma en la que éstos interactúan y la estructura de datos que utilizan. Sin embargo, en un sentido más amplio, los componentes se generalizan para que representen los elementos de un sistema grande y sus interacciones.

Una meta del diseño del software es obtener una aproximación arquitectónica de un sistema. Ésta sirve como estructura a partir de la cual se realizan las actividades de diseño más detalladas. Un conjunto de patrones arquitectónicos permite que el ingeniero de software resuelva problemas de diseño comunes.

Cita:

"Una arquitectura del software es el producto del trabajo de desarrollo que tiene la rentabilidad más alta para una inversión en cuanto a calidad, secuencia de actividades y costo."

Len Bass *et al.*

Shaw y Garlan [Sha95a] describen un conjunto de propiedades que deben especificarse como parte del diseño de la arquitectura:

Propiedades estructurales. Este aspecto de la representación del diseño arquitectónico define los componentes de un sistema (módulos, objetos, filtros, etc.) y la manera en la que están agrupados e interactúan unos con otros. Por ejemplo, los objetos se agrupan para que encapsulen tanto datos como el procedimiento que los manipula e interactúen invocando métodos.

Propiedades extrafuncionales. La descripción del diseño arquitectónico debe abordar la forma en la que la arquitectura del diseño satisface los requerimientos de desempeño, capacidad, confiabilidad, seguridad y adaptabilidad, así como otras características del sistema.

Familias de sistemas relacionados. El diseño arquitectónico debe basarse en patrones repetibles que es común encontrar en el diseño de familias de sistemas similares. En esencia, el diseño debe tener la capacidad de reutilizar bloques de construcción arquitectónica.

CONSEJO

No deje al azar la arquitectura. Si lo hace, pasará el resto del proyecto forzándola para que se ajuste al diseño. Diseñe la arquitectura explícitamente.

Dada la especificación de estas propiedades, el diseño arquitectónico se representa con el uso de uno o más de varios modelos diferentes [Gar95]. Los *modelos estructurales* representan la arquitectura como un conjunto organizado de componentes del programa. Los *modelos de marco* aumentan el nivel de abstracción del diseño, al tratar de identificar patrones de diseño arquitectónico repetibles que se encuentran en tipos similares de aplicaciones. Los *modelos dinámicos* abordan los aspectos estructurales de la arquitectura del programa e indican cómo cambia la

5 Sin embargo, debe notarse que un conjunto de operaciones puede reemplazarse con otro, en tanto la función que implica la abstracción de procedimiento sea la misma. Por tanto, los pasos requeridos para implementar *abrir* cambiarían mucho si la puerta fuera automática y tuviera un sensor.

estructura o la configuración del sistema en función de eventos externos. Los *modelos del proceso* se centran en el diseño del negocio o proceso técnico al que debe dar acomodo el sistema. Por último, los *modelos funcionales* se usan para representar la jerarquía funcional de un sistema.

Para representar estos modelos, se ha desarrollado cierto número de *lenguajes de descripción arquitectónica* (LDA) [Sha95b]. Aunque han sido propuestos muchos LDA diferentes, la mayoría tiene mecanismos para describir los componentes del sistema y la manera en la que se conectan entre sí.

Debe observarse que hay un debate acerca del papel que tiene la arquitectura en el diseño. Algunos investigadores afirman que la obtención de la arquitectura del software debe separarse del diseño y que ocurre entre las acciones de la ingeniería de requerimientos y las del diseño más convencional. Otros piensan que la definición de la arquitectura es parte integral del proceso de diseño. En el capítulo 9 se estudia la forma en la que se caracteriza la arquitectura del software y su papel en el diseño.

Cita:

"Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente, por lo que describe el núcleo de la solución de ese problema, en forma tal que puede usarse ésta un millón de veces sin repetir lo mismo ni una sola vez."

Christopher Alexander

8.3.3 Patrones

Brad Appleton define un *patrón de diseño* de la manera siguiente: "Es una mezcla con nombre propio de puntos de vista que contienen la esencia de una solución demostrada para un problema recurrente dentro de cierto contexto de necesidades en competencia" [App00]. Dicho de otra manera, un patrón de diseño describe una estructura de diseño que resuelve un problema particular del diseño dentro de un contexto específico y entre "fuerzas" que afectan la manera en la que se aplica y en la que se utiliza dicho patrón.

El objetivo de cada patrón de diseño es proporcionar una descripción que permita a un diseñador determinar 1) si el patrón es aplicable al trabajo en cuestión, 2) si puede volverse a usar (con lo que se ahorra tiempo de diseño) y 3) si sirve como guía para desarrollar un patrón distinto en funciones o estructura. En el capítulo 12 se estudian los patrones de diseño.

8.3.4 División de problemas

La *división de problemas* es un concepto de diseño que sugiere que cualquier problema complejo puede manejarse con más facilidad si se subdivide en elementos susceptibles de resolverse u optimizarse de manera independiente. Un *problema* es una característica o comportamiento que se especifica en el modelo de los requerimientos para el software. Al separar un problema en sus piezas más pequeñas y por ello más manejables, se requiere menos esfuerzo y tiempo para resolverlo.

El argumento para separar los problemas puede llevarse demasiado lejos. Si se divide un problema en un número muy grande de problemas muy pequeños, será fácil resolver cada uno de éstos, pero unificarlos en la solución (integración) será muy difícil.

Si para dos problemas, p_1 y p_2, la complejidad que se percibe para p_1 es mayor que la percibida para p_2, entonces se concluye que el esfuerzo requerido para resolver p_1 es mayor que el necesario para resolver p_2. Como caso general, este resultado es intuitivamente obvio. Lleva más tiempo resolver un problema difícil.

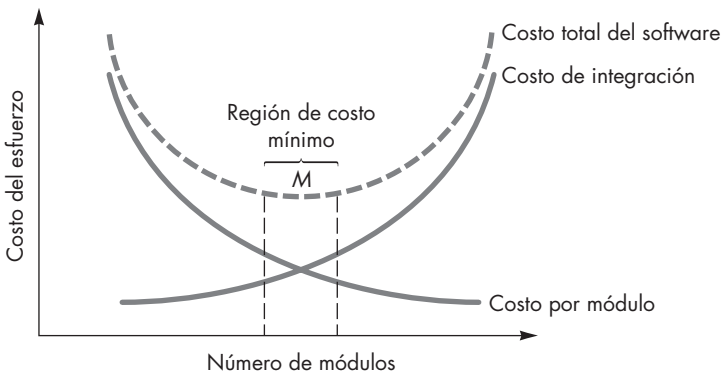
También se concluye que cuando se combinan dos problemas, con frecuencia la complejidad percibida es mayor que la suma de la complejidad tomada por separado. Esto lleva a la estrategia de divide y vencerás, pues es más fácil resolver un problema complejo si se divide en elementos manejables. Esto tiene implicaciones importantes en relación con la modularidad del software.

La división de problemas se manifiesta en otros conceptos de diseño relacionados: modularidad, aspectos, independencia de funcionamiento y mejora. Cada uno de éstos se estudiará en las secciones siguientes.

8.3.5 Modularidad

La modularidad es la manifestación más común de la división de problemas. El software se divide en componentes con nombres distintos y abordables por separado, en ocasiones llamados *módulos*, que se integran para satisfacer los requerimientos del problema.

FIGURA 8.2
Modularidad y costo del software



Se ha dicho que "la modularidad es el único atributo del software que permite que un programa sea manejable en lo intelectual" [Mye78]. El software monolítico (un programa grande compuesto de un solo módulo) no es fácil de entender para un ingeniero de software. El número de trayectorias de control, alcance de referencia, número de variables y complejidad general haría que comprenderlo fuera casi imposible. En función de las circunstancias, el diseño debe descomponerse en muchos módulos con la esperanza de que sea más fácil entenderlos y, en consecuencia, reducir el costo requerido para elaborar el software.

Según el punto de vista de la división de problemas, sería posible concluir que si el software se dividiera en forma indefinida, el esfuerzo requerido para desarrollarlo ¡sería despreciable por pequeño! Desafortunadamente, hay otras fuerzas que entran en juego y que hacen que esta conclusión sea (tristemente) inválida. De acuerdo con la figura 8.2, el esfuerzo (costo) de desarrollar un módulo individual de software disminuye conforme aumenta el número total de módulos. Dado el mismo conjunto de requerimientos, tener más módulos significa tamaños individuales más pequeños. Sin embargo, a medida que se incrementa el número de módulos, el esfuerzo (costo) asociado con su integración también aumenta. Estas características llevan a una curva de costo total como la que se muestra en la figura. Existe un número, *M*, de módulos que arrojarían el mínimo costo de desarrollo, pero no se dispone de las herramientas necesarias para predecir *M* con exactitud.

? ¿Cuál es el número correcto de módulos para un sistema dado?

Las curvas que aparecen en la figura 8.2 constituyen una guía útil al considerar la modularidad. Deben hacerse módulos, pero con cuidado para permanecer en la cercanía de *M*. Debe evitarse hacer pocos o muchos módulos. Pero, ¿cómo saber cuál es la cercanía de *M*? ¿Cuán modular debe hacerse el software? Las respuestas a estas preguntas requieren la comprensión de otros conceptos de diseño que se analizan más adelante en este capítulo.

Debe hacerse un diseño (y el programa resultante) con módulos, de manera que el desarrollo pueda planearse con más facilidad, que sea posible definir y desarrollar los incrementos del software, que los cambios se realicen con más facilidad, que las pruebas y la depuración se efectúen con mayor eficiencia y que el mantenimiento a largo plazo se lleve a cabo sin efectos colaterales de importancia.

8.3.6 Ocultamiento de información

PUNTO CLAVE
El objetivo de ocultar la información es esconder los detalles de las estructuras de datos y el procesamiento tras una interfaz de módulo. No es necesario que los usuarios de éste los conozcan.

El concepto de modularidad lleva a una pregunta fundamental: "¿Cómo descomponer una solución de software para obtener el mejor conjunto de módulos?" El principio del ocultamiento de información sugiere que los módulos se "caractericen por decisiones de diseño que se oculten (cada una) de las demás". En otras palabras, deben especificarse y diseñarse módulos, de forma que la información (algoritmos y datos) contenida en un módulo sea inaccesible para los que no necesiten de ella.

El ocultamiento implica que la modularidad efectiva se logra definiendo un conjunto de módulos independientes que intercambien sólo aquella información necesaria para lograr la función del software. La abstracción ayuda a definir las entidades de procedimiento (o informativas) que constituyen el software. El ocultamiento define y hace cumplir las restricciones de acceso tanto a los detalles de procedimiento como a cualquier estructura de datos local que utilice el módulo [Ros75].

El uso del ocultamiento de información como criterio de diseño para los sistemas modulares proporciona los máximos beneficios cuando se requiere hacer modificaciones durante las pruebas, y más adelante, al dar mantenimiento al software. Debido a que la mayoría de los datos y detalles del procedimiento quedan ocultos para otras partes del software, es menos probable que los errores inadvertidos introducidos durante la modificación se propaguen a distintos sitios dentro del software.

8.3.7 Independencia funcional

El concepto de independencia funcional es resultado directo de la separación de problemas y de los conceptos de abstracción y ocultamiento de información. En escritos cruciales sobre el diseño de software, Wirth [Wir71] y Parnas [Par72] mencionan técnicas de mejora que promueven la independencia modular. Los trabajos posteriores de Stevens, Myers y Constantine [Ste74] dan solidez al concepto.

La independencia funcional se logra desarrollando módulos con funciones "miopes" que tengan "aversión" a la interacción excesiva con otros módulos. Dicho de otro modo, debe diseñarse software de manera que cada módulo resuelva un subconjunto específico de requerimientos y tenga una interfaz sencilla cuando se vea desde otras partes de la estructura del programa. Es lógico preguntar por qué es importante la independencia.

? ¿Por qué debe tratarse de crear módulos independientes?

El software con modularidad eficaz, es decir, con módulos independientes, es más fácil de desarrollar porque su función se subdivide y las interfaces se simplifican (cuando el desarrollo es efectuado por un equipo hay que considerar las ramificaciones). Los módulos independientes son más fáciles de mantener (y probar) debido a que los efectos secundarios causados por el diseño o por la modificación del código son limitados, se reduce la propagación del error y es posible obtener módulos reutilizables. En resumen, la independencia funcional es una clave para el buen diseño y éste es la clave de la calidad del software.

PUNTO CLAVE

La cohesión es un indicador cualitativo del grado en el que un módulo se centra en hacer una sola cosa.

La independencia se evalúa con el uso de dos criterios cualitativos: la cohesión y el acoplamiento. La *cohesión* es un indicador de la fortaleza relativa funcional de un módulo. El *acoplamiento* lo es de la independencia relativa entre módulos.

La cohesión es una extensión natural del concepto de ocultamiento de información descrito en la sección 8.3.6. Un módulo cohesivo ejecuta una sola tarea, por lo que requiere interactuar poco con otros componentes en otras partes del programa. En pocas palabras, un módulo cohesivo debe (idealmente) hacer sólo una cosa. Aunque siempre debe tratarse de lograr mucha cohesión (por ejemplo, una sola tarea), con frecuencia es necesario y aconsejable hacer que un componente de software realice funciones múltiples. Sin embargo, para lograr un buen diseño hay que evitar los componentes "esquizofrénicos" (módulos que llevan a cabo funciones no relacionadas).

PUNTO CLAVE

El acoplamiento es un indicador cualitativo del grado en el que un módulo está conectado con otros y con el mundo exterior.

El acoplamiento es una indicación de la interconexión entre módulos en una estructura de software, y depende de la complejidad de la interfaz entre módulos, del grado en el que se entra o se hace referencia a un módulo y de qué datos pasan a través de la interfaz. En el diseño de software, debe buscarse el mínimo acoplamiento posible. La conectividad simple entre módulos da como resultado un software que es más fácil de entender y menos propenso al "efecto de oleaje" [Ste74], ocasionado cuando ocurren errores en un sitio y se propagan por todo el sistema.

8.3.8 Refinamiento

CONSEJO

Existe la tendencia a pasar de inmediato a los detalles e ignorar los pasos del refinamiento. Esto genera errores y hace que el diseño sea mucho más difícil de revisar. Realice refinamiento stepwise.

El refinamiento stepwise es una estrategia de diseño propuesta originalmente por Niklaus Wirth [Wir71]. Un programa se elabora por medio del refinamiento sucesivo de los detalles del procedimiento. Se desarrolla una jerarquía con la descomposición de un enunciado macroscópico de la función (abstracción del procedimiento) en forma escalonada hasta llegar a los comandos del lenguaje de programación.

En realidad, el refinamiento es un proceso de *elaboración*. Se comienza con un enunciado de la función (o descripción de la información), definida en un nivel de abstracción alto. Es decir, el enunciado describe la función o información de manera conceptual, pero no dice nada sobre los trabajos internos de la función o de la estructura interna de la información. Después se trabaja sobre el enunciado original, dando más y más detalles conforme tiene lugar el refinamiento (elaboración) sucesivo.

La abstracción y el refinamiento son conceptos complementarios. La primera permite especificar internamente el procedimiento y los datos, pero elimina la necesidad de que los "extraños" conozcan los detalles de bajo nivel. El refinamiento ayuda a revelar estos detalles a medida que avanza el diseño. Ambos conceptos permiten crear un modelo completo del diseño conforme éste evoluciona.

8.3.9 Aspectos

Cita:

"Es difícil leer un libro sobre los principios de la magia sin echar una mirada de vez en cuando a la portada para asegurarse de que no es un texto sobre diseño de software."

Bruce Tognazzini

Conforme tiene lugar el análisis de los requerimientos, surge un conjunto de "preocupaciones" que "incluyen requerimientos, casos de uso, características, estructuras de datos, calidad del servicio, variantes, fronteras de las propiedades intelectuales, colaboraciones, patrones y contratos" [AOS07]. Idealmente, un modelo de requerimientos se organiza de manera que permita aislar cada preocupación (requerimiento) a fin de considerarla en forma independiente. Sin embargo, en la práctica, algunas de estas preocupaciones abarcan todo el sistema y no es fácil dividirlas en compartimientos.

Cuando comienza el diseño, los requerimientos son refinados en una representación de diseño modular. Considere dos requerimientos, *A* y *B*. El *A interfiere* con el *B* "si se ha elegido una descomposición [refinamiento] en la que *B* no puede satisfacerse sin tomar en cuenta a *A*" [Ros04].

Por ejemplo, considere dos requerimientos para la *webapp* **CasaSeguraAsegurada.com**. El requerimiento *A* se describe con el caso de uso AVC-DVC analizado en el capítulo 6. Un refinamiento del diseño se centraría en aquellos módulos que permitieran que usuarios registrados accedieran al video de cámaras situadas en un espacio. El requerimiento *B* es de seguridad y establece que *un usuario registrado debe ser validado antes de que use* **CasaSeguraAsegurada.com**. Este requerimiento es aplicable a todas las funciones disponibles para los usuarios registrados de *CasaSegura*. Cuando ocurre el refinamiento del diseño, *A** es una representación del diseño para el requerimiento *A*, y *B** es otra para el requerimiento *B*. Por tanto, *A** y *B** son representaciones de las preocupaciones, y *B* interfiere* con *A**.

PUNTO CLAVE

Una preocupación de interferencia es alguna característica del sistema que se aplica a través de muchos requerimientos distintos.

Un *aspecto* es una representación de una preocupación de interferencia. Entonces, la representación del diseño, *B**, del requerimiento *un usuario registrado debe ser validado antes de que use* **CasaSeguraAsegurada.com** es un aspecto de la *webapp CasaSegura*. Es importante identificar aspectos, de modo que el diseño les pueda dar acomodo conforme sucede el refinamiento y la división en módulos. En un contexto ideal, un aspecto se implementa como módulo (componente) separado y no como fragmentos de software "dispersos" o "regados" en muchos componentes [Ban06]. Para lograr esto, la arquitectura del diseño debe apoyar un mecanismo para definir aspecto: un módulo que permita implementar la preocupación en todas aquellas con las que interfiera.

8.3.10 Rediseño

WebRef
En la dirección **www.refactoring.com**, se encuentran recursos excelentes para el rediseño.

WebRef
En **http://c2.com/cgi/wiki?RefactoringPatterns**, se encuentran varios patrones de rediseño.

Una actividad de diseño importante que se sugiere para muchos métodos ágiles (véase el capítulo 3) es el *rediseño*, técnica de reorganización que simplifica el diseño (o código) de un componente sin cambiar su función o comportamiento. Fowler [Fow00] define el rediseño del modo siguiente: "Es el proceso de cambiar un sistema de software en forma tal que no se altera el comportamiento externo del código [diseño], pero sí se mejora su estructura interna."

Cuando se rediseña el software, se examina el diseño existente en busca de redundancias, elementos de diseño no utilizados, algoritmos ineficientes o innecesarios, estructuras de datos mal construidas o inapropiadas y cualquier otra falla del diseño que pueda corregirse para obtener un diseño mejor. Por ejemplo, una primera iteración de diseño tal vez genere un componente con poca cohesión (realiza tres funciones que tienen poca relación entre sí). Después de un análisis cuidadoso, se decide rediseñar el componente en tres componentes separados, cada uno con mucha cohesión. El resultado será un software más fácil de integrar, de probar y de mantener.

CASASEGURA

Conceptos de diseño

La escena: Cubículo de Vinod, cuando comienza el modelado del diseño.

Participantes: Vinod, Jamie y Ed, miembros del equipo de ingeniería del software de *CasaSegura*. También Shakira, nueva integrante del equipo.

La conversación:

[Los cuatro miembros del equipo acaban de regresar de un seminario matutino llamado "Aplicación de los conceptos básicos del diseño", ofrecido por una profesora local de ciencias de la computación.]

Vinod: ¿Les dejó algo el seminario?

Ed: Sabíamos la mayor parte de lo que trató, pero creo que no fue mala idea escucharlo de nuevo.

Jamie: Cuando estudiaba la carrera de ciencias de la computación, nunca entendí, en realidad, por qué era tan importante, como decían, ocultar información.

Vinod: Por… la línea de base… es una técnica para reducir la propagación del error en un programa. En realidad, la independencia funcional hace lo mismo.

Shakira: Yo no estudié una carrera de computación, así que mucho de lo que dijo el instructor fue nuevo para mí. Soy capaz de generar buen código y rápido. No veo por qué es tan importante todo eso.

Jamie: He visto tu trabajo, Shak, y aplicas de manera natural mucho de lo que se habló… ésa es la razón por la que funcionan bien tus diseños y códigos.

Shakira (sonríe): Bueno, siempre trato de realizar la partición del código, hacer que se aboque a una cosa, construir interfaces sencillas y restringidas, reutilizar código siempre que se pueda… esa clase de cosas.

Ed: Modularidad, independencia funcional, ocultamiento, patrones… ya veo.

Jamie: Todavía recuerdo el primer curso de programación que tomé… nos enseñaron a refinar el código en forma iterativa.

Vinod: Lo mismo puede aplicarse al diseño, ya sabes.

Vinod: Los únicos conceptos que no había escuchado antes fueron los de "aspectos" y "rediseño".

Shakira: Eso se utiliza en programación extrema.

Ed: Sí. No es muy diferente del refinamiento, sólo que lo haces una vez terminado el diseño o código. Si me preguntan, diré que es algo así como una etapa de optimización del software.

Jamie: Volvamos al diseño de *CasaSegura*. Pienso que mientras desarrollemos el modelo de su diseño, debemos poner estos conceptos en nuestra lista de revisión.

Vinod: Estoy de acuerdo. Pero es importante que todos nos comprometamos a pensar en ellos al hacer el diseño.

8.3.11 Conceptos de diseño orientados a objeto

El paradigma de la orientación a objeto (OO) se utiliza mucho en la ingeniería de software moderna. El apéndice 2 está pensado para aquellos lectores que no estén familiarizados con los conceptos de diseño OO, tales como clases y objetos, herencia, mensajes y polimorfismo, entre otros.

8.3.12 Clases de diseño

El modelo de requerimientos define un conjunto de clases de análisis (capítulo 6). Cada una describe algún elemento del dominio del problema y se centra en aspectos de éste que son visibles para el usuario. El nivel de abstracción de una clase de análisis es relativamente alto.

Conforme el diseño evoluciona, se definirá un conjunto de *clases de diseño* que refinan las clases de análisis, dando detalles del diseño que permitirán que las clases se implementen y generen una infraestructura para el software que apoye la solución de negocios. Pueden desarrollarse cinco tipos diferentes de clases de diseño, cada una de las cuales representa una capa distinta de la arquitectura del diseño [Amb01]:

? ¿Qué tipos de clases crea el diseñador?

- *Clases de usuario de la interfaz*. Definen todas las abstracciones necesarias para la interacción humano-computadora (IHC). En muchos casos, la IHC ocurre dentro del contexto de una *metáfora* (por ejemplo, cuaderno de notas, formato de orden, máquina de fax, etc.) y las clases del diseño para la interfaz son representaciones visuales de los elementos de la metáfora.
- *Clases del dominio de negocios*. Es frecuente que sean refinamientos de las clases de análisis definidas antes. Las clases identifican los atributos y servicios (métodos) que se requieren para implementar algunos elementos del dominio de negocios.
- *Clases de proceso*. Implantan abstracciones de negocios de bajo nivel que se requieren para administrar por completo las clases de dominio de negocios.
- *Clases persistentes*. Representan almacenamientos de datos (por ejemplo, una base de datos) que persistirán más allá de la ejecución del software.
- *Clases de sistemas*. Implantan las funciones de administración y control del software que permiten que el sistema opere y se comunique dentro de su ambiente de computación y con el mundo exterior.

A medida que se forma la arquitectura, el nivel de abstracción se reduce cuando cada clase de análisis se transforma en una representación del diseño. Es decir, las clases de análisis representan objetos de datos (y servicios asociados que se aplican a éstos) que usan la terminología del dominio del negocio. Las clases de diseño presentan muchos más detalles técnicos como guía para su implementación.

Arlow y Neustadt [Arl02] sugieren que se revise cada clase de diseño para asegurar que esté "bien formada". Definen cuatro características de las clases de diseño bien formadas:

? ¿Qué es una clase de diseño "bien formada"?

Completa y suficiente. Una clase de diseño debe ser el encapsulado total de todos los atributos y métodos que sea razonable esperar (con base en una interpretación comprensible del nombre de la clase) y que existan para la clase. Por ejemplo, la clase **Escena** definida para el software de la edición de video será completa sólo si contiene todos los atributos y métodos que se asocian de manera razonable con la creación de una escena de video. La suficiencia asegura que la clase de diseño contiene sólo los métodos que bastan para lograr el objetivo de la clase, ni más ni menos.

Primitivismo. Los métodos asociados con una clase de diseño deben centrarse en el cumplimiento de un servicio para la clase. Una vez implementado el servicio con un método, la clase no debe proveer otro modo de hacer lo mismo. Por ejemplo, la clase **VideoClip** para el software de la edición de video tal vez tenga los atributos **punto-inicial** y **punto-final** que indiquen los puntos de inicio y fin del corto (observe que el video original cargado en el sistema puede ser más extenso que el corto utilizado). Los métodos *EstablecerPuntoInicial ( )* y *EstablecerPuntoFinal ( )* proporcionan los únicos medios para establecer los puntos de comienzo y terminación del corto.

Mucha cohesión. Una clase de diseño cohesiva tiene un conjunto pequeño y centrado de responsabilidades; para implementarlas emplea atributos y métodos de objetivo único. Por

ejemplo, la clase **VideoClip** quizá contenga un conjunto de métodos para editar el corto de video. La cohesión se mantiene en tanto cada método se centre sólo en los atributos asociados con el corto.

Poco acoplamiento. Dentro del modelo de diseño, es necesario que las clases de diseño colaboren una con otra. Sin embargo, la colaboración debe mantenerse en un mínimo aceptable. Si un modelo de diseño está muy acoplado (todas las clases de diseño colaboran con todas las demás), el sistema es difícil de implementar, probar y mantener con el paso del tiempo. En general, las clases de diseño dentro de un subsistema deben tener sólo un conocimiento limitado de otras clases. Esta restricción se llama *Ley de Demeter* [Lie03] y sugiere que un método sólo debe enviar mensajes a métodos que están en clases vecinas.[6]

CASASEGURA

Refinamiento de una clase de análisis en una clase de diseño

La escena: El cubículo de Ed, cuando comienza el modelado del diseño.

Participantes: Vinod y Ed, miembros del equipo de ingeniería de software de *CasaSegura*.

La conversación:

[Ed está trabajando en la clase **PlanodelaPlanta** (véanse el recuadro en la sección 6.5.3 y la figura 6.10) y la ha refinado para el modelo del diseño.]

Ed: Entonces recuerdas la clase **PlanodelaPlanta**, ¿verdad? Se usa como parte de las funciones de vigilancia y administración de la casa.

Vinod (afirma con la cabeza): Sí, recuerdo que la usamos como parte de nuestros análisis CRC para la administración de la casa.

Ed: Así es. De cualquier modo, la estoy mejorando para el diseño. Quiero mostrarte cómo implantaremos en realidad la clase **PlanodelaPlanta**. Mi idea es implementarla como un conjunto de listas ligadas [una estructura de datos específica] de modo que… tuve que refinar la clase de análisis **PlanodelaPlanta** (véase la figura 6.10) y, en verdad, simplificarla.

Vinod: La clase de análisis sólo mostraba cosas en el dominio del problema, bueno, en la pantalla de la computadora, visibles para el usuario final, ¿de acuerdo?

Ed: Sí, pero para la clase de diseño **PlanodelaPlanta**, he tenido que agregar algunas cosas específicas para la implantación. Necesité mostrar que **PlanodelaPlanta** es un agregado de segmentos —de ahí la clase **Segmento**— y que la clase **Segmento** está compuesta de listas para segmentos de pared, ventanas, puertas, etc. La clase **Cámara** colabora con **PlanodelaPlanta** y, obviamente, hay muchas cámaras en el piso.

Vinod: Ah… veamos la ilustración de esta nueva clase de diseño, **PlanodelaPlanta**.

[Ed muestra a Vinod el diagrama que aparece en la figura 8.3.]

Vinod: Bien, ya veo lo que tratas de hacer. Esto te permite modificar el plano de la planta con facilidad porque los nuevos temas se agregan, o eliminan de la lista (el agregado), sin problemas.

Ed (asiente): Sí, creo que funcionará.

Vinod: También yo.

8.4 EL MODELO DEL DISEÑO

El modelo del diseño puede verse en dos dimensiones distintas, como se ilustra en la figura 8.4. La *dimensión del proceso* indica la evolución del modelo del diseño conforme se ejecutan las tareas de éste como parte del proceso del software. La *dimensión de la abstracción* representa el nivel de detalle a medida que cada elemento del modelo de análisis se transforma en un equivalente de diseño y luego se mejora en forma iterativa. En relación con la figura 8.4, la línea punteada indica la frontera entre los modelos de análisis y de diseño. En ciertos casos, es posible hacer una distinción clara entre ambos modelos. En otros, el modelo de análisis se mezcla poco a poco con el de diseño y la distinción es menos obvia.

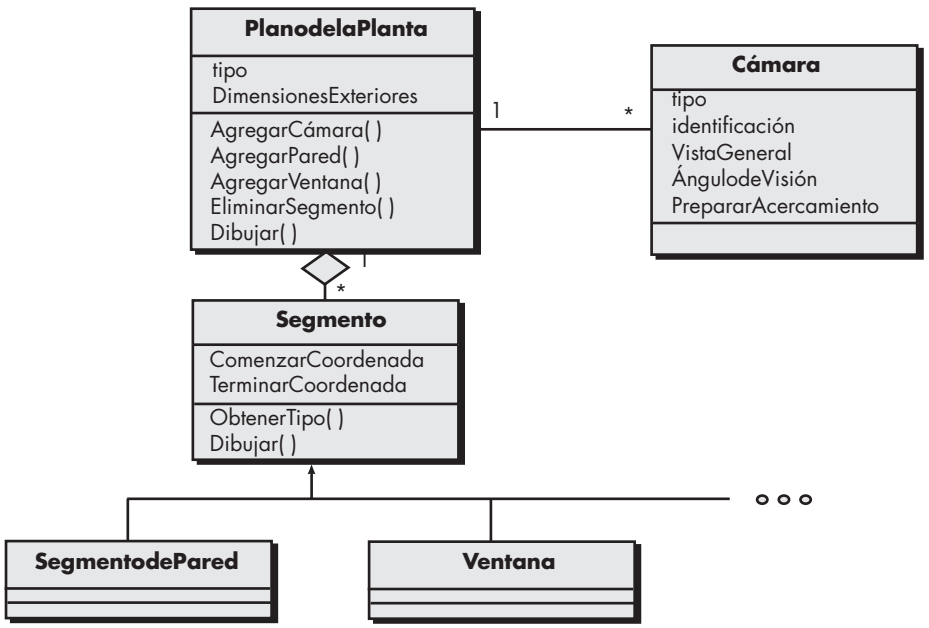
Los elementos del modelo de diseño usan muchos de los diagramas UML[7] que se utilizaron en el modelo del análisis. La diferencia es que estos diagramas se refinan y elaboran como parte

6 Una manera menos formal de la Ley de Demeter es: "cada unidad debe hablar sólo con sus amigas: no hablar con extraños".

7 En el apéndice 1 se encuentra un método de enseñanza sobre los conceptos y notación básica del UML.

FIGURA 8.3

Clase de diseño para PlanodelaPlanta y composición del agregado para ella (véase el análisis en el recuadro)



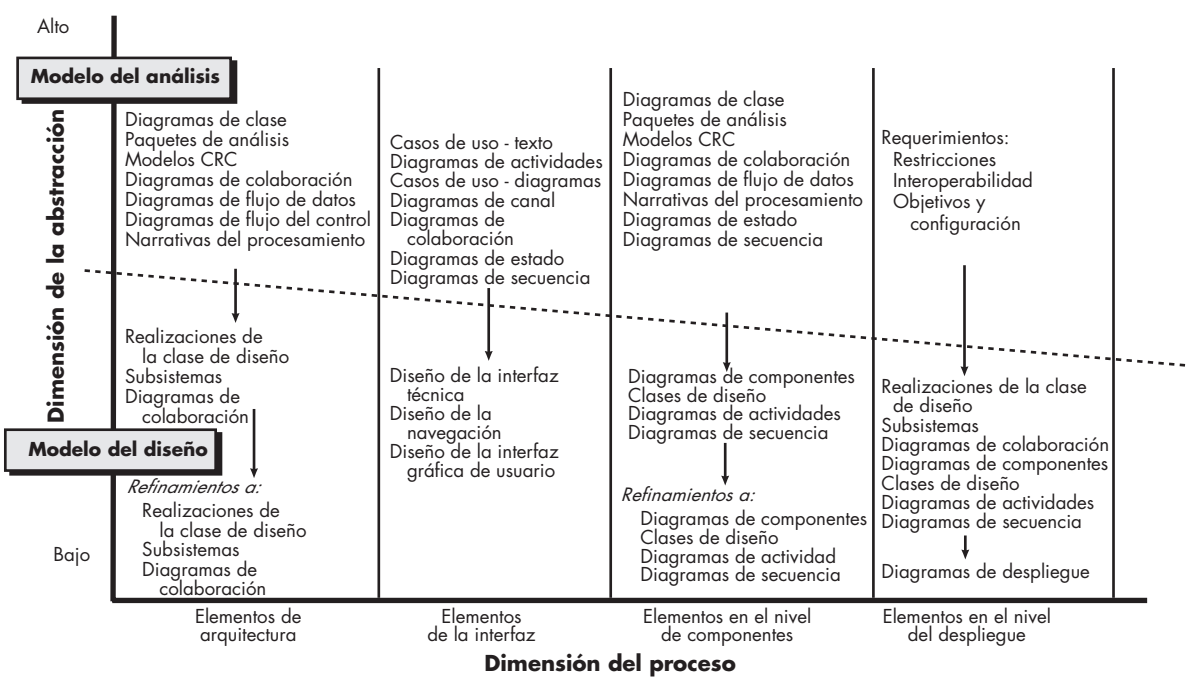
PUNTO CLAVE

El modelo de diseño tiene cuatro elementos principales: datos, arquitectura, componentes e interfaz.

del diseño; se dan más detalles específicos de la implantación y se hace énfasis en la estructura y en el estilo arquitectónico, en los componentes que residen dentro de la arquitectura y en las interfaces entre los componentes y el mundo exterior.

No obstante, debe observarse que los elementos del modelo indicados a lo largo del eje horizontal no siempre se desarrollan en forma secuencial. En la mayoría de los casos, el diseño preliminar de la arquitectura establece la etapa y va seguido del diseño de la interfaz y del dise-

FIGURA 8.4 Dimensiones del modelo de diseño



Cita:

"Las preguntas acerca de si el diseño es necesario o digno de pagarse están más allá de la discusión: el diseño es inevitable. La alternativa al buen diseño es el mal diseño, no la falta de diseño."

Douglas Martin

ño del nivel de los componentes, los cuales con frecuencia ocurren en paralelo. El modelo de despliegue por lo general se retrasa hasta que el diseño haya sido desarrollado por completo.

Es posible aplicar patrones de diseño en cualquier punto de este proceso (véase el capítulo 12). Estos patrones permiten aplicar el conocimiento del diseño a problemas específicos del dominio que han sido encontrados y resueltos por otras personas.

8.4.1 Elementos del diseño de datos

Igual que otras actividades de la ingeniería de software, el diseño de datos (en ocasiones denominado *arquitectura de datos*) crea un modelo de datos o información que se representa en un nivel de abstracción elevado (el punto de vista del usuario de los datos). Este modelo de los datos se refina después en forma progresiva hacia representaciones más específicas de la implementación que puedan ser procesadas por el sistema basado en computadora. En muchas aplicaciones de software, la arquitectura de los datos tendrá una influencia profunda en la arquitectura del software que debe procesarlo.

PUNTO CLAVE

En el nivel de la arquitectura (aplicación), el diseño de los datos se centra en archivos o bases de datos; en el de los componentes, el diseño de datos considera las estructuras de datos que se requieren para implementar objetos de datos locales.

La estructura de los datos siempre ha sido parte importante del diseño de software. En el nivel de componentes del programa, del diseño de las estructuras de datos y de los algoritmos requeridos para manipularlos, es esencial la creación de aplicaciones de alta calidad. En el nivel de la aplicación, la traducción de un modelo de datos (obtenido como parte de la ingeniería de los requerimientos) a una base de datos es crucial para lograr los objetivos de negocios de un sistema. En el nivel de negocios, el conjunto de información almacenada en bases de datos incompatibles y reorganizados en un "data warehouse" permite la minería de datos o descubrimiento de conocimiento que tiene un efecto en el éxito del negocio en sí. En cada caso, el diseño de los datos juega un papel importante. El diseño de datos se estudia con más detalle en el capítulo 9.

8.4.2 Elementos del diseño arquitectónico

Cita:

"Puede usarse una goma en la mesa de dibujo o un marro en el sitio construido."

Frank Lloyd Wright

El *diseño de la arquitectura* del software es el equivalente del plano de una casa. Éste ilustra la distribución general de las habitaciones, su tamaño, forma y relaciones entre ellas, así como las puertas y ventanas que permiten el movimiento entre los cuartos. El plano da una visión general de la casa. Los elementos del diseño de la arquitectura dan la visión general del software.

El modelo arquitectónico [Sha96] proviene de tres fuentes: 1) información sobre el dominio de la aplicación del software que se va a elaborar, 2) los elementos específicos del modelo de requerimientos, tales como diagramas de flujo de datos o clases de análisis, sus relaciones y colaboraciones para el problema en cuestión y 3) la disponibilidad de estilos arquitectónicos (capítulo 9) y sus patrones (capítulo 12).

Por lo general, el elemento de diseño arquitectónico se ilustra como un conjunto de sistemas interconectados, con frecuencia obtenidos de paquetes de análisis dentro del modelo de requerimientos. Cada subsistema puede tener su propia arquitectura (por ejemplo, la interfaz gráfica de usuario puede estar estructurada de acuerdo con un estilo de arquitectura preexistente para interfaces de usuario). En el capítulo 9 se presentan técnicas para obtener elementos específicos del modelo arquitectónico.

Cita:

"El público está más familiarizado con el mal diseño que con el buen diseño. En realidad, está condicionado para que prefiera el mal diseño porque es con el que vive. Lo nuevo le parece amenazador; lo viejo le da seguridad."

Paul Rand

8.4.3 Elementos de diseño de la interfaz

El diseño de la interfaz para el software es análogo al conjunto de trazos (y especificaciones) detalladas para las puertas, ventanas e instalaciones de una casa. Tales dibujos ilustran el tamaño y forma de puertas y ventanas, la manera en la que operan, la forma en la que llegan las instalaciones de servicios (agua, electricidad, gas, teléfono, etc.) a la vivienda y se distribuyen entre las habitaciones indicadas en el plano. Indican dónde está el timbre de la puerta, si se usará un intercomunicador para anunciar la presencia de un visitante y cómo se va a instalar el

sistema de seguridad. En esencia, los planos (y especificaciones) detallados para las puertas, ventanas e instalaciones externas nos dicen cómo fluyen las cosas y la información hacia dentro y fuera de la casa y dentro de los cuartos que forman parte del plano. Los elementos de diseño de la interfaz del software permiten que la información fluya hacia dentro y afuera del sistema, y cómo están comunicados los componentes que son parte de la arquitectura.

PUNTO CLAVE

Hay tres partes para el elemento de diseño de la interfaz: la interfaz de usuario, las interfaces dirigidas hacia el sistema externo a la aplicación y las interfaces orientadas hacia los componentes dentro de ésta.

Hay tres elementos importantes del diseño de la interfaz: 1) la interfaz de usuario (IU), 2) las interfaces externas que tienen que ver con otros sistemas, dispositivos, redes y otros productores o consumidores de información y 3) interfaces internas que involucran a los distintos componentes del diseño. Estos elementos del diseño de la interfaz permiten que el software se comunique externamente y permita la comunicación y colaboración internas entre los componentes que constituyen la arquitectura del software.

El diseño de la IU (denominada cada vez con más frecuencia *diseño de la usabilidad*) es una acción principal de la ingeniería de software y se estudia con detalle en el capítulo 11. El diseño de la usabilidad incorpora elementos estéticos (como distribución, color, gráficos, mecanismos de interacción, etc.), elementos ergonómicos (por ejemplo, distribución y colocación de la información, metáforas, navegación por la IU, etc.) y elementos técnicos (como patrones de la IU y patrones reutilizables). En general, la IU es un subsistema único dentro de la arquitectura general de la aplicación.

Cita:

"De vez en cuando apártate, relájate un poco, para que cuando regreses al trabajo tu criterio sea más seguro. Toma algo de distancia porque entonces el trabajo parece más pequeño y es posible apreciar una porción mayor con una sola mirada, de modo que se detecta con facilidad la falta de armonía y proporción."

Leonardo da Vinci

El diseño de interfaces externas requiere información definitiva sobre la entidad a la que se envía información o desde la que se recibe. En todo caso, esta información debe recabarse durante la ingeniería de requerimientos (capítulo 5) y verificarse una vez que comienza el diseño de la interfaz.[8] El diseño de interfaces externas debe incorporar la revisión en busca de errores y (cuando sea necesario) las medidas de seguridad apropiadas.

El diseño de las interfaces internas se relaciona de cerca con el diseño de componentes (véase el capítulo 10). Las realizaciones del diseño de las clases de análisis representan todas las operaciones y esquemas de mensajería que se requieren para permitir la comunicación y colaboración entre las operaciones en distintas clases. Cada mensaje debe diseñarse para que contenga la información que se requiere transmitir y los requerimientos específicos de la función de la operación que se ha solicitado. Si para el diseño se elige el enfoque clásico de un proceso de entrada-salida, la interfaz de cada componente del software se diseña con base en las representaciones del flujo de datos y en la funcionalidad descrita en una narrativa de procesamiento.

WebRef

En la dirección **www.useit.com**, se encuentra información sumamente valiosa sobre el diseño de la IU.

En ciertos casos, una interfaz se modela en forma muy parecida a la de una clase. En el UML se define *interfaz* del modo siguiente [OMG03a]: "Es un especificador para las operaciones visibles desde el exterior [públicas] de una clase, un componente u otro clasificador (incluso subsistemas), sin especificar su estructura interna." En pocas palabras, una interfaz es un conjunto de operaciones que describen alguna parte del comportamiento de una clase y dan acceso a aquéllas.

Por ejemplo, la función de seguridad de *CasaSegura* hace uso del panel de control que permite que el propietario controle ciertos aspectos de la función de seguridad. En una versión avanzada del sistema, las funciones del panel de control podrían implementarse a través de un PDA inalámbrico o un teléfono móvil.

Cita:

"Un error común que comete la gente cuando trata de diseñar algo a prueba de tontos es subestimar la ingenuidad de los completamente tontos."

Douglas Adams

La clase **PaneldeControl** (véase la figura 8.5) proporciona el comportamiento asociado con un teclado, por lo que debe implementar las operaciones *LeerTeclazo ( )* y *DecodificarTecla ( )*. Si estas operaciones se van a dar a otras clases (en este caso, **PDAInalámbrico** y **TeléfonoMóvil**), es útil definir una interfaz como la de la figura. La interfaz, llamada **Teclado**, se ilustra como un estereotipo <<interfaz>> o como un círculo pequeño con leyenda y conectado a la clase

8 Las características de la interfaz pueden cambiar con el tiempo. Por tanto, un diseñador debe cerciorarse de que la especificación para ella sea exacta y completa.

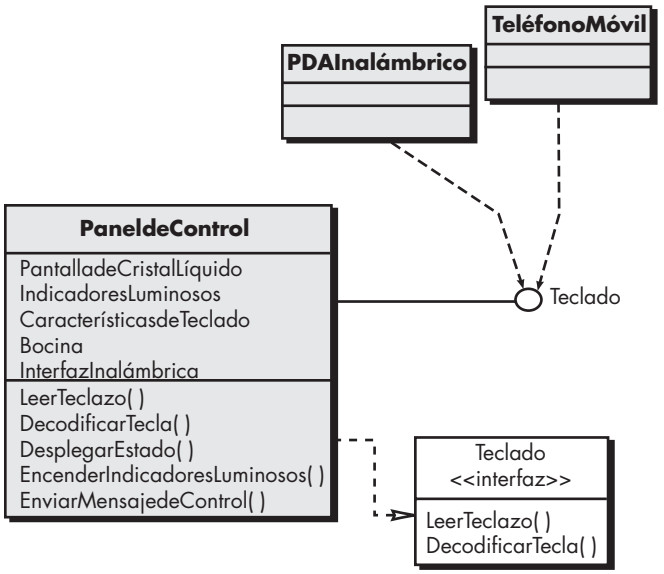


FIGURA 8.5
Representación de la interfaz para PaneldeControl

con una línea. La interfaz se define sin atributos y con el conjunto de operaciones que sean necesarias para lograr el comportamiento de un teclado.

La línea punteada con un triángulo abierto en su extremo (en la figura 8.5) indica que la clase **PaneldeControl** proporciona las operaciones de **Teclado** como parte de su comportamiento. En UML, esto se caracteriza como una *realización.* Es decir, parte del comportamiento de **PaneldeControl** se implementará con la realización de las operaciones de **Teclado**. Éstas se darán a otras clases que accedan a la interfaz.

8.4.4 Elementos del diseño en el nivel de los componentes

El diseño en el nivel de los componentes del software es el equivalente de los planos (y especificaciones) detallados de cada habitación de la casa. Estos dibujos ilustran el cableado y la plomería de cada cuarto, la ubicación de cajas eléctricas e interruptores, grifos, coladeras, regaderas, tinas, drenajes, gabinetes y closets. También describen el tipo de piso que se va a usar, las molduras que se van a aplicar y todos los detalles asociados con una habitación. El diseño de componentes para el software describe por completo los detalles internos de cada componente. Para lograrlo, este diseño define estructuras de datos para todos los objetos de datos locales y detalles algorítmicos para todo el procesamiento que tiene lugar dentro de un componente, así como la interfaz que permite el acceso a todas las operaciones de los componentes (comportamientos).

Cita:

"Los detalles no son los detalles. Constituyen el diseño."

Charles Eames

En el contexto de la ingeniería de software orientada a objeto, un componente se representa en forma de diagrama UML, como se ilustra en la figura 8.6. En ésta, aparece un componente llamado **AdministraciónSensor** (parte de la función de seguridad de *CasaSegura*). Una flecha punteada conecta al componente con una clase llamada **Sensor**, a él asignada. El compo-

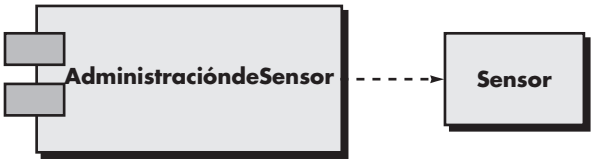


FIGURA 8.6
Diagrama de componente UML

nente **AdministracióndeSensor** lleva a cabo funciones asociadas con los sensores de *CasaSegura*, incluso su vigilancia y configuración. En el capítulo 10 se analizan más los diagramas de componentes.

Los detalles del diseño de un componente se modelan en muchos niveles de abstracción diferentes. Se utiliza un diagrama de actividades UML para representar la lógica del procesamiento. El flujo detallado del procedimiento para un componente se representa con seudocódigo (representación que se parece a un lenguaje de programación y que se describe en el capítulo 10) o con alguna otra forma diagramática (como un diagrama de flujo o de cajas). La estructura algorítmica sigue las reglas establecidas para la programación estructurada (por ejemplo, un conjunto de construcciones restringidas de procedimiento). Las estructuras de datos, seleccionadas con base en la naturaleza de los objetos de datos que se van a procesar, por lo general se modelan con el empleo de seudocódigo del lenguaje de programación que se usará para la implementación.

8.4.5 Elementos del diseño del despliegue

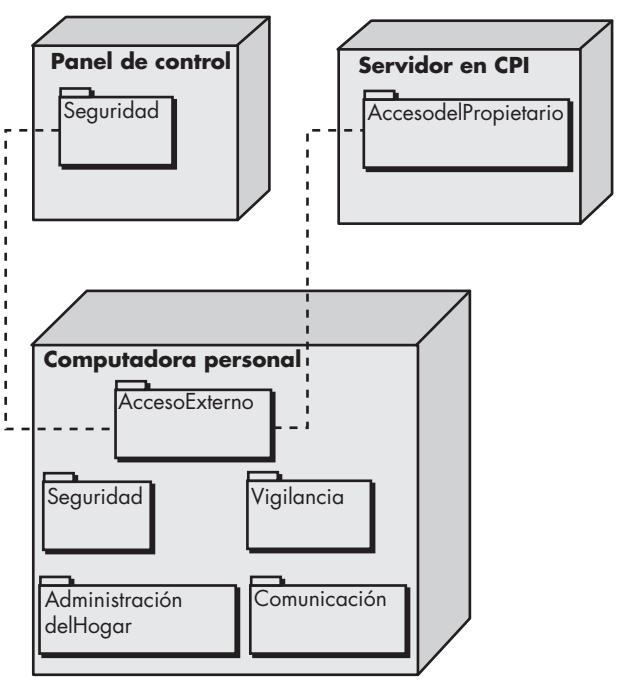
Los elementos del diseño del despliegue indican la forma en la que se acomodarán la funcionalidad del software y los subsistemas dentro del ambiente físico de la computación que lo apoyará. Por ejemplo, los elementos del producto *CasaSegura* se configuran para que operen dentro de tres ambientes de computación principales: una PC en la casa, el panel de control de *CasaSegura* y un servidor alojado en CPI Corp. (que provee el acceso al sistema a través de internet).

PUNTO CLAVE

Los diagramas de despliegue comienzan en forma de descriptor, donde el ambiente de despliegue se describe en términos generales. Después se utilizan formas de instancia y se describen explícitamente los elementos de la configuración.

Durante el diseño se desarrolla un diagrama de despliegue que después se refina, como se ilustra en la figura 8.7. En ella aparecen tres ambientes de computación (en realidad habría más, con sensores y cámaras, entre otros). Se indican los subsistemas (funcionalidad) que están alojados dentro de cada elemento de computación. Por ejemplo, la computadora personal aloja subsistemas que implementan características de seguridad, vigilancia, administración del hogar y comunicaciones. Además, se ha diseñado un subsistema de acceso externo para manejar todos los intentos de acceder al sistema *CasaSegura* desde un lugar externo. Cada subsistema se elaboraría para que indicara los componentes que implementa.

FIGURA 8.7
Diagrama de despliegue UML



El diagrama que aparece en la figura 8.7 es un *formato descriptor*. Esto significa que el diagrama de despliegue muestra el ambiente de computación, pero no indica de manera explícita los detalles de la configuración. Por ejemplo, no se da mayor identificación de la "computadora personal". Puede ser una Mac o basarse en Windows, una estación de trabajo Sun o un sistema Linux. Estos detalles se dan cuando se regrese al diagrama de despliegue en el *formato de instancia* en las etapas finales del diseño, o cuando comience la construcción. Se identifica cada instancia del despliegue (una configuración específica, llamada *configuración del hardware*).

8.5 Resumen

El diseño del software comienza cuando termina la primera iteración de la ingeniería de requerimientos. El objetivo del diseño del software es aplicar un conjunto de principios, conceptos y prácticas que llevan al desarrollo de un sistema o producto de alta calidad. La meta del diseño es crear un modelo de software que implantará correctamente todos los requerimientos del usuario y causará placer a quienes lo utilicen. Los diseñadores del software deben elegir entre muchas alternativas de diseño y llegar a la solución que mejor se adapte a las necesidades de los participantes en el proyecto.

El proceso de diseño va de una visión "panorámica" del software a otra más cercana que define el detalle requerido para implementar un sistema. El proceso comienza por centrarse en la arquitectura. Se definen los subsistemas, se establecen los mecanismos de comunicación entre éstos, se identifican los componentes y se desarrolla la descripción detallada de cada uno. Además, se diseñan las interfaces externa, interna y de usuario.

Los conceptos de diseño han evolucionado en los primeros 60 años de trabajo de la ingeniería de software. Describen atributos de software de computadora que debe presentarse sin importar el proceso que se elija para hacer la ingeniería, los métodos de diseño que se apliquen o los lenguajes de programación que se utilicen. En esencia, los conceptos de diseño ponen el énfasis en la necesidad de la abstracción como mecanismo para crear componentes reutilizables de software, en la importancia de la arquitectura como forma de entender mejor la estructura general de un sistema, en los beneficios de la ingeniería basada en patrones como técnica de diseño de software con capacidad comprobada, en el valor de la separación de problemas y de la modularidad eficaz como forma de elaborar software más entendible, más fácil de probar y de recibir mantenimiento, en las consecuencias de ocultar información como mecanismo para reducir la propagación de los efectos colaterales cuando hay errores, en el efecto de la independencia funcional como criterio para construir módulos eficaces, en el uso del refinamiento como mecanismo de diseño, en una consideración de los aspectos que interfieren con los requerimientos del sistema, en la aplicación del rediseño para optimizar el diseño obtenido y en la importancia de las clases orientadas a objetos y de las características relacionadas con ellos.

El modelo del diseño incluye cuatro elementos distintos. Conforme se desarrolla cada uno, surge una visión más completa del diseño. El elemento arquitectónico emplea información obtenida del dominio de la aplicación, del modelo de requerimientos y de los catálogos disponibles de patrones y estilos para obtener una representación estructural completa del software, de sus subsistemas y componentes. Los elementos del diseño de la interfaz modelan las interfaces internas y externas y la de usuario. Los elementos en el nivel de componentes definen cada uno de los módulos (componentes) que constituyen la arquitectura. Por último, los elementos del diseño albergan la arquitectura, sus componentes y las interfaces dirigidas hacia la configuración física en la que se alojará el software.

Problemas y puntos por evaluar

8.1. Cuando se "escribe" un programa, ¿se diseña software? ¿En qué difieren el diseño de software y la codificación?

8.2. Si el diseño del software no es un programa (y no lo es), entonces, ¿qué es?

8.3. ¿Cómo se evalúa la calidad del diseño del software?

8.4. Estudie el conjunto de tareas presentado para el diseño. ¿Dónde se evalúa la calidad en dicho conjunto? ¿Cómo se logra? ¿Cómo se consiguen los atributos de calidad estudiados en la sección 8.2.1?

8.5. Dé ejemplos de tres abstracciones de datos y de las abstracciones de procedimiento que se usan para manipularlas.

8.6. Describa con sus propias palabras la arquitectura de software.

8.7. Sugiera un patrón de diseño que encuentre en una categoría de objetos cotidianos (por ejemplo, electrónica de consumo, automóviles, aparatos, etc.). Describa el patrón en forma breve.

8.8. Describa con sus propias palabras la separación de problemas. ¿Hay algún caso en el que no sea apropiada la estrategia de divide y vencerás? ¿Cómo afecta esto al argumento a favor de la modularidad?

8.9. ¿Cuándo debe implementarse un diseño modular como software monolítico? ¿Cómo se logra esto? ¿El rendimiento es la única justificación para la implementación de software monolítico?

8.10. Analice la relación entre el concepto de ocultamiento de información como atributo de la modularidad efectiva y el de independencia de los módulos.

8.11. ¿Cómo se relacionan los conceptos de acoplamiento y portabilidad del software? Dé ejemplos que apoyen su punto de vista.

8.12. Aplique un "enfoque de refinamiento stepwise" para desarrollar tres niveles distintos de abstracciones del procedimiento para uno o más de los programas siguientes: *a*) un revisor de la escritura que, dada una cantidad numérica de dinero, imprima ésta en las palabras que se requieren normalmente en un cheque. *b*) una resolución en forma iterativa de las raíces de una ecuación trascendente. *c*) un algoritmo de programación de tareas simples para un sistema operativo.

8.13. Considere el software requerido para implementar la capacidad de navegación (con un GPS) en un dispositivo móvil de comunicación portátil. Describa dos o tres preocupaciones de interferencia que se presentarían. Analice la manera en la que se representaría como aspecto una de estas preocupaciones.

8.14. ¿"Rediseñar" significa que se modifica todo el diseño en forma iterativa? Si no es así, ¿qué significa?

8.15. Describa en breves palabras cada uno de los cuatro elementos del modelo del diseño.

Lecturas adicionales 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 la bibliografía sobre el diseño y una "obligación" de lectura para quien diseñe cualquier cosa que utilicen los humanos. Adams (*Conceptual Blockbusting*, 3a. ed., Addison-Wesley, 1986) escribió un libro cuya lectura es esencial para los diseñadores que deseen ampliar su forma de pensar. Por último, el texto clásico de Polya (*How to Solve It*, 2a. ed., Princeton University Press, 1988) proporciona un proceso general de solución de problemas que ayuda a los diseñadores de software cuando se enfrentan a problemas complejos.

En la misma tradición, Winograd *et al.* (*Bringing Design to Software*, Addison-Wesley, 1996) analizan los diseños de software que funcionan, los que no y su por qué. Un libro fascinante editado por Wixon y Ramsey (*Field Methods Casebook for Software Design*, Wiley, 1996) sugiere métodos de investigación de campo (muy parecidos a los de los antropólogos) para entender cómo hacen el trabajo los usuarios finales y luego diseñar el software que satisfaga sus necesidades. Beyer y Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs*, Academic Press, 1997) ofrecen otro punto de vista del diseño de software que integra al consumidor o usuario en cada aspecto del proceso de diseño. Bain (*Emergent Design*, Addison-Wesley, 2008) acopla los patrones, el rediseño y el desarrollo orientado a pruebas en un enfoque de diseño eficaz.

Un tratamiento exhaustivo del diseño en el contexto de la ingeniería de software es presentado por Fox (*Introduction to Software Engineering Design*, Addison-Wesley, 2006) y Zhu (*Software Design Methodology*, Butterworth-Heinemann, 2005). McConnell (*Code Complete*, 2a. ed., Microsoft Press, 2004) plantea un estudio excelente de los aspectos prácticos del diseño de software de alta calidad. Robertson (*Simple Program Design*, 3a. ed., Boyd y Fraser Publishing, 1999) presenta un análisis introductorio del diseño de software, útil para quienes se inician en el estudio del tema. Budgen (*Software Design*, 2a. ed., Addison-Wesley, 2004) presenta

varios métodos populares de diseño y los compara entre sí. Fowler *y sus colegas* (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) estudian técnicas para la optimización incremental de diseños de software. Rosenberg y Stevens (*Use Case Driven Object Modeling with UML*, Apress, 2007) estudian el desarrollo de diseños orientados a objeto con el empleo de casos de uso como fundamento.

En una antología editada por Freeman y Wasserman (*Software Design Techniques*, 4a. ed., IEEE, 1983), hay una excelente revisión histórica del diseño de software. Esta edición contiene muchas reimpresiones de los artículos clásicos que forman la base de las tendencias actuales del diseño del software. Card y Glass (*Measuring Software Design Quality*, Prentice-Hall, 1990) presentan mediciones de la calidad procedentes de los campos de la técnica y la administración.

En internet hay una variedad amplia de fuentes de información acerca del diseño de software. En el sitio web del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, se encuentra una lista actualizada de referencias existentes en la red mundial que son relevantes para el diseño de software y para la ingeniería de diseño.