- **Modelo de contenido:** identifica el espectro completo de contenido que dará la *webapp*. El contenido incluye datos de texto, gráficos e imágenes, video y sonido.
- **Modelo de interacción:** describe la manera en que los usuarios interactúan con la *webapp*.
- **Modelo funcional:** define las operaciones que se aplicarán al contenido de la *webapp* y describe otras funciones de procesamiento que son independientes del contenido pero necesarias para el usuario final.
- **Modelo de navegación:** define la estrategia general de navegación para la *webapp*.
- **Modelo de configuración:** describe el ambiente e infraestructura en la que reside la *webapp*.

Es posible desarrollar cada uno de estos modelos con el empleo de un esquema de representación (llamado con frecuencia "lenguaje") que permite que su objetivo y estructura se comuniquen y evalúen con facilidad entre los miembros del equipo de ingeniería de web y otros participantes. En consecuencia, se identifica una lista de aspectos clave (como errores, omisiones, inconsistencias, sugerencias de mejora o modificaciones, puntos de aclaración, etc.) para trabajar sobre ellos.

### 7.5.4 Modelo del contenido de las *webapps*

El modelo de contenido incluye elementos estructurales que dan un punto de vista importante de los requerimientos del contenido de una *webapp*. Estos elementos estructurales agrupan los objetos del contenido y todas las clases de análisis, entidades visibles para el usuario que se crean o manipulan cuando éste interactúa con la *webapp*.[15]

El contenido puede desarrollarse antes de la implementación de la *webapp*, mientras ésta se construye o cuando ya opera. En cualquier caso, se incorpora por referencia de navegación en la estructura general de la *webapp*. Un *objeto de contenido* es una descripción de un producto en forma de texto, un artículo que describe un evento deportivo, una fotografía tomada en éste, la respuesta de un usuario en un foro de análisis, una representación animada de un logotipo corporativo, una película corta de un discurso o una grabación en audio para una presentación con diapositivas. Los objetos de contenido pueden almacenarse como archivos separados, incrustarse directamente en páginas web u obtenerse en forma dinámica de una base de datos. En otras palabras, un objeto de contenido es cualquier aspecto de información cohesiva que se presente al usuario final.

Los objetos de contenido se determinan directamente a partir de casos de uso, estudiando la descripción del escenario respecto de referencias directas e indirectas al contenido. Por ejemplo, se establece en **CasaSeguraAsegurada.com** una *webapp* que da apoyo a *CasaSegura*. Un caso de uso, *Comprar componentes seleccionados de CasaSegura*, describe el escenario que se requiere para comprar un componente de *CasaSegura* y que contiene la siguiente oración:

> Podré obtener información descriptiva y de precios de cada componente del producto.

El modelo de contenido debe ser capaz de describir el objeto de contenido **Componente**. En muchas circunstancias, para definir los requerimientos para el contenido que debe diseñarse e implementarse, es suficiente una lista sencilla de los objetos de contenido, junto con la descripción breve de cada uno. Sin embargo, en ciertos casos, el modelo de contenido se beneficia de un análisis más rico que ilustre en forma gráfica las relaciones entre los objetos de contenido y la jerarquía que mantiene una *webapp*.

Por ejemplo, tome en cuenta el *árbol de datos* [Sri01] creado por el componente **CasaSeguraAsegurada.com** que aparece en la figura 7.10. El árbol representa una jerarquía de informa-

15 Las clases de análisis se estudiaron en el capítulo 6.

www.FreeLibros.me

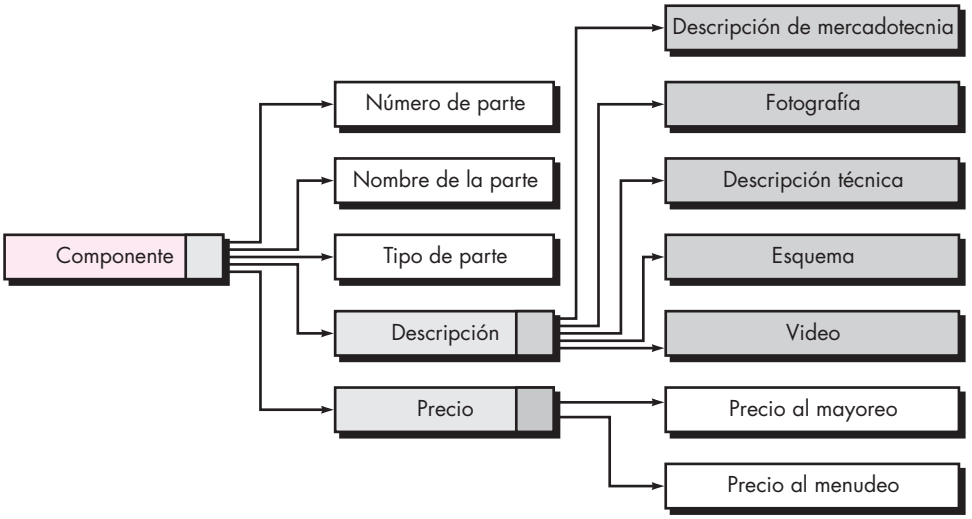



**FIGURA 7.10**
**Árbol de datos para el componente CasaSeguraAsegurada.com**

ción que se utiliza para describir un componente. Los aspectos de datos simples o compuestos (uno o más valores de los datos) se representan con rectángulos sin sombra. Los objetos de contenido se representan con rectángulos con sombra. En la figura, **descripción** está definida por cinco objetos (los rectángulos sombreados). En ciertos casos, uno o más de estos objetos se mejorará más conforme se expanda el árbol de datos.

Es posible crear un árbol de datos para cualquier contenido que se componga de múltiples objetos de contenido y aspectos de datos. El árbol de datos se desarrolla como un esfuerzo para definir relaciones jerárquicas entre los objetos de contenido y para dar un medio de revisión del contenido a fin de que se descubran las omisiones e inconsistencias antes de que comience el diseño. Además, el árbol de datos sirve como base para diseñar el contenido.

### 7.5.5 Modelo de la interacción para *webapps*

La gran mayoría de *webapps* permiten una "conversación" entre un usuario final y funcionalidad, contenido y comportamiento de la aplicación. Esta conversación se describe con el uso de un modelo de *interacción* que se compone de uno o más de los elementos siguientes: 1) casos de uso, 2) diagramas de secuencia, 3) diagramas de estado[16] y 4) prototipos de la interfaz de usuario.

En muchas instancias, basta un conjunto de casos de uso para describir la interacción en el nivel del análisis (durante el diseño se introducirán más mejoras y detalles). Sin embargo, cuando la secuencia de interacción es compleja e involucra múltiples clases de análisis o muchas tareas, es conveniente ilustrarla de forma más rigurosa mediante un diagrama.

El formato de la interfaz de usuario, el contenido que presenta, los mecanismos de interacción que implementa y la estética general de las conexiones entre el usuario y la *webapp* tienen mucho que ver con la satisfacción de éste y con el éxito conjunto del software. Aunque se afirme que la creación de un prototipo de interfaz de usuario es una actividad de diseño, es una buena idea llevarla a cabo durante la creación del modelo de análisis. Entre más pronto se revise la representación física de la interfaz de usuario, más probable es que los consumidores finales obtengan lo que desean. En el capítulo 11 se estudia con detalle el diseño de interfaces de usuario.

16 Los diagramas de secuencia y los de estado se modelan con el empleo de notación UML. Los diagramas de estado se describen en la sección 7.3. Para mayores detalles, consulte el apéndice 1.

www.FreeLibros.me

Como hay muchas herramientas para construir *webapps* baratas y poderosas en sus funciones, es mejor crear el prototipo de la interfaz con el empleo de ellas. El prototipo debe implementar los vínculos de navegación principales y representar la pantalla general en forma muy parecida a la que se construirá. Por ejemplo, si van a ponerse a disposición del usuario final cinco funciones principales del sistema, el prototipo debe representarlas tal como las verá cuando entre por primera vez a la *webapp*. ¿Se darán vínculos gráficos? ¿Dónde se desplegará el menú de navegación? ¿Qué otra información verá el usuario? Preguntas como éstas son las que debe responder el prototipo.

### 7.5.6 Modelo funcional para las *webapps*

Muchas *webapps* proporcionan una amplia variedad de funciones de computación y manipulación que se asocian directamente con el contenido (porque lo utilizan o porque lo producen) y es frecuente que sean un objetivo importante de la interacción entre el usuario y la *webapp*. Por esta razón, deben analizarse los requerimientos funcionales y modelarlos cuando sea necesario.

El *modelo funcional* enfrenta dos elementos de procesamiento de la *webapp*, cada uno de los cuales representa un nivel distinto de abstracción del procedimiento: 1) funciones observables por los usuarios que entrega la *webapp* a éstos y 2) las operaciones contenidas en las clases de análisis que implementan comportamientos asociados con la clase.

La funcionalidad observable por el usuario agrupa cualesquiera funciones de procesamiento que inicie directamente el usuario. Por ejemplo, una *webapp* financiera tal vez implemente varias funciones de finanzas (como una calculadora de ahorros para una colegiatura universitaria o un fondo para el retiro). Estas funciones en realidad se implementan con el uso de operaciones dentro de clases de análisis, pero desde el punto de vista del usuario final; el resultado visible es la función (más correctamente, los datos que provee la función).

En un nivel más bajo de abstracción del procedimiento, el modelo de requerimientos describe el procesamiento que se realizará por medio de operaciones de clase de análisis. Estas operaciones manipulan los atributos de clase y se involucran como clases que colaboran entre sí para lograr algún comportamiento que se desea.

Sin que importe el nivel de abstracción del procedimiento, el diagrama de actividades UML se utiliza para representar detalles de éste. En el nivel de análisis, los diagramas de actividades deben usarse sólo donde la funcionalidad sea relativamente compleja. Gran parte de la complejidad de muchas *webapps* ocurre no en las funciones que proveen, sino en la naturaleza de la información a que se accede y en las formas en las que se manipula.

Un ejemplo de complejidad relativa de la funcionalidad para **CasaSeguraAsegurada.com** se aborda en un caso de uso llamado *Obtener recomendaciones para la distribución de sensores en mi espacio*. El usuario ya ha desarrollado la distribución del espacio que se vigilará y en este caso de uso selecciona dicha distribución y solicita ubicaciones recomendables para los sensores dentro de ella. **CasaSeguraAsegurada.com** responde con la representación gráfica de la distribución por medio de información adicional acerca de la ubicación recomendable para los sensores. La interacción es muy sencilla, el contenido es algo más complejo, pero la funcionalidad subyacente es muy sofisticada. El sistema debe realizar un análisis relativamente complejo de la planta del piso para determinar el conjunto óptimo de sensores. Debe examinar las dimensiones de la habitación, la ubicación de puertas y ventanas, y coordinar éstas con la capacidad y especificaciones de los sensores. ¡No es una tarea fácil! Para describir el procesamiento de este caso de uso se utiliza un conjunto de diagramas de actividades.

El segundo ejemplo es el caso de uso *Controlar cámaras*. En éste, la interacción es relativamente sencilla, pero existe el potencial de una funcionalidad compleja, dado que dicha operación "sencilla" requiere una comunicación compleja con dispositivos ubicados en posiciones remotas y a los que se accede por internet. Una complicación adicional se relaciona con la ne-

www.FreeLibros.me

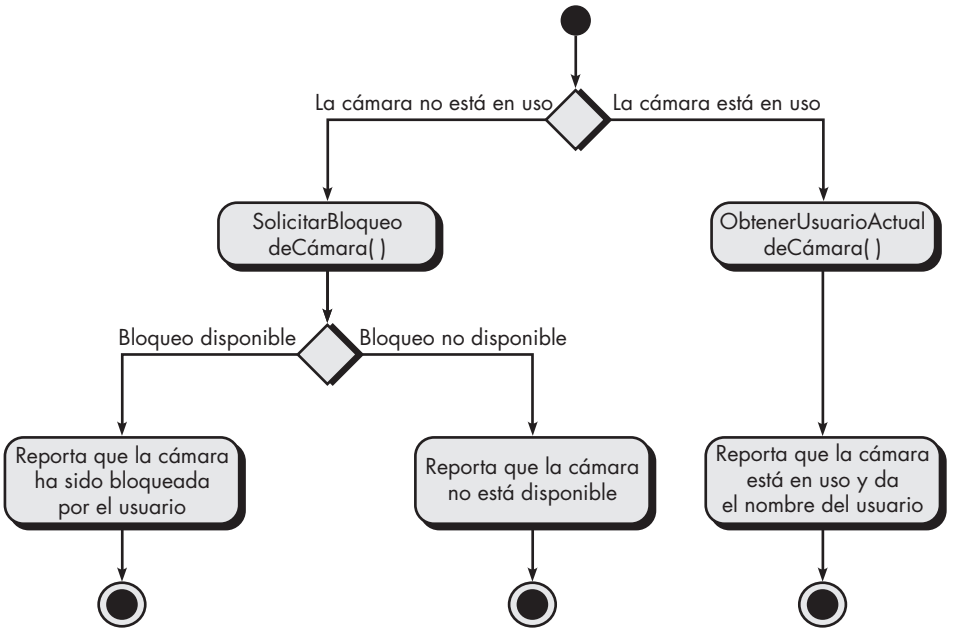



**FIGURA 7.11**
**Diagrama de actividades para la operación *TomarControl deCámara( )***

gociación del control cuando varias personas autorizadas tratan de vigilar o controlar un mismo sensor al mismo tiempo.

La figura 7.11 ilustra el diagrama de actividades para la operación *TomarControldeCámara* que forma parte de la clase de análisis **Cámara** usada dentro del caso de uso *Controlar cámaras*. Debe observarse que con el flujo de procedimiento se invocan dos operaciones adicionales: *SolicitarBloqueodeCámara ( )*, que trata de bloquear la cámara para este usuario, y *ObtenerUsuarioActualdeCámara ( )*, que recupera el nombre del usuario que controla en ese momento la cámara. Los detalles de construcción indican cómo se invocan estas operaciones, y los de la interfaz para cada operación no se señalan hasta que comienza el diseño de la *webapp*.

### 7.5.7 Modelos de configuración para las *webapps*

En ciertos casos, el modelo de configuración no es sino una lista de atributos del lado del servidor y del lado del cliente. Sin embargo, para *webapps* más complejas, son varias las dificultades de configuración (por ejemplo, distribuir la carga entre servidores múltiples, arquitecturas caché, bases de datos remotas, distintos servidores que atienden a varios objetos en la misma página web, etc.) que afectan el análisis y diseño. El *diagrama de despliegue UML* se utiliza en situaciones en las que deben considerarse arquitecturas de configuración compleja.

Para **CasaSeguraAsegurada.com**, deben especificarse el contenido y funcionalidad públicos a fin de que sean accesibles a través de todos los clientes principales de web (como aquéllos con 1 por ciento o más de participación en el mercado).[17] A la inversa, es aceptable restringir las funciones más complejas de control y vigilancia (que sólo es accesible para los usuarios tipo **Propietario**) a un conjunto más pequeño de clientes. El modelo de configuración para **CasaSeguraAsegurada.com** también especificará la operación cruzada con las bases de datos de productos y aplicaciones de vigilancia.

17 La determinación de la participación en el mercado para los navegadores es notoriamente problemática y varía en función de cuál fuente se utilice. No obstante, en el momento de escribir este libro, Internet Explorer y Firefox eran los únicos que sobrepasaban 30 por ciento, y Mozilla, Opera y Safari los únicos que superaban de manera consistente 1 por ciento.

www.FreeLibros.me

### 7.5.8 Modelado de la navegación

Para modelar la navegación se considera cómo navegará cada categoría de usuario de un elemento de la *webapp* (como un objeto de contenido) a otro. La mecánica de navegación se define como parte del diseño. En esa etapa debe centrarse la atención en los requerimientos generales de navegación. Deben considerarse las preguntas siguientes:

- ¿Ciertos elementos deben ser más fáciles de alcanzar (requieren menos pasos de navegación) que otros? ¿Cuál es la prioridad de presentación?
- ¿Debe ponerse el énfasis en ciertos elementos para forzar a los usuarios a navegar en esa dirección?
- ¿Cómo deben manejarse los errores en la navegación?
- ¿Debe darse prioridad a la navegación hacia grupos de elementos relacionados y no hacia un elemento específico?
- ¿La navegación debe hacerse por medio de vínculos, acceso basado en búsquedas o por otros medios?
- ¿Debe presentarse a los usuarios ciertos elementos con base en el contexto de acciones de navegación previas?
- ¿Debe mantenerse un registro de usuarios de la navegación?
- ¿Debe estar disponible un mapa completo de la navegación (en oposición a un solo vínculo para "regresar" o un apuntador dirigido) en cada punto de la interacción del usuario?
- ¿El diseño de la navegación debe estar motivado por los comportamientos del usuario más comunes y esperados o por la importancia percibida de los elementos definidos de la *webapp*?
- ¿Un usuario puede "guardar" su navegación previa a través de la *webapp* para hacer expedito el uso futuro?
- ¿Para qué categoría de usuario debe diseñarse la navegación óptima?
- ¿Cómo deben manejarse los vínculos externos hacia la *webapp*? ¿Con la superposición de la ventana del navegador existente? ¿Como nueva ventana del navegador? ¿En un marco separado?

Estas preguntas y muchas otras deben plantearse y responderse como parte del análisis de la navegación.

Usted y otros participantes también deben determinar los requerimientos generales para la navegación. Por ejemplo, ¿se dará a los usuarios un "mapa del sitio" y un panorama de toda la estructura de la *webapp*? ¿Un usuario puede hacer una "visita guiada" que resalte los elementos más importantes (objetos y funciones de contenido) con que se disponga? ¿Podrá acceder un usuario a los objetos o funciones de contenido con base en atributos definidos de dichos elementos (por ejemplo, un usuario tal vez desee acceder a todas las fotografías de un edificio específico o a todas las funciones que permiten calcular el peso)?

## 7.6 Resumen

Los modelos orientados al flujo se centran en el flujo de objetos de datos a medida que son transformados por las funciones de procesamiento. Derivados del análisis estructurado, los modelos orientados al flujo usan el diagrama de flujo de datos, notación de modelación que ilustra la manera en la que se transforma la entrada en salida cuando los objetos de datos se mueven a través del sistema. Cada función del software que transforme datos es descrita por la

www.FreeLibros.me

especificación o narrativa de un proceso. Además del flujo de datos, este elemento de modelación también muestra el flujo del control, representación que ilustra cómo afectan los eventos al comportamiento de un sistema.

El modelado del comportamiento ilustra el comportamiento dinámico. El modelo de comportamiento utiliza una entrada basada en el escenario, orientada al flujo y elementos basados en clases para representar los estados de las clases de análisis y al sistema como un todo. Para lograr esto, se identifican los estados y se definen los eventos que hacen que una clase (o el sistema) haga una transición de un estado a otro, así como las acciones que ocurren cuando se efectúa dicha transición. Los diagramas de estado y de secuencia son la notación que se emplea para modelar el comportamiento.

Los patrones de análisis permiten a un ingeniero de software utilizar el conocimiento del dominio existente para facilitar la creación de un modelo de requerimientos. Un patrón de análisis describe una característica o función específica del software que puede describirse con un conjunto coherente de casos de uso. Especifica el objetivo del patrón, la motivación para su uso, las restricciones que limitan éste, su aplicabilidad en distintos dominios de problemas, la estructura general del patrón, su comportamiento y colaboraciones, así como información suplementaria.

El modelado de los requerimientos para las *webapps* utiliza la mayoría, si no es que todos, los elementos de modelado que se estudian en el libro. Sin embargo, dichos elementos se aplican dentro de un conjunto de modelos especializados que se abocan al contenido, interacción, función, navegación y configuración cliente-servidor en la que reside la *webapp*.

## Problemas y puntos por evaluar

**7.1.** ¿Cuál es la diferencia fundamental entre el análisis estructurado y las estrategias orientadas a objetos para hacer el análisis de los requerimientos?

**7.2.** En un diagrama de flujo de datos, ¿una flecha representa un flujo del control u otra cosa?

**7.3.** ¿Qué es la "continuidad del flujo de información" y cómo se aplica cuando se mejora el diagrama de flujo de datos?

**7.4.** ¿Cómo se utiliza el análisis gramatical en la creación de un DFD?

**7.5.** ¿Qué es una especificación del control?

**7.6.** ¿Son lo mismo una PSPEC y un caso de uso? Si no es así, explique las diferencias.

**7.7.** Hay dos tipos diferentes de "estados" que los modelos del comportamiento pueden representar. ¿Cuáles son?

**7.8.** ¿En qué difiere un diagrama de secuencia de un diagrama de estado? ¿En qué se parecen?

**7.9.** Sugiera tres patrones de requerimientos para un teléfono inalámbrico moderno y escriba una descripción breve de cada uno. ¿Estos patrones podrían usarse para otros equipos? Dé un ejemplo.

**7.10.** Seleccione uno de los patrones desarrollados en el problema 7.9 y desarrolle una descripción del patrón razonablemente completa, similar en contenido y estilo a la que se presentó en la sección 7.4.2.

**7.11.** ¿Cuánto modelado del análisis piensa que se requeriría para **CasaSeguraAsegurada.com**? ¿Se necesitaría cada uno de los tipos de modelo descritos en la sección 7.5.3?

**7.12.** ¿Cuál es el propósito del modelo de interacción para una *webapp*?

**7.13.** Un modelo funcional de *webapp* debe retrasarse hasta el diseño. Diga los pros y contras de este argumento.

**7.14.** ¿Cuál es el propósito de un modelo de configuración?

**7.15.** ¿En qué difiere el modelo de navegación del modelo de interacción?

www.FreeLibros.me

## Lecturas adicionales y fuentes de información

Se han publicado decenas de libros sobre el análisis estructurado. Todos cubren el tema de manera adecuada, pero algunos son excelentes. DeMarco y Plauger escribieron un clásico (*Structured Analysis and System Specification*, Pearson, 1985) que sigue siendo una buena introducción a la notación básica. Los libros escritos por Kendall y Kendall (*Systems Analysis and Design*, 5a. ed., Prentice-Hall, 2002), Hoffer *et al*. (*Modern Systems Analysis and Design*, Addison-Wesley, 3a. ed., 2001), Davis y Yen (*The Information System Consultant's Handbook: Systems Analysis and Design*, CRC Press, 1998) y Modell (*A Professional's Guide to Systems Analysis*, 2a. ed., McGraw-Hill, 1996) son buenas referencias. El escrito por Yourdon (*Modern Structured Analysis*, Yourdon-Press, 1989) sobre el tema está entre las fuentes más exhaustivas publicadas hasta la fecha.

El modelado del comportamiento presenta un punto de vista dinámico e importante del comportamiento de un sistema. Los libros de Wagner *et al.* (*Modeling Software with Finite State Machines: A Practical Approach*, Auerbach, 2006) y Boerger y Staerk (*Abstract State Machines*, Springer, 2003) presentan un análisis completo de los diagramas de estado y de otras representaciones del comportamiento.

La mayoría de textos escritos sobre patrones de software se centran en el diseño de éste. Sin embargo, los libros de Evans (*Domain-Driven Design*, Addison-Wesley, 2003) y Fowler ([Fow03] y ([Fow97]) abordan específicamente los patrones de análisis.

Pressman y Lowe presentan un tratamiento profundo del modelado del análisis para *webapps* [Pre08]. Los artículos contenidos dentro de una antología editada por Murugesan y Desphande (*Web Engineering: Managing Diversity and Complexity of Web Application Development*, Springer, 2001) analizan distintos aspectos de los requerimientos para las *webapps*. Además, la edición anual de *Proceedings of the International Conference on Web Engineering* analiza en forma regular aspectos del modelado de los requerimientos.

En internet hay una amplia variedad de fuentes de información sobre modelado de los requerimientos. En el sitio web del libro, **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, se encuentra una lista actualizada de referencias que hay en la red mundial, relevantes para el modelado del análisis.

www.FreeLibros.me

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.



www.FreeLibros.me

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.*

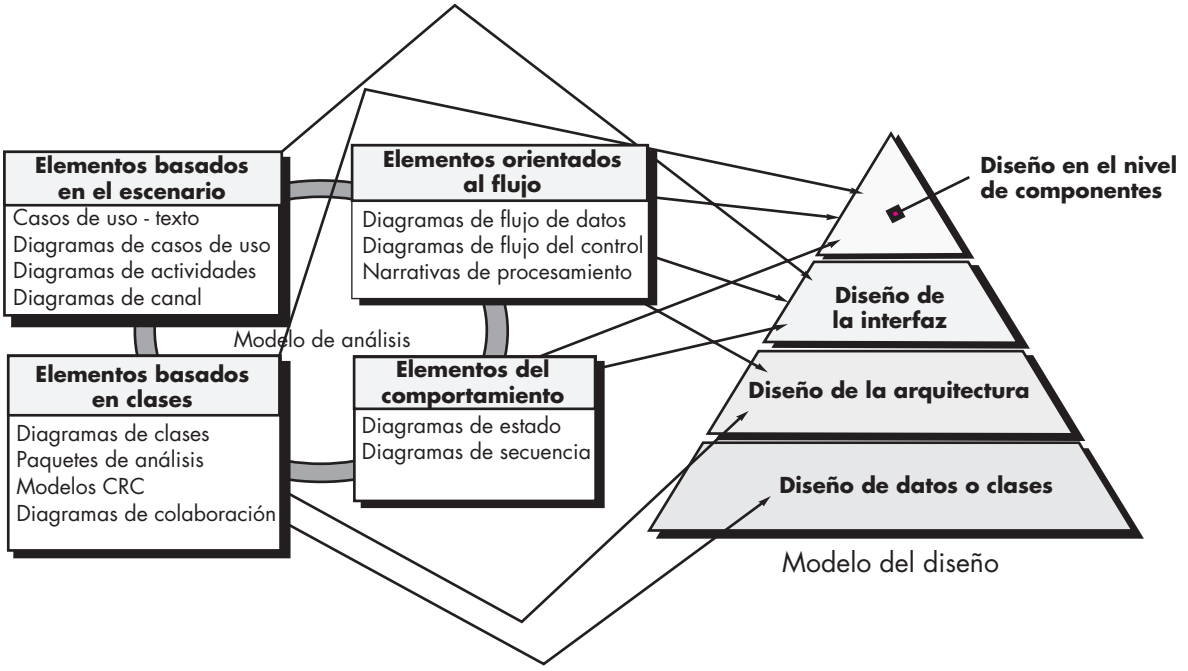
El diseño de datos o clases transforma los modelos de clases (capítulo 6) en realizaciones de clases de diseño y en las estructuras de datos que se requieren para implementar el software. Los objetos y relaciones definidos en el diagrama CRC y el contenido detallado de los datos ilustrado por los atributos 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].

www.FreeLibros.me

**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.

www.FreeLibros.me

**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-

www.FreeLibros.me

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.

www.FreeLibros.me

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.

www.FreeLibros.me

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,

www.FreeLibros.me

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.

www.FreeLibros.me

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.

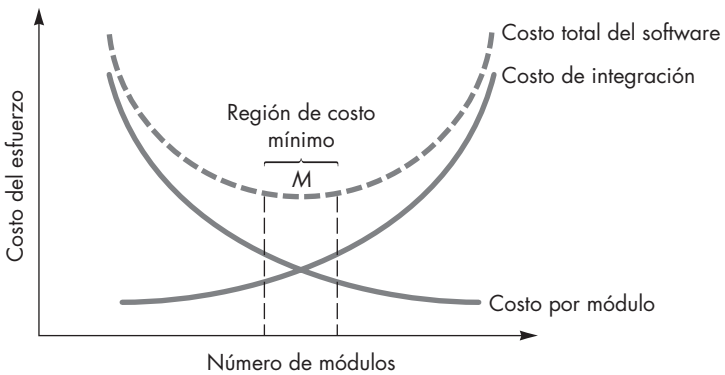
La división de problemas se manifiesta en otros conceptos de diseño relacionados: modularidad, aspectos, independencia de funcionamiento y mejora. Cada uno de éstos se estudiará en las secciones siguientes.

### 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.

www.FreeLibros.me

**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.

www.FreeLibros.me

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.

www.FreeLibros.me

### 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.

www.FreeLibros.me

### 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.

www.FreeLibros.me

### 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

www.FreeLibros.me

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

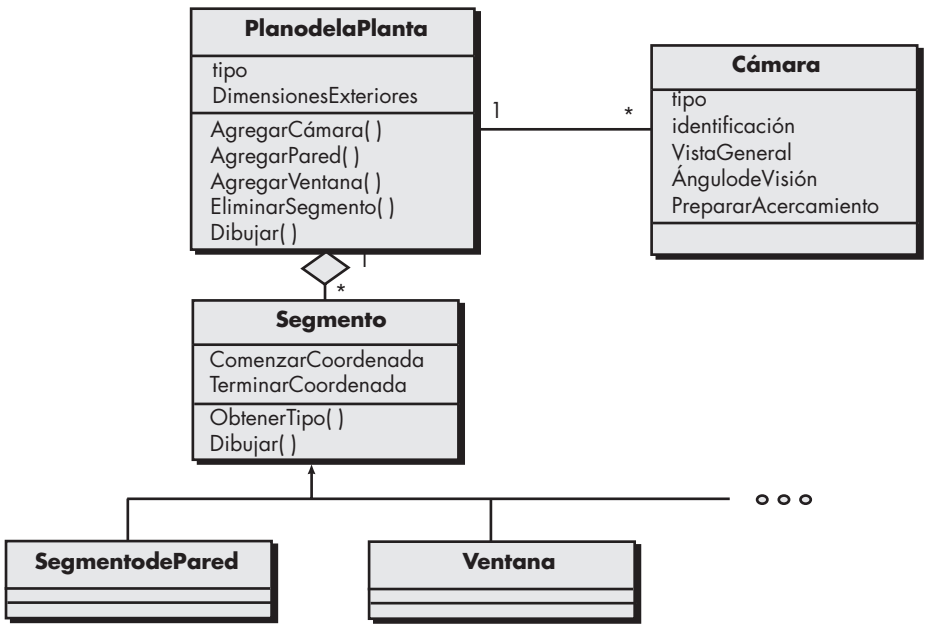
6 Una manera menos formal de la Ley de Demeter es: "cada unidad debe hablar sólo con sus amigas: no hablar con extraños".

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

www.FreeLibros.me

**FIGURA 8.3**

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



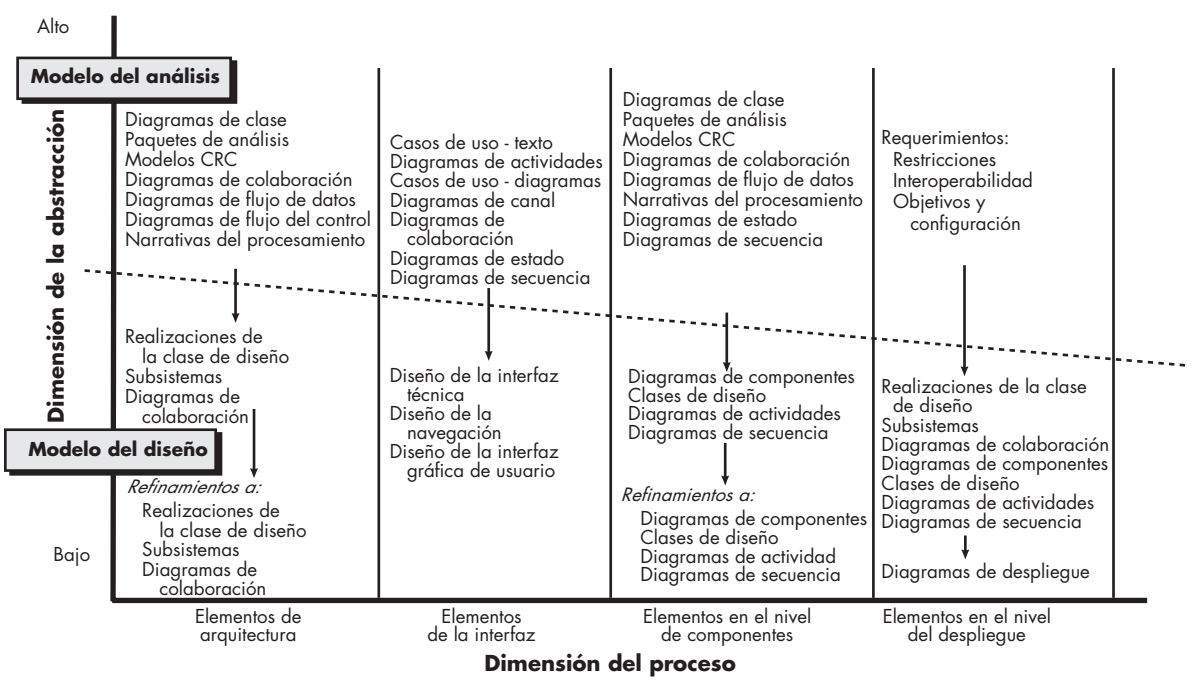

**PUNTO CLAVE**

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




www.FreeLibros.me

**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

www.FreeLibros.me

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.

www.FreeLibros.me

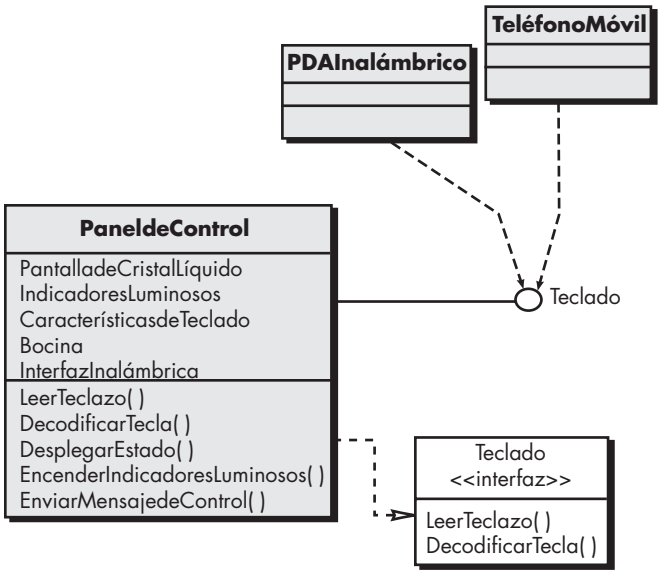



**FIGURA 8.5**
Representación de la interfaz para PaneldeControl

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-

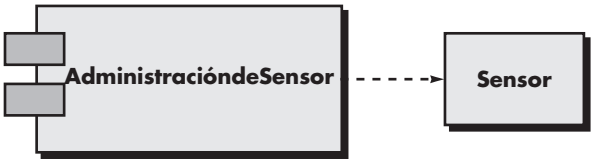



**FIGURA 8.6**
Diagrama de componente UML

www.FreeLibros.me

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

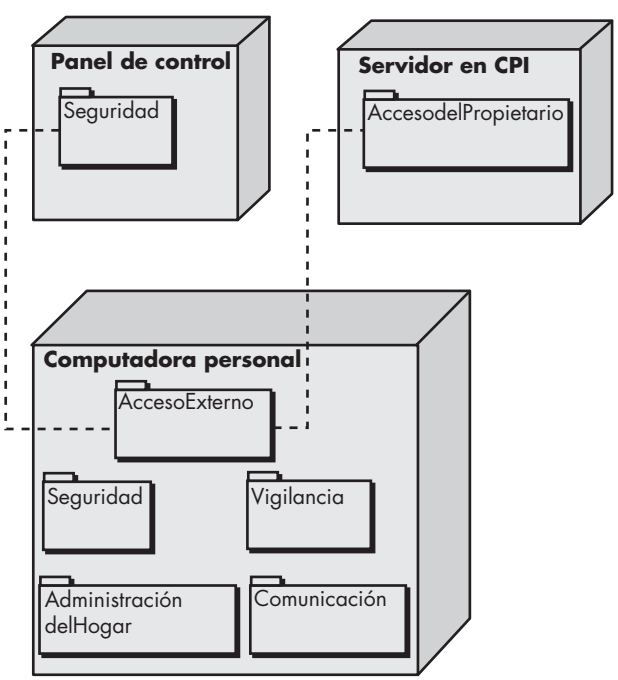
Los elementos del diseño del despliegue indican la forma en la que se acomodarán la funcionalidad del software y los subsistemas dentro del ambiente físico de la computación que lo apoyará. Por ejemplo, los elementos del producto *CasaSegura* se configuran para que operen dentro de tres ambientes de 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**




www.FreeLibros.me

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?

www.FreeLibros.me

**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

www.FreeLibros.me

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.

www.FreeLibros.me

CAPÍTULO

# 9 DISEÑO DE LA ARQUITECTURA

CONCEPTOS CLAVE



Se ha descrito al diseño como un proceso de etapas múltiples en el que, a partir de los requerimientos de información, se sintetizan las representaciones de los datos y la estructura del programa, las características de la interfaz y los detalles del procedimiento. Esta descripción la amplía Freeman [Fre80], como sigue:

> El diseño es una actividad que tiene que ver con la toma de decisiones importantes, con frecuencia de naturaleza estructural. Comparte con la programación el objetivo de abstraer una representación de la información y de las secuencias de procesamiento, pero en los extremos el grado de detalle es muy distinto. El diseño elabora representaciones coherentes y bien planeadas de programas, que se concentran en las relaciones de las partes en el nivel más alto y en las operaciones lógicas involucradas en los niveles bajos.

Como se señaló en el capítulo 8, el diseño está motivado por la información. Los métodos de diseño del software provienen de los tres dominios del modelo de análisis. Los dominios de datos, funciones y comportamiento sirven como guía para la creación del diseño del software.

En este capítulo se presentan los métodos requeridos para crear "representaciones coherentes y bien planeadas" de las capas de datos y de la arquitectura del modelo de diseño. El objetivo es brindar un enfoque sistemático para la obtención del diseño arquitectónico, el plano preliminar a partir del cual se elabora el software.

## UNA MIRADA RÁPIDA

**¿Qué es?** El diseño arquitectónico representa la estructura de los datos y de los componentes del programa que se requieren para construir un sistema basado en computadora. Considera el estilo de arquitectura que adoptará el sistema, la estructura y las propiedades de los componentes que lo constituyen y las interrelaciones que ocurren entre sus componentes arquitectónicos.

**¿Quién lo hace?** Aunque es un ingeniero de software quien puede diseñar tanto los datos como la arquitectura, es frecuente que si deben construirse sistemas grandes y complejos, el trabajo lo realicen especialistas. El diseñador de una base de datos o data warehouse crea la arquitectura de los datos para un sistema. El "arquitecto del sistema" selecciona un estilo arquitectónico apropiado a partir de los requerimientos obtenidos durante el análisis de los datos.

**¿Por qué es importante?** El lector no intentaría construir una casa sin un plano, ¿o sí? Tampoco comenzaría los planos con el dibujo de la plomería del lugar. Antes de preocuparse por los detalles, necesitaría tener el panorama general: la casa en sí. Eso es lo que hace el diseño arquitectónico, da el panorama y asegura que sea el correcto.

**¿Cuáles son los pasos?** El diseño de la arquitectura comienza con el diseño de los datos y continúa con la obtención de una o más representaciones de la estructura arquitectónica del sistema. Se analizan alternativas de estilos o patrones arquitectónicos para llegar a la estructura más adecuada para los requerimientos del usuario y para los atributos de calidad. Una vez seleccionada la alternativa, se elabora la arquitectura con el empleo de un método de diseño.

**¿Cuál es el producto final?** Durante el diseño arquitectónico se crea un modelo de arquitectura que incluye la arquitectura de los datos y la estructura del programa. Además, se describen las propiedades y relaciones (interacciones) que hay entre los componentes.

**¿Cómo me aseguro de que lo hice bien?** En cada etapa se revisan los productos del trabajo del diseño del software para que sean claros, correctos, completos y consistentes con los requerimientos y entre sí.



www.FreeLibros.me

## 9.1 ARQUITECTURA DEL SOFTWARE

En un libro clave sobre el tema, Shaw y Garlan [Sha96] plantean lo siguiente sobre la arquitectura del software:

> Desde el primer programa que se dividió en módulos, los sistemas de software han tenido arquitecturas y los programadores han sido los responsables de las interacciones entre los módulos y las propiedades globales del ensamble. Históricamente, las arquitecturas han estado implícitas: accidentes de implementación o sistemas heredados del pasado. Los desarrolladores de buen software han adoptado con frecuencia uno o varios patrones de arquitectura como estrategias para la organización del sistema, pero los utilizan de manera informal y no tienen manera de hacerlos explícitos en el sistema resultante.

En el presente, la arquitectura de software eficaz y su representación y diseño explícitos se han vuelto los temas dominantes en la ingeniería de software.

### 9.1.1 ¿Qué es la arquitectura?

**Cita:**

"La arquitectura de un sistema es un marco general que describe su forma y estructura: sus componentes y la manera en la que ajustan entre sí".

**Jerrold Grochow**

Cuando se piensa en la arquitectura de una construcción, llegan a la mente muchos atributos distintos. En el nivel más sencillo, se considera la forma general de la estructura física. Pero, en realidad, la arquitectura es mucho más que eso. Es la manera en la que los distintos componentes del edificio se integran para formar un todo cohesivo. Es la forma en la que la construcción se adapta a su ambiente y se integra a los demás edificios en la vecindad. Es el grado en el que el edificio cumple con su propósito y en el que satisface las necesidades del propietario. Es la sensación estética de la estructura —el efecto visual de la edificación— y el modo en el que se combinan texturas, colores y materiales para crear la fachada en el exterior y el "ambiente de vida" en el interior. Es los pequeños detalles: diseño de las lámparas, tipo de piso, color de las cortinas... la lista es casi interminable. Y, finalmente, es arte.

**PUNTO CLAVE**

La arquitectura del software debe modelar la estructura de un sistema y la manera en la que los datos y componentes del procedimiento colaboran entre sí.

Pero la arquitectura también es algo más. Son los "miles de decisiones, tanto grandes como pequeñas" [Tyt05]. Algunas de éstas se toman en una etapa temprana del diseño y tienen un efecto profundo en todas las demás acciones. Otras se dejan para más adelante, con lo que se eliminan las restricciones prematuras que llevarían a una mala implementación del estilo arquitectónico.

Pero, ¿qué es la arquitectura del software? Bass, Clements y Kazman [Bas03] definen este término tan elusivo de la manera siguiente:

> La arquitectura del software de un programa o sistema de cómputo es la estructura o estructuras del sistema, lo que comprende a los componentes del software, sus propiedades externas visibles y las relaciones entre ellos.

La arquitectura no es el software operativo. Es una representación que permite 1) analizar la efectividad del diseño para cumplir los requerimientos establecidos, 2) considerar alternativas arquitectónicas en una etapa en la que hacer cambios al diseño todavía es relativamente fácil y 3) reducir los riesgos asociados con la construcción del software.

**Cita:**

"Cásate con tu arquitectura de prisa, arrepiéntete en tu tiempo libre."

**Barry Boehm**

Esta definición pone el énfasis en el papel de los "componentes del software" en cualquier representación arquitectónica. En el contexto del diseño de la arquitectura, un componente del software puede ser algo tan simple como un módulo de programa o una clase orientada a objeto, pero también puede ampliarse para que incluya bases de datos y "middleware" que permitan la configuración de una red de clientes y servidores. Las propiedades de los componentes son aquellas características necesarias para entender cómo interactúan unos componentes con otros. En el nivel arquitectónico, no se especifican las propiedades internas (por ejemplo, detalles de un algoritmo). Las relaciones entre los componentes pueden ser tan simples como una invocación de procedimiento de un módulo a otro o tan complejos como un protocolo de acceso a una base de datos.

www.FreeLibros.me

Ciertos miembros de la comunidad de la ingeniería de software [Kaz03] hacen una diferencia entre las acciones asociadas con la obtención de una arquitectura de software (lo que el autor llama "diseño de la arquitectura") y las que se aplican para obtener el diseño del software. Como dijo un revisor de esta edición:

> Hay una diferencia entre los términos *arquitectura* y *diseño*. Un *diseño* es una instancia de una *arquitectura*, similar a un objeto que es una instancia de una clase. Por ejemplo, considere la arquitectura de cliente-servidor. Con esta arquitectura es posible diseñar de muchos modos un sistema de software basado en red, con el uso de una plataforma Java (Java EE) o Microsoft (estructura .NET). Entonces, hay una arquitectura, pero con base en ella pueden crearse muchos diseños. Así, no es válido mezclar "arquitectura" y "diseño".

**WebRef**

En la dirección **www2.umassd.edu/SECenter/SAResources.html**, se encuentran apuntadores útiles para muchos sitios de arquitectura de software.

Aunque el autor está de acuerdo con que un diseño de software es una instancia de una arquitectura específica de software, los elementos y estructuras que se definen como parte de ésta son el origen de todo diseño que evolucione a partir de ellos. El diseño comienza con la consideración de la arquitectura.

En este libro, el diseño de la arquitectura de software considera dos niveles de la pirámide del diseño (véase la figura 8.1): el diseño de los datos y el de la arquitectura. En el contexto del análisis precedente, el diseño de los datos permite representar el componente de datos de la arquitectura con definiciones de sistemas y clase convencionales (que incluyen atributos y operaciones) en sistemas orientados a objeto. El diseño arquitectónico se centra en la representación de la estructura de los componentes del software, sus propiedades e interacciones.

### 9.1.2 ¿Por qué es importante la arquitectura?

**Cita:**

**"La arquitectura es demasiado importante para dejarla en manos de una sola persona, no importa cuán brillante sea."**

**Scott Ambler**

En un libro dedicado a la arquitectura del software, Bass *et al*. [Bas03] identifican tres razones clave por las que es importante la arquitectura del software:

- Las representaciones de la arquitectura del software permiten la comunicación entre todas las partes (participantes) interesadas en el desarrollo de un sistema basado en computadora.
- La arquitectura resalta las primeras decisiones que tendrán un efecto profundo en todo el trabajo de ingeniería de software siguiente y, también importante, en el éxito último del sistema como entidad operacional.
- La arquitectura "constituye un modelo relativamente pequeño y asequible por la vía intelectual sobre cómo está estructurado el sistema y la forma en la que sus componentes trabajan juntos" [Bas03].

El modelo del diseño de la arquitectura y los patrones arquitectónicos contenidos dentro de éste son transferibles. Es decir, los géneros, estilos y patrones arquitectónicos pueden aplicarse al diseño de otros sistemas y representan un conjunto de abstracciones que permite a los ingenieros de software describir la arquitectura en formas predecibles.

**PUNTO CLAVE**

El modelo arquitectónico da un punto de vista de la Gestalt del sistema, lo que permite que el ingeniero de software lo examine como un todo.

### 9.1.3 Descripciones arquitectónicas

Cada uno de nosotros tiene una imagen mental de lo que significa la palabra *arquitectura*. Sin embargo, la realidad es que tiene significados diferentes para distintas personas. La conclusión es que los diversos participantes verán una arquitectura desde puntos de vista diferentes motivados por varios conjuntos de preocupaciones. Esto implica que una descripción arquitectónica en realidad es un conjunto de productos del trabajo que reflejan puntos de vista distintos del sistema.

Por ejemplo, el arquitecto de un gran edificio de oficinas debe trabajar con distintos participantes. La preocupación principal del propietario de la edificación (un participante) es garantizar el placer estético y que brinde suficiente espacio de oficinas e infraestructura para garantizar su rentabilidad. Por tanto, el arquitecto debe desarrollar una descripción con el empleo de pers-

www.FreeLibros.me

pectivas del edificio que se apeguen a las preocupaciones del dueño. Los puntos de vista empleados son dibujos del edificio en tres dimensiones (para ilustrar el aspecto estético) y un conjunto de planos en dos dimensiones que expliquen la preocupación por el espacio de oficinas y la infraestructura.


*El esfuerzo debe centrarse en representaciones de la arquitectura que guiarán todos los demás aspectos del diseño. Hay que dedicar tiempo a revisar con cuidado la arquitectura. Un error aquí tendrá un efecto negativo a largo plazo.*

Pero el espacio de oficinas tiene muchos otros participantes, incluido el fabricante de acero estructural que proveerá dicho material para la estructura del edificio. Necesita información arquitectónica detallada sobre el acero que soportará al edificio, incluso de las vigas tipo I, sus dimensiones, conectividad, materiales y muchos otros detalles. A estas preocupaciones se abocan diferentes productos del trabajo que representan distintos puntos de vista de la arquitectura. Los planos especializados (otro punto de vista) de la estructura de acero de la edificación se centran sólo en una de las muchas preocupaciones del fabricante.

La descripción de la arquitectura de un sistema basado en software debe tener características análogas a las mencionadas para el edificio de oficinas. Tyree y Ackerman [Tyr05] recalcan esto así: "Los desarrolladores desean lineamientos claros y decisivos sobre la forma de proceder con el diseño. Los consumidores desean la comprensión clara de los cambios ambientales que deben ocurrir y las garantías de que la arquitectura satisfará las necesidades de negocios. Otros arquitectos desean una comprensión clara y notable de los aspectos clave de la arquitectura." Cada uno de estos "deseos" se refleja en un punto de vista diferente representado con el uso de una perspectiva distinta.

IEEE Computer Society hizo la propuesta IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, [IEE00], con los siguientes objetivos: 1) establecer un marco conceptual con un vocabulario que se use durante el diseño de la arquitectura del software, 2) proporcionar lineamientos detallados para representar una descripción arquitectónica y 3) estimular las mejores prácticas del diseño arquitectónico.

El estándar IEEE define una *descripción arquitectónica* (DA) como "un conjunto de productos para documentar una arquitectura". La descripción en sí se representa con el uso de perspectivas múltiples, donde cada *perspectiva* es "una representación del sistema completo desde el punto de vista de un conjunto de preocupaciones relacionadas [de un participante]". Una *perspectiva* se crea de acuerdo con reglas y convenciones definidas en un *punto de vista*: "especificación de las convenciones para construir y usar una perspectiva" [IEE00]. En este capítulo se estudian varios productos del trabajo que se utilizan para desarrollar distintas perspectivas de la arquitectura del software.

### 9.1.4 Decisiones arquitectónicas

Cada perspectiva desarrollada como parte de una descripción arquitectónica se aboca a una preocupación de un participante específico. Para desarrollar cada perspectiva (y la descripción arquitectónica en su conjunto), el arquitecto del sistema considera varias alternativas y decide en última instancia las características arquitectónicas específicas que satisfagan del mejor modo la preocupación. Entonces, las decisiones arquitectónicas en sí mismas se consideran una perspectiva de la arquitectura. Las razones por las que se tomaron las decisiones dan una visión de un sistema y su concordancia con las preocupaciones del participante.

Como arquitecto simbólico, el lector puede usar el formato sugerido en el recuadro para documentar cada decisión importante. Al hacerlo, presenta la racionalidad de su trabajo y deja un registro histórico que será útil cuando deban hacerse modificaciones del diseño.

## 9.2 Géneros arquitectónicos

Aunque los principios subyacentes del diseño arquitectónico se aplican a todos los tipos de la arquitectura, con frecuencia será el *género* arquitectónico el que dicte el enfoque específico para la estructura que deba construirse. En el contexto del diseño arquitectónico, el *género* implica

www.FreeLibros.me

INFORMACIÓN

### *Formato para la descripción de una decisión arquitectónica*

Toda decisión arquitectónica de importancia puede documentarse para que posteriormente la revisen los participantes que deseen entender la descripción de la arquitectura propuesta. El formato que se presenta en este recuadro es una versión adaptada y abreviada de otro, propuesto por Tyree y Ackerman [Tyr05].

**Aspecto del diseño:** Se describen los aspectos del diseño arquitectónico que se van a abordar.

**Resolución:** Se establece el enfoque escogido para abordar el aspecto de diseño.

**Categoría:** Se especifica la categoría de diseño a que se aboca el aspecto y la resolución (por ejemplo, diseño de datos, estructura del contenido, estructura del componente, integración, presentación, etcétera).

**Suposiciones:** Se indican cualesquiera suposiciones que ayuden a dar forma a la decisión.

**Restricciones:** Se especifican todas las restricciones ambientales que ayuden a conformar la decisión (como los estándares tecnológicos, patrones disponibles y aspectos relacionados con el diseño).

**Alternativas:** Se describen con brevedad las alternativas de diseño arquitectónico que se consideraron y la razón por la que se rechazaron.

**Argumento:** Se establece por qué se eligió la resolución sobre las demás alternativas.

**Implicaciones:** Se indican las consecuencias que tendrá la toma de la decisión en el diseño. ¿Cómo afectará la resolución a otros aspectos del diseño de la arquitectura? ¿La resolución restringe de algún modo al diseño?

**Decisiones relacionadas:** ¿Qué otros documentos se relacionan con esta decisión?

**Preocupaciones relacionadas:** ¿Qué otros requerimientos se relacionan con esta decisión?

**Productos finales:** Se indica dónde se reflejará esta decisión en la descripción arquitectónica.

**Notas:** Se hace referencia a las anotaciones del equipo u otra clase de documentación que se haya empleado para tomar la decisión.

PUNTO CLAVE

Diversos estilos arquitectónicos pueden aplicarse a un género específico (también conocido como dominio de aplicación).

una categoría específica dentro del dominio general del software. Dentro de cada categoría hay varias subcategorías. Por ejemplo, dentro del género *edificios*, se encuentran los siguientes estilos generales: casas, condominios, edificios de departamentos, edificios de oficinas, edificios industriales, bodegas, etc. Dentro de cada estilo general habrá estilos más específicos (véase la sección 9.3). Cada estilo tendrá una estructura que puede describirse con el uso de un conjunto de patrones predecibles.

En su texto evolutivo *Handbook of Software Architecture* [Boo08], Grady Booch sugiere los siguientes géneros arquitectónicos para sistemas basados en software:

- **Inteligencia artificial:** Sistemas que simulan o incrementan la cognición humana, su locomoción u otros procesos orgánicos.
- **Comerciales y no lucrativos:** Sistemas que son fundamentales para la operación de una empresa de negocios.
- **Comunicaciones:** Sistemas que proveen la infraestructura para transferir y manejar datos, para conectar usuarios de éstos o para presentar datos en la frontera de una infraestructura.
- **Contenido de autor:** Sistemas que se emplean para crear o manipular artefactos de texto o multimedios.
- **Dispositivos:** Sistemas que interactúan con el mundo físico a fin de brindar algún servicio puntual a un individuo.
- **Entretenimiento y deportes:** Sistemas que administran eventos públicos o que proveen una experiencia grupal de entretenimiento.
- **Financieros:** Sistemas que proporcionan la infraestructura para transferir y manejar dinero y otros títulos.
- **Juegos:** Sistemas que dan una experiencia de entretenimiento a individuos o grupos.

www.FreeLibros.me

**Cita:**

"Programar sin una arquitectura o diseño general en mente es como explorar una caverna sólo con una linterna: no sabes dónde has estado, a dónde vas ni dónde estás."

**Danny Thorpe**

- **Gobierno:** Sistemas que dan apoyo a la conducción y operaciones de una institución política local, estatal, federal, global o de otro tipo.
- **Industrial:** Sistemas que simulan o controlan procesos físicos.
- **Legal:** Sistemas que dan apoyo a la industria jurídica.
- **Médicos:** Sistemas que diagnostican, curan o contribuyen a la investigación médica.
- **Militares:** Sistemas de consulta, comunicaciones, comando, control e inteligencia (o C4I), así como de armas ofensivas y defensivas.
- **Sistemas operativos:** Sistemas que están inmediatamente instalados en el hardware para dar servicios de software básico.
- **Plataformas:** Sistemas que se encuentran en los sistemas operativos para brindar servicios avanzados.
- **Científicos:** Sistemas que se emplean para hacer investigación científica y aplicada.
- **Herramientas:** Sistemas que se utilizan para desarrollar otros sistemas.
- **Transporte:** Sistemas que controlan vehículos acuáticos, terrestres, aéreos o espaciales.
- **Utilidades:** Sistemas que interactúan con otro software para brindar algún servicio específico.

Desde el punto de vista del diseño arquitectónico, cada género representa un desafío único. Por ejemplo, considere la arquitectura del software de un sistema de juego. Esta clase de sistemas, en ocasiones llamados *aplicaciones interactivas de inmersión*, requieren el cómputo de algoritmos intensivos, gráficas avanzadas en computadora, fuentes de datos continuas en multimedios, interactividad en tiempo real a través de dispositivos de entrada convencionales y no convencionales, y otras preocupaciones especializadas.

Alexander Francois [Fra03] sugiere una arquitectura del software para *inmerpresencia*[1] que se aplica a un ambiente de juegos. Él describe la arquitectura de la manera siguiente:

> La ASI (Arquitectura de Software de Inmerpresencia) es un modelo nuevo de arquitectura de software para diseñar, analizar e implementar aplicaciones que realizan un procesamiento distribuido, asíncrono y paralelo de flujos de datos generales. El objetivo de la ASI es proveer un marco universal para la implementación distribuida de algoritmos y su fácil integración en sistemas complejos [...] El modelo de procesamiento de datos extensibles subyacentes y el modelo de procesamiento híbrido (depósito y transmisión de mensajes compartidos), asíncrono y en paralelo permiten la manipulación natural y eficiente de flujos de datos generales con el uso indistinto de librerías ya existentes o código original. La modularidad del estilo facilita el desarrollo de código distribuido, pruebas y reutilización, así como el diseño e integración rápida del sistema y su mantenimiento y evolución.

El análisis detallado del ASI queda fuera del alcance de este libro. No obstante, es importante reconocer que el género de sistemas de juegos puede abordarse con un estilo arquitectónico (véase la sección 9.3) diseñado específicamente para resolver preocupaciones de sistemas de juego. Si el lector tiene especial interés, consulte [Fra03].

## 9.3 Estilos arquitectónicos

Cuando un constructor usa la frase "vestíbulo central colonial" para describir una casa, la mayor parte de personas familiarizadas con viviendas en Estados Unidos se hará una imagen general de ella y de cuál es su probable distribución. El constructor usó un *estilo arquitectónico* como mecanismo descriptivo para diferenciar la casa de otros estilos (por ejemplo, de dos aguas, finca

1 Francois utiliza el término *inmerpresencia* para denotar aplicaciones interactivas de inmersión.

www.FreeLibros.me

**Cita:**

"En el fondo de la mente de todo artista hay un patrón o tipo de arquitectura."

**G. K. Chesterton**

campestre, Cabo Cod, etc.). Pero, lo que es más importante, el estilo arquitectónico también es una plantilla para la construcción. Deben definirse más detalles, especificar sus dimensiones finales, agregar características personalizadas, determinar los materiales de construcción, pero el estilo (un "vestíbulo central colonial") orienta al constructor en su trabajo.

El software construido para sistemas basados en computadora también tiene uno de muchos estilos arquitectónicos. Cada estilo describe una categoría de sistemas que incluye 1) un conjunto de componentes (como una base de datos o módulos de cómputo) que realizan una función requerida por el sistema, 2) un conjunto de conectores que permiten la "comunicación, coordinación y cooperación" entre los componentes, 3) restricciones que definen cómo se integran los componentes para formar el sistema y 4) modelos semánticos que permiten que un diseñador entienda las propiedades generales del sistema al analizar las propiedades conocidas de sus partes constituyentes [Bas03].

**¿Qué es un estilo arquitectónico?**

Un estilo arquitectónico es una transformación que se impone al diseño de todo el sistema. El objetivo es establecer una estructura para todos los componentes del sistema. En el caso en el que ha de hacerse la reingeniería de una arquitectura ya existente (véase el capítulo 29), la imposición de un estilo arquitectónico dará como resultado cambios fundamentales en la estructura del software, incluida la reasignación de las funciones de los componentes [Bos00].

**WebRef**

Los estilos arquitectónicos basados en atributos (ABAS) pueden usarse como bloques de construcción para las arquitecturas de software. En la dirección **www.sei.cmu.edu/architecture/abas.html**, hay información al respecto.

Un patrón arquitectónico, como un estilo de arquitectura, impone la transformación del diseño de una arquitectura. Sin embargo, un patrón difiere de un estilo en varias formas fundamentales: 1) el alcance del patrón es menos amplio y se centra en un aspecto de la arquitectura más que en el total de ésta, 2) un patrón impone una regla a la arquitectura, describe la manera en la que el software manejará ciertos aspectos de su funcionalidad en el nivel de la infraestructura (por ejemplo, la concurrencia) [Bos00], 3) los patrones arquitectónicos (véase la sección 9.4) tienden a abocarse a aspectos específicos del comportamiento en el contexto de la arquitectura (por ejemplo, cómo manejarán la sincronización o las interrupciones las aplicaciones en tiempo real). Los patrones se utilizan junto con un estilo arquitectónico para dar forma a la estructura

**INFORMACIÓN**

### *Estructuras arquitectónicas canónicas*

En esencia, la arquitectura del software representa una estructura en la que cierta colección de entidades (con frecuencia llamados *componentes*) está conectada por un conjunto de relaciones definidas (usualmente llamadas *conectores*). Tanto las componentes como los conectores están asociados con un conjunto de *propiedades* que permiten que el diseñador diferencie los tipos de componentes y conectores que pueden usarse. Pero, ¿qué clases de estructuras (componentes, conectores y propiedades) se utilizan para describir una arquitectura? Bass y Kazman [Bas03] sugieren cinco estructuras canónicas o fundamentales:

**Estructura funcional.** Los componentes representan entidades de función o procesamiento. Los conectores representan interfaces que proveen la capacidad de "usar" o "pasar datos a" un componente. Las propiedades describen la naturaleza de los componentes y la organización de las interfaces.

**Estructura de implementación.** "Los componentes son paquetes, clases, objetos, procedimientos, funciones, métodos, etc., que son vehículos para empacar funciones en varios niveles de abstracción" [Bas03]. Los conectores incluyen la capacidad de pasar datos y control, compartir datos, "usar" y "ser una instancia de". Las propiedades se centran en las características de la calidad (por ejemplo, facilidad de recibir mantenimiento, ser reutilizables, etc.) que surgen cuando se implementa la estructura.

**Estructura de concurrencia.** Los componentes representan "unidades de concurrencia" que están organizadas como tareas o trayectorias paralelas. "Las relaciones [conectores] incluyen sincronizarse-con, tiene-mayor-prioridad-que, envía-datos-a, no-corre-sin y no-corre-con. Las propiedades relevantes para esta estructura incluyen prioridad, anticipación y tiempo de ejecución" [Bas03].

**Estructura física.** Esta estructura es similar al modelo de despliegue desarrollado como parte del diseño. Los componentes son el hardware físico en el que reside el software. Los conectores son las interfaces entre los componentes del hardware y las propiedades incluyen la capacidad, ancho de banda y rendimiento, entre otros atributos.

**Estructura de desarrollo.** Esta estructura define los componentes, productos del trabajo y otras fuentes de información que se requieren a medida que avanza la ingeniería de software. Los conectores representan las relaciones entre los productos del trabajo; las propiedades identifican las características de cada aspecto.

Cada una de estas estructuras presenta un punto de vista de la arquitectura del software y expone información que es útil para el equipo de software a medida que realiza la modelación y construcción.

www.FreeLibros.me

general de un sistema. En la sección 9.3.1 se consideran los estilos y patrones arquitectónicos comunes para el software.

### 9.3.1 Breve taxonomía de estilos de arquitectura

**Cita:**

"El uso de patrones y estilos de diseño está presente en todas las disciplinas de la ingeniería."

**Mary Shaw y David Garlan**

Aunque en los últimos 60 años se han creado millones de sistemas basados en computadoras, la gran mayoría se clasifica en un número relativamente pequeño de estilos de arquitectura:

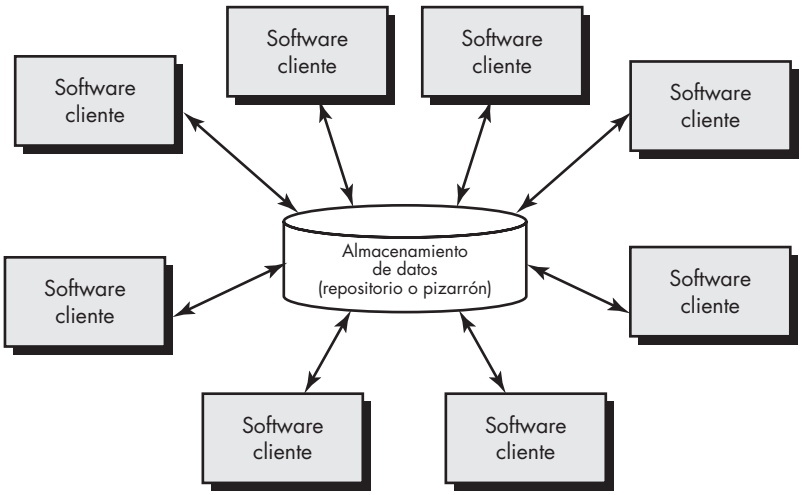
**Arquitecturas centradas en los datos.** En el centro de esta arquitectura se halla un almacenamiento de datos (como un archivo o base de datos) al que acceden con frecuencia otros componentes que actualizan, agregan, eliminan o modifican los datos de cierto modo dentro del almacenamiento. La figura 9.1 ilustra un estilo común centrado en datos. El software cliente accede a un repositorio (depósito) central. En ciertos casos éste es pasivo. Es decir, el software cliente accede a los datos en forma independiente de cualesquiera cambios que éstos sufran o de las acciones del software de otro cliente. Una variante de este enfoque transforma el depósito en un "pizarrón" que envía notificaciones al software cliente cuando hay un cambio en datos de interés del cliente.

Las arquitecturas centradas en datos promueven la *integrabilidad* [Bas03]. Es decir, los componentes del software pueden ser cambiados y agregarse otros nuevos, del cliente, a la arquitectura sin problemas con otros clientes (porque los componentes del cliente operan en forma independiente). Además, pueden pasarse datos entre clientes con el uso de un mecanismo de pizarrón (el componente pizarrón sirve para coordinar la transferencia de información entre clientes). Los componentes del cliente ejecutan los procesos con independencia.

**Arquitecturas de flujo de datos.** Esta arquitectura se aplica cuando datos de entrada van a transformarse en datos de salida a través de una serie de componentes computacionales o manipuladores. Un patrón de tubo y filtro (véase la figura 9.2) tiene un conjunto de componentes, llamados *filtros*, conectados por *tubos* que transmiten datos de un componente al siguiente. Cada filtro trabaja en forma independiente de aquellos componentes que se localizan arriba o abajo del flujo; se diseña para esperar una entrada de datos de cierta forma y produce datos de salida (al filtro siguiente) en una forma especificada. Sin embargo, el filtro no requiere ningún conocimiento de los trabajos que realizan los filtros vecinos.

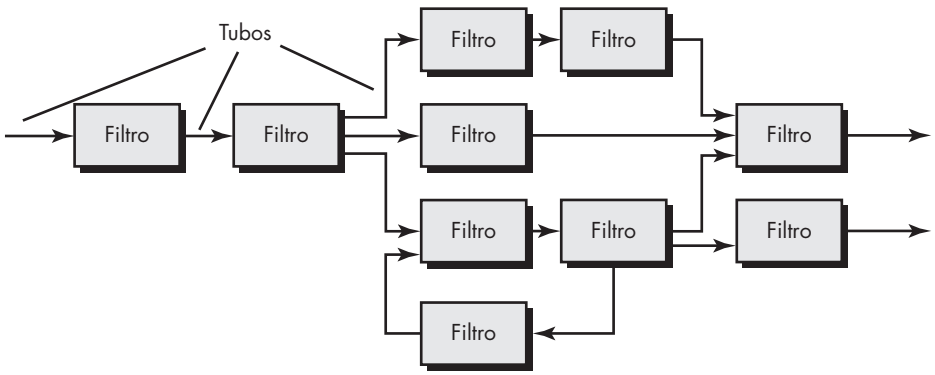
Si el flujo de datos degenera en una sola línea de transformaciones, se denomina lote secuencial. Esta estructura acepta un lote de datos y luego aplica una serie de componentes secuenciales (filtros) para transformarlos.

**FIGURA 9.1**
**Arquitectura centrada en datos**




www.FreeLibros.me

**FIGURA 9.2**
**Arquitectura de flujo de datos**




Tubos y filtros

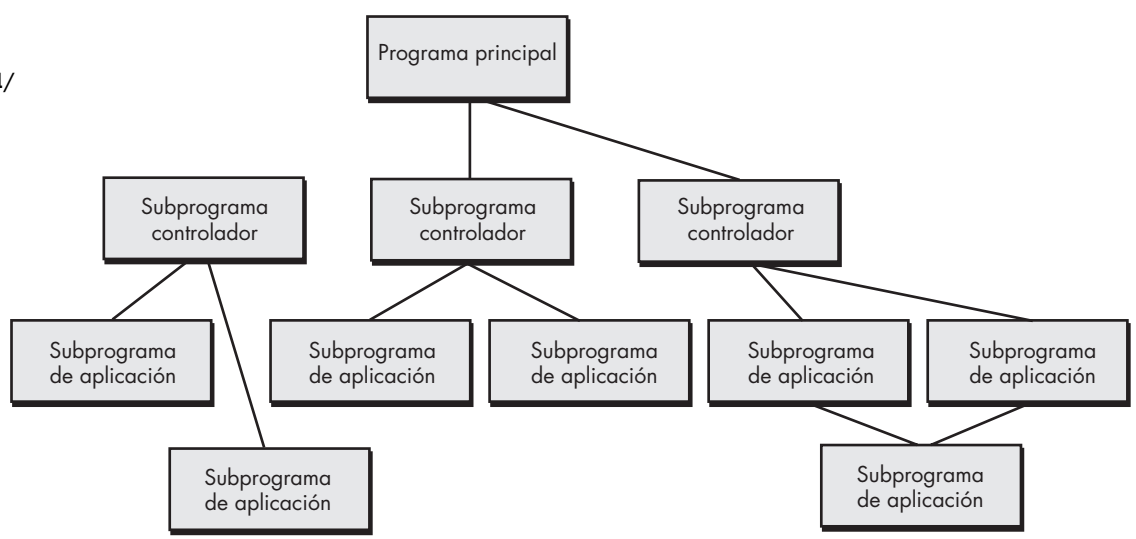
**Arquitecturas de llamar y regresar.** Este estilo arquitectónico permite obtener una estructura de programa que es relativamente fácil de modificar y escalar. Dentro de esta categoría existen varios subestilos [Bas03]:

- *Arquitecturas de programa principal/subprograma*. Esta estructura clásica de programa descompone una función en una jerarquía de control en la que un programa "principal" invoca cierto número de componentes de programa que a su vez invocan a otros. La figura 9.3 ilustra una arquitectura de este tipo.
- *Arquitecturas de llamada de procedimiento remoto.* Los componentes de una arquitectura de programa principal/subprograma están distribuidos a través de computadoras múltiples en una red.

**Arquitecturas orientadas a objetos.** Los componentes de un sistema incluyen datos y las operaciones que deben aplicarse para manipularlos. La comunicación y coordinación entre los componentes se consigue mediante la transmisión de mensajes.

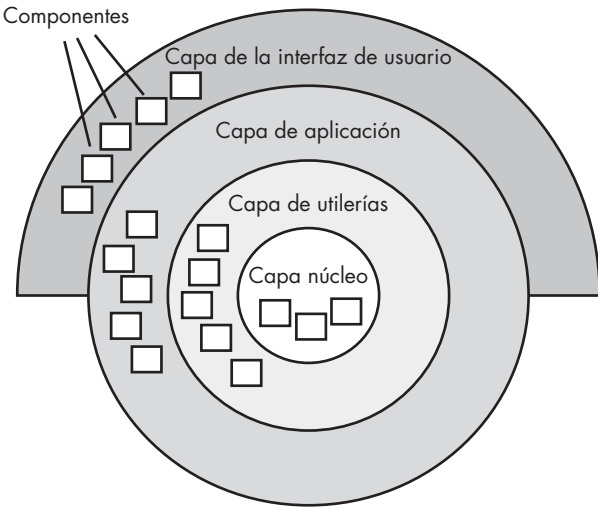
**Arquitecturas en capas.** En la figura 9.4 se ilustra la estructura básica de una arquitectura en capas. Se define un número de capas diferentes; cada una ejecuta operaciones que se aproximan progresivamente al conjunto de instrucciones de máquina. En la capa externa, los compo-

**FIGURA 9.3**
**Arquitectura de programa principal/subprograma**




www.FreeLibros.me

**FIGURA 9.4**
**Arquitectura en capas**




nentes atienden las operaciones de la interfaz de usuario. En la interna, los componentes realizan la interfaz con el sistema operativo. Las capas intermedias proveen servicios de utilerías y funciones de software de aplicación.

Estos estilos arquitectónicos tan sólo son un pequeño subconjunto de los que están disponibles.[2] Una vez que la ingeniería de requerimientos revela las características y restricciones del sistema que se va a elaborar, se elige el estilo arquitectónico o la combinación de patrones que se ajusten mejor a esas características y restricciones. En muchos casos, más de un patrón es apropiado y es posible diseñar y evaluar estilos arquitectónicos alternativos. Por ejemplo, en muchas aplicaciones de bases de datos se combina un estilo en capas (apropiado para la mayoría de sistemas) con una arquitectura centrada en datos.

### 9.3.2 Patrones arquitectónicos

**Cita:**

"Tal vez está en la planta baja. Déjame subir a revisar."

**M. C. Escher**

A medida que se desarrolle el modelo de requerimientos, se observará que el software debe enfrentar cierto número de problemas amplios que abarcan toda la aplicación. Por ejemplo, el modelo de requerimientos para virtualmente cualquier aplicación de comercio electrónico se enfrenta con el siguiente problema: *¿Cómo ofrecer una amplia variedad de bienes a un grupo extenso de consumidores y permitir que los adquieran en línea?*

El modelo de requerimientos también define un contexto en el que debe responderse esta pregunta. Por ejemplo, un negocio de comercio electrónico que venda equipo de golf de consumo operará en un contexto diferente que otro que venda equipo industrial de precio alto a corporaciones medianas y pequeñas. Además, hay varias limitantes y restricciones que afectan la manera en la que se aborda este problema para resolverlo.

Los patrones arquitectónicos se abocan a un problema de aplicación específica dentro de un contexto dado y sujeto a limitaciones y restricciones. El patrón propone una solución arquitectónica que sirve como base para el diseño de la arquitectura.

En una sección anterior, se dijo que la mayor parte de aplicaciones caen dentro de un dominio o género específico, y que para éstas son apropiados uno o más estilos de arquitectura. Por ejemplo, el estilo de arquitectura general para una aplicación podría ser el de llamar y regresar o el que está orientado a objeto. Pero dentro de ese estilo se encontrará un conjunto de problemas comu-

2 Para un estudio detallado de estilos y patrones arquitectónicos, véase [Bus07], [Gor06], [Roz05], [Bas03], [Bos00] y [Hof00].

www.FreeLibros.me

**CasaSegura**

### *Elección de un estilo de arquitectura*

**La escena:** Cubículo de Jamie, cuando comienza la modelación del diseño.

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

**La conversación:**

**Ed (frunce el ceño):** Hemos estado modelando la función de seguridad con el empleo de UML, ya sabes, clases, relaciones y demás. Así que supongo que la arquitectura orientada a objeto[3] es la elección apropiada.

**Jamie:** ¿Pero…?

**Ed:** Pero… tengo problemas para visualizar lo que es una arquitectura orientada a objeto. Entiendo la arquitectura de llamar y regresar, que es algo así como una jerarquía de proceso convencional, pero la orientada a objeto… no sé, parece algo amorfo.

**Jamie (sonríe):** Amorfo, ¿eh?

**Ed:** Sí… lo que quiero decir es que no visualizo una estructura real, sólo clases de diseño que flotan en el espacio.

**Jamie:** Bueno, eso no es cierto. Hay jerarquías de clase… piensa en la jerarquía (agregado) que hicimos para el objeto **Plano** [figura 8.3]. Una arquitectura orientada a objetos es una combinación de esta estructura y de las interconexiones —ya sabes, colaboraciones— entre las clases. Puede mostrarse con la descripción completa de los atributos y operaciones, los mensajes que hay y la estructura de las clases.

**Ed:** Voy a dedicar una hora a mapear una arquitectura de llamar y regresar; luego volveré a considerar la orientada a objeto.

**Jamie:** Doug no tendrá problemas con eso. Dijo que deben considerarse alternativas arquitectónicas. Por cierto, no hay ninguna razón para no usar estas arquitecturas combinadas entre sí.

**Ed:** Bueno. Estoy en eso.

nes que se abordan mejor con patrones arquitectónicos específicos. En el capítulo 12 se presentan algunos de estos problemas y se hace un estudio más completo de los patrones de arquitectura.

### 9.3.3 Organización y refinamiento

Debido a que es frecuente que el proceso de diseño permita varias alternativas de arquitectura, es importante establecer un conjunto de criterios de diseño que se usan para evaluar el diseño arquitectónico obtenido. Las preguntas que siguen [Bas03] dan una visión del estilo de arquitectura:

**? ¿Cómo se evalúa el estilo arquitectónico que se haya obtenido?**

**Control.** ¿Cómo se administra el control dentro de la arquitectura? ¿Existe una jerarquía de control distinta? Si es así, ¿cuál es el papel de los componentes dentro de esta jerarquía de control? ¿Cómo transfieren el control los componentes dentro del sistema? ¿Cómo lo comparten? ¿Cuál es la topología del control (por ejemplo, la forma geométrica que adopta el control)? ¿El control está sincronizado o los componentes operan en forma asincrónica?

**Datos.** ¿Cómo se comunican los datos entre los componentes? ¿El flujo de datos es continuo o los objetos de datos pasan al sistema en forma esporádica? ¿Cuál es el modo de transferencia de datos (pasan de un componente a otro o se dispone de ellos globalmente para compartirse entre los componentes del sistema)? ¿Existen componentes de datos (como un pizarrón o depósito) y, si así fuera, cuál es su papel? ¿Cómo interactúan los componentes funcionales con los componentes de datos? ¿Los componentes de datos son pasivos o activos (el componente de datos actúa activamente con otros componentes del sistema)? ¿Cómo interactúan los datos y el control dentro del sistema?

3 Podría afirmarse que la arquitectura de *CasaSegura* debe considerarse en un nivel más alto que la que se comenta. *CasaSegura* tiene varios subsistemas: vigilancia de las funciones del hogar, vigilancia del sitio de la empresa y el subsistema que corre en la PC del propietario. Dentro de los subsistemas son comunes los procesos concurrentes (por ejemplo, los que vigilan sensores) y los que manejan eventos. En este nivel, algunas decisiones de arquitectura se toman durante la ingeniería del producto, pero el diseño arquitectónico dentro de la ingeniería de software muy bien debe considerar estos aspectos.

www.FreeLibros.me

Estas preguntas dan al diseñador una evaluación temprana de la calidad del diseño y proporcionan el fundamento para hacer análisis más detallados de la arquitectura.

## 9.4 Diseño arquitectónico

**Cita:**

"Un doctor puede sepultar sus errores, pero un arquitecto sólo puede aconsejar a su cliente que siembre enredaderas."

**Frank Lloyd Wright**

Cuando comienza el diseño arquitectónico, el software que se va a desarrollar debe situarse en contexto, es decir, el diseño debe definir las entidades externas (otros sistemas, dispositivos, personas, etc.) con las que interactúa el software y la naturaleza de dicha interacción. Esta información por lo general se adquiere a partir del modelo de los requerimientos y de toda la que se reunió durante la ingeniería de éstos. Una vez que se ha modelado el contexto y descrito todas las interfaces externas del software, se identifica un conjunto de arquetipos de arquitectura. Un *arquetipo* es una abstracción (similar a una clase) que representa un elemento de comportamiento del sistema. El conjunto de arquetipos provee una colección de abstracciones que deben modelarse en cuanto a la arquitectura si el sistema ha de construirse, pero los arquetipos por sí mismos no dan suficientes detalles para la implementación. Por tanto, el diseñador especifica la estructura del sistema, definiendo y refinando los componentes del software que implementan cada arquetipo. Este proceso sigue en forma iterativa hasta que se obtiene una estructura arquitectónica completa. En las secciones que siguen se estudia cada una de estas tareas de diseño arquitectónico con un poco más de detalle.

### 9.4.1 Representación del sistema en contexto

**PUNTO CLAVE**

El contexto arquitectónico representa la manera en la que el software interactúa con las entidades externas a sus fronteras.

En el nivel de diseño arquitectónico, el arquitecto del software usa un *diagrama de contexto arquitectónico* (DCA) para modelar la manera en la que el software interactúa con entidades más allá de sus fronteras. La estructura general del diagrama de contexto arquitectónico se ilustra en la figura 9.5.

**? ¿Cómo interactúan los sistemas uno con el otro?**

En relación con dicha figura, los sistemas que interactúan con el *sistema objetivo* (aquel para el que va a desarrollarse un diseño arquitectónico) están representados como sigue:

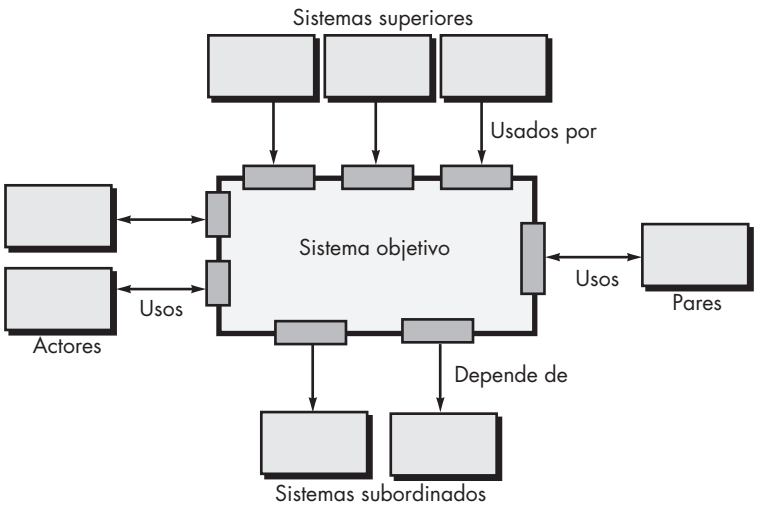
- *Sistemas superiores*: aquellos que utilizan al sistema objetivo como parte de algún esquema de procesamiento de alto nivel.
- *Sistemas subordinados*: los que son usados por el sistema objetivo y proveen datos o procesamiento que son necesarios para completar las funciones del sistema objetivo.
- *Sistemas entre iguales*: son los que interactúan sobre una base de igualdad (por ejemplo, la información se produce o consume por los iguales y por el sistema objetivo).
- *Actores*: entidades (personas, dispositivos, etc.) que interactúan con el sistema objetivo mediante la producción o consumo de información que es necesaria para el procesamiento de los requerimientos.

Cada una de estas entidades externas se comunica con el sistema objetivo a través de una interfaz (rectángulos pequeños sombreados).

Para ilustrar el empleo del DCA, considere la función de seguridad del hogar del producto *CasaSegura*. El controlador general del producto *CasaSegura* y el sistema basado en internet son superiores respecto de la función de seguridad y se muestran por arriba de la función en la figura 9.6. La función de vigilancia es un *sistema entre iguales* y en las versiones posteriores del producto usa (es usada por) la función de seguridad del hogar. El propietario y los paneles de control son actores que producen y consumen información usada o producida por el software de seguridad. Por último, los sensores se utilizan por el software de seguridad y se muestran como subordinados a éste.

www.FreeLibros.me

**FIGURA 9.5**
**Diagrama de contexto arquitectónico**
*Fuente:* Adaptado de [Bos00].




Como parte del diseño arquitectónico, tendrían que especificarse los detalles de cada interfaz mostrada en la figura 9.6. En esta etapa deben identificarse todos los datos que fluyen hacia dentro y fuera del sistema objetivo.

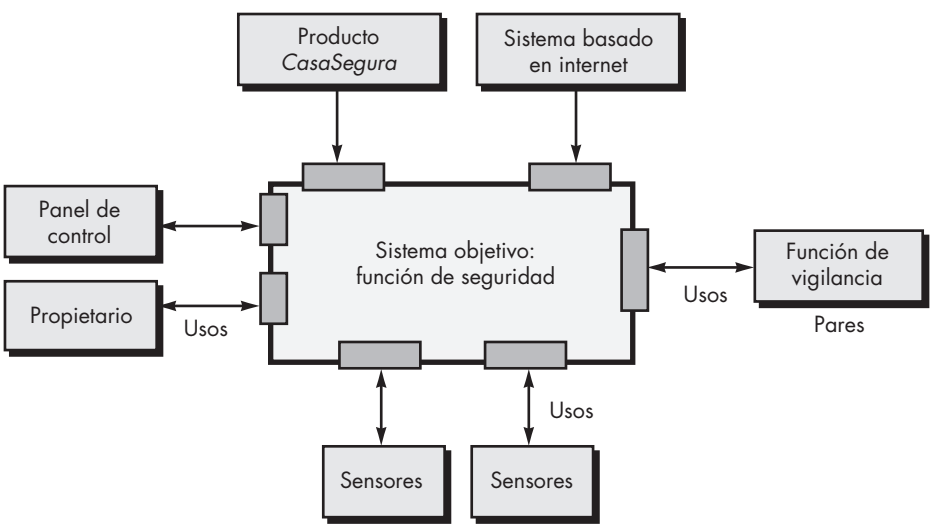
### 9.4.2 Definición de arquetipos

PUNTO CLAVE
Los arquetipos son los bloques constructivos de un diseño arquitectónico.

Un *arquetipo* es una clase o un patrón que representa una abstracción fundamental de importancia crítica para el diseño de una arquitectura para el sistema objetivo. En general, se requiere de un conjunto relativamente pequeño de arquetipos a fin de diseñar sistemas incluso algo complejos. La arquitectura del sistema objetivo está compuesta de estos arquetipos, que representan elementos estables de la arquitectura, pero que son implementadas en muchos modos diferentes con base en el comportamiento del sistema.

En muchos casos, los arquetipos se obtienen con el estudio de las clases de análisis definidas como parte del modelo de los requerimientos. Al continuar el análisis de la función de seguridad de *CasaSegura*, se definirán los arquetipos siguientes:

**FIGURA 9.6**
**Diagrama de contexto arquitectónico para la función de seguridad de *CasaSegura***




www.FreeLibros.me

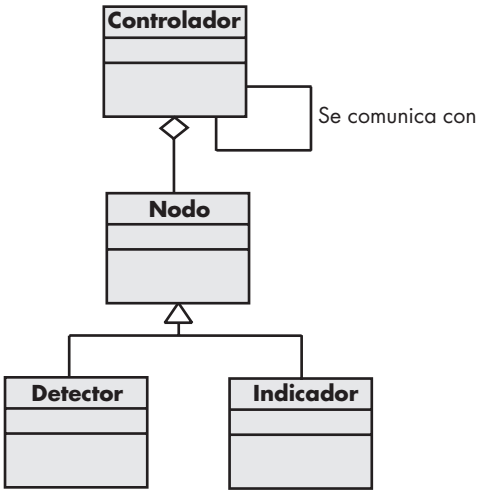



**FIGURA 9.7**
**Relaciones de UML para los arquetipos de la función de seguridad de *CasaSegura***
*Fuente:* Adaptado de [Bos00].

- **Nodo.** Representa una colección cohesiva de elementos de entrada y salida de la función de seguridad del hogar. Por ejemplo, un nodo podría comprender 1) varios sensores y 2) varios indicadores de alarma (salida).
- **Detector.** Abstracción que incluye todos los equipos de detección que alimentan con información al sistema objetivo.
- **Indicador.** Abstracción que representa todos los mecanismos (como la sirena de la alarma, luces, campana, etc.) que indican que está ocurriendo una condición de alarma.
- **Controlador.** Abstracción que ilustra el mecanismo que permite armar o desarmar un nodo. Si los controladores residen en una red, tienen la capacidad de comunicarse entre sí.

En la figura 9.7 se muestra cada uno de estos arquetipos con el empleo de notación UML. Recuerde que los arquetipos constituyen la base de la arquitectura, pero son abstracciones que deben refinarse a medida que avanza el diseño arquitectónico. Por ejemplo, **Detector** se refinaría en una jerarquía de clase de sensores.

### 9.4.3 Refinamiento de la arquitectura hacia los componentes

**Cita:**

"La estructura de un sistema de software provee la ecología en la que el código nace, crece y muere. Un hábitat bien diseñado permite la evolución exitosa de todos los componentes necesarios en un sistema de software."

**R. Pattis**

Conforme la arquitectura se refina hacia los componentes, comienza a emerger la estructura del sistema. Pero, ¿cómo se eligen estos componentes? Para responder esta pregunta se comienza con las clases descritas como parte del modelo de requerimientos.[4] Estas clases de análisis representan entidades dentro del dominio de aplicación (negocio) que deben enfrentarse dentro de la arquitectura del software. Entonces, el dominio de aplicación es una fuente para la obtención y refinamiento de los componentes. Otra fuente es el dominio de la infraestructura. La arquitectura debe albergar muchas componentes de la infraestructura que hagan posible los componentes de la aplicación, pero que no tengan conexión con el dominio de ésta. Por ejemplo, los componentes de administración de memoria, de comunicación, de base de datos y de administración de tareas con frecuencia están integrados en la arquitectura del software.

Las interfaces ilustradas en el diagrama de contexto de la arquitectura (sección 9.4.1) implican uno o más componentes especializados que procesan los datos que fluyen a través de la

4 Si se elige un enfoque convencional (no orientado a objeto), los componentes se obtienen a partir de los datos del modelo del flujo. Este enfoque se estudia brevemente en la sección 9.6.

www.FreeLibros.me

interfaz. En ciertos casos (por ejemplo, una interfaz de usuario gráfica) debe diseñarse una arquitectura completa con muchos componentes para el subsistema.

Al seguir con el ejemplo de la función de seguridad de *CasaSegura*, debe definirse el conjunto de componentes de alto nivel que se aboque a las funciones siguientes:

- *Administración de la comunicación externa*: coordina la comunicación de la función de seguridad con entidades externas, tales como otros sistemas basados en internet y la notificación externa de una alarma.
- *Procesamiento del panel de control*: administra toda la funcionalidad del panel de control.
- *Administración de detectores*: coordina el acceso a todos los detectores del sistema.
- *Procesamiento de alarmas*: verifica y actúa en todas las condiciones de alarma.

Cada uno de estos componentes de alto nivel tendría que elaborarse en forma iterativa para después posicionarlo dentro de la arquitectura general de *CasaSegura*. Para cada uno se definirían las clases de diseño (con los atributos y operaciones apropiadas). Sin embargo, es importante observar que los detalles de diseño de todos los atributos y operaciones no se especificarían hasta abordar el diseño en el nivel de componentes (véase el capítulo 10).

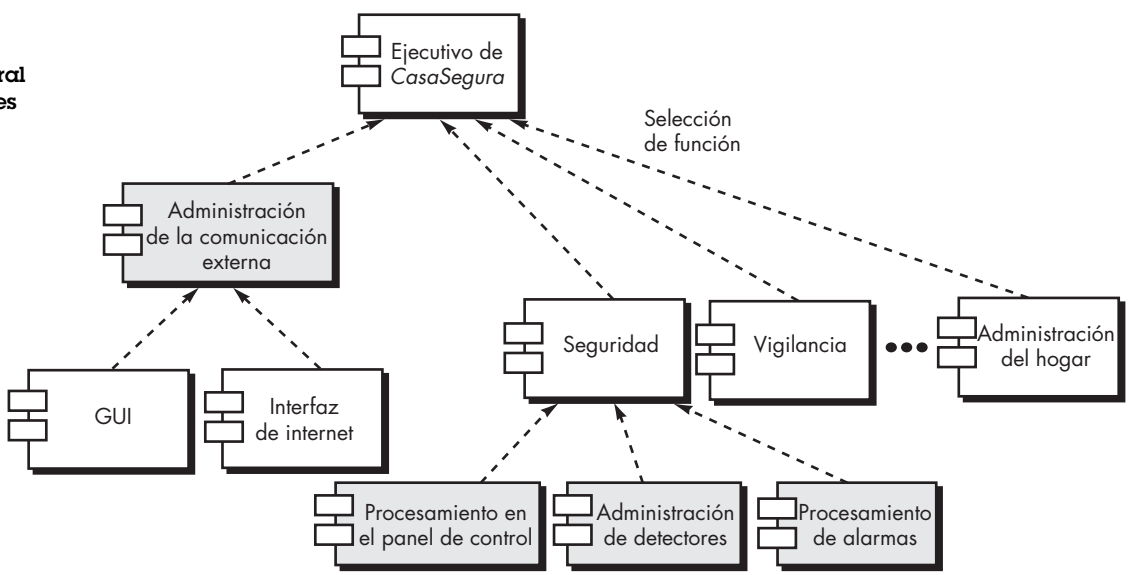
La estructura arquitectónica general (representada como diagrama de componentes UML) se ilustra en la figura 9.8. Las transacciones son adquiridas por *administración de la comunicación externa* a medida que se mueven desde los componentes que procesan la GUI de *CasaSegura* y la interfaz de internet. Esta información es administrada por un componente ejecutivo de *CasaSegura* que selecciona la función apropiada del producto (la seguridad, en este caso). El componente *procesamiento en el panel de control* interactúa con el propietario para activar o desactivar la función de seguridad. El componente *administración de detectores* prueba los sensores para detectar una condición de alarma y el componente *procesamiento de alarmas* produce la salida cuando se detecta una alarma.

### 9.4.4 Descripción de las instancias del sistema

El diseño arquitectónico modelado hasta este momento todavía es de nivel relativamente alto. Se ha representado el contexto del sistema, se definieron los arquetipos que indican las abstracciones importantes dentro del dominio del problema, es visible la estructura general del sistema

**FIGURA 9.8**

**Estructura arquitectónica general para los componentes de alto nivel de *CasaSegura***




www.FreeLibros.me

y están identificadas las componentes principales del software. Sin embargo, es necesario más refinamiento (recuerde que todo el diseño es iterativo).

Para lograr esto, se desarrollan instancias de la arquitectura. Esto significa que la arquitectura se aplica a un problema específico con objeto de demostrar que tanto ella como sus componentes son apropiados.

La figura 9.9 ilustra instancias de la arquitectura de *CasaSegura* para el sistema de seguridad. Los componentes ilustrados en la figura 9.8 se elaboraron para mostrar más detalles. Por ejemplo, el componente *administración de detectores* interactúa con un componente de infraestructura *programador* que implementa la prueba de cada objeto *sensor* usado por el sistema de seguridad. Una elaboración similar se lleva a cabo para cada uno de los componentes representados en la figura 9.8.

**HERRAMIENTAS DE SOFTWARE**

### *Diseño arquitectónico*

**Objetivo:** Las herramientas de diseño arquitectónico modelan la estructura general del software mediante la representación de la interfaz de los componentes, de sus dependencias y relaciones, así como de sus interacciones.

**Mecánica:** La mecánica de las herramientas varía. En la mayoría de casos, la capacidad de diseño de la arquitectura es parte de las funciones provistas por herramientas automatizadas para el modelado del análisis y el diseño.

**Herramientas representativas:**[5]

*Adalon*, desarrollada por Synthis Corp. (**www.synthis.com**), es una herramienta de diseño especializada para diseñar y construir arquitecturas de componentes específicos basados en web.

*ObjectiF*, desarrollada por microTOOL GmbH (**www.microtool.de/objectiF/en/**), es una herramienta de diseño basada en UML que conduce a arquitecturas (como Coldfusion, J2EE, Fusebox, etc.) adecuadas para la ingeniería de software basada en componentes (véase el capítulo 29).

*Rational Rose*, desarrollada por Rational (**www-306.ibm.com/software/rational/**), es una herramienta de diseño basada en UML que proporciona apoyo a todos los aspectos del diseño de la arquitectura.

## 9.5 EVALUACIÓN DE LOS DISEÑOS ALTERNATIVOS PARA LA ARQUITECTURA

En su libro sobre evaluación de las arquitecturas de software, Clements *et al.* [Cle03] afirman lo siguiente:

> Para decirlo sin rodeos, una arquitectura es una apuesta, una adivinanza sobre el éxito de un sistema. ¿No sería agradable saber por adelantado si se apostó a un ganador en vez de esperar hasta que el sistema esté casi terminado para saber si cumplirá con los requerimientos o no? Si usted compra un sistema o paga por su desarrollo, ¿no querría tener alguna seguridad de que va por el camino correcto? Si usted mismo es el arquitecto, ¿no le gustaría tener una buena manera de validar sus intuiciones y experiencia para que pudiera dormir por la noche con la certeza de que la confianza puesta en su diseño está bien fundamentada?

En verdad, las respuestas a estas preguntas tendrían un valor. El diseño da como resultado varias alternativas de arquitectura, cada una de las cuales se evalúa para determinar cuál es la más apropiada para el problema por resolver. En las secciones que siguen se presentan dos enfoques diferentes para la evaluación de diseños arquitectónicos alternativos. El primero utiliza un método iterativo para evaluar negociaciones en el diseño. El segundo aplica una técnica seudocuantitativa para evaluar la calidad del diseño.

5 Las herramientas mencionadas aquí no son obligatorias; sólo son una muestra en esta categoría. En la mayoría de casos, sus nombres son marcas registradas por sus desarrolladores respectivos.

www.FreeLibros.me

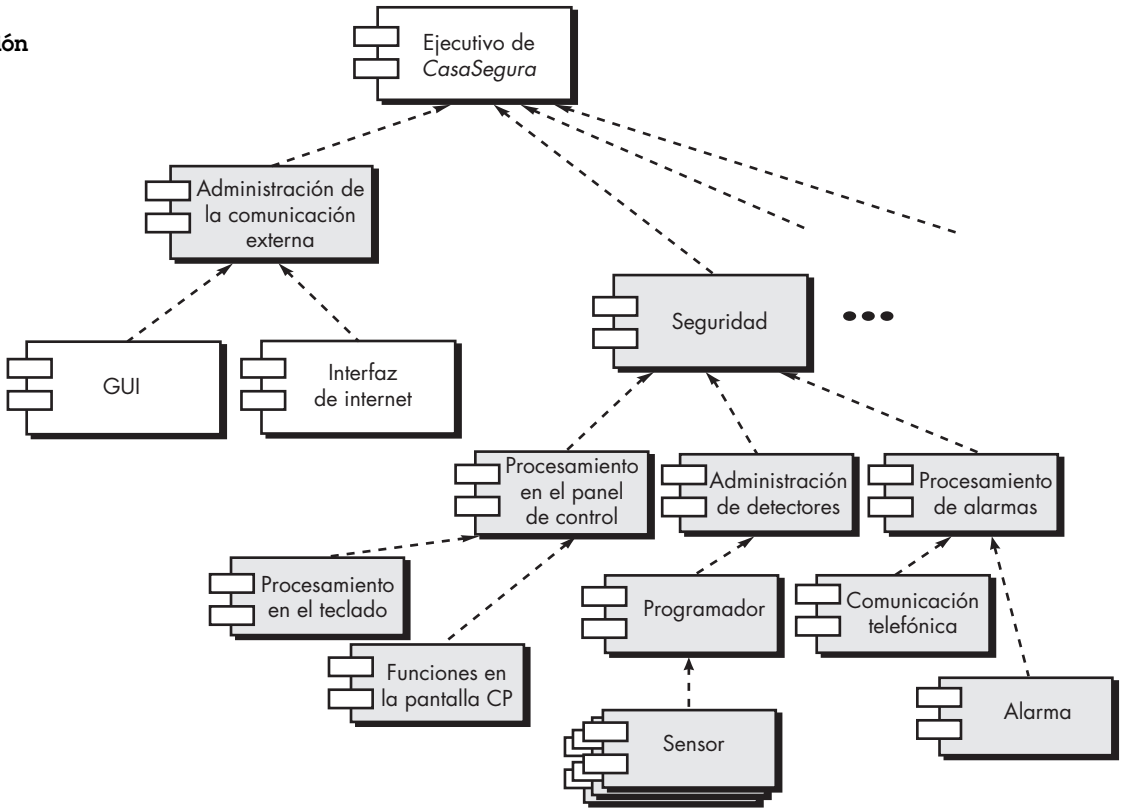



**FIGURA 9.9**
**Instancias de la función de seguridad con la elaboración de componentes**

### 9.5.1 Método de la negociación para analizar la arquitectura

**WebRef**
En la dirección **www.sei.cmu.edu/activities/architecture/ata_method.html**, puede obtenerse mucha información sobre el ATAM.

El Instituto de Ingeniería de Software (SEI, por sus siglas en inglés) desarrolló el *método de la negociación para analizar la arquitectura* (Architecture trade-off analysis method -ATAM) [Kaz98], que establece un proceso de evaluación iterativo para arquitecturas de software. Las actividades de análisis del diseño que se mencionan a continuación se llevan a cabo en forma iterativa:

1. *Escenarios de investigación.* Se desarrolla un conjunto de casos de uso (véanse los capítulos 5 y 6) para representar al sistema desde el punto de vista del usuario.
2. *Obtención de los requerimientos y restricciones, y descripción del ambiente*. Esta información se determina como parte de la ingeniería de requerimientos y se utiliza para estar seguros de que se han detectado todas las preocupaciones de los participantes.
3. *Descripción de los estilos o patrones de arquitectura elegidos para abordar los escenarios y requerimientos.* Debe describirse el estilo arquitectónico con el empleo de las siguientes perspectivas arquitectónicas:
   - *Perspectiva modular* para el análisis de las asignaciones de trabajo con componentes y grado en el que se logra el ocultamiento de información.
   - *Perspectiva del proceso* para el análisis del desempeño del sistema.
   - *Perspectiva del flujo de datos* para analizar el grado en el que la arquitectura satisface los requerimientos funcionales.
4. *Evaluación de los atributos de calidad, considerando cada atributo por separado.* El número de atributos de la calidad elegidos para el análisis es una función del tiempo disponible para la revisión y el grado en el que los atributos de calidad son relevantes para el sistema en cuestión. Los atributos de calidad para evaluar el diseño arquitectónico

www.FreeLibros.me

incluyen confiabilidad, desempeño, seguridad, facilidad de mantenimiento, flexibilidad, facilidad de hacer pruebas, portabilidad, reutilización e interactuación.

5. *Identificación de la sensibilidad de los atributos de calidad de varios atributos arquitectónicos para un estilo de arquitectura específico.* Eso se lleva a cabo haciendo cambios pequeños en la arquitectura a fin de determinar la sensibilidad que tiene un atributo de calidad, por ejemplo, el desempeño ante el cambio. Cualesquiera atributos que se vean afectados en forma significativa por la variación de la arquitectura se denominan *puntos sensibles*.
6. *Crítica de las arquitecturas candidatas (desarrollado en el paso 3) con el uso del análisis de sensibilidad realizado en el paso 5.* El SEI describe este enfoque de la manera siguiente [Kaz98]:

> Una vez determinados los puntos sensibles de la arquitectura, la detección de puntos de negociación es simplemente la identificación de los elementos de la arquitectura a los que atributos múltiples son sensibles. Por ejemplo, el desempeño de una arquitectura cliente-servidor podría ser muy sensible al número de servidores (aumenta el desempeño, en cierto rango, con el incremento del número de servidores) [...] Entonces, el número de servidores es un punto de negociación con respecto de esta arquitectura.

Estos seis pasos representan la primera iteración ATAM. Con base en los resultados de los pasos 5 y 6, se eliminan algunas arquitecturas alternativas o se modifican una o varias de las restantes a fin de representarlas con más detalle para después volver a aplicar el ATAM.[6]

## CasaSegura

### *Evaluación de la arquitectura*

**La escena:** Oficina de Doug Miller, cuando avanza la modelación del diseño arquitectónico.

**Participantes:** Vinod, Jamie y Ed, miembros del equipo de ingeniería de software de *CasaSegura*, y Doug Miller, gerente del grupo de ingeniería de software.

**La conversación:**

**Doug:** Sé que están desarrollando un par de diferentes arquitecturas para el producto *CasaSegura*, y eso es bueno. Mi pregunta es, ¿cómo vamos a elegir la mejor?

**Ed:** Estoy trabajando en un estilo de llamada y regreso, luego alguno de los dos, Jamie o yo, derivará a una arquitectura OO.

**Doug:** Muy bien. ¿Y cómo podemos elegir?

**Jamie:** En mi último año de estudios de ciencias de la computación, tomé un curso de diseño y recuerdo que había varios modos de hacerlo.

**Vinod:** Los hay, pero son algo académicos. Miren, pienso que podemos evaluarlos y escoger el correcto con el uso de casos y escenarios.

**Doug:** ¿No es lo mismo?

**Vinod:** No si hablas de evaluar la arquitectura. Ya tenemos un conjunto completo de casos de uso. Así que los aplicaremos a las dos arquitecturas y veremos cómo reacciona cada sistema y cómo funcionan los componentes y conectores en el contexto del caso de uso.

**Ed:** Buena idea. Nos aseguramos de no dejar nada fuera.

**Vinod:** Es cierto, pero también nos dice si el diseño arquitectónico tiene rizos o si el sistema tiene que volver sobre sí mismo en un lazo para hacer el trabajo.

**Jamie:** Los escenarios no sólo son otro nombre de los casos de uso.

**Vinod:** No, en este caso, un escenario implica algo diferente.

**Doug:** Hablas de un escenario de calidad o de un escenario de cambio, ¿verdad?

**Vinod:** Sí. Lo que hacemos es regresar con los participantes y preguntarles cómo les gustaría que *CasaSegura* cambiara, digamos, en los próximos tres años. Ya sabes, nuevas versiones, características, esa clase de cosas. Construimos un conjunto de escenarios de cambio. También desarrollamos un conjunto de escenarios de calidad que defina los atributos que nos gustaría ver en la arquitectura del software.

**Jamie:** Y los aplicamos a las alternativas.

**Vinod:** Exacto. Elegiremos el estilo que mejor maneje los casos de uso y los escenarios.

6 El *Método de Análisis de la Arquitectura del Software* (MAAS) es una alternativa al ATAM y es de mucha utilidad para los lectores interesados en el análisis arquitectónico. Puede descargarse un artículo sobre el MAAS de la dirección www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html

www.FreeLibros.me

### 9.5.2 Complejidad arquitectónica

Una técnica útil para evaluar la complejidad general de una arquitectura propuesta es considerar las dependencias entre los componentes dentro de la arquitectura. Estas dependencias están motivadas por el flujo de la información o por el control dentro del sistema. Zhao [Zha98] sugiere tres tipos de dependencias:

> Las *dependencias compartidas* representan relaciones entre consumidores que usan los mismos recursos o productores que producen para los mismos consumidores. Por ejemplo, para dos componentes **u** y **v**, si **u** y **v** se refieren a los mismos datos globales, entonces existe una relación de dependencia compartida entre **u** y **v**.
>
> Las *dependencias de flujo* representan relaciones de dependencia entre productores y consumidores de recursos. Por ejemplo, para dos componentes **u** y **v**, si **u** debe completarse para que el control pase a **v** (prerrequisito), o si **u** se comunica con **v** por medio de parámetros, entonces existe una relación de dependencia de flujo entre **u** y **v**.
>
> Las *dependencias de restricción* representan restricciones en el flujo relativo del control entre un conjunto de actividades. Por ejemplo, si dos componentes **u** y **v** no pueden ejecutarse al mismo tiempo (son mutuamente excluyentes), entonces existe una relación de dependencia de restricción entre **u** y **v**.

Las dependencias compartidas y el flujo propuestos por Zhao son similares al concepto de acoplamiento estudiado en el capítulo 8. El acoplamiento es un importante concepto de diseño aplicable en el nivel arquitectónico y de componente. En el capítulo 23 se estudian criterios de medida sencillos para evaluar el acoplamiento.

### 9.5.3 Lenguajes de descripción arquitectónica

El arquitecto de una casa tiene un conjunto de herramientas y notación estandarizadas que permiten que el diseño se represente sin ambigüedades y que sea comprensible. Aunque el arquitecto de software dispone de la notación UML, para un enfoque más formal de la especificación del diseño arquitectónico se necesitan otras formas de diagramas y algunas herramientas relacionadas.

El *lenguaje de descripción arquitectónica* (LDA) provee la semántica y sintaxis para describir una arquitectura de software. Hofmann *et al*. [Hof01] sugieren que un LDA debe brindar al diseñador la capacidad de desintegrar los componentes arquitectónicos, integrar componentes in-

HERRAMIENTAS DE SOFTWARE

#### *Lenguajes de descripción arquitectónica*

El resumen siguiente de varios LDA importantes fue preparado por Richard Land (Lan02) y se publica con el permiso de su autor. Debe observarse que los primeros cinco LDA fueron desarrollados con fines de investigación y no son productos comerciales.

**Rapide (http://poset.stanford.edu/rapide/)** construye a partir del concepto de conjuntos parcialmente ordenados, con lo que genera estructuras de programación muy nuevas (pero aparentemente poderosas).

**UniCon (www.cs.cmu.edu/~UniCon)** es "un lenguaje de descripción arquitectónica que busca ayudar a los diseñadores a definir arquitecturas de software en términos de abstracciones que les parezcan útiles".

**Aesop (www.cs.cmu.edu/~able/aesop/)** aborda el problema de la reutilización del estilo. Con este lenguaje es posible definir estilos y usarlos cuando se construye un sistema real.

**Wright (www.cs.cmu.edu/~able/wright/)** es un lenguaje formal que incluye los elementos siguientes: *componentes* con *puertos*, *conectores* con *roles* y *pegamento* para unir roles con puertos. Los estilos arquitectónicos se formalizan en el lenguaje con predicados, lo que permite revisiones estáticas para determinar la consistencia y completitud de una arquitectura.

**Acme (www.cs.cmu.edu/~acme/)** puede considerarse como un LDA de segunda generación, ya que su objetivo es identificar una clase de mínimo común denominador de los LDA.

**UML (www.uml.org/)** incluye muchos de los artefactos necesarios para hacer descripciones arquitectónicas: procesos, nodos, perspectivas, etc. UML es apropiado para hacer descripciones informales tan sólo porque se trata de un estándar ampliamente comprendido. Sin embargo, carece de toda la fortaleza que se necesita para hacer una descripción arquitectónica adecuada.

www.FreeLibros.me

dividuales en bloques arquitectónicos más grandes y representar las interfaces (mecanismos de conexión) que hay entre los componentes. Una vez establecidas las técnicas descriptivas basadas en lenguaje para el diseño de la arquitectura, es más probable que, a medida que el diseño evoluciona, se obtengan métodos de evaluación eficaces para las arquitecturas.

## 9.6 MAPEO DE LA ARQUITECTURA CON EL USO DEL FLUJO DE DATOS

Los estilos arquitectónicos analizados en la sección 9.3.1 representan arquitecturas radicalmente distintas. Por ello, no sorprende que no exista un mapeo exhaustivo que efectúe la transición del modelo de requerimientos a varios estilos de arquitectura. En realidad, no hay un mapeo práctico para ciertos estilos y el diseñador debe enfocar la traducción de los requerimientos a su diseño con el empleo de las técnicas descritas en la sección 9.4.

Para ilustrar un enfoque al mapeo arquitectónico, considere la arquitectura denominada de llamada y regreso, estructura muy común para muchos tipos de sistemas. La arquitectura de llamada y regreso reside dentro de otras más sofisticadas que ya se analizaron en este capítulo. Por ejemplo, la arquitectura de uno o más componentes de una arquitectura cliente-servidor podría denominarse de llamada y regreso.

Una técnica de mapeo llamada *diseño estructurado* [You79] se caracteriza con frecuencia como método de diseño orientado al flujo porque provee una transición conveniente de un diagrama de flujo de datos (véase el capítulo 7) a la arquitectura del software.[7] La transición del flujo de la información (representada con el diagrama de flujo de datos o DFD) a estructura de programa se consigue como parte de un proceso de seis pasos: 1) se establece el tipo de flujo de información, 2) se indican las fronteras del flujo, 3) se mapea el DFD en la estructura del programa, 4) se define la jerarquía del control, 5) se refina la estructura resultante con el empleo de criterios de medición para el diseño y heurísticos, y 6) se mejora y elabora la descripción arquitectónica.

Como ejemplo breve del mapeo de flujo de datos, se presenta uno de "transformación" paso a paso para una pequeña parte de la función de seguridad *CasaSegura*.[8] Con objeto de realizar el mapeo, debe determinarse el tipo de flujo de la información. Un tipo de flujo de información se llama *flujo de transformación* si presenta una cualidad lineal. Los datos fluyen al sistema con una *trayectoria de flujo de entrada* en la que se transforman de una representación del mundo exterior a una forma internalizada. Una vez internalizados, se procesan en un *centro de transformación*. Por último, salen del sistema por una *trayectoria de flujo de salida* que transforma los datos a una forma del mundo exterior.[9]

### 9.6.1 Mapeo de transformación

El mapeo de transformación es un conjunto de pasos de diseño que permite mapear un DFD con características de flujo de transformación en un estilo arquitectónico específico. Para ilustrar este enfoque, de nuevo consideraremos la función de seguridad de *CasaSegura*.[10] Un elemento del modelo de análisis es un conjunto de diagramas de flujo de datos que describen el flujo de información dentro de la función de seguridad. Para mapear estos diagramas de flujo de datos en una arquitectura de software deben darse los siguientes pasos de diseño:

---

7 Debe observarse que durante el método de mapeo también se utilizan otros elementos del modelo de requerimientos.

8 En el sitio web del libro se presenta un análisis más detallado del diseño estructurado.

9 En este ejemplo no se considera otro tipo importante de flujo de información, pero se aborda en el ejemplo de diseño estructurado que se presenta en el sitio web del libro.

10 Sólo se considera la parte de la función de seguridad de *CasaSegura* que utiliza el panel de control. Aquí no se incluyen otras características analizadas en el libro.

www.FreeLibros.me

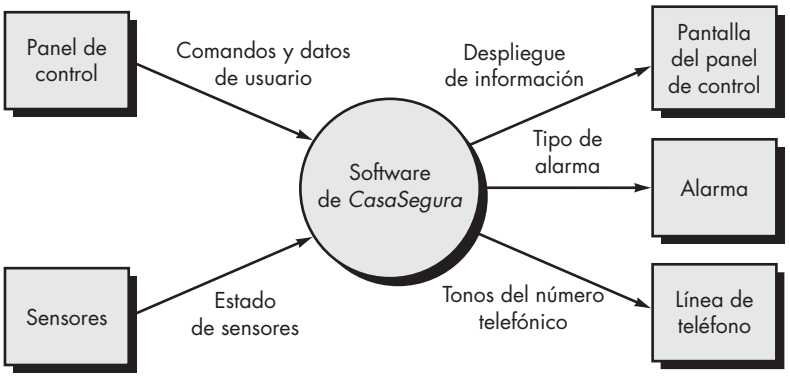


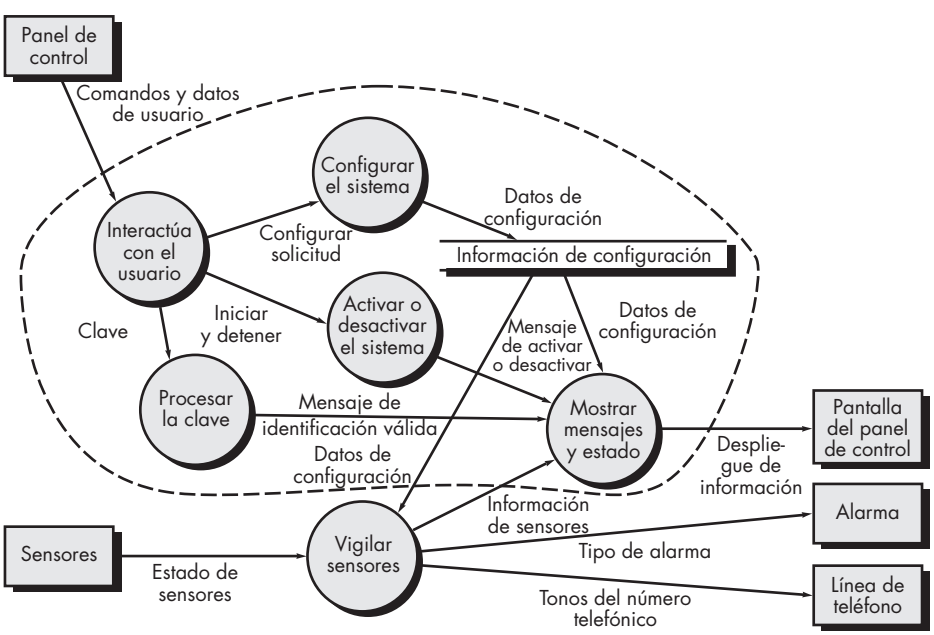

**FIGURA 9.10**
**Diagrama de flujo de datos para la función de seguridad de *CasaSegura***

**Paso 1. Revisión del modelo del sistema fundamental.** El modelo del sistema fundamental o diagrama de contexto ilustra la función de seguridad como una transformación única, y representa a los productores y consumidores externos de los datos que fluyen hacia dentro y fuera de la función. La figura 9.10 ilustra un modelo de contexto de nivel 0, y la figura 9.11 muestra el flujo de datos refinado para la función de seguridad.


*Si en este momento se mejora más el diagrama de flujo de datos, trate de obtener burbujas que presenten mucha cohesión.*

**Paso 2. Revisar y mejorar los diagramas de flujos de datos para el software.** La información obtenida del modelo de requerimientos se refina para producir más detalles. Por ejemplo, se estudia el DFD de nivel 2 para *vigilar sensores* (véase la figura 9.12) y se obtiene el diagrama de flujo de datos de nivel 3 que se presenta en la figura 9.13. En el nivel 3, cada transformación en el diagrama de flujo de datos presenta una cohesión relativamente grande (consulte el capítulo 8). Es decir, el proceso implícito por una transformación realiza una sola y distintiva función que se implementa como componente en el software de *CasaSegura*. Entonces, el DFD de la figura 9.13 contiene detalles suficientes para hacer un "primer corte" en el diseño de la arquitectura del subsistema *vigilar sensores*, y se continúa con más refinamiento.

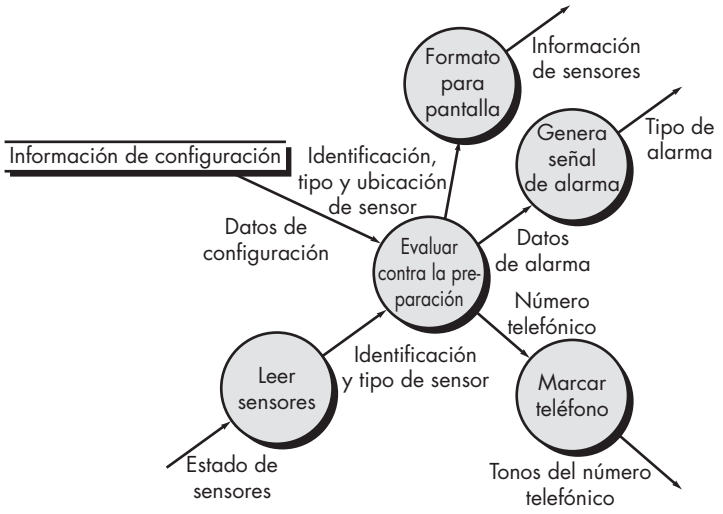
**FIGURA 9.11**
**Diagrama de flujo de datos de nivel 1 para la función de seguridad de *CasaSegura***




www.FreeLibros.me

**FIGURA 9.12**

**Diagrama de flujo de datos de nivel 2 que mejora la transformación *vigilar sensores***




**PUNTO CLAVE**

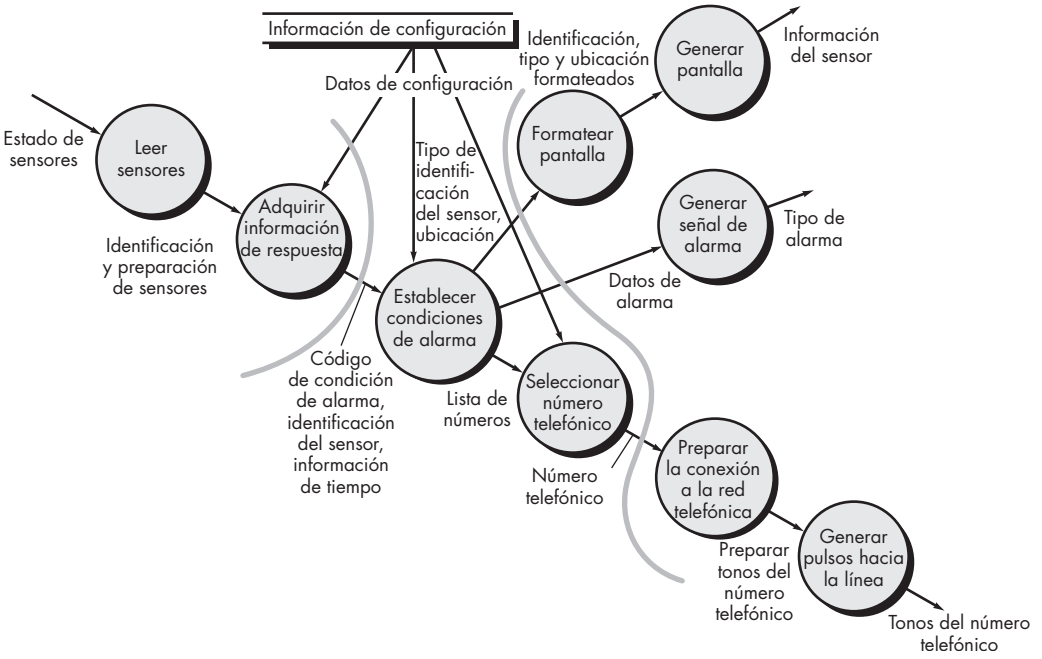
Será frecuente encontrar varios tipos de flujo de datos dentro del mismo modelo orientado al flujo. Los flujos se dividen y la estructura del programa se obtiene con el uso del mapeo apropiado.

**Paso 3. Determinar si el DFD tiene características de flujo de transformación o de transacción.**[11] Al evaluar el DFD (véase la figura 9.13) se observa que los datos entran al software por una trayectoria de ingreso y lo abandonan por tres trayectorias de salida. Por tanto, se adoptará una característica general de transformación para el flujo de la información.

**Paso 4. Aísle el centro de transformación, especificando las fronteras de entrada y salida del flujo.** Los datos de entrada fluyen por una trayectoria en la que la información pasa de su forma externa a su forma interna; el flujo de salida convierte los datos internalizados a su forma externa. Las fronteras de los flujos de entrada y salida quedan abiertas a la interpretación.

**FIGURA 9.13**

**Diagrama de flujo de datos de nivel 3 para *vigilar sensores* con fronteras del flujo**




11 En el flujo de transacción, un solo concepto de datos, llamado *transacción*, ocasiona que el flujo de datos se ramifique a través de cierto número de trayectorias definidas por la naturaleza de la transacción.

www.FreeLibros.me

CONSEJO

*En un esfuerzo por explorar estructuras alternativas para el programa, varíe las fronteras del flujo. Esto toma poco tiempo y da una perspectiva importante.*

Es decir, diferentes diseñadores tal vez seleccionen como ubicación de la frontera diferentes puntos en el flujo. De hecho, es posible obtener soluciones alternativas al diseño si se varía la colocación de las fronteras del flujo. Aunque debe tenerse cuidado al seleccionar las fronteras, la variación de una burbuja a lo largo de la trayectoria del flujo tendrá por lo general poco efecto en la estructura final del programa.

Para el ejemplo, en la figura 9.13 se ilustran las fronteras del flujo como curvas sombreadas que corren de arriba abajo a través del flujo. Las transformaciones (burbujas) que constituyen el centro de transformación quedan dentro de esas dos fronteras sombreadas. Es posible dar argumentos para reajustar una frontera (por ejemplo, podría proponerse una frontera para el flujo de entrada que separara *leer sensores* de *adquirir información de respuesta*). En este diseño, el énfasis en esta etapa de diseño debe estar en la selección de fronteras razonables y no en la iteración extensa en la colocación de las divisiones.

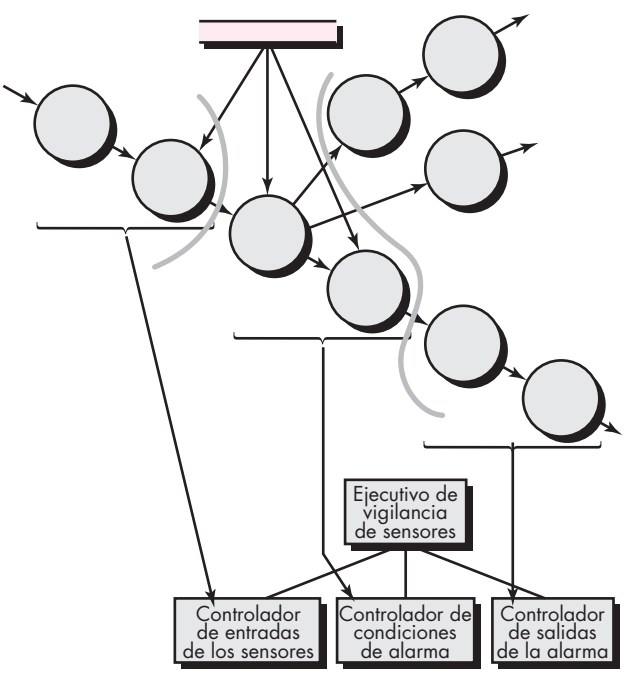
**Paso 5. Realizar el "rediseño de primer nivel".** La arquitectura del programa obtenida con este mapeo da como resultado una distribución del control de arriba abajo. El *rediseño* lleva a una estructura de programa en la que los componentes de alto nivel ejecutan la toma de decisiones y los de bajo nivel realizan la mayor parte del trabajo de entrada, computación y salida. Los componentes de nivel medio llevan a cabo cierto control y moderan las cantidades de trabajo.

Cuando se encuentra una transformación, se mapea un diagrama de flujo de datos para una estructura específica (una de llamar y regresar) que provea control para el procesamiento de información de entrada, transformación y salida. En la figura 9.14 se ilustra este rediseño de primer nivel para el subsistema *vigilar sensores*. Un controlador principal (llamado *ejecutivo de vigilancia de sensores*) reside en la parte superior de la estructura del programa y coordina las siguientes funciones de control subordinadas:

- Un controlador del procesamiento de la información de entrada, llamado *controlador de entradas de los sensores*, coordina la recepción de todos los datos que llegan.
- Un controlador del flujo de transformación, llamado *controlador de condiciones de la alarma*, supervisa todas las operaciones sobre los datos en forma internalizada (por ejemplo, un módulo que invoque varios procedimientos de transformación de datos.

**FIGURA 9.14**

**Rediseño de primer nivel para *vigilar sensores***




www.FreeLibros.me

CONSEJO

*En esta etapa no se debe ser dogmático. Tal vez sea necesario establecer dos o más controladores para el procesamiento de las entradas o la computación, con base en la complejidad del sistema que se va a elaborar. Si el sentido común sugiere este enfoque, ¡adelante!*

- Un controlador de procesamiento de información de salida, llamado *controlador de salidas de la alarma*, coordina la producción de información de salida.

Aunque la figura 9.14 sugiere una estructura de tres dientes, los flujos complejos que hay en sistemas grandes tal vez requieran dos o más módulos de control para cada una de las funciones de control generales descritas con anterioridad. El número de módulos en el primer nivel debe limitarse al mínimo necesario para ejecutar las funciones de control y mantener buenas características de independencia de funciones.

**Paso 6. Realizar "rediseño de segundo nivel".** El rediseño de segundo nivel se logra con el mapeo de transformaciones individuales (burbujas) de un diagrama de flujo de datos en módulos apropiados dentro de la arquitectura. Se comienza en la frontera del centro de transformación y se avanza hacia afuera a lo largo de las trayectorias de entrada y salida; las transformaciones se mapean en niveles subordinados de la estructura del software. En la figura 9.15 se presenta el enfoque general del rediseño de segundo nivel.

CONSEJO

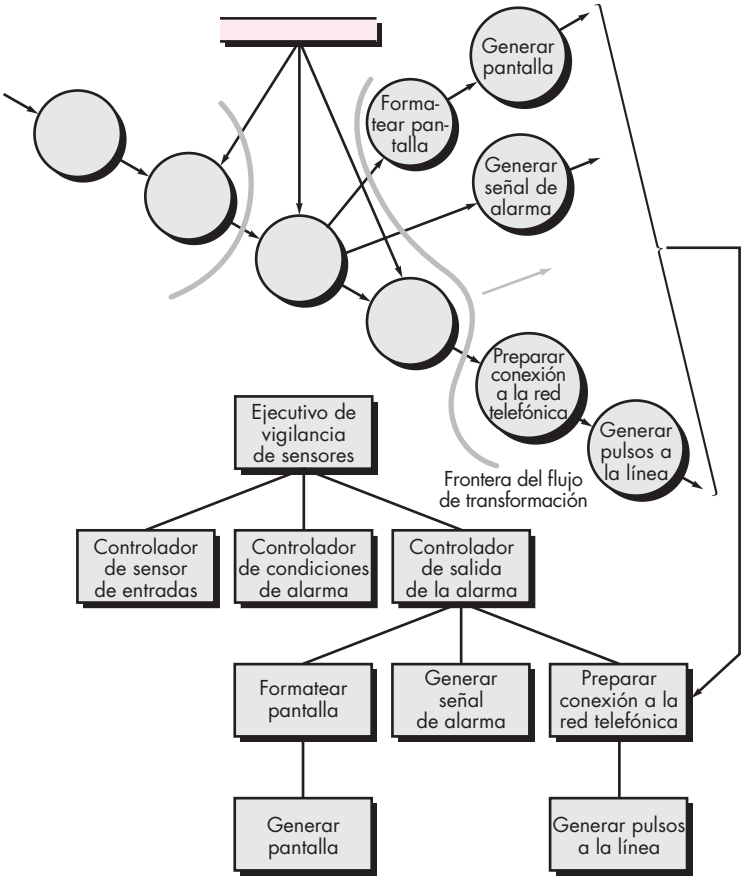
*Hay que eliminar los módulos de control redundantes. Es decir, si un módulo de control no hace nada más que controlar a otro módulo, su función controladora debe llevarse a un módulo de nivel más alto.*

Aunque la figura 9.15 ilustra un mapeo uno a uno entre las transformaciones del diagrama de flujo y los módulos de software, es frecuente que haya distintos mapeos. Es posible combinar y representar dos o incluso tres burbujas como un solo componente, o una sola burbuja puede expandirse a dos o más componentes. Son consideraciones prácticas y mediciones de calidad del diseño las que dictan el resultado del rediseño de segundo nivel. La revisión y refinamiento tal vez produzcan cambios en esta estructura, pero sirven como diseño de "primera iteración".

El rediseño de segundo nivel para el flujo de entrada se presenta de la misma manera. El rediseño se logra de nuevo avanzando hacia afuera a partir de la frontera del centro de trans-

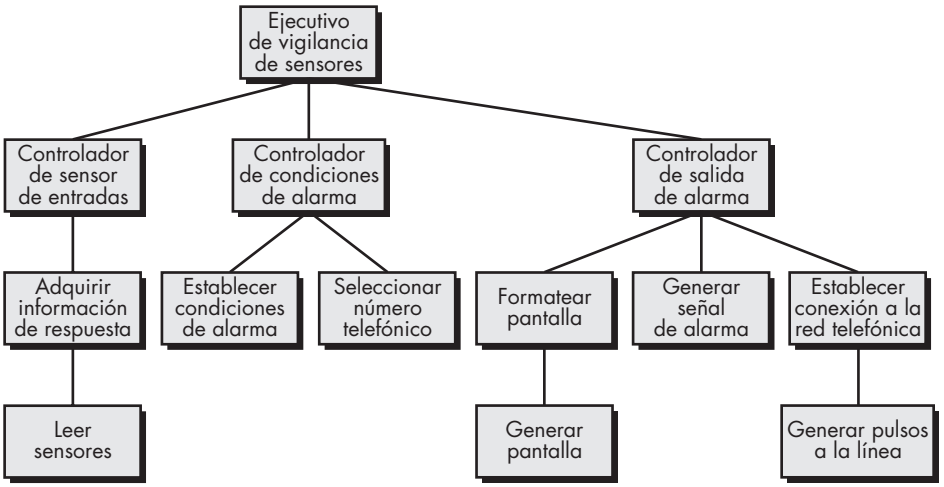
**FIGURA 9.15**

**Rediseño de segundo nivel para *vigilar sensores***




www.FreeLibros.me

**FIGURA 9.16**
**Estructura de primera iteración para *vigilar sensores***




**CONSEJO**

*Conserve módulos "trabajadores" en un nivel bajo de la estructura del programa. Esto llevará a una arquitectura fácil de mantener.*

formación en el lado del flujo de entrada. El centro de transformación del software del subsistema *vigilar sensores* se mapea de modo un poco distinto. Cada conversión de datos o transformación de cálculo de la parte de transformación del diagrama de flujo de datos se mapea en un módulo subordinado al controlador de la transformación. En la figura 9.16 se presenta la arquitectura completa de primera iteración.

Los componentes así mapeados que se aprecian en la figura 9.16 representan un diseño inicial de la arquitectura del software. Aunque los componentes reciben su nombre de manera que se implique la función, para cada uno debe escribirse una narración breve (adaptada de la especificación del proceso desarrollada para la transformación de datos y que se generó durante el modelado de los requerimientos). La narración describe la interfaz del componente, las estructuras internas de los datos, un relato de las funciones y el análisis breve de las restricciones y otros rasgos especiales (como los archivos de entrada y salida, características que dependen del hardware, requerimientos especiales de tiempo, etcétera).

**Cita:**

"Hacerlo tan sencillo como sea posible. Pero no más."

**Albert Einstein**

**Paso 7. Refinar la arquitectura de primera iteración con el empleo de heurísticos de diseño para mejorar la calidad del software.** Siempre es posible refinar la arquitectura de primera iteración, aplicando conceptos de independencia de funciones (véase el capítulo 8). Los componentes hacen explosión o implosión para producir un rediseño sensible, la separación de problemas, buena cohesión, acoplamiento mínimo y, lo más importante, una estructura que puede implementarse sin dificultad, probar sin confusión y mantener sin grandes problemas.

Los refinamientos son impuestos por el análisis y los métodos de evaluación descritos en la sección 9.5, así como por consideraciones prácticas y el sentido común. Por ejemplo, hay ocasiones en las que el controlador para el flujo de datos de entrada es totalmente innecesario, cuando se requiere algún procesamiento de las entradas en un componente subordinado al controlador de la transformación, cuando no es posible evitar mucho acoplamiento debido a datos globales o cuando no se logran características estructurales óptimas. El arbitraje final lo constituyen los requerimientos del software acoplados con el criterio humano.

El objetivo de los siete pasos anteriores es desarrollar una representación arquitectónica del software. Es decir, una vez definida la estructura, se evalúa y mejora considerándola como un todo. Las modificaciones que se hacen en este momento exigen poco trabajo adicional, pero tienen un efecto profundo en la calidad del software.

Debe hacerse una pausa y considerar la diferencia entre el enfoque de diseño descrito y el proceso de "escribir programas". Si el código es la única representación del software, usted y

www.FreeLibros.me

**CasaSegura**

### *Refinación de la arquitectura de primer corte*

**La escena:** El cubículo de Jamie, cuando comienza la modelación del diseño.

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

**La conversación:**

[Ed acaba de terminar el diseño de primer corte del subsistema de vigilancia de sensores. Se detiene para solicitar la opinión de Jamie.]

**Ed:** Pues bien, aquí está la arquitectura que obtuve.

[Ed muestra a Jamie la figura 9.16, y ella la estudia unos momentos.]

**Jamie:** Está muy bien, pero creo que podemos hacer algo para que sea más sencilla… y mejor.

**Ed:** ¿Como qué?

**Jamie:** Bueno, ¿por qué usaste el componente *controlador de sensores de entrada*?

**Ed:** Porque se necesita un controlador para el mapeo.

**Jamie:** No en realidad. El controlador no hace gran cosa, ya que estamos manejando una sola trayectoria de flujo para los datos de entrada. Puede eliminarse el controlador sin que pase nada.

**Ed:** Puedo vivir con eso. Lo cambiaré y…

**Jamie (sonríe):** Espera… También podemos hacer la implosión de los componentes *establecer condiciones de alarma* y *seleccionar número telefónico*. El controlador de la transformación que presentas en realidad no es necesario y la poca disminución en la cohesión es tolerable.

**Ed:** Simplificación, ¿eh?

**Jamie:** Sí. Y al hacer refinamientos sería buena idea hacer la implosión de los componentes *formatear pantalla* y *generar pantalla*. El formateo de la pantalla para el panel de control es algo sencillo. Puede definirse un nuevo módulo llamado *producir pantalla*.

**Ed (dibuja):** Entonces, ¿esto es lo que piensas que debemos hacer?

[Muestra a Jamie la figura 9.17.]

**Jamie:** Es un buen comienzo.

sus colegas tendrán grandes dificultades para evaluarlo o mejorarlo en un nivel global u holístico y, en verdad, tendrán dificultades porque "los árboles no los dejarán ver el bosque".

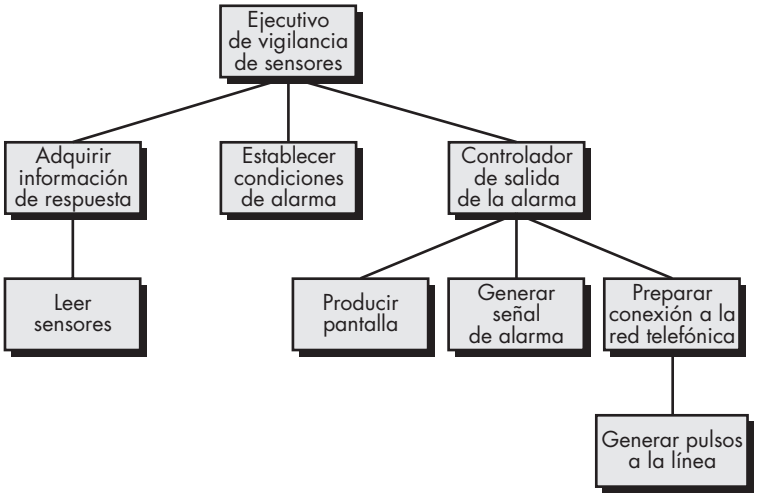
### 9.6.2 Refinamiento del diseño arquitectónico

**? ¿Qué pasa después de que se generó la arquitectura?**

Cualquier análisis del refinamiento del diseño debería ir precedido de este comentario: "Recuerde que un 'diseño óptimo' que no funcione tiene un mérito cuestionable." Debe ocuparse de desarrollar una representación del software que satisfaga todos los requerimientos funcionales y de desempeño, y darle mérito según sus mediciones y heurísticos de diseño.

Debe estimularse el refinamiento de la arquitectura del software durante las primeras etapas del diseño. Como ya se dijo en este capítulo, hay estilos alternativos para la arquitectura que es posible obtener, refinar y evaluar en busca del "mejor" enfoque. Éste, dirigido a la optimización,

**Figura 9.17**
**Estructura refinada del programa para *vigilar sensores***




www.FreeLibros.me

es uno de los verdaderos beneficios que se logran con el desarrollo de una representación de la arquitectura del software.

Es importante observar que la sencillez estructural con frecuencia refleja tanto elegancia como eficiencia. El refinamiento del diseño debe perseguir el menor número de componentes que sea consistente con la modularidad efectiva y la estructura de datos menos compleja que cumpla de modo adecuado con los requerimientos de información.

## 9.7 Resumen

La arquitectura del software proporciona una visión holística del sistema que se va a construir. Ilustra la estructura y organización de los componentes del software, sus propiedades y conexiones. Los componentes del software incluyen módulos de programa y las distintas representaciones de datos que manipula éste. Por tanto, el diseño de los datos es parte integral de la generación de la arquitectura del software. Ésta subraya las primeras decisiones respecto del diseño y provee un mecanismo para considerar los beneficios de las estructuras alternativas para el sistema.

Dentro de un género arquitectónico dado, hay varios estilos y patrones diferentes disponibles para el ingeniero de software. Cada estilo describe una categoría de sistemas que agrupa un conjunto de componentes que realizan una función requerida por el sistema; un grupo de conectores que permiten comunicación, coordinación y cooperación entre los componentes; restricciones que definen cómo pueden integrarse éstos para formar el sistema y modelos semánticos que permiten que un diseñador entienda las propiedades generales del sistema.

En un sentido general, el diseño arquitectónico se obtiene con el empleo de cuatro pasos. En primer lugar, el sistema debe representarse en contexto. Es decir, el diseñador debe definir las entidades externas con las que interactúa el software y la naturaleza de la interacción. Una vez especificado el contexto, el diseñador debe identificar un conjunto de abstracciones de alto nivel, llamadas *arquetipos*, que representan elementos fundamentales del comportamiento o función del sistema. Ya que se definieron las abstracciones, el diseño comienza a avanzar cerca del dominio de la implementación. Se identifican los componentes y se representan dentro del contexto de una arquitectura que les da apoyo. Por último, se desarrollan instancias específicas de la arquitectura para "probar" el diseño en el contexto del mundo real.

Como ejemplo sencillo del diseño arquitectónico, el método del mapeo presentado en este capítulo usa las características del flujo de datos para obtener un estilo arquitectónico de uso muy común. El diagrama de flujo de datos se mapea en la estructura del programa con el uso del enfoque del mapeo de la transformación. Éste se aplica a un flujo de información que presente fronteras distintas entre los datos de entrada y los de salida. El diagrama de flujo de datos se mapea en una estructura que asigna el control a la entrada, al procesamiento y a la salida junto con tres jerarquías de módulos diseñados por separado. Una vez que se tiene la arquitectura, se elabora y analiza mediante criterios de calidad.

## Problemas y puntos por evaluar

**9.1.** Con el uso de la arquitectura de una casa o edificio como metáfora, establezca comparaciones con la arquitectura del software. ¿En qué se parecen las disciplinas de la arquitectura clásica y la del software? ¿En qué difieren?

**9.2.** Diga dos o tres ejemplos de aplicaciones para cada uno de los estilos arquitectónicos mencionados en la sección 9.3.1.

**9.3.** Algunos de los estilos arquitectónicos citados en la sección 9.3.1 tienen naturaleza jerárquica, mientras que otros no. Elabore una lista de cada tipo. ¿Cómo se implementarían los estilos arquitectónicos que no son jerárquicos?

www.FreeLibros.me

**9.4.** Los términos *estilo arquitectónico*, *patrón arquitectónico* y *marco* (que no se estudia en este libro) surgen con frecuencia en los análisis de la arquitectura del software. Investigue y describa en qué difiere cada uno de ellos de los demás.

**9.5.** Seleccione una aplicación con la que esté familiarizado. Responda cada una de las preguntas planteadas para el control y los datos de la sección 9.3.3.

**9.6.** Investige el ATAM (en [Kaz98]) y presente un análisis detallado de los seis pasos presentados en la sección 9.5.1.

**9.7.** Si no lo ha hecho, termine el problema 6.6. Use los métodos de diseño descritos en este capítulo para desarrollar una arquitectura del software para el SSRB.

**9.8.** Utilice un diagrama de flujo y una narración del procesamiento para describir un sistema basado en computadora que tenga distintas características de transformación del flujo. Defina las fronteras del sistema y mapee el diagrama de flujo de los datos en una arquitectura del software con el empleo de la técnica descrita en la sección 9.6.1.

## Lecturas adicionales y fuentes de información

La bibliografía sobre arquitectura del software ha crecido mucho en la última década. Los libros escritos por Gorton (*Essential Software Architecture*, Springer, 2006), Reekie y McAdam (*A Software Architecture Primer*, Angophora Press, 2006), Albin (*The Art of Software Architecture*, Wiley, 2003) y Bass *et al.* (*Software Architecture in Practice*, 2a. ed., Addison-Wesley, 2002), presentan introducciones útiles a un área del conocimiento con muchos retos intelectuales.

Buschman *et al.* (*Pattern-Oriented Software Architecture*, Wiley, 2007) y Kuchana (*Software Architecture Design Patterns in Java*, Auerbach, 2004) estudian aspectos orientados al patrón del diseño arquitectónico. Rozanski y Woods (*Software Systems Architecture*, Addison-Wesley, 2005), Fowler (*Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003), Clements *et al.* (*Documenting Software Architecture: View and Beyond*, Addison-Wesley, 2002), Bosch [Bos00] y Hofmeister *et al.* [Hof00] hacen análisis profundos de la arquitectura del software.

Hennesey y Patterson (*Computer Architecture*, 4a. ed., Morgan-Kaufmann, 2007) adoptan un punto de vista notable, por ser cuantitativo, para los aspectos del diseño de la arquitectura del software. Clements *et al.* (*Evaluating Software Architectures*, Addison-Wesley, 2002) analizan los aspectos asociados con la evaluación de alternativas arquitectónicas y la selección de la mejor arquitectura para un dominio dado de problemas.

Los libros dedicados a la implementación sobre la arquitectura abordan el diseño arquitectónico dentro de un ambiente o tecnología específicos de desarrollo. Marks y Bell (*Sevice-Oriented Architecture*, Wiley, 2006) estudian el enfoque de diseño que relaciona los negocios y los recursos computacionales con los requerimientos definidos por los clientes. Stahl *et al*. (*Model-Driven Software Development*, Wiley, 2006) analizan la arquitectura dentro del contexto de los enfoques de modelado dirigidos al dominio. Radaideh y Al-Ameed (*Architecture of Reliable Web Applications Software*, GI Global, 2007) consideran arquitecturas apropiadas para *webapps*. Clements y Norhrop (*Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001) estudian el diseño de arquitecturas que dan apoyo a líneas de productos de software. Shanley (*Protected Mode Software Architecture*, Addison-Wesley, 1996) proporciona una guía del diseño arquitectónico para cualquier persona que diseñe sistemas basados en PC que operen en tiempo real, o para sistemas operativos de tareas múltiples o manejadores de dispositivos.

La investigación actual sobre arquitectura del software se documenta cada año en *Proceedings of the International Workshop on Software Architecture*, publicación patrocinada por ACM y otras organizaciones de cómputo, y en *Proceedings of the International Conference on Software Engineering*.

En internet se encuentra una amplia variedad de fuentes de información sobre el diseño arquitectónico. En el sitio web del libro, **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, se halla una lista actualizada de referencias que hay en la red mundial, relevantes para el diseño de la arquitectura.

www.FreeLibros.me

CAPÍTULO

# 10 DISEÑO EN EL NIVEL DE COMPONENTES

CONCEPTOS CLAVE



El diseño en el nivel de componentes tiene lugar una vez terminado el diseño de la arquitectura. En esta etapa se ha establecido la estructura general de los datos y del programa del software. El objetivo es traducir el modelo del diseño a software operativo. Pero el nivel de abstracción del modelo de diseño existente es relativamente alto y el del programa operativo es bajo. La traducción es difícil y está abierta a la introducción de errores sutiles que son difíciles de detectar y de corregir en las etapas posteriores del proceso del software. En una conferencia famosa, Edsgar Dijkstra, investigador importante que ha contribuido mucho a nuestra comprensión del diseño de software, dijo [Dij72]:

> El software parece ser diferente de muchos otros productos en los que la regla es que a mejor calidad se da un mayor precio. Aquellos que desean un software en verdad confiable descubrirán que deben encontrar un medio para evitar de inicio la mayoría de los posibles errores; como resultado, el proceso de programación se hace más barato [...] los programadores eficaces no tienen que perder su tiempo depurando errores: no deben introducirlos al arrancar.

Aunque estas palabras fueron expresadas hace muchos años, siguen siendo ciertas. Al traducir el modelo del diseño a código fuente, deben seguirse principios de diseño que no sólo hagan la traducción sino que también eviten la "introducción de errores desde el principio".

Es posible representar el diseño en el nivel del componente con el uso de un lenguaje de programación. En esencia, el programa se crea con el empleo del modelo de diseño arquitectónico como guía. Un enfoque alternativo consiste en representar el diseño en el nivel de los componentes con alguna representación intermedia (gráfica, tabular o basada en texto) que se traduzca con facilidad a código fuente. Sin que importe el mecanismo utilizado para representar

UNA MIRADA RÁPIDA

**¿Qué es?** Durante el diseño arquitectónico, se define un conjunto completo de componentes de software. Pero las estructuras internas de datos y detalles de procesamiento de cada componente no están representadas en un nivel de abstracción cercano al código. El diseño en el nivel de componentes define las estructuras de datos, algoritmos, características de la interfaz y mecanismos de comunicación asignados a cada componente del software.

**¿Quién lo hace?** Un ingeniero de software es quien realiza el diseño en el nivel de componentes.

**¿Por qué es importante?** Antes de elaborar el software, se tiene que ser capaz de determinar si funcionará. El diseño en el nivel de componentes lo representa en forma tal que permite revisar los detalles del diseño para garantizar su corrección y su consistencia con otras representaciones del diseño (por ejemplo, los datos y el diseño de la arquitectura y la interfaz). Esto proporciona un medio para evaluar si funcionarán las estructuras de datos, interfaces y algoritmos.

**¿Cuáles son los pasos?** Las representaciones de diseño de datos, arquitectura e interfaces constituyen el fundamento para el diseño en el nivel de componentes. La definición de clase o narrativa de procesamiento de cada componente se traduce a un diseño detallado que utiliza formas de diagrama o basadas en texto que especifican las estructuras de datos internas, los detalles de la interfaz local y la lógica del procesamiento. La notación del diseño incluye diagramas UML y formatos complementarios. Se especifica el diseño del procedimiento con el empleo de construcciones de programación estructurada. Con frecuencia es posible obtener componentes de software reutilizable ya existentes, en lugar de construir nuevos.

**¿Cuál es el producto final?** El producto principal que se genera en esta etapa es el diseño de cada componente, representado con notación gráfica, tabular o basada en texto.

**¿Cómo me aseguro de que lo hice bien?** Se efectúa la revisión del diseño. Esto se hace para determinar durante las primeras etapas de diseño si las estructuras de datos, interfaces, secuencias de procesamiento y condiciones lógicas son correctas y si producirán los datos apropiados o la transformación del control asignado al componente.



www.FreeLibros.me

a éste, las estructuras de datos, interfaces y algoritmos definidos deben apegarse a varios lineamientos de diseño bien establecidos que ayudan a evitar los errores conforme evoluciona el diseño del procedimiento. En este capítulo se estudian estos lineamientos y los métodos disponibles para cumplirlos.

## 10.1 ¿Qué es un componente?

**Cita:**

"Los detalles no son detalles. Son el diseño."

**Charles Eames**

Un *componente* es un bloque de construcción de software de cómputo. Con más formalidad, la *Especificación OMG del Lenguaje de Modelado Unificado* [OMG03a] define un componente como "una parte modular, desplegable y sustituible de un sistema, que incluye la implantación y expone un conjunto de interfaces".

Como se dijo en el capítulo 9, los componentes forman la arquitectura del software y, en consecuencia, juegan un papel en el logro de los objetivos y de los requerimientos del sistema que se va a construir. Como los componentes se encuentran en la arquitectura del software, deben comunicarse y colaborar con otros componentes y con entidades (otros sistemas, dispositivos, personas, etc.) que existen fuera de las fronteras del software.

El verdadero significado del término *componente* difiere en función del punto de vista del ingeniero de software que lo use. En las secciones que siguen, se estudian tres visiones importantes de lo que es un componente y cómo se emplea en el desarrollo de la modelación del diseño.

### 10.1.1 Una visión orientada a objetos

**PUNTO CLAVE**

Desde un punto de vista orientado a objetos, un componente es un conjunto de clases que colaboran.

En el contexto de la ingeniería de software orientada a objetos, un componente contiene un conjunto de clases que colaboran.[1] Cada clase dentro de un componente se elabora por completo para que incluya todos los atributos y operaciones relevantes para su implantación. Como parte de la elaboración del diseño, también deben definirse todas las interfaces que permiten que las clases se comuniquen y colaboren con otras clases de diseño. Para lograr esto, se comienza con el modelo de requerimientos y se elaboran clases de análisis (para los componentes que se relacionan con el dominio del problema) y clases de infraestructura (para los componentes que dan servicios de apoyo para el dominio del problema).

Para ilustrar el proceso de la elaboración del diseño, considere el software que se va a elaborar para un taller de impresión avanzada. El objetivo general del software es obtener los requerimientos que plantea el cliente en el mostrador, presupuestar un trabajo de impresión y después pasar éste a una instalación automatizada de producción. Durante la ingeniería de requerimientos se obtuvo una clase de análisis llamada **ImprimirTrabajo**. En la parte superior de la figura 10.1 aparecen los atributos y operaciones definidos durante el análisis. En el diseño de la arquitectura se definió **ImprimirTrabajo** como un componente dentro de la arquitectura del software y está representado con la notación abreviada UML[2] que se muestra en la parte media derecha de la figura. Observe que **ImprimirTrabajo** tiene dos interfaces, *CalcularTrabajo*, que provee la capacidad de obtener el costo del trabajo, e *IniciarTrabajo*, que pasa el trabajo a través de las instalaciones de producción. Éstas se encuentran representadas con los símbolos de "paleta" que aparecen en el lado izquierdo de la caja del componente.

**CONSEJO**

*Recuerde que el modelado del análisis y del diseño son acciones iterativas. Es probable que la elaboración de la clase de análisis original requiera de etapas adicionales, que con frecuencia van seguidas de etapas de modelado del diseño que representan la clase de diseño elaborada (los detalles del componente).*

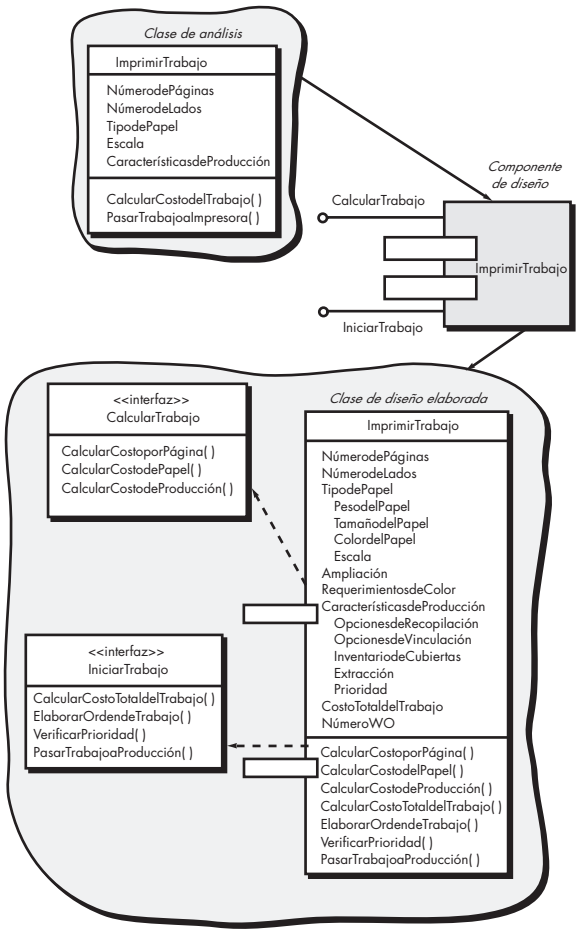
El diseño en el nivel del componente comienza en este punto. Deben elaborarse los detalles del componente **ImprimirTrabajo** para que den información suficiente que guíe la implantación. La clase de análisis original se lleva a cabo para dar cuerpo a todos los atributos y operaciones requeridos para implantar la clase así como el componente **ImprimirTrabajo**. En la parte inferior derecha de la figura 10.1, la clase de diseño elaborada **ImprimirTrabajo** contiene

1 En ciertos casos, un componente contiene una sola clase.

2 Los lectores que no estén familiarizados con la notación UML deben consultar el apéndice 1.

www.FreeLibros.me

**FIGURA 10.1**
**Elaboración de un componente de diseño**




información más detallada de los atributos, así como una descripción amplia de las operaciones que se necesitan para implantar el componente. Las interfaces *CalcularTrabajo* e *IniciarTrabajo* implican comunicación y colaboración con otros componentes (que no se muestran). Por ejemplo, la operación *CalcularCostoporPágina ( )* (que forma parte de la interfaz *CalcularTrabajo*) podría colaborar con un componente llamado **TabladeValuación** que contuviera información sobre los precios del trabajo. La operación *VerificarPrioridad ( )* (parte de la interfaz *IniciarTrabajo*) quizá colabore con un componente de nombre **FiladeTrabajos** para determinar los tipos y prioridades de trabajos que están en espera de su producción.

Esta actividad de elaboración se aplica a cada componente definido como parte del diseño de la arquitectura. Una vez concluida, se aplica más elaboración a cada atributo, operación e interfaz. Deben especificarse las estructuras de datos apropiadas para cada atributo. Además, se diseñan los detalles algorítmicos requeridos para implantar la lógica del procesamiento asociada con cada operación. Este diseño del procedimiento se analiza más adelante, en este capítulo. Por último, se diseñan los mecanismos requeridos para implantar la interfaz. Para el software orientado a objetos, esto incluye la descripción de todos los mensajes que se requieren para efectuar la comunicación dentro del sistema.

### 10.1.2 La visión tradicional

En el contexto de la ingeniería de software tradicional, un componente es un elemento funcional de un programa que incorpora la lógica del procesamiento, las estructuras de datos internas que

www.FreeLibros.me

**Cita:**

"Invariablemente se observa que un sistema complejo que funciona ha evolucionado a partir de un sistema sencillo que funcionaba."

**John Gall**

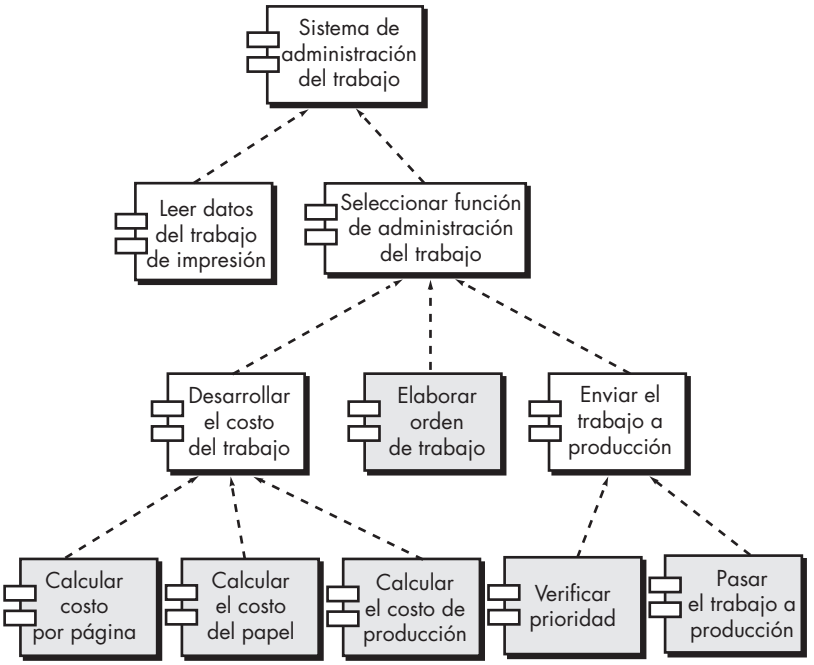
se requieren para implantar la lógica del procesamiento y una interfaz que permite la invocación del componente y el paso de los datos. Dentro de la arquitectura del software se encuentra un componente tradicional, también llamado *módulo*, que tiene tres funciones importantes: 1) como *componente de control* que coordina la invocación de todos los demás componentes del dominio del problema, 2) como *componente del dominio del problema* que implanta una función completa o parcial que requiere el cliente y 3) como *componente de infraestructura* que es responsable de las funciones que dan apoyo al procesamiento requerido en el dominio del problema.

Igual que los componentes orientados a objetos, los componentes tradicionales del software provienen del modelo de análisis. Sin embargo, en este caso, el elemento de datos orientado al flujo del modelo de análisis sirve de base para su obtención. Cada transformación (burbuja) representada en los niveles más bajos del diagrama de flujo de datos se mapea (véase la sección 9.6) en una jerarquía de módulos. Cerca de la parte superior de la jerarquía (arquitectura del programa) se hallan los componentes del control (módulos) y hacia la parte inferior de ella tienden a encontrarse los del dominio del problema. Para lograr una modularidad efectiva, cuando se elaboran los componentes se aplican conceptos de diseño, como la independencia de funciones (véase el capítulo 8).

Para ilustrar este proceso de elaboración del diseño de componentes tradicionales, considere otra vez el software que debe elaborarse para un taller de impresión avanzada. Durante el modelado de los requerimientos se obtendrá un conjunto de diagramas de flujo de datos. Suponga que éstos se mapean en la arquitectura que se aprecia en la figura 10.2. Cada rectángulo representa un componente del software. Observe que los que están sombreados son equivalentes en su función y operaciones a los definidos para la clase **ImprimirTrabajo** que se analizó en la sección 10.1.1. Sin embargo, en este caso, cada operación se representa como módulo aislado que se invoca como se indica en la figura. Para controlar el procesamiento se utilizan otros módulos, por lo que son componentes de control.

Cada módulo de la figura 10.2 se elabora durante el diseño en el nivel de componentes. La interfaz del módulo se define explícitamente. Es decir, se representa todo objeto de datos o

**FIGURA 10.2**
**Gráfica de la estructura de un sistema tradicional**




www.FreeLibros.me


*Conforme se elabora el diseño para cada componente del software, la atención pasa al diseño de estructuras de datos específicas y al diseño del procedimiento para manipularlas. Sin embargo, no hay que olvidar la arquitectura que debe albergar los componentes o las estructuras de datos globales que den servicio a muchos componentes.*

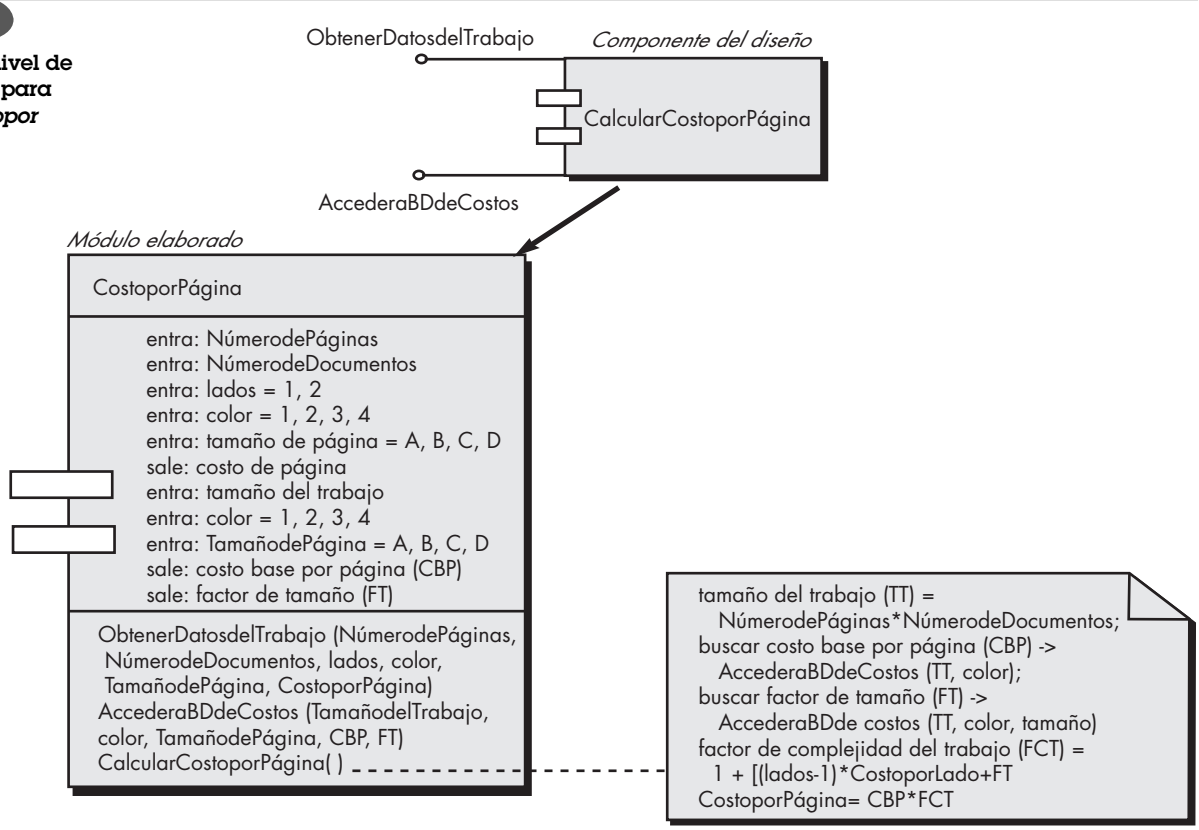
control que fluya a través de la interfaz. Se definen las estructuras de datos que se utilicen en el interior del módulo. El algoritmo que permite que el módulo cumpla su función prevista se diseña con el empleo del enfoque de refinamiento por etapas que se estudió en el capítulo 8. El comportamiento del módulo se representa en ocasiones con un diagrama de estado.

Para ilustrar este proceso, considere el módulo *CalcularCostoporPágina*. El objetivo de este módulo es calcular el costo de impresión por página con base en las especificaciones dadas por el cliente. Los datos requeridos para realizar esta función son: **número de páginas en el documento, número total de documentos que se va a producir, impresión por uno o dos lados, requerimientos de color y requerimientos de tamaño**. Estos datos se pasan a *CalcularCostoporPágina* a través de la interfaz del módulo. *CalcularCostoporPágina* usa estos datos para determinar el costo por página con base en el tamaño y complejidad del trabajo, que es función de todos los datos proporcionados al módulo a través de la interfaz. El costo por página es inversamente proporcional al tamaño del trabajo y directamente proporcional a su complejidad.

La figura 10.3 representa el diseño en el nivel de componentes con el uso de notación UML modificada. El módulo *CalcularCostoporPágina* accede a los datos invocando el módulo *ObtenerDatosdelTrabajo*, que permite que todos los datos relevantes pasen al componente, y una interfaz de base de datos, *AccederBDdeCostos*, que permite que el módulo acceda a una base de datos que contiene todos los costos de impresión. A medida que avanza el diseño, se elabora el módulo *CalcularCostoporPágina* para que provea los detalles del algoritmo y de la interfaz (véase la figura 10.3). Los detalles del algoritmo se representan con el uso del texto de seudocódigo que aparece en la figura, o con un diagrama de actividades UML. Las interfaces se representan como una colección de objetos de datos o conceptos de entrada y salida. La elaboración del diseño continúa hasta que haya detalles suficientes que guíen la construcción del componente.

**FIGURA 10.3**

**Diseño en el nivel de componentes para *CalcularCostopor Página***




www.FreeLibros.me

### 10.1.3 Visión relacionada con el proceso

La visión orientada a objetos y la tradicional del diseño en el nivel de componentes, presentadas en las secciones 10.1.1 y 10.1.2, suponen que el componente se diseña desde la nada. Es decir, que se crea un nuevo componente con base en las especificaciones obtenidas del modelo de requerimientos. Por supuesto, existe otro enfoque.

En las últimas dos décadas, la comunidad de la ingeniería de software ha puesto el énfasis en la necesidad de elaborar sistemas que utilicen componentes de software o patrones de diseño ya existentes. En esencia, a medida que avanza el trabajo de diseño se dispone de un catálogo de diseño probado o de componentes en el nivel de código. Conforme se desarrolla la arquitectura del software, se escogen del catálogo componentes o patrones de diseño y se usan para construir la arquitectura. Como estos componentes fueron construidos teniendo en mente lo reutilizable, se dispone totalmente de la descripción de su interfaz, de las funciones que realizan y de la comunicación y colaboración que requieren. En la sección 10.6 se estudian algunos aspectos importantes de la ingeniería de software basada en componentes.

INFORMACIÓN

*Estándares y marcos basados en componentes*

Uno de los elementos clave que conducen al éxito o fracaso de la ingeniería de estándares basados en componentes es su disponibilidad, que en ocasiones recibe el nombre de *middleware*. El *middleware* es una colección de componentes de infraestructura que permiten que los componentes del dominio del problema se comuniquen entre sí a través de una red o dentro de un sistema complejo. Los ingenieros de software que deseen usar el desarrollo basado en componentes como proceso de software pueden elegir entre los estándares siguientes:

**OMG CORBA-www.corba.org/**
**Microsoft COM-www.microsoft.com/com/tech/complus.asp**
**Microsoft.NET-http://msdn2.microsoft.com/en-us/netframework/default.aspx**
**Sun JavaBeans-http://java.sun.com/products/ejb/**

Estos sitios web tienen una amplia variedad de métodos de enseñanza, documentos en limpio, herramientas y recursos generados con dichos estándares importantes de middleware.

## 10.2 DISEÑO DE COMPONENTES BASADOS EN CLASE

Como ya se dijo, el diseño en el nivel de componentes se basa en la información desarrollada como parte del modelo de requerimientos (capítulos 6 y 7) y se representa como parte del modelo arquitectónico (véase el capítulo 9). Cuando se escoge un enfoque de ingeniería orientado al software, el diseño en el nivel de componentes se centra en la elaboración de clases específicas del dominio del problema y en el refinamiento de las clases de infraestructura contenidas en el modelo de requerimientos. La descripción detallada de los atributos, operaciones e interfaces que emplean dichas clases es el detalle de diseño que se requiere como precursor de la actividad de construcción.

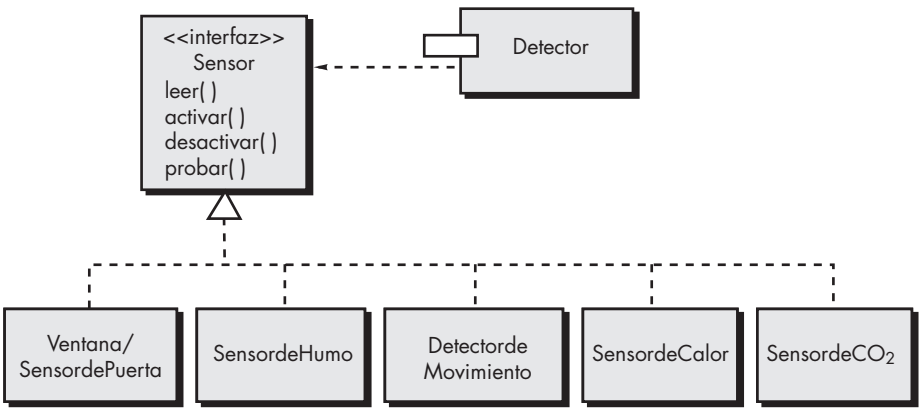
### 10.2.1 Principios básicos del diseño

Hay cuatro principios básicos que son aplicables al diseño en el nivel de componentes y que han sido ampliamente aceptados para la aplicación de la ingeniería de software orientada a objetos. La motivación subyacente para aplicar estos principios es crear diseños que sean más factibles de cambiar, así como reducir la propagación de efectos colaterales cuando se hagan cambios. Estos principios pueden usarse como guía cuando se desarrolle cada componente del software.

**Principio Abierto-Cerrado (PAC).** *"Un módulo [componente] debe ser abierto para la extensión pero cerrado para la modificación"* [Mar00]. Este enunciado parece ser una contradicción, pero representa una de las características más importantes de un buen diseño en el nivel de

www.FreeLibros.me

**FIGURA 10.4**
**Seguimiento del PAC**




componentes. Dicho en pocas palabras, debe especificarse el componente en forma tal que permita extenderlo (dentro del dominio funcional a que está dirigido) sin necesidad de hacerle modificaciones internas (en el nivel del código o de la lógica). Para lograr esto, se crean abstracciones que sirven como búfer entre la funcionalidad que sea probable extender y la clase de diseño en sí.

Por ejemplo, suponga que la función de seguridad *CasaSegura* utiliza la clase **Detector** que debe revisar el estado de cada tipo de sensor de seguridad. Es probable que, conforme pase el tiempo, crezca el número y tipos de sensores de seguridad. Si la lógica de procesamiento interno se implementa como una secuencia de comandos si-entonces-en otro caso, cada uno dirigido a un tipo diferente de sensor, cuando se agregue uno nuevo se requerirá una lógica de procesamiento interno adicional (otro si-entonces-en otro caso). Esto sería una violación del PAC.

## CASASEGURA

### *El PAC en acción*

**La escena:** El cubículo de Vinod.

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

**La conversación:**

**Vinod:** Me acaba de llamar Doug [gerente del equipo]. Dice que mercadotecnia quiere agregar un nuevo sensor.

**Shakira (con sonrisa cómplice):** No otra vez, por favor…

**Vinod:** Sí… y no vas a creer con lo que salieron…

**Shakira:** Sorpréndeme.

**Vinod (ríe):** Lo llaman *sensor de angustia del perro*.

**Shakira:** ¿Qué dijiste?

**Vinod:** Es para la gente que deja su mascota en departamentos o condominios o casas que están muy cerca una de otra. El perro comienza a ladrar, el vecino se enoja y se queja. Con este sensor, si el perro ladra durante, digamos, más de un minuto, el sensor hace sonar una alarma especial que llama al propietario a su teléfono móvil.

**Shakira:** Bromeas, ¿verdad?

**Vinod:** No, no. Doug quiere saber cuánto tiempo nos llevaría agregar eso a la función de seguridad.

**Shakira (piensa un momento):** No mucho… mira [muestra a Vinod la figura 10.4]. Hemos aislado las clases reales sensor atrás de la interfaz **sensor**. Cuando tengamos las especificaciones para el sensor perrito, será pan comido agregarlo. Lo único que tendré que hacer es crear un componente apropiado para él… mmm, una clase. El componente **Detector** no cambiará en absoluto.

**Vinod:** Entonces diré a Doug que no hay problema.

**Shakira:** Conociendo a Doug, nos estará vigilando; yo no le daría el asunto del perrito hasta la siguiente entrega.

**Vinod:** No está mal, pero podrías implantarlo ahora si él quisiera, ¿o no?

**Shakira:** Sí, la forma en la que diseñamos la interfaz me permite hacerlo sin problemas.

**Vinod (piensa un momento):** ¿Has oído hablar del Principio Abierto-Cerrado?

**Shakira (encoge los hombros):** Nunca.

**Vinod (sonríe):** No hay problema.

www.FreeLibros.me

En la figura 10.4 se ilustra una forma de seguir el PAC para la clase **Detector**. La interfaz *sensor* presenta una consistente visión de los componentes sensores para los detectores. Si se agregara un nuevo tipo de sensor, no se requeriría hacer ningún cambio para la clase **Detector** (componente). Se preservaría el PAC.

**Principio de sustitución de Liskov (PSL).** *"Las subclases deben ser sustituibles por sus clases de base"* [Mar00]. Este principio de diseño, originalmente propuesto por Barbara Liskov [Lis88], sugiere que un componente que use una clase de base debe funcionar bien si una clase derivada de la clase base pasa al componente. El PSL demanda que cualquier clase derivada de una clase de base debe respetar cualquier contrato implícito entre la clase de base y los componentes que la usan. En el contexto de este análisis, un "contrato" es una *precondición* que debe ser verdadera antes de que el componente use una clase de base y una *poscondición* que debe ser verdadera después de ello. Cuando se crean clases derivadas hay que asegurarse de que respeten la precondición y la poscondición.


*Si omite el diseño y pasa al código, sólo recuerde que el diseño es la última "concreción". Estaría violando el PID.*

**Principio de Inversión de la Dependencia (PID).** *"Dependa de las abstracciones. No dependa de las concreciones"* [Mar00]. Como se vio en el estudio del PAC, las abstracciones son el lugar en el que es posible ampliar un diseño sin muchas dificultades. Entre más dependa un componente de otros componentes concretos (y no de abstracciones tales como una interfaz), más difícil será ampliarlo.

**Principio de segregación de la interfaz (PSI).** *"Es mejor tener muchas interfaces específicas del cliente que una sola de propósito general"* [Mar00]. Hay muchas instancias en las que múltiples componentes del cliente usan las operaciones que provee una clase servidor. El PSI sugiere que debe crearse una interfaz especializada que atienda a cada categoría principal de clientes. En la interfaz de ese cliente, sólo deben especificarse aquellas operaciones que sean relevantes para una categoría particular de clientes.

Por ejemplo, considere la clase **PlanodelaCasa** que se usó en las funciones de seguridad y vigilancia de *CasaSegura* (véase el capítulo 6). Para las funciones de seguridad, **PlanodelaCasa** se utiliza sólo durante las actividades de configuración y emplea las operaciones *SituarDispositivo( )*, *MostrarDispositivo( )*, *AgruparDispositivo( )* y *QuitarDispositivo( )* para situar, mostrar, agrupar y quitar sensores del plano de la casa. La función de vigilancia de *CasaSegura* usa las cuatro operaciones mencionadas para la seguridad, pero también requiere operaciones especiales para administrar cámaras: *MostrarFOV( )* y *MostrarIdentificacióndeDispositivo( )*. Entonces, el PSI sugiere que los componentes cliente de las dos funciones de *CasaSegura* tienen interfaces especializadas definidas para ellas. La interfaz para la seguridad incluiría sólo las operaciones *SituarDispositivo( )*, *MostrarDispositivo( )*, *AgruparDispositivo( )* y *QuitarDispositivo( )*. La interfaz para vigilancia incorporaría esas mismas operaciones pero también *MostrarFOV( )* y *MostrarIdentificacióndeDispositivo( )*.

Aunque los principios de diseño en el nivel de componentes son una guía útil, los componentes no existen en el vacío. En muchos casos, los componentes o clases individuales están organizados en subsistemas o paquetes. Es razonable preguntar cómo debe ocurrir esta actividad de agrupamiento. ¿Exactamente cómo deben organizarse los componentes conforme avanza el diseño? Martin [Mar00] propone principios adicionales de agrupamiento que son aplicables al diseño en el nivel de componentes:


Para que los componentes sean reutilizables, su diseño requiere algo más que un buen diseño técnico. También exige mecanismos efectivos de configuración (véase el capítulo 22).

**Principio de equivalencia de la liberación de la reutilización (PER).** *"El gránulo de reutilización es el gránulo de liberación"* [Mar00]. Cuando las clases o componentes se diseñan para ser reutilizables, existe un contrato implícito que se establece entre el desarrollador de la entidad reutilizable y las personas que la emplearán. El desarrollador se compromete a establecer un sistema que controle la liberación para que dé apoyo y mantenimiento a las versiones anteriores de la entidad mientras los usuarios se actualizan poco a poco con la versión más nueva.

www.FreeLibros.me

En lugar de abordar cada clase individual, es frecuente que sea mejor agrupar las que sean reutilizables en paquetes que puedan manejarse y controlar a medida que evolucionen las nuevas versiones.

**Principio de cierre común (PCC).** *"Las clases que cambian juntas pertenecen a lo mismo"* [Mar00]. Las clases deben empacarse en forma cohesiva. Es decir, cuando las clases se agrupan como parte de un diseño, deben estar dirigidas a la misma área de funciones o comportamiento. Cuando deba cambiar alguna característica de dicha área, es probable que sólo aquellas clases que haya dentro del paquete requieran modificación. Esto lleva a un control de cambios y a un manejo de la liberación más eficaces.

**Principio de la reutilización común (PRC).** *"Las clases que no se reutilizan juntas no deben agruparse juntas"* [Mar00]. Cuando cambia una o más clases dentro de un paquete, cambia el número de liberación del paquete. Entonces, todas las demás clases o paquetes que permanecen en el paquete que cambió deben actualizarse con la liberación más reciente y someterse a pruebas a fin de garantizar que la nueva versión opera sin problemas. Si las clases no se agrupan de manera cohesiva, es posible que se cambie una clase sin relación junto con las demás que hay dentro del paquete. Esto generará integración y pruebas innecesarias. Por esta razón, sólo las clases que se reutilicen juntas deben incluirse dentro de un paquete.

### 10.2.2 Lineamientos de diseño en el nivel de componentes

Además de los principios estudiados en la sección 10.2.1, conforme avanza el diseño en el nivel de componentes se aplican lineamientos prácticos a los componentes, a sus interfaces y a las características de dependencia y herencia que tengan algún efecto en el diseño resultante. Ambler [Amb02b] sugiere los lineamientos siguientes:

**? ¿Qué es lo que hay que tomar en cuenta al dar nombre a los componentes?**

**Componentes.** Deben establecerse convenciones para dar nombre a los componentes que se especifique que forman parte del modelo arquitectónico, para luego mejorarlos y elaborarlos como parte del modelo en el nivel de componentes. Los nombres de los componentes arquitectónicos deben provenir del dominio del problema y significar algo para todos los participantes que vean el modelo arquitectónico. Por ejemplo, el nombre de la clase **PlanodelaCasa** tiene un significado para todos los que lo lean, aunque no tengan formación técnica. Por otro lado, los componentes de infraestructura o clases elaboradas en el nivel de componentes deben recibir un nombre que tenga un significado específico de la implantación. Si como parte de la implantación de **PlanodelaCasa** va a administrarse una lista vinculada, es apropiada la operación *AdministrarLista( )*, aun si una persona sin capacitación técnica pudiera interpretarlo mal.[3]

Pueden usarse estereotipos para ayudar a identificar la naturaleza de los componentes en el nivel de diseño detallado. Por ejemplo, **<<infraestructura>>** debiera usarse para identificar un componente de infraestructura, **<<basededatos>>** podría emplearse para identificar una base de datos que dé servicio a una o más clases de diseño o a todo el sistema; se usaría **<<tabla>>** para identificar una tabla dentro de una base de datos.

**Interfaces.** Las interfaces dan información importante sobre la comunicación y la colaboración (también nos ayudan a cumplir el PAC). Sin embargo, la representación sin restricciones de las interfaces tiende a complicar los diagramas de componentes. Ambler [Amb02c] recomienda que 1) si los diagramas aumentan en complejidad, en lugar del enfoque formal del UML con recuadro y flecha, debe representarse la interfaz con una paleta; 2) en aras de la consistencia, las interfaces deben fluir a partir del lado izquierdo del recuadro del componente; 3) sólo deben aparecer aquellas interfaces que sean relevantes para el componente que se está considerando,

3 No es probable que alguien de mercadotecnia o de la organización cliente (de tipo no técnico) analice la información de diseño detallado.

www.FreeLibros.me

aun si estuvieran disponibles otras. Estas recomendaciones buscan simplificar la naturaleza visual de los diagramas UML de componentes.

**Dependencias y herencia.** Para tener una mejor legibilidad, es buena idea modelar las dependencias de izquierda a derecha y la herencia de abajo (clases obtenidas) hacia arriba (clases base). Además, las interdependencias de componentes deben representarse por medio de interfaces y no con la dependencia componente a componente. Según la filosofía del PAC, esto ayudará a hacer que sea más fácil dar mantenimiento al sistema.

### 10.2.3 Cohesión

En el capítulo 8 se describió la cohesión como la "unidad de objetivo" de un componente. En el contexto del diseño en el nivel de componentes para los sistemas orientados a objetos, la *cohesión* implica que un componente o clase sólo contiene atributos y operaciones que se relacionan de cerca uno con el otro y con la clase o componente en sí. Lethbridge y Laganiére [Let01] definen varios tipos diferentes de cohesión (se listan en función del nivel de cohesión):[4]

**Funcional.** Lo tienen sobre todo las operaciones; este nivel de cohesión ocurre cuando un componente realiza un cálculo y luego devuelve el resultado.

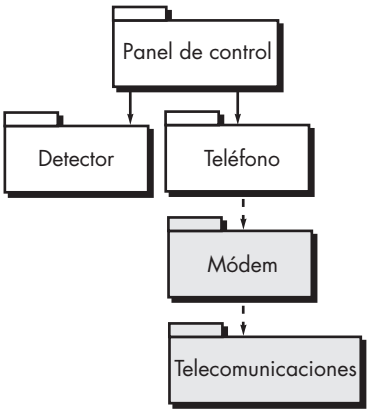

*Aunque es instructivo entender los distintos niveles de cohesión, es más importante tener presente el concepto general cuando se diseñen componentes. Mantenga la cohesión tan grande como sea posible.*

**De capa.** Lo tienen los paquetes, componentes y clases; este tipo de cohesión ocurre cuando una capa más alta accede a los servicios de otra más baja, pero ésta no tiene acceso a las superiores. Por ejemplo, considere el requerimiento de la función de seguridad de *CasaSegura* para hacer una llamada telefónica si se detecta una alarma. Podría definirse un conjunto de paquetes en capas, como se aprecia en la figura 10.5. Los paquetes sombreados contienen componentes de infraestructura. Es posible realizar el acceso del paquete del panel de control hacia abajo.

**De comunicación.** Todas las operaciones que acceden a los mismos datos se definen dentro de una clase. En general, tales clases se centran únicamente en los datos en cuestión, acceden a ellos y los guardan.

Las clases y componentes que tienen cohesión funcional, de capa y comunicación son relativamente fáciles de implantar, probar y mantener. Siempre que sea posible, deben alcanzarse estos niveles de cohesión. Sin embargo, es importante notar que en ocasiones hay aspectos pragmáticos del diseño y de la implantación que obligan a optar por niveles de cohesión más bajos.

**FIGURA 10.5**
**Cohesión de capa**




4 En general, entre más alto sea el nivel de cohesión, el componente es más fácil de implantar, probar y mantener.

www.FreeLibros.me

**CasaSegura**

### *La cohesión en acción*

**La escena:** Cubículo de Jamie.

**Participantes:** Jamie y Ed, miembros del equipo de ingeniería de software que trabajan en la función de vigilancia.

**La conversación:**

**Ed:** Tengo un diseño de primer corte del componente **cámara**.

**Jamie:** ¿Quieres revisarlo rápido?

**Ed:** Sí… pero en realidad quisiera que me dijeras algo.

(Con señas, Jamie lo invita a que continúe.)

**Ed:** Originalmente definimos cinco operaciones para **cámara**. Mira…

*DeterminarTipo( )* dice el tipo de cámara.

*CambiarUbicación( )* permite mover la cámara por el plano de la casa.

*MostrarIdentificación( )* obtiene la identificación de la cámara y la muestra cerca de su ícono.

*MostrarVista( )* presenta el campo de visión de la cámara en forma gráfica.

*MostrarAcercamiento( )* muestra gráficamente la amplificación de la cámara.

**Ed:** Las diseñé por separado y son operaciones muy simples. Por eso pensé que sería una buena idea combinar todas las operaciones de la pantalla en una sola que denominé *MostrarCámara( )* y que mostrará la identificación, vista y acercamiento. ¿Cómo la ves?

**Jamie (hace una mueca):** No estoy seguro de que sea una buena idea.

**Ed (frunce el seño):** ¿Por qué? Todas esas pequeñas operaciones pueden dar dolores de cabeza.

**Jamie:** El problema de que las combinemos es que se pierde cohesión, ya sabes, la operación *MostrarCámara( )* no tendrá un único objetivo.

**Ed (un poco exasperado):** ¿Y qué? Todo este asunto requerirá menos de 100 líneas de código fuente, si acaso. Será más fácil de implantar… creo.

**Jamie:** ¿Y qué pasa si decidimos cambiar la forma en la que representamos el campo de visión?

**Ed:** Sólo se pasa a la operación *MostrarCámara( )* y se hace la modificación.

**Jamie:** ¿Qué hay con los efectos colaterales?

**Ed:** ¿Qué quieres decir?

**Jamie:** Bueno, digamos que se hace el cambio, pero, sin darnos cuenta, se genera un problema al mostrar en la pantalla la identificación.

**Ed:** No sería tan torpe.

**Jamie:** Tal vez no, pero, ¿qué tal si alguien de apoyo tiene que hacer la modificación dentro de dos años? Tal vez no entenderá la operación tan bien como tú, y, ¿quién sabe?, podría ser torpe.

**Ed:** Entonces, ¿estás en contra?

**Jamie:** Tú eres el diseñador… es tu decisión… sólo asegúrate de que entiendes las consecuencias de la poca cohesión.

**Ed (piensa un momento):** Tal vez haga operaciones de pantalla separadas.

**Jamie:** Buena decisión.

### 10.2.4 Acoplamiento

En el estudio anterior del análisis y el diseño, se dijo que la comunicación y la colaboración eran elementos esenciales de cualquier sistema orientado a objetos. Sin embargo, esta característica tan importante (y necesaria) tiene un lado oscuro. A medida que aumentan la comunicación y colaboración (es decir, conforme se eleva la "conectividad" entre las clases), la complejidad del sistema también se incrementa. Y si la complejidad aumenta, también crece la dificultad de implantar, probar y dar mantenimiento al software.


*A medida que se elabora el diseño de cada componente del software, la atención pasa al diseño de las estructuras específicas de los datos y al diseño de procedimientos para manipularlas. Sin embargo, no hay que olvidar la arquitectura que debe albergar los componentes de las estructuras globales de los datos, que tal vez atiendan a muchos componentes.*

El *acoplamiento* es la medición cualitativa del grado en el que las clases se conectan una con otra. Conforme las clases (y componentes) se hacen más interdependientes, el acoplamiento crece. Un objetivo importante del diseño en el nivel de componente es mantener el acoplamiento tan bajo como sea posible.

El acoplamiento de las clases se manifiesta de varias maneras. Lethbridge y Laganiére [Let01] definen las siguientes categorías de acoplamiento:

**Acoplamiento de contenido.** Tiene lugar cuando un componente "modifica subrepticiamente datos internos en otro componente" [Let01]. Esto viola el ocultamiento de la información, concepto básico del diseño.

**Acoplamiento común.** Sucede cuando cierto número de componentes hacen uso de una variable global. Aunque a veces esto es necesario (por ejemplo, para establecer valores de-

www.FreeLibros.me

finidos que se utilizan en toda la aplicación), el acoplamiento común lleva a la propagación incontrolada del error y a efectos colaterales imprevistos cuando se hacen los cambios.

**Acoplamiento del control.** Tiene lugar si la operación *A( )* invoca a la operación *B( )* y pasa una bandera de control a *B*. La bandera "dirige" entonces el flujo de la lógica dentro de *B*. El problema con esta forma de acoplamiento es que un cambio no relacionado en *B* puede dar como resultado la necesidad de cambiar el significado de la bandera de control que pasa *A*. Si esto se pasa por alto ocurrirá un error.

**Acoplamiento de molde.** Se presenta cuando se declara a **ClaseB** como un tipo para un argumento de una operación de **ClaseA**. Como **ClaseB** ahora forma parte de la definición de **ClaseA**, la modificación del sistema se vuelve más compleja.

**Acoplamiento de datos.** Ocurre si las operaciones pasan cadenas largas de argumentos de datos. El "ancho de banda" de la comunicación entre clases y componentes crece y la complejidad de la interfaz se incrementa. Se hace más difícil hacer pruebas y dar mantenimiento.

**Acoplamiento de rutina de llamada.** Tiene lugar cuando una operación invoca a otra. Este nivel de acoplamiento es común y con frecuencia muy necesario. Sin embargo, aumenta la conectividad del sistema.

**Acoplamiento de tipo de uso.** Ocurre si el componente **A** usa un tipo de datos definidos en el componente **B** (esto ocurre siempre que "una clase declara una variable de instancia o una variable local como si tuviera otra clase para su tipo" [Let01]). Si cambia la definición de tipo, también debe cambiar todo componente que la utilice.

**Acoplamiento de inclusión o importación.** Pasa cuando el componente **A** importa o incluye un paquete o el contenido del componente **B**.

**Acoplamiento externo.** Sucede si un componente se comunica o colabora con componentes de infraestructura (por ejemplo, funciones del sistema operativo, capacidad de la base de datos, funciones de telecomunicación, etc.). Aunque este tipo de acoplamiento es necesario, debe limitarse a un número pequeño de componentes o clases dentro de un sistema.

El software debe tener comunicación interna y externa. Por tanto, el acoplamiento es un hecho de la vida. Sin embargo, el diseñador debe trabajar para reducirlo siempre que sea posible, y entender las ramificaciones que tiene el acoplamiento abundante cuando no puede evitarse.

## CasaSegura

### *El acoplamiento en acción*

**La escena:** Cubículo de Shakira.

**Participantes:** Vinod y Shakira, miembros del equipo de software de CasaSegura, que trabajan en la función de seguridad.

**La conversación:**

**Shakira:** Tuve lo que considero una gran idea… entonces lo pensé un poco y me pareció que no era tan buena. Al final la deseché, pero pensé en hacerla para ustedes.

**Vinod:** Seguro. ¿Cuál es la idea?

**Shakira:** Bueno, cada uno de los sensores reconoce una condición de alarma de algún tipo, ¿verdad?

**Vinod (sonríe):** Por eso se llaman sensores, Shakira.

**Shakira (exasperada):** Sarcasmo, Vinod, tienes que mejorar tus habilidades interpersonales.

**Vinod:** ¿Decías?

**Shakira:** Bien, de cualquier modo, me pregunté… por qué no crear una operación dentro de cada objeto de sensor llamada *HacerLlamada( )* que colaboraría directamente con el componente **SaleLlamada**, bueno, con una interfaz hacia el componente **SaleLlamada**.

**Vinod (pensativo):** Quieres decir, ¿eso en vez de hacer que esa colaboración ocurra fuera de un componente como **PaneldeControl** o algún otro?

www.FreeLibros.me

**Shakira:** Sí… Pero entonces me dije que eso significaba que cada objeto de sensor estaría conectado al componente **SaleLlamada**, y que eso querría decir que estaría acoplado de manera indirecta con el mundo exterior y… bueno, pensé que sólo complicaría las cosas.

**Vinod:** Estoy de acuerdo. En este caso, es mejor idea dejar que la interfaz del sensor pase información a **PaneldeControl** y que inicie la llamada de salida. Además, diferentes sensores tal vez darían como resultado números telefónicos distintos. Tú no querrías que el sensor guardara esa información, porque si cambiara…

**Shakira:** No lo sentí bien.

**Vinod:** La heurística del diseño para el acoplamiento nos dice que no está bien.

**Shakira:** Pues sí.

## 10.3 Realización del diseño en el nivel de componentes

**Cita:**

"Si hubiera tenido más tiempo habría escrito una carta más breve."

**Blas Pascal**

Antes, en este capítulo, se dijo que el diseño en el nivel de componentes es de naturaleza elaborativa. Debe transformarse la información de los modelos de requerimientos y arquitectónico a una representación de diseño que dé suficientes detalles para guiar la actividad de construcción (codificación y pruebas). Los pasos siguientes representan un conjunto de tareas comunes para el diseño en el nivel de componentes cuando se aplica a un sistema orientado a objetos.

**Paso 1. Identificar todas las clases de diseño que correspondan al dominio del problema.** Con el uso del modelo de requerimientos y arquitectónico, se elabora cada clase de análisis y componente de la arquitectura según se describió en la sección 10.1.1.

**CONSEJO**

*Si se trabaja en un ambiente sin espías, los primeros tres pasos se dirigen a refinar los objetos de datos y las funciones de procesamiento (transformaciones) identificadas como parte del modelo de requerimientos.*

**Paso 2. Identificar todas las clases de diseño que correspondan al dominio de la infraestructura.** Estas clases no están descritas en el modelo de los requerimientos y con frecuencia se pierden a partir del modelo arquitectónico; sin embargo, deben describirse en este punto. Como se dijo, las clases y componentes en esta categoría incluyen componentes de la interfaz gráfica de usuario (con frecuencia disponibles como componentes reutilizables), componentes del sistema operativo y componentes de administración de objetos y datos.

**Paso 3. Elaborar todas las clases de diseño que no sean componentes reutilizables.** La elaboración requiere que se describan en detalle todas las interfaces, atributos y operaciones necesarios para implantar la clase. Mientras se realiza esta tarea, deben considerarse los heurísticos del diseño (como la cohesión y el acoplamiento del componente).

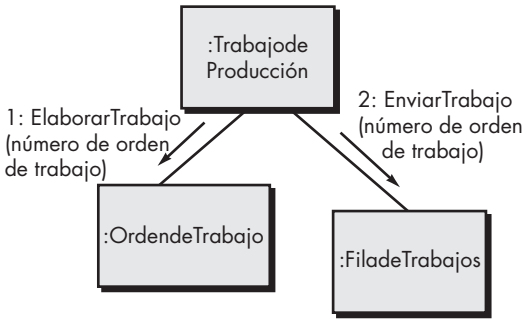
**Paso 3*a*. Especificar detalles del mensaje cuando colaboren clases o componentes.** El modelo de requerimientos utiliza un diagrama de colaboración para mostrar la forma en la que las clases de análisis colaboran una con la otra. A medida que avanza el diseño en el nivel de componentes, en ocasiones es útil mostrar los detalles de estas colaboraciones especificando las estructuras de los mensajes que pasan entre los objetos de un sistema. Aunque esta actividad de diseño es opcional, se usa como precursor de la especificación de interfaces que muestren el modo en el que se comunican y colaboran los componentes del sistema.

La figura 10.6 ilustra un diagrama sencillo de colaboración para el sistema de impresión que ya se mencionó. Los objetos **TrabajodeProducción**, **OrdendeTrabajo** y **FiladeTrabajos** colaboran para preparar un trabajo de impresión a fin de ejecutar una secuencia de producción. Los mensajes entre los objetos se transmiten como lo ilustran las flechas en la figura. Durante la modelación de los requerimientos, los mensajes se especifican como se aprecia. Sin embargo, conforme el diseño avanza, cada mensaje se elabora expandiendo su sintaxis de la manera siguiente [Ben02]:

```
[condición de guardia] expresión de secuencia (devuelve valor) :=
nombre del mensaje (lista de argumentos)
```

www.FreeLibros.me

**FIGURA 10.6**
**Diagrama de colaboración con mensajería**




donde **[condición de guardia]** se escribe en Lenguaje de Restricción de Objetos (LRO)[5] y especifica cualesquiera condiciones que deban cumplirse antes de que pueda enviarse el mensaje; **expresión de secuencia** es un valor entero (u otro indicador de ordenación, por ejemplo, 3.1.2) que indica el orden secuencial en el que se envía el mensaje; **(devuelve valor)** es el nombre de la información que devuelve la operación invocada por el mensaje; **nombre del mensaje** identifica la operación que va a invocarse y **(lista de argumentos)** es la lista de atributos que se pasan a la operación.

**Paso 3*b*. Identificar interfaces apropiadas para cada componente.** En el contexto del diseño en el nivel de componentes, una interfaz UML es un "grupo de operaciones visibles externamente (para el público). La interfaz no contiene estructura interna, ni atributos ni asociaciones..." [Ben02]. Dicho con más formalidad, una interfaz es el equivalente de una clase abstracta que provee una conexión controlada entre clases de diseño. En la figura 10.1 se ilustra la elaboración de interfaces. En esencia, las operaciones definidas para la clase de diseño se clasifican en una o más clases abstractas. Cada operación dentro de la clase abstracta (la interfaz) debe ser cohesiva, es decir, debe tener un procesamiento que se centre en una función o subfunción limitada.

En relación con la figura 10.1, puede afirmarse que la interfaz *IniciarTrabajo* no tiene suficiente cohesión. En realidad, ejecuta tres subfunciones diferentes: elaborar una orden de trabajo, verificar la prioridad del trabajo y pasar el trabajo a producción. La interfaz debe rediseñarse. Un enfoque podría consistir en volver a estudiar las clases de diseño y definir una nueva, **OrdendeTrabajo**, que se haría cargo de todas las actividades asociadas con la formación de una orden de trabajo. La operación *ElaborarOrdendeTrabajo( )* se vuelve parte de esa clase. De manera similar, se definiría una clase **FiladeTrabajos** que incorporaría la operación *VerificarPrioridad( )*. Una clase **TrabajodeProducción** podría incorporar toda la información asociada con un trabajo que pasara a las instalaciones de producción. La interfaz *IniciarTrabajo* podría adoptar la forma que se muestra en la figura 10.7. Ahora la interfaz *IniciarTrabajo* es cohesiva y se centra en una función. Las interfaces asociadas con **TrabajodeProducción, OrdendeTrabajo** y **FiladeTrabajos** también tienen un solo objetivo.

**Paso 3*c*. Elaborar atributos y definir tipos y estructuras de datos requeridos para implantarlos.** En general, las estructuras y tipos de datos usados para definir atributos se definen en el contexto del lenguaje de programación que se va a usar para la implantación. El UML define un tipo de datos del atributo que usa la siguiente sintaxis:

**nombre : tipo-de-expresión = valor-inicial {cadena de propiedades}**

5 En el apéndice 1 se analiza brevemente el LRO.

www.FreeLibros.me

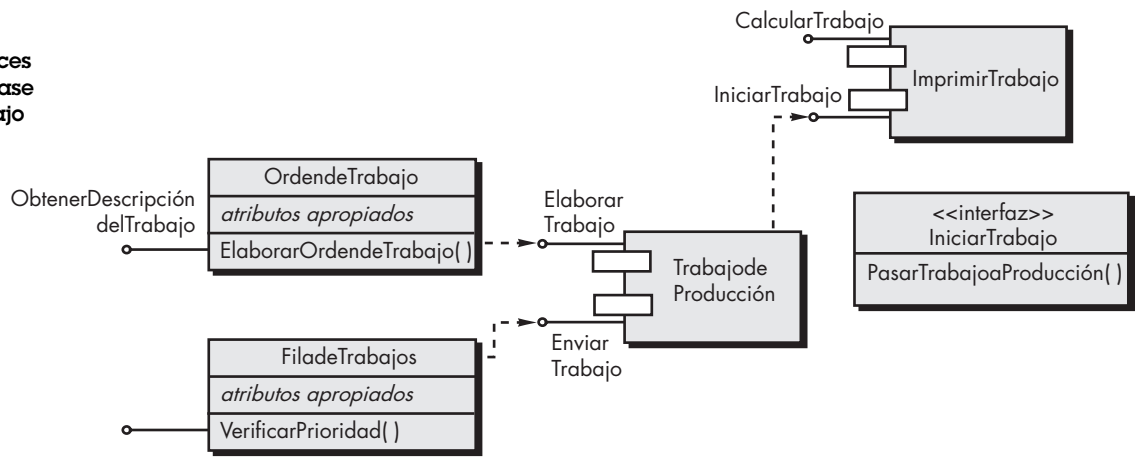



**FIGURA 10.7**
**Rediseño de interfaces y definiciones de clase para ImprimirTrabajo**

donde **nombre** es el nombre del atributo, **tipo-de-expresión** es el tipo de datos, **valor-inicial** es el valor que toma el atributo cuando se crea un objeto y **cadena de propiedades** define una propiedad o característica del atributo.

Durante la primera iteración del diseño en el nivel de componentes, los atributos normalmente se describen por su nombre. De nuevo, en la figura 10.1, la lista de atributos para **ImprimirTrabajo** sólo enlista los nombres de los atributos. No obstante, a medida que avanza la elaboración del diseño, cada atributo se define con el formato de atributos UML mencionado. Por ejemplo, **Tipodepapel-peso** se define del modo siguiente:

**Tipodepapel-peso: cadena = "A" {contiene 1 de 4 valores – A, B, C o D}**

que define a **Tipodepapel-peso** como una variable de cadena que se inicializa en el valor A y que puede adoptar uno de cuatro valores del conjunto {A, B, C, D}.

Si un atributo aparece en forma repetida en varias clases de diseño y tiene una estructura relativamente compleja, es mejor crear una clase separada para que lo albergue.

**Paso 3*d*. Describir en detalle el flujo del procesamiento dentro de cada operación.** Esto se logra con el uso de seudocódigo basado en lenguaje de programación o con un diagrama UML de actividades. Cada componente del software se elabora a través de cierto número de iteraciones que apliquen paso a paso el concepto de refinamiento (capítulo 8).

La primera iteración define cada operación como parte de la clase de diseño. En cada caso, la operación debe caracterizarse en forma tal que asegure que haya mucha cohesión; es decir, la operación debe estar dirigida a la ejecución de una sola función o subfunción. La siguiente iteración no hace más que expandir el nombre de la operación. Por ejemplo, la operación *CalcularCostodelPapel( )* que aparece en la figura 10.1 se expande de la manera siguiente:

**CalcularCostodelPapel (peso, tamaño, color): numérico**

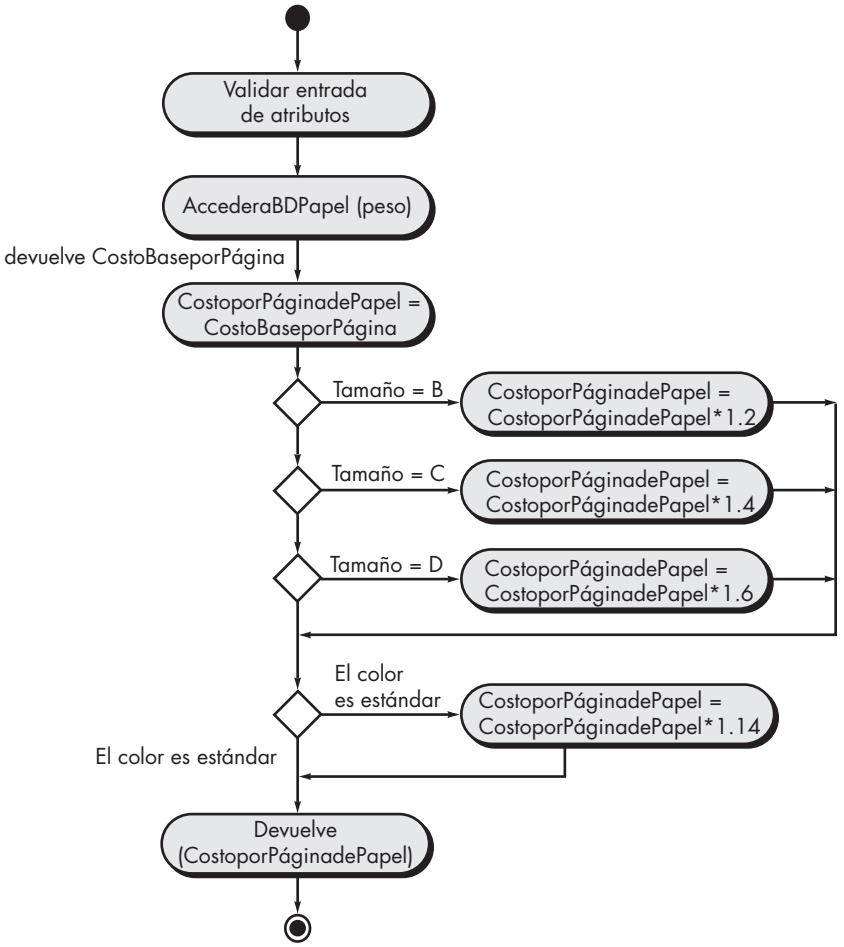
Esto indica que *CalcularCostodelPapel( )* requiere como entrada los atributos **peso**, **tamaño** y **color**, y da como salida un valor numérico (valor en dólares).


*Para refinar el diseño del componente, utilice una elaboración stepwise. Siempre pregunte, "¿hay una forma de simplificar esto y que aun así se logre el mismo resultado?"*

Si el algoritmo requerido para implantar *CalcularCostodelPapel( )* es sencillo y entendido por todos, tal vez no sea necesaria una mayor elaboración del diseño. El ingeniero de software que haga la codificación proveerá los detalles necesarios para implantar la operación. Sin embargo, si el algoritmo fuera más complejo o difícil de entender, se requeriría elaborar más el diseño. La figura 10.8 ilustra un diagrama UML de actividades para *CalcularCostodelPapel( )*. Cuando se usan diagramas de actividades para especificar el diseño en el nivel de componentes, por lo

www.FreeLibros.me

**FIGURA 10.8**
**Diagrama de actividades UML para calcular *CostodelPapel( )***




general se representan en un nivel de abstracción que es algo mayor que el código fuente. En la sección 10.5.3 se estudia un enfoque alternativo: el uso de seudocódigo para hacer las especificaciones del diseño.

**Paso 4. Describir las fuentes persistentes de datos (bases de datos y archivos) e identificar las clases requeridas para administrarlos.** Es normal que las bases de datos y archivos trasciendan la descripción del diseño de un componente individual. En la mayoría de casos, estos almacenamientos persistentes de datos se especifican al inicio como parte del diseño de la arquitectura. No obstante, a medida que avanza la elaboración del diseño, es frecuente que sea útil dar detalles adicionales sobre la estructura y organización de dichas fuentes persistentes de datos.

**Paso 5. Desarrollar y elaborar representaciones del comportamiento para una clase o componente.** Los diagramas de estado UML fueron utilizados como parte del modelo de los requerimientos para representar el comportamiento observable desde el exterior del sistema y el más localizado de las clases de análisis individuales. Durante el diseño en el nivel de componentes, en ocasiones es necesario modelar el comportamiento de una clase de diseño.

El comportamiento dinámico de un objeto (instancias de una clase de diseño cuando el programa se ejecuta) se ve afectado por eventos externos a él y por el estado en curso (modo de comportamiento) del objeto. Para entender el comportamiento dinámico de un objeto, deben estudiarse todos los casos de uso que sean relevantes para la clase de diseño a lo largo de su

www.FreeLibros.me

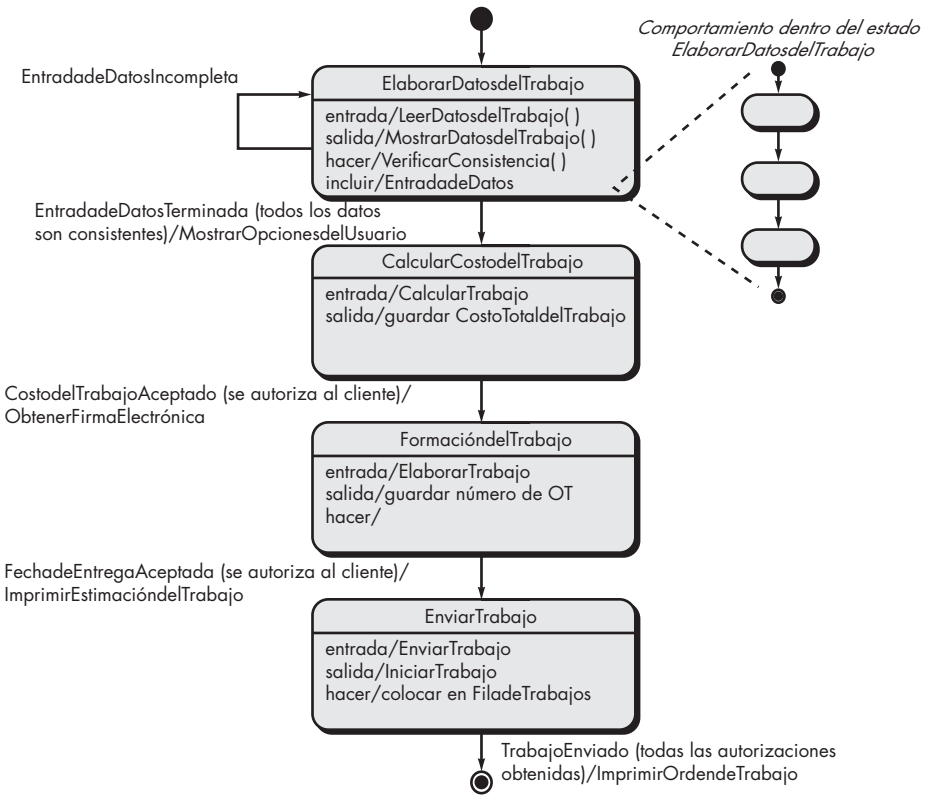



**FIGURA 10.9**
**Fragmento de un diagrama de estado para la clase ImprimirTrabajo**

vida. Estos casos de uso dan información que ayuda a delinear los eventos que afectan al objeto y los estados en los que éste reside conforme pasa el tiempo y suceden los eventos. Las transiciones entre estados (dictadas por los eventos) se representan con un diagrama de estado [Ben02] como el que se ilustra en la figura 10.9.

La transición de un estado al otro (representada por un rectángulo con esquinas redondeadas) ocurre como consecuencia de un evento que tiene la forma siguiente:

**Nombre-del-evento (lista-de-parámetros) [guardar-condición] / expresión de acción**

donde **nombre-del-evento** identifica al evento, **lista-de-parámetros** incorpora datos asociados con el evento, **guardar-condición** se escribe en Lenguaje de Restricción de Objetos (LRO) y especifica una condición que debe cumplirse para que pueda ocurrir el evento, y **expresión de acción** define una acción que ocurre a medida que toma lugar la transición.

En la figura 10.9, cada estado define acciones de *entrada/* y *salida/* que ocurren cuando sucede la transición al estado y fuera de éste, respectivamente. En la mayor parte de los casos, estas acciones corresponden a operaciones que son relevantes para la clase que se modela. El indicador *hacer/* proporciona un mecanismo para señalar actividades que tienen lugar mientras se está en el estado y el indicador *incluir/* proporciona un medio para elaborar el comportamiento incrustando más detalles del diagrama de estado en la definición de un estado.

Es importante observar que el modelo del comportamiento con frecuencia contiene información que no es obvia a primera vista en otros modelos del diseño. Por ejemplo, el análisis cuidadoso del diagrama de estado en la figura 10.9 indica que el comportamiento dinámico de la clase **ImprimirTrabajo** es contingente mientras no se obtengan dos aprobaciones de datos de costo y programación para el trabajo de impresión. Sin las aprobaciones (la condición de guar-

www.FreeLibros.me

dar garantiza que el cliente está autorizado para aprobar), el trabajo de impresión no puede enviarse porque no hay forma de alcanzar el estado *EnviarTrabajo*.

**Paso 6. Elaborar diagramas de despliegue para dar más detalles de la implantación.** Los diagramas de despliegue (véase el capítulo 8) se utilizan como parte del diseño de la arquitectura y se representan en forma de descriptor. De este modo, las funciones principales de un sistema (que con frecuencia se representan como subsistemas) se representan en el contexto del ambiente de computación que las contendrá.

Durante el diseño en el nivel de componentes, pueden elaborarse diagramas de despliegue que representen la ubicación de paquetes de componentes clave. Sin embargo, en un diagrama de componentes, éstos por lo general no se representan de manera individual a fin de evitar la complejidad del diagrama. En ciertos casos, se elaboran diagramas de despliegue en forma de instancia en ese momento. Esto significa que se especifican los ambientes del hardware y el sistema operativo que se emplearán, y se indica la ubicación de los paquetes de componentes dentro de este ambiente.

**Paso 7. Rediseñar cada representación del diseño en el nivel de componentes y siempre considerar alternativas.** En este libro se hace hincapié en que el diseño es un proceso iterativo. El primer modelo en el nivel de componentes que se crea no será tan completo, consistente o exacto como la enésima iteración que se realice. Es esencial rediseñar a medida que se ejecuta el trabajo de diseño.

Además, no debe adoptarse una visión de túnel. Siempre hay soluciones alternativas para el diseño y los mejores diseñadores toman en cuenta todas ellas (o a la mayoría) antes de decidirse por el modelo de diseño final. Desarrolle las alternativas y estudie con cuidado cada una, con los principios y conceptos del diseño presentados en el capítulo 8 y en éste.

## 10.4 Diseño en el nivel de componentes para webapps

Es frecuente que cuando se trata de sistemas y aplicaciones basados en web, la frontera entre el contenido y la función sea borrosa. Por tanto, es razonable preguntar: ¿qué es un componente de *webapps*?

En el contexto de este capítulo, un componente de *webapp* es 1) una función cohesiva bien definida que manipula contenido o da procesamiento de cómputo o de datos para un usuario final o 2) un paquete cohesivo de contenido y funciones que brindan al usuario final alguna capacidad solicitada. Entonces, el diseño en el nivel de componentes de *webapps* con frecuencia incorpora elementos de diseño del contenido y de las funciones.

### 10.4.1 Diseño del contenido en el nivel de componente

El diseño del contenido en el nivel de componentes se centra en objetos de contenido y en la forma en la que se empacan para su presentación a un usuario final de *webapps*. Por ejemplo, considere una capacidad de vigilancia con video dentro de **CasaSeguraAsegurada.com**. Entre muchas otras capacidades, el usuario selecciona y controla cualesquiera cámaras representadas en el plano de la casa, solicita imágenes instantáneas de todas las cámaras y muestra un video desde cualquiera de ellas. Además, el usuario tiene la posibilidad de abrir el ángulo o de hacer acercamientos con una cámara por medio de los íconos apropiados de control.

Para la capacidad de vigilancia con video, pueden definirse varios componentes potenciales de contenido: 1) objetos de contenido que representen la distribución del espacio (el plano de la casa) con íconos adicionales que representen la ubicación de sensores y cámaras de video, 2) el conjunto de imágenes instantáneas de video (cada una es un objeto de datos separado) y 3) la ventana del video de una cámara específica. Cada uno de estos componentes recibe un nombre por separado y se manipula como paquete.

www.FreeLibros.me

Considere el plano de la casa con cuatro cámaras colocadas estratégicamente en una casa. A solicitud del usuario, se captura una toma de video desde cada cámara y se identifica como un objeto de contenido generado en forma dinámica, **TomadeVideo*N***, donde *N* identifica las cámaras 1 a 4. Un componente de contenido, llamado **ImágenesInstantáneas**, combina los cuatro objetos de contenido **TomadeVideo*N*** y los muestra en la página de vigilancia con video.

La formalidad del diseño del contenido en el nivel de componentes debe adaptarse a las características de la *webapp* que se va a elaborar. En muchos casos, los objetos de contenido no necesitan estar organizados como componentes y pueden manipularse en forma individual. Sin embargo, a medida que aumentan el tamaño y la complejidad (de la *webapp*, los objetos de contenido y sus interrelaciones), es necesario organizar el contenido en forma que sea fácil hacer referencia y manipular el diseño.[6] Además, si el contenido es muy dinámico (por ejemplo, el de un sitio de subastas en línea), es importante establecer un modelo estructural claro que incorpore los componentes del contenido.

### 10.4.2 Diseño de las funciones en el nivel de componentes

Las aplicaciones web modernas proporcionan funciones de procesamiento cada vez más sofisticadas que: 1) producen un procesamiento localizado que genera contenido y capacidad de navegación en forma dinámica; 2) dan capacidad de computación o procesamiento de datos que resultan adecuados para el dominio del negocio de la *webapp*; 3) brindan consultas y acceso avanzado a una base de datos, y 4) establecen interfaces de datos con sistemas corporativos externos. Para lograr las capacidades anteriores (y muchas otras), se diseñan componentes de la *webapp* que tengan forma similar a la de los componentes del software convencional.

Las funciones de la *webapp* se entregan como una serie de componentes desarrollados en paralelo con la arquitectura de la información que garantice que sean consistentes. En esencia, se comienza con la consideración del modelo de requerimientos y de la arquitectura inicial de la información, para luego estudiar el modo en el que las funciones afectan la interacción del usuario con la aplicación, la información que se presenta y las tareas que el usuario realiza.

Durante el diseño de la arquitectura, el contenido y funciones de la *webapp* se combinan para crear una *arquitectura funcional*, que es una representación del dominio de funciones de la *webapp* y describe los componentes funcionales clave de la *webapp* y la forma en la que interactúan una con otra.

Por ejemplo, las funciones de abrir el ángulo y hacer acercamientos de la capacidad de vigilancia con video para **CasaSeguraAsegurada.com** se implementan como las operaciones *recorrer( )* y *zoom( )*, que forman parte de la clase **Cámara**. En cualquier caso, la funcionalidad implícita para *recorrer( )* y *zoom( )* deben implantarse como módulos dentro de **CasaSeguraAsegurada.com**.

## 10.5 Diseño de componentes tradicionales

PUNTO CLAVE

La programación estructurada es una técnica de diseño que limita el flujo de la lógica a tres construcciones: secuencia, condición y repetición.

Los fundamentos del diseño en el nivel de componentes para el software tradicional[7] se establecieron a principios de la década de 1960 y se formalizaron con el trabajo de Edsger Dijkstra *et al.* ([Boh66], [Dij65] y [Dij76b]). A finales de esa época, ellos propusieron el empleo de un conjunto de construcciones lógicas restringidas con las que pudiera elaborarse cualquier programa.

6 Los componentes del contenido también pueden volverse a utilizar en otras *webapps*.

7 Un componente tradicional de software implementa un elemento de procesamiento abocado a una función o subfunción en el dominio del problema, o cierta capacidad en el dominio de la infraestructura. Los componentes tradicionales, llamados con frecuencia módulos, procedimientos o subrutinas, no incluyen datos en la misma forma en la que lo hacen los componentes orientados a objetos.

www.FreeLibros.me

Las construcciones ponían el énfasis en el "mantenimiento del dominio funcional". Es decir, cada construcción tenía una estructura lógica predecible que se introducía al principio y salía por el final, lo que permitía que el lector siguiera el flujo del procedimiento con más facilidad.

Las construcciones son secuencia, condición y repetición. La *secuencia* implementa pasos de procesamiento que son esenciales en la especificación de cualquier algoritmo. La *condición* proporciona el medio para seleccionar un procesamiento con base en algún suceso lógico y la *repetición* permite la ejecución de lazos. Estas tres construcciones son fundamentales para la *programación estructurada*, técnica importante del diseño en el nivel de componentes.

Las construcciones estructuradas fueron propuestas para limitar el diseño del software orientado al procedimiento a un número pequeño de estructuras lógicas predecibles. La medición de la complejidad (véase el capítulo 23) indica que el uso de las construcciones estructuradas reduce la complejidad del programa y con ello mejora la legibilidad y la facilidad de realizar pruebas y de dar mantenimiento. El uso de un número limitado de construcciones lógicas también contribuye a un proceso de comprensión humana que los sicólogos denominan *lotificación*. Para entender este proceso, considere el lector la forma en la que está leyendo esta página. No necesita leer las letras en lo individual, sino reconocer patrones o grupos de letras que forman palabras o frases. Las construcciones estructuradas son grupos lógicos que permiten al lector reconocer elementos de procedimiento de un módulo, en vez de leer el diseño o el código línea por línea. La comprensión mejora cuando se encuentran patrones lógicos que es fácil reconocer.

Con tan sólo las tres construcciones estructuradas es posible diseñar e implantar cualquier programa, sin importar el área de aplicación o complejidad técnica. Sin embargo, debe notarse que su empleo dogmático con frecuencia ocasiona dificultades prácticas. En la sección 10.5.1 se estudia esto con más detalles.

### 10.5.1 Notación gráfica de diseño

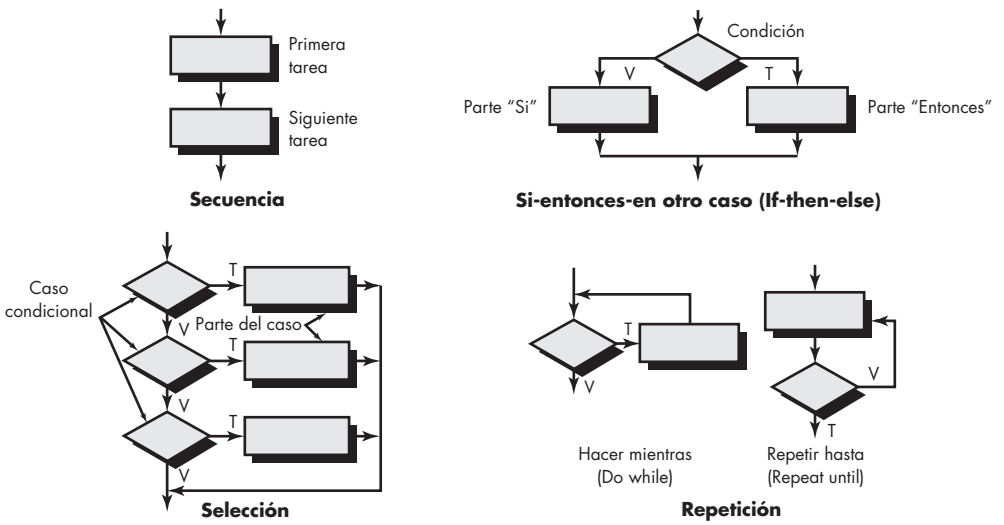
"Una imagen vale más que mil palabras", pero es importante saber de qué imagen se trata y cuáles serían las mil palabras. No hay duda de que herramientas gráficas, como el diagrama UML de actividades o el diagrama de flujo, constituyen patrones gráficos útiles que ilustran fácilmente detalles de procedimiento. No obstante, si se hace mal uso de las herramientas gráficas, surge una imagen equivocada que conduce al software equivocado.

El diagrama de actividades permite representar la secuencia, condición y repetición —todos los elementos de que consta la programación estructurada— y es descendiente de un diseño gráfico anterior (que todavía se utiliza mucho) llamado *diagrama de flujo*. Como cualquier diagrama de actividades, el de flujo es muy simple. Se emplea un rectángulo para indicar un paso de procesamiento. Un rombo representa una condición lógica y las flechas indican el flujo del control. La figura 10.10 ilustra tres construcciones estructuradas. La *secuencia* se representa como dos cajas de procesamiento conectadas por una línea (flecha) de control. La *condición*, también llamada *si-entonces-en otro caso*, se ilustra como un rombo de decisión que, si se da el estado de "verdadero", hace que ocurra la *parte del entonces*, y si el estado es "falso", se invoca el procesamiento de la *parte en otro caso*. La *repetición* se representa con el uso de dos formas ligeramente distintas. Las pruebas *hacer mientras*, prueban una condición y ejecutan repetidamente un lazo de tareas mientras la condición sea verdadera. La parte *repetir hasta* primero ejecuta el lazo de la tarea y después prueba una condición y repite la tarea hasta que la condición se vuelve falsa. La construcción *selección* (o *caso seleccionar*) que se aprecia en la figura en realidad es una extensión de la cláusula *si-entonces-en otro caso*. Un parámetro se somete a prueba por medio de decisiones sucesivas hasta que ocurre una condición de "verdadero" y se ejecuta el procesamiento por la trayectoria de la *parte caso*.

En general, el uso dogmático de construcciones estructuradas introduce ineficiencia cuando se requiere una salida de un grupo de lazos o condiciones anidadas. Más importante aún, la complicación adicional de todas las pruebas lógicas y de la ruta de salida llega a oscurecer el

www.FreeLibros.me

**FIGURA 10.10**
**Construcciones de los diagramas de flujo**




flujo de control del software, aumenta la posibilidad de errores y tiene un efecto negativo en la legibilidad y facilidad de dar mantenimiento al software. ¿Qué hacer?

Hay dos opciones: 1) rediseñar la representación del procedimiento de modo que no se requiera que la "rama de salida" esté en una ubicación anidada en el flujo del control o 2) violar en forma controlada las construcciones estructuradas; es decir, diseñar una rama restringida fuera del flujo anidado. Es obvio que la opción 1 es el enfoque ideal, pero la 2 se consigue sin violar el espíritu de la programación estructurada.

### 10.5.2 Notación del diseño tabular

**CONSEJO**

*Úsese una tabla de decisión cuando dentro de un componente se halle un conjunto complejo de condiciones y acciones.*

En muchas aplicaciones de software, se requiere un módulo para evaluar una combinación compleja de combinaciones de condiciones y seleccionar acciones apropiadas con base en éstas. Las *tablas de decisión* [Hur83] proporcionan una notación que traduce las acciones y condiciones (descritas en la narración del procesamiento o caso de uso) a una forma tabular. Es difícil malinterpretar la tabla e incluso se puede usar como entrada legible por una máquina que la use en un algoritmo dirigido por aquélla.

En la figura 10.11 se muestra la organización de la tabla de decisión. La tabla se divide en cuatro secciones. El cuadrante superior izquierdo contiene la lista de todas las condiciones. El inferior izquierdo contiene la lista de condiciones que son posibles con base en combinaciones de las condiciones. Los cuadrantes del lado derecho forman una matriz que indica combinaciones de las condiciones y las correspondientes acciones que ocurrirán para una combinación específica de éstas. Por tanto, cada columna de la matriz se interpreta como *regla de procesamiento.* Para desarrollar una tabla de decisión se emplean los pasos siguientes:

**? ¿Cómo elaboro una tabla de decisión?**

1. Enlistar todas las acciones asociadas con un procedimiento (o componente) específico.
2. Enlistar todas las condiciones (o decisiones tomadas) durante la ejecución del procedimiento.
3. Asociar conjuntos específicos de condiciones con acciones específicas, con la eliminación de las combinaciones o con condiciones imposibles; de manera alternativa, desarrollar toda posible permutación de las condiciones.
4. Definir reglas indicando qué acciones suceden para un conjunto dado de condiciones.

Para ilustrar el uso de una tabla de decisión, considere el siguiente extracto de un caso de uso informal propuesto para el sistema del taller de impresión:

www.FreeLibros.me

**FIGURA 10.11**
**Nomenclatura de una tabla de decisión**

| | Reglas | | | | | |
|---|---|---|---|---|---|---|
| **Condiciones** | **1** | **2** | **3** | **4** | **5** | **6** |
| Cliente regular | T | T | | | | |
| Cliente plateado | | | T | T | | |
| Cliente dorado | | | | | T | T |
| Descuento especial | F | T | F | T | F | T |
| **Acciones** | | | | | | |
| Sin descuento | ✓ | | | | | |
| Aplicar 8% de descuento | | | ✓ | ✓ | | |
| Aplicar 15% de descuento | | | | | ✓ | ✓ |
| Aplicar un porcentaje adicional de descuento | | ✓ | | ✓ | | ✓ |

Se definen tres tipos de clientes: regular, plateado y dorado (que se asignan de acuerdo con la cantidad de negocios que realice el cliente con el taller de impresión en un periodo de 12 meses). Un cliente regular recibe tarifas normales por la impresión y entrega. Uno plateado obtiene un descuento de 8 por ciento sobre todas sus compras y es colocado por delante de todos los clientes regulares en la fila de trabajos. Un cliente dorado recibe 15 por ciento de descuento sobre los precios de lista y es puesto adelante de los clientes regulares y de los plateados en la fila de trabajos. Además de los descuentos anteriores, se aplica un descuento especial de *x* porcentaje a cualquier cliente según el criterio de la administración.

La figura 10.11 ilustra una representación de tabla de decisión del caso de uso anterior. Cada una de las seis reglas indica una de las seis condiciones posibles. Por regla general, la tabla de decisión se usa de manera eficaz para dar otra notación de diseño orientado al procedimiento.

### 10.5.3 Lenguaje de diseño del programa

El *lenguaje de diseño del programa* (LDP), también llamado *castellano estructurado* o *seudocódigo*, incorpora la estructura lógica de un lenguaje de programación y la expresividad de forma libre de un lenguaje natural (como el castellano). Se incrusta el texto de la narración (en castellano) en una sintaxis de programación parecida al idioma. Para mejorar la aplicación del LDP se utilizan herramientas automatizadas (ver [Cai03]).

Una sintaxis básica de LDP debe incluir construcciones para: definir los componentes, describir la interfaz, hacer la declaración de los datos, estructurar bloques, hacer construcciones condicionales, de repetición y de entrada y salida (E/S). Debe observarse que el LDP se amplía a fin de que incluya palabras clave para hacer un procesamiento de tareas múltiples o concurrentes, manejar interrupciones, sincronía entre procesos y muchas otras características. El diseño de la aplicación para el que se utilizará el LDP es lo que debe dictar la forma final para el lenguaje del diseño. En el ejemplo siguiente se presenta el formato y semántica de algunas de estas construcciones de LDP.

Para ilustrar el uso del LDP, piense en un diseño orientado al procedimiento para la función de seguridad de *CasaSegura* que se estudió en secciones anteriores. El sistema vigila alarmas de incendio, humo, robo, inundación y temperatura (por ejemplo, si el sistema de calefacción falla cuando el propietario está fuera durante el invierno), genera un sonido de alarma y hace una llamada al servicio de vigilancia con la generación de un mensaje de voz sintetizada.

Recuerde que el LDP *no* es un lenguaje de programación. Se adapta según se requiera sin preocuparse por errores de sintaxis. Sin embargo, el diseño para el software de vigilancia ten-

www.FreeLibros.me

dría que revisarse (¿ve algún problema el lector?) y mejorarse más para poder escribir código. El siguiente LDP[8] muestra la elaboración del diseño del procedimiento para una versión temprana de un componente de administración de una alarma.

```
componente AdministrarAlarma;
El objetivo de este componente es administrar los interruptores del panel de control y las entradas desde los
sensores por tipo, y actuar en cualquier condición de alarma que se encuentre.
    Establecer valores de inicio para EstadodelSistema (valor devuelto), todos los grupos de datos inician
    todos los puertos del sistema y reinician todo el hardware
    verificar los InterruptoresdelPaneldeControl (ipc)
        si ipc = "probar" entonces invocar que alarma se coloque en "on"
        si ipc = "AlarmaApagada" entonces invocar que alarma se coloque en "off"
        si ipc = "NuevoValorAsignado" entonces invocar EntradadelTeclado
        si ipc = "AlarmaContraRoboApagada" invocar DesactivarAlarma;
        •
        •
        •
        prestablecido para ipc = ninguno
    reiniciar todos los ValordeSeñal e interruptores
    hacer para todos los sensores
        invocar el procedimiento VerificarSensor con la devolución de ValordeSeñal
        si ValordeSeñal > asignar [TipodeAlarma]
            entonces MensajeTelefónico = mensaje [TipodeAlarma]
                iniciar SonidodeAlarma en "on" para SegundosdeTiempo de alarma
                iniciar estado del sistema = "CondicióndeAlarma"
                ComienzaPar
                    invocar procedimiento de alarma con "on", SegundosdeTiempo de alarma;
                    invocar procedimiento de teléfono con TipodeAlarma, NúmeroTelefónico
                TerminaPar
            en otro caso salta
        TerminaSi
    TerminaHacerPara
termina AdministrarAlarma
```

Observe que el diseñador del componente **AdministrarAlarma** usó la construcción `ComienzaPar... TerminaPar`, que especifica un bloque paralelo. Todas las tareas especificadas dentro de `ComienzaPar` se ejecutan en paralelo. En este caso, no se consideran los detalles de la implantación.

## 10.6 Desarrollo basado en componentes

En el contexto de la ingeniería de software, la reutilización es una idea tanto antigua como nueva. Los programadores han reutilizado ideas, abstracciones y procesos desde los primeros días de la computación, pero el enfoque inicial que aplicaban era *ad hoc*. Hoy en día, los sistemas basados en computadoras, complejos y de alta calidad, deben elaborarse en plazos muy cortos y demandan un enfoque más organizado de la reutilización.

8 El nivel de detalle representado por el LDP se define localmente. Algunas personas prefieren la descripción más natural orientada al lenguaje, mientras que otras escogen algo más cercano al código.

www.FreeLibros.me

La *ingeniería de software basada en componentes* (ISBC) es un proceso que pone el énfasis en el diseño y construcción de sistemas basados en computadora que emplean "componentes" reutilizables de software. Clements [Cle95] describe la ISBC del modo siguiente:

> La ISBC adopta la filosofía "compra, no hagas" propuesta por Fred Brooks y otros. Del mismo modo que las subrutinas de los primeros tiempos liberaron al programador de pensar en los detalles, la ISBC traslada el énfasis de la programación a la combinación de sistemas de software. La implantación ha dado paso a la integración como el aspecto importante.

Pero surgen varias preguntas. ¿Es posible construir sistemas complejos con el ensamble de componentes reutilizables de software procedentes de un catálogo? ¿Se logra esto en forma eficaz en cuanto a costo y tiempo? ¿Pueden establecerse incentivos apropiados que estimulen a los ingenieros de software a reutilizar, más que a reinventar? ¿La dirección está dispuesta a incurrir en los gastos adicionales asociados a la creación de componentes de software reutilizables? ¿Es posible crear la biblioteca de componentes necesarios para conseguir la reutilización en forma tal que sea accesible a quienes los necesitan? ¿Quienes necesitan los componentes que ya existen pueden encontrarlos?

Cada vez más, la respuesta a estas preguntas es "sí". En el resto de esta sección se estudian algunos aspectos que deben tomarse en cuenta para hacer que la ISBC tenga éxito en una organización de ingeniería de software.

**Cita:**

"La ingeniería del dominio trata de encontrar aspectos en común en los sistemas a fin de identificar los componentes aplicables a muchos de ellos…"

**Paul Clements**

### 10.6.1 Ingeniería del dominio

El objetivo de la *ingeniería del dominio* es identificar, construir, catalogar y diseminar un conjunto de componentes de software que sean aplicables al software existente y al del futuro en un dominio particular de aplicaciones.[9] La meta general es establecer mecanismos que permitan que los ingenieros de software compartan dichos componentes —los reutilicen— cuando trabajen en sistemas nuevos o ya existentes. La ingeniería del dominio incluye tres actividades principales: análisis, construcción y diseminación.

**CONSEJO**

*El proceso de análisis que se estudia en esta sección se centra en los componentes reutilizables. Sin embargo, el análisis de sistemas con componentes comerciales separados (CCS) completos (por ejemplo, aplicaciones de comercio electrónico y automatización de la fuerza de ventas) también forma parte del análisis del dominio.*

El enfoque general del *análisis del dominio* se caracteriza con frecuencia dentro del contexto de la ingeniería de software orientada a objetos. Los pasos de este proceso se definen como sigue:

1. Definir el dominio que se va a investigar.
2. Clasificar los aspectos extraídos del dominio.
3. Reunir una muestra representativa de aplicaciones en el dominio.
4. Analizar cada aplicación en la muestra y definir clases de análisis.
5. Desarrollar un modelo de los requerimientos para las clases.

Es importante observar que el análisis del dominio es aplicable a cualquier paradigma de la ingeniería de software y se aplica tanto al desarrollo convencional como al orientado a objetos.

### 10.6.2 Calificación, adaptación y combinación de los componentes

La ingeniería del dominio genera la biblioteca de componentes reutilizables que se requieren para la práctica de la ingeniería de software basada en componentes. Algunos de estos componentes reutilizables se desarrollan en la propia empresa, mientras que otros provienen de aplicaciones existentes y otros más se adquieren de terceras personas.

Desafortunadamente, la existencia de componentes reutilizables no garantiza que éstos se integren con facilidad o eficacia en la arquitectura escogida para una nueva aplicación. Es por

9 En el capítulo 9 se mencionan géneros arquitectónicos que identifican dominios específicos de aplicación.

www.FreeLibros.me

esta razón que cuando se propone el empleo de un componente se aplica una secuencia de acciones de desarrollo basadas en componentes.

**Calificación de componentes.** La calificación de componentes garantiza que un componente candidato ejecute la función requerida, "encaje" en forma adecuada en el estilo arquitectónico (véase el capítulo 9) especificado para el sistema y tenga las características de calidad (rendimiento, confiabilidad, usabilidad) que se requieren para la aplicación.

La descripción de la interfaz proporciona información útil sobre la operación y uso de un componente de software, pero no toda la que se requiere para determinar si el componente propuesto es en verdad capaz de reutilizarse con eficacia en una nueva aplicación. Entre los muchos factores que se consideran durante la calificación de un componente se encuentran los que se mencionan a continuación [Bro96]:

**¿Cuáles son los factores que se consideran durante la calificación de componentes?**

- Aplicación de programación de la interfaz (API).
- Herramientas de desarrollo e integración requeridas por el componente.
- Requerimientos durante la puesta en marcha, incluidos el uso de recursos (como la memoria de almacenamiento), sincronía o velocidad y protocolo de redes.
- Requerimientos de servicio, incluidos interfaces del sistema operativo y apoyo de otros componentes.
- Características de seguridad, incluidos controles de acceso y protocolo de autentificación.
- Suposiciones incrustadas en el diseño, incluido el empleo de algoritmos, numéricos o no, específicos.
- Manejo de excepciones.

Cada uno de estos factores es relativamente fácil de evaluar cuando se proponen componentes reutilizables desarrollados en la propia empresa. Si durante el desarrollo de un componente se aplican buenas prácticas de ingeniería de software, es posible responder las preguntas que están implícitas en la lista. Sin embargo, es mucho más difícil determinar los trabajos internos de los componentes comerciales separados (CCS) o de los adquiridos a terceras personas porque la única información disponible es la especificación de la interfaz en sí misma.

**Adaptación de componentes.** En la situación ideal, la ingeniería del dominio crea una biblioteca de componentes que se integra con facilidad en la arquitectura de una aplicación. La implicación de una "integración fácil" es que 1) se han implementado métodos consistentes de administración de recursos para todos los componentes que hay en la biblioteca; 2) para todos los componentes existen actividades comunes, tales como administración de datos, y 3) se han implementado de manera consistente interfaces dentro de la arquitectura y con el ambiente externo.

**CONSEJO**

*Además de evaluar si se justifica el costo de adaptación para la reutilización, también debe analizarse si es posible lograr la funcionalidad que se requiere de manera rentable.*

En realidad, incluso si un componente ha sido calificado para el uso dentro de una aplicación de arquitectura, surgen conflictos en una o más de las áreas mencionadas. Para evitar estos conflictos, en ocasiones se emplea una técnica de adaptación llamada *envoltura de componentes* [Bro96]. Cuando un equipo de software tiene acceso total al diseño y código interno de un componente (que con frecuencia no es el caso a menos que se utilicen componentes CCS de fuente abierta), se aplica la *envoltura de caja blanca*. Como su contraparte en las pruebas del software (véase el capítulo 18), la envoltura de caja blanca estudia los detalles del procesamiento interno del componente y hace modificaciones en el nivel de código para eliminar cualquier conflicto. Se aplica la *envoltura de caja gris* cuando la biblioteca de componentes proporciona un lenguaje de extensión del componente o API que permite que los conflictos se eliminen o se anulen. La *envoltura de caja negra* requiere la introducción de procesamiento previo y posterior en la interfaz del componente para eliminar o anular los conflictos. Debe determinarse si se justifica el

www.FreeLibros.me

esfuerzo para envolver de manera adecuada un componente o si en vez de ello debe hacerse la ingeniería de un componente especializado (diseñado para eliminar los conflictos que surjan).

**Combinación de componentes.** La tarea de combinar componentes ensambla componentes calificados, adaptados y con la ingeniería necesaria para incluirse en la arquitectura establecida para una aplicación. Para lograr esto, debe establecerse una infraestructura que ligue los componentes en un sistema operativo. La infraestructura (por lo general una biblioteca de componentes especializados) provee un modelo para la coordinación de componentes y servicios específicos que permite que éstos se coordinen entre sí y ejecuten tareas comunes.

Como el efecto potencial de la reutilización y la ISBC en la industria del software es enorme, varias compañías y consorcios han propuesto estándares para el software de los componentes.[10]

**WebRef**

En la dirección **ww.omg.org** se encuentra la información más reciente sobre la arquitectura de transacciones de objetos comunes solicitados (GAO/ATOCS).

**GAO/ATOCS.** El Grupo de Administración de Objetos publicó un documento titulado *arquitectura de intercambio de objetos comunes solicitados* (GAO/ATOCS). Un intercambio de objetos solicitados (IOS) proporciona varios servicios que permiten que los componentes reutilizables (objetos) se comuniquen con otros componentes, sin importar su ubicación dentro de un sistema.

**WebRef**

La información más reciente sobre MCO y .NET se obtiene en la dirección **www.microsoft.com/COM** y **msdn2.microsoft.com/en-us/netframeworkdefault.aspx.**

**MCO de Microsoft y .NET.** Microsoft desarrolló un *modelo de componentes de objetos* (MCO) que proporciona las especificaciones para el uso de componentes producidos por varios vendedores dentro de un campo de aplicación y que son compatibles con el sistema operativo Windows. Desde el punto de vista de la aplicación, "la atención no se pone en la forma de implementar [los objetos MCO], sino en el hecho de que el objeto tiene una interfaz que es compatible con el sistema y que usa el sistema de componentes para comunicarse con otros objetos MCO" [Har98a]. La estructura .NET de Microsoft incluye MCO y da una biblioteca de clases reutilizables que cubre una amplia variedad de dominios de aplicación.

**WebRef**

La información más reciente sobre JavaBeans se obtiene en **java.sun.com/products/javabeans/docs/**.

**Componentes JavaBeans de Sun.** El sistema de componentes JavaBeans es una infraestructura de la ISBC, portátil e independiente de la plataforma, desarrollada con el uso del lenguaje de programación Java. El sistema de componentes JavaBeans incluye un conjunto de herramientas llamado *Kit de Desarrollo Bean* (KDB) que permite que los desarrolladores 1) analicen la forma en la que funcionan los Beans (componentes) existentes; 2) personalicen su comportamiento y apariencia; 3) establezcan mecanismos para la coordinación y comunicación; 4) desarrollen Beans especiales para el uso en una aplicación específica, y 5) prueben y evalúen el comportamiento Bean.

Ninguno de estos estándares domina la industria. Aunque muchos desarrolladores se han estandarizado teniendo como modelo uno, es probable que las organizaciones grandes de software elijan el empleo de un estándar con base en las categorías y plataformas de aplicación que se elijan.

### 10.6.3 Análisis y diseño para la reutilización

Aunque la ISBC promueve el uso de componentes de software ya existentes, hay veces en las que deben desarrollarse otros nuevos para integrarlos con CCS disponibles y con otros propios. Como estos nuevos componentes se vuelven miembros de la biblioteca de la empresa de componentes reutilizables, debe hacerse la ingeniería para su reutilización.

Los conceptos de diseño, tales como abstracción, ocultamiento, independencia de funciones, refinamiento y programación estructurada, así como los métodos orientados a objeto, pruebas,

10 Greg Olsen [Ols06] hace un análisis excelente de los esfuerzos pasados y presentes de la industria para hacer de la ISBC una realidad.

www.FreeLibros.me

aseguramiento de la calidad del software (ACS) y métodos de comprobación de la corrección (véase el capítulo 21), contribuyen a la creación de componentes de software reutilizables. En esta subsección se estudian aspectos específicos de la reutilización que son complementarios de las prácticas sólidas de la ingeniería de software.

El modelo de requerimientos se analiza para determinar qué elementos apuntan hacia componentes reutilizables ya existentes. Los elementos del modelo de los requerimientos se comparan con los componentes reutilizables en un proceso que en ocasiones se conoce como "ajuste de especificaciones" [Bel95]. Si el ajuste de especificaciones señala hacia un componente que ya existe y que se ajusta a las necesidades de la aplicación en cuestión, se extrae el componente de la biblioteca de reutilización (repositorio) y se emplea en el diseño de un nuevo sistema. Si no es posible encontrar el componente (por ejemplo, no hay coincidencia), se crea uno nuevo. Es en este punto —cuando se comienza a crear el componente nuevo— donde debe considerarse el *diseño para la reutilización* (DPR).

Como ya se dijo, el DPR requiere que se apliquen conceptos y principios sólidos del diseño de software (véase el capítulo 8). Pero las características del dominio de la aplicación también deben tomarse en cuenta. Binder [Bin93] sugiere varios aspectos clave[11] que constituyen la base del diseño para la reutilización:


*El DPR es muy difícil si hay que hacer las interfaces o integrar los componentes con sistemas heredados o con sistemas múltiples cuya arquitectura y protocolos de interfaz son incongruentes.*

**Datos estándar.** El dominio de la aplicación debe investigarse y tienen que identificarse las estructuras de datos globales estándar (como las de archivos o una base de datos completa). Todos los componentes del diseño se caracterizan para hacer uso de estas estructuras de datos estándar.

**Protocolos de interfaz estándar.** Deben establecerse tres niveles de protocolos de interfaz: la naturaleza de las interfaces intramodulares, el diseño de las interfaces externas técnicas (no humanas) y la interfaz humano-computadora.

**Plantillas de programa.** Se elige un estilo arquitectónico (véase el capítulo 9) que sirve como plantilla para el diseño de la arquitectura del nuevo software.

Una vez establecidos los datos estándar, interfaces y plantillas del programa, se tiene una estructura en la cual crear el diseño. Los componentes nuevos que conforman ésta tienen una probabilidad mayor de tener un uso posterior.

### 10.6.4 Clasificación y recuperación de componentes

Considere una biblioteca universitaria grande. Se encuentran disponibles cientos de miles de libros, revistas y otras fuentes de información. Pero para acceder a dichas fuentes debe desarrollarse un esquema de clasificación. Para navegar en este enorme volumen de información, los bibliotecarios han definido un esquema de clasificación que incluye un código de la Biblioteca del Congreso, palabras clave, nombres de los autores y otras entradas para el índice. Todos ellos permiten que el usuario encuentre fácil y rápidamente la fuente que necesita.

Ahora, considere un repositorio grande de componentes en el que se encuentran decenas de miles de componentes de software reutilizables. Pero, ¿cómo se encuentra el que se necesita? Al tratar de responder esta pregunta surge otra: ¿cómo describimos los componentes de software en términos no ambiguos y clasificables? Éstas son preguntas difíciles para las que todavía no hay una respuesta definitiva. En esta sección se estudian las direcciones actuales que permitirán que los ingenieros de software naveguen a través de las bibliotecas de la reutilización.

Un componente de software reutilizable puede describirse de muchas formas, pero la descripción ideal incluye lo que Tracz [Tra95] llama *modelo 3C*: concepto, contenido y contexto. El *concepto* de un componente de software es "la descripción de lo que hace el componente"

11 En general, las preparaciones del DPR deben tomarse como parte de la ingeniería del dominio.

www.FreeLibros.me

[Whi95]. Se describe por completo la interfaz al componente y se identifica la semántica, representada en el contexto de las condiciones previas y posteriores. El concepto debe comunicar el objetivo del componente. El *contenido* de un componente describe cómo se lleva a cabo el concepto. En esencia, el contenido es información que queda oculta a los usuarios casuales y que sólo necesitan conocer aquellos que pretenden modificar o probar el componente. El *contexto* coloca un componente de software reutilizable en su dominio de aplicabilidad. Es decir, al especificar las características conceptuales, operativas y de implantación, el contexto permite que un ingeniero de software encuentre el componente apropiado para que se cumplan los requerimientos de la aplicación.

Concepto, contenido y contexto deben traducirse en un esquema concreto de especificación para que tengan uso práctico. Se han escrito decenas de textos y artículos sobre esquemas de clasificación para componentes de software reutilizables (por ejemplo, consulte en [Cec06] el panorama de las tendencias actuales).

La clasificación permite encontrar y recuperar componentes que son candidatos a la reutilización, pero debe existir un ambiente propicio para integrarlos con eficacia. Éste tiene las características siguientes:

**? ¿Cuáles son las características clave de un ambiente de reutilización de componentes?**

- Una base de datos capaz de almacenar componentes de software y la información de clasificación necesaria para recuperarlos.
- Un sistema de administración de la biblioteca que dé acceso a la base de datos.
- Un sistema de recuperación de componentes de software (por ejemplo, un agente de solicitud de objetos) que permita que la aplicación de un cliente recupere componentes y servicios del servidor de la biblioteca.
- Herramientas de ISBC que apoyen la integración de componentes reutilizados en un diseño o implantación nuevos.

Cada una de estas funciones interactúa con los confines de una biblioteca de reutilización o se halla incrustada en ella.

La *biblioteca de reutilización* es un elemento de un repositorio mayor de software (véase el capítulo 22) y brinda herramientas para el almacenamiento de componentes de software, así

**HERRAMIENTAS DE SOFTWARE**

***ISBC***

**Objetivo:** Ayudar a modelar, diseñar, revisar e integrar los componentes de software como parte de un sistema mayor.

**Mecánica:** Las mecánicas de las herramientas varían. En general, las herramientas de ISBC ayudan en una o más de las siguientes capacidades: especificar y modelar la arquitectura del software, investigar y seleccionar los componentes de software disponibles; integrar los componentes.

**Herramientas representativas**[12]

*ComponentSource* (**www.componentsource.com**) proporciona una variedad amplia de componentes (y herramientas) de software CCS apoyada dentro de muchos estándares de componentes distintos.

*Component Manager*, desarrollado por Flashline (**www.flashline.com**), "es una aplicación que permite, promueve y mide la reutilización de componentes de software".

*Select Component Factory*, desarrollado por Select Business Solutions (**www.selectbs.com**), "es un conjunto integrado de productos para diseñar software, revisar el diseño, administrar servicios o componentes, manejar requerimientos y generar código".

*Software Through Pictures-ACD*, distribuido por Aonix (**www.aonix.com**), permite el modelado exhaustivo con el uso de UML para el modelo OMG orientado a la arquitectura, un enfoque de la ISBC abierto y neutral para el vendedor.

12 Las herramientas mencionadas aquí no son obligatorias, sólo son una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

www.FreeLibros.me

como una amplia variedad de productos finales reutilizables (especificaciones, diseños, patrones, estructuras, fragmentos de código, casos de prueba, guías del usuario, etc.). La biblioteca incluye una base de datos y las herramientas necesarias para hacer consultas y recuperar los componentes. El esquema de clasificación de componentes es la base de las consultas a la biblioteca.

**WebRef**

En la dirección **www.cbd-hq.com/** se encuentra un conjunto exhaustivo de recursos de la ISBC.

Es frecuente que las consultas se clasifiquen con el empleo del elemento de contexto del modelo 3C mencionado. Si una consulta inicial da como resultado una lista grande de componentes candidatos, se afina la consulta para depurar la lista. Después se extrae la información de concepto y contenido (una vez encontrados los componentes candidatos) que ayuden a seleccionar el componente apropiado.

## 10.7 Resumen

El proceso de diseño en el nivel de componentes incluye una secuencia de actividades que reduce poco a poco el nivel de abstracción con el que se representa el software. El diseño en el nivel de componentes ilustra en definitiva al software en un nivel de abstracción cercano al código.

Es posible adoptar tres puntos de vista diferentes en el nivel de diseño, en función de la naturaleza del software que se va a desarrollar. El enfoque orientado a objetos se centra en la elaboración de clases de diseño que provienen tanto del dominio del problema como de la infraestructura. El punto de vista tradicional mejora tres tipos diferentes de componentes o módulos: los de control, los del dominio del problema y los de la infraestructura. En ambos casos se aplican los principios y conceptos básicos del diseño que llevan a un software de alta calidad. Cuando se considera al diseño en el nivel de componentes desde un punto de vista del proceso, se llega a componentes de software reutilizables y a patrones de diseño que son elementos cruciales de la ingeniería de software basada en componentes.

Conforme se elaboran las clases, varios principios y conceptos importantes guían al diseñador. Las ideas agrupadas en el Principio Abierto-Cerrado y en el Principio de Inversión de la Dependencia, así como conceptos tales como el acoplamiento y la cohesión, guían al ingeniero de software en la construcción de componentes de software susceptibles de someterse a prueba, implantarse y recibir mantenimiento. Para hacer el diseño en el nivel de componentes en este contexto, se elaboran las clases especificando detalles de la mensajería, identificando las interfaces apropiadas, elaborando atributos y definiendo estructuras de datos que las implementen, describiendo el flujo del procesamiento dentro de cada operación y representando el comportamiento en el nivel de clase o componente. Una actividad esencial en cada caso es el diseño iterativo (rediseñar).

El diseño tradicional en el nivel de componentes requiere la representación de estructuras de datos, interfaces y algoritmos para un módulo de programa con detalle suficiente para guiar la generación del código fuente del lenguaje de programación. A fin de lograr esto, el diseñador usa una de varias notaciones de diseño que representan los detalles en el nivel de componente en un formato gráfico, tabular o basado en texto.

El diseño en el nivel de componentes para *webapps* considera tanto el contenido como la funcionalidad tal como es entregada por un sistema basado en web. El diseño del contenido en el nivel de componentes se centra en objetos de contenido y en la manera en la que se empacan para su presentación en una *webapp* al usuario final. El diseño funcional para las *webapps* se centra en funciones de procesamiento que manipulan contenido, realizan cálculos, consultan y acceden a una base de datos y establecen interfaces con otros sistemas. Se aplican todos los principios y lineamientos del diseño en el nivel de componentes.

La programación estructurada es una filosofía de diseño orientado al procedimiento que limita el número y tipo de construcciones lógicas usadas para representar los detalles algorítmi-

www.FreeLibros.me

cos. El objetivo de la programación estructurada es auxiliar al diseñador en la definición de algoritmos que sean menos complejos y por ello más fáciles de leer, probar y mantener.

La ingeniería de software basada en componentes identifica, construye, cataloga y disemina un conjunto de componentes de software en un dominio particular de aplicación. Después, estos componentes se califican, adaptan e integran para usarlos en un sistema nuevo. Los componentes reutilizables deben diseñarse dentro de un ambiente que establezca para cada dominio de aplicación estructuras de datos estándar, protocolos de interfaz y arquitecturas de programa.

## Problemas y puntos por evaluar

**10.1.** En ocasiones resulta difícil definir el término *componente*. Primero dé una definición general y luego otras más explícitas para el software orientado a objetos y para el tradicional. Por último, elija tres lenguajes de programación con los que esté familiarizado e ilustre la manera en la que cada uno define un componente.

**10.2.** ¿Por qué son necesarios los componentes de control en el software tradicional y por qué en general no se requieren en el orientado a objetos?

**10.3.** Describa con sus propias palabras el PAC. ¿Por qué es importante crear abstracciones que sirvan como interfaz entre los componentes?

**10.4.** Describa el PID con sus propias palabras. ¿Qué pasaría si un diseñador dependiera demasiado de las concreciones?

**10.5.** Seleccione tres componentes que haya desarrollado recientemente y evalúe los tipos de cohesión que presente cada uno. Si tuviera que definir el beneficio principal de la cohesión, ¿cuál sería?

**10.6.** Elija tres componentes que haya elaborado hace poco y evalúe los tipos de acoplamiento que tenga cada uno. Si definiera el principal beneficio del poco acoplamiento, ¿qué diría?

**10.7.** ¿Es razonable decir que los componentes del dominio del problema nunca deben tener acoplamiento externo? Si está de acuerdo, ¿qué tipos de componente tendrían acoplamiento externo?

**10.8.** Desarrolle 1) una clase de diseño elaborada, 2) descripciones de interfaz, 3) un diagrama de actividades para una de las operaciones dentro de la clase de diseño y 4) un diagrama de estado detallado para una de las clases de *CasaSegura* que se estudiaron en los capítulos anteriores.

**10.9.** ¿Son lo mismo el refinamiento *stepwise* y el rediseño? Si no es así, ¿en qué difieren?

**10.10.** ¿Qué es un componente de *webapp*?

**10.11.** Seleccione una parte pequeña de un programa existente (de 50 a 75 líneas de código). Separe las construcciones de programación estructurada con cuadros que dibuje alrededor de ellas en el código fuente. ¿El extracto de programa tiene construcciones que violan la filosofía de la programación estructurada? Si es así, rediseñe el código para que se apegue a las construcciones de programación estructurada. Si no es así, ¿qué observa en los cuadros que dibujó?

**10.12.** Todos los lenguajes modernos de programación implementan las construcciones de programación estructurada. Dé ejemplos de tres lenguajes de programación.

**10.13.** Seleccione un componente codificado pequeño y represéntelo con 1) un diagrama de actividades, 2) un diagrama de flujo, 3) una tabla de decisión y 4) LDP.

**10.14.** ¿Por qué es importante la "lotificación" en el proceso de revisión del diseño en el nivel de componentes?

## Lecturas adicionales y fuentes de información

En los últimos años se han publicado muchos libros sobre el desarrollo basado en componentes y acerca de su reutilización. Apperly *et al.* (*Service- and Component-Based Development*, Addison-Wesley, 2003), Heineman y Councill (*Component Based Software Engineering*, Addison-Wesley, 2001), Brown (*Large Scale Compo-*

www.FreeLibros.me

*nent-Based Development*, Prentice-Hall, 2000), Allen (*Realizing e-Business with Components*, Addison-Wesley, 2000), Herzum y Sims (*Business Component Factory*, Wiley, 1999), Allen, Frost y Yourdon (*Component-Based Development for Enterprise Systems: Applying the Select Perspective*, Cambridge University Press, 1998) cubren todos los aspectos importantes del proceso de la ISBC. Cheesman y Daniels (*UML Components*, Addison-Wesley, 2000) estudian la ISBC con énfasis en el UML.

Gao *et al.* (*Testing and Quality Assurance for Component-Based Software*, Artech House, 2006) y Gross (*Component-Based Software Testing with UML*, Springer, 2005) estudian las pruebas y el aseguramiento de la calidad en los sistemas basados en componentes.

En los años recientes han sido publicadas decenas de libros que describen estándares de la industria basados en componentes. Estas obras se dirigen a los trabajos internos de los estándares, pero también consideran muchos temas importantes de la ISBC.

El trabajo de Linger, Mills y Witt (*Structured Programming-Theory and Practice*, Addison-Wesley, 1979) permanece como el análisis definitivo del tema. El texto contiene un buen LDP y también estudios detallados de las ramificaciones de la programación estructurada. Otros libros que se centran en el diseño orientado al procedimiento para sistemas tradicionales incluyen los de Robertson (*Simple Program Design*, 3a. ed., Course Technology, 2000), Farrell (*A Guide to Programming Logic and Design*, Course Technology, 1999), Bentley (*Programming Pearls*, 2a. ed., Addison-Wesley, 1999) y Dahl (*Structured Programming*, Academic Press, 1997).

Son relativamente pocos los libros dedicados en exclusivo al diseño en el nivel de componentes. En general, los libros de lenguajes de programación están dirigidos al diseño del procedimiento con cierto detalle, pero siempre en el contexto del lenguaje que trata el libro. Son cientos los títulos disponibles.

En internet hay una amplia variedad de fuentes de información sobre diseño en el nivel de componentes. En el sitio web del libro, **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, existe una lista actualizada de referencias en la red mundial que son relevantes para el diseño en el nivel de componentes.

www.FreeLibros.me

CAPÍTULO

# 11

# DISEÑO DE LA INTERFAZ DE USUARIO



Vivimos en un mundo de productos de alta tecnología, y virtualmente todos ellos —electrónica para el consumidor, equipo industrial, sistemas corporativos, sistemas militares, software de computadoras personales y *webapps*— requieren interacción humana. Si un producto ha de alcanzar el éxito, debe tener buena *usabilidad*: medición cualitativa de la facilidad y eficiencia con la que un humano emplea las funciones y características que ofrece el producto de alta tecnología.

La usabilidad importa, ya sea que una interfaz haya sido diseñada para un reproductor de música digital o para el sistema de control de armas de un avión de combate. Si los mecanismos de la interfaz están bien diseñados, el usuario se desliza por la interacción a un ritmo suave que hace que el trabajo se realice sin esfuerzo. Pero si la interfaz fue mal concebida, el usuario avanza y retrocede, y el resultado final es frustración y poca eficiencia en el trabajo.

Durante las tres primeras décadas de la era de la computación, la usabilidad no era la preocupación dominante de quienes elaboraban software. En su libro clásico sobre diseño, Donald Norman [Nor88] afirmaba que ya era tiempo de cambiar de actitud:

> Para producir tecnología que se adapte a los seres humanos, es necesario estudiar a éstos. Pero en la actualidad tendemos a estudiar sólo a la primera. El resultado es que se exige a las personas que se adapten a la tecnología. Es tiempo de que esta tendencia se revierta, es el momento de que la tecnología se adapte a las personas.

A medida que los tecnólogos estudiaban la interacción humana, surgieron dos aspectos dominantes. El primero fue que se identificaron *reglas doradas* (que se estudian en la sección 11.1). Éstas se aplican a toda interacción humana con productos de la tecnología. El segundo fue que se definió un conjunto de *mecanismos de interacción* que permitieron a los diseñadores de software construir sistemas que implantaban en forma correcta las reglas doradas. Esos meca-

## UNA MIRADA RÁPIDA

**¿Qué es?** El diseño de la interfaz de usuario crea un medio eficaz de comunicación entre los seres humanos y la computadora. Siguiendo un conjunto de principios de diseño de la interfaz, el diseño identifica los objetos y acciones de ésta y luego crea una plantilla de pantalla que constituye la base del prototipo de la interfaz de usuario.

**¿Quién lo hace?** Un ingeniero de software diseña la interfaz de usuario con la aplicación de un proceso iterativo que sigue principios de diseño predefinidos.

**¿Por qué es importante?** Si el software es difícil de usar, fuerza al usuario a cometer errores, o si frustra sus esfuerzos para alcanzar las metas, entonces no le gustará, sin que importe el poder computacional que tenga, el contenido que entregue o las funciones que ofrezca. La interfaz tiene que estar bien hecha porque moldea la percepción que el usuario tiene del software.

**¿Cuáles son los pasos?** El diseño de la interfaz de usuario comienza con la identificación de los requerimientos del usuario, la tarea y el ambiente. Una vez identificadas las tareas del usuario, se crean y analizan los escenarios para éste y se define un conjunto de objetos y acciones de la interfaz. Esto forma la base para crear una plantilla de la pantalla que ilustra el diseño gráfico y la colocación de los iconos, la definición de textos descriptivos, la especificación y títulos de las ventanas, y la especificación de aspectos mayores y menores del menú. Con el empleo de herramientas, se hace el prototipo, se implementa en definitiva el modelo del diseño y se evalúa la calidad del resultado.

**¿Cuál es el producto final?** Se crean los escenarios del usuario y se generan los formatos de la pantalla. Se desarrolla un prototipo de la interfaz y se modifica de manera iterativa.

**¿Cómo me aseguro de que lo hice bien?** Los usuarios "prueban" un prototipo de la interfaz y la retroalimentación de esta prueba se utiliza para la siguiente modificación iterativa del prototipo.



www.FreeLibros.me

nismos de interacción, llamados de manera colectiva *interfaz gráfica del usuario* (GUI), eliminaron algunos de los problemas más notables asociados con las interfaces humanas. Pero aun en un "mundo Windows", todos hemos encontrado interfaces de usuario que son difíciles de aprender y usar, que son confusas y van contra la intuición, que no perdonan y, en muchos casos, resultan frustrantes por completo. No obstante, alguien dedicó tiempo y energía a la elaboración de estas interfaces, y no es probable que su constructor haya creado dichos problemas con toda intención.

## 11.1 Las reglas doradas

En su libro sobre el diseño de la interfaz, Theo Mandel [Man97] acuñó tres *reglas doradas*:

1. Dejar el control al usuario.
2. Reducir la carga de memoria del usuario.
3. Hacer que la interfaz sea consistente.

En realidad, estas reglas doradas constituyen la base de un conjunto de principios de diseño de la interfaz de usuario que guían este aspecto tan importante del diseño del software.

### 11.1.1 Dejar el control al usuario

**Cita:**

"Es mejor diseñar la experiencia del usuario que corregirla."

**Jon Meads**

Durante una sesión para recabar los requerimientos de un nuevo y gran sistema de información, se preguntó a una usuaria clave acerca de los atributos de la interfaz gráfica basada en ventanas.

"Lo que realmente me gustaría", respondió con solemnidad, "es un sistema que lea mi mente. Que sepa lo que quiero hacer antes de que necesite hacerlo y que sea fácil para mí obtener eso que quiero. Eso es todo, sólo eso".

Mi primera reacción fue afirmar con la cabeza y sonreír, pero me detuve un momento. No había absolutamente nada descabellado en la solicitud de la usuaria. Quería un sistema que reaccionara a sus necesidades y la ayudara para que las cosas se hicieran. Deseaba controlar la computadora, no que ésta la controlara a ella.

La mayor parte de limitaciones y restricciones que impone un diseñador pretenden simplificar el modo de interacción. Pero, ¿para quién?

Como diseñador, tal vez se sienta tentado a introducir restricciones y limitantes que simplifiquen la implantación de la interfaz. El resultado puede ser una interfaz fácil de construir, pero que sea frustrante utilizar. Mandel [Man97] define cierto número de principios de diseño que permiten que el usuario tenga el control:

**Definir modos de interacción de manera que no se obligue al usuario a realizar acciones innecesarias o no deseadas.** Un modo de interacción es el estado actual de la interfaz. Por ejemplo, si en el menú de un procesador de textos se selecciona *revisar ortografía*, el software pasa al modo de revisión de la ortografía. No hay razón para obligar al usuario a permanecer en este modo si acaso desea hacer una pequeña edición del texto. El usuario debe poder entrar y salir del modo con poco o ningún esfuerzo.

**Dar una interacción flexible.** Debido a que diferentes usuarios tienen distintas preferencias para la interacción, debe darse la posibilidad de elegir. Por ejemplo, el software debe permitir que el usuario interactúe por medio de comandos introducidos con el teclado, el ratón, una pluma digitalizadora, una pantalla sensible al tacto o un mecanismo de reconocimiento de voz. Pero no todas las acciones son accesibles a través de cualquier mecanismo de interacción. Por ejemplo, piénsese en la dificultad de usar comandos del teclado (o entradas con la voz) para hacer un dibujo complicado.

www.FreeLibros.me

**Permitir que la interacción del usuario sea interrumpible y también reversible.** El usuario debe poder suspender la secuencia de su trabajo (aun cuando consista en una secuencia de acciones) para hacer otra cosa (sin perder el trabajo realizado hasta ese momento). También debe poder "deshacer" cualquier acción.

**Facilitar la interacción a medida que aumenta la habilidad y permitir que aquélla se personalice.** Es frecuente que los usuarios realicen la misma secuencia de interacciones en forma repetida. Es benéfico diseñar un mecanismo de "macros" que permita que un usuario avanzado personalice la interfaz para facilitar la interacción.

**Cita:**

"Siempre deseé que mi computadora fuera tan fácil de usar como mi teléfono. Mi deseo se ha vuelto realidad. Ya no sé cómo se utiliza mi teléfono."

Bjarne Stronstrup (iniciador de C++)

**Ocultar los tecnicismos internos al usuario ocasional.** La interfaz de usuario debe introducirlo al mundo virtual de la aplicación. El usuario no debe tener que preocuparse del sistema operativo, de las funciones de administración de archivos ni de ninguna otra tecnología de computación secreta. En esencia, la interfaz no debe requerir que el usuario interactúe en un nivel "interno" de la máquina (nunca debería tener que escribir comandos del sistema operativo desde una aplicación de software).

**Diseñar la interacción directa con objetos que aparezcan en la pantalla.** El usuario tiene la sensación de control cuando puede manipular los objetos que se necesitan a fin de ejecutar un trabajo en la misma forma en la que lo haría si el objeto fuera algo físico. Por ejemplo, una interfaz de aplicación que le permita "estirar" un objeto (modificar su tamaño) es una implementación de manipulación directa.

### 11.1.2 Reducir la necesidad de que el usuario memorice

Entre más cosas tenga que recordar el usuario, más fácil será que cometa errores al interactuar con el sistema. Es por esto que una interfaz de usuario bien diseñada no sobrecarga la memoria del usuario. Siempre que sea posible, el sistema debe "recordar" la información pertinente y ayudar al usuario con un escenario de interacción que lo ayude a recordar. Mandel [Man97] define los siguientes principios de diseño que permiten que una interfaz reduzca la necesidad de que el usuario memorice:

**Reducir la demanda de memoria de corto plazo.** Cuando los usuarios se involucran en tareas complejas, la demanda de memoria de corto plazo es significativa. La interfaz debe diseñarse para disminuir la necesidad de recordar acciones, entradas y resultados del pasado. Esto se logra dando claves visuales que permitan al usuario reconocer acciones anteriores, en lugar de que tenga que recordarlas.

**Hacer que lo preestablecido sea significativo.** Lo que al principio se dé por preestablecido debe tener sentido para el usuario promedio, pero éste debería poder especificar sus preferencias individuales. Sin embargo, debe disponerse de la opción de "reiniciar" para restablecer los valores originales.

**Definir atajos que sean intuitivos.** Cuando se utilice nemotecnia para ejecutar una función del sistema (como la secuencia Ctrl-B para invocar la función de buscar), debe estar ligada con la acción, de modo que sea fácil de recordar (por ejemplo, con la primera letra de la tarea que se va a realizar).

**La distribución visual de la interfaz debe basarse en una metáfora del mundo real.** Por ejemplo, un sistema de pagos debe usar una metáfora de chequera y talonario que guíe al usuario a través del proceso de pago. Esto permite que el usuario se base en claves visuales que comprende bien, en vez de tener que memorizar una secuencia críptica de interacciones.

**Revelar información de manera progresiva.** La interfaz debe estar organizada de manera jerárquica. Es decir, la información acerca de una tarea, objeto o comportamiento debe presen-

www.FreeLibros.me

tarse primero en un nivel de generalización elevado. Después de que con el ratón el usuario manifieste interés, deben darse más detalles. Un ejemplo, común para muchas aplicaciones de procesamiento de textos, es la función de subrayar. La función en sí es una de varias en el menú *estilo del texto*. No obstante, no se enlista cada una de las herramientas para subrayar. El usuario debe hacer *clic* en la opción de subrayar; después se presentan todas las opciones para esta función (una raya, doble raya, línea punteada, etcétera).

**CASASEGURA**

### *Violación de la regla dorada de la interfaz de usuario*

**La escena:** Cubículo de Vinod, cuando comienza el diseño de la interfaz de usuario.

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

**La conversación:**

**Jamie:** He estado pensando en la interfaz de la función de vigilancia.

**Vinod (sonríe):** Es bueno pensar.

**Jamie:** Creo que tal vez podamos simplificar algunas cosas.

**Vinod:** ¿Qué quieres decir?

**Jamie:** Bueno, qué tal si eliminamos el plano de la casa. Es bueno, pero va a requerir mucho esfuerzo de desarrollo. En vez de eso, puede pedirse al usuario que especifique la cámara que quiere ver y que luego despliegue el video en una ventana.

**Vinod:** ¿Cómo recordaría el propietario cuántas cámaras hay y dónde están?

**Jamie (algo irritado):** Él es el propietario; debe saberlo.

**Vinod:** ¿Y si no es así?

**Jamie:** Debería.

**Vinod:** Eso no es lo que estamos discutiendo… ¿qué pasa si lo olvida?

**Jamie:** Bueno, podríamos darle una lista de cámaras y sus ubicaciones.

**Vinod:** Es posible, pero, ¿por qué debería pedir una lista?

**Jamie:** Bueno, damos la lista la pida o no.

**Vinod:** Eso está mejor. Al menos no tendrá que recordar cosas que le podemos dar.

**Jamie (piensa unos instantes):** Pero, ¿te gusta o no el plano de la casa?

**Vinod:** Mmm.

**Jamie:** ¿Cuál piensas que le agrade más a mercadotecnia?

**Vinod:** Bromeas, ¿verdad?

**Jamie:** No.

**Vinod:** Mmm… el plano… adoran las características bonitas de los productos… no les importa cuál es más fácil de elaborar.

**Jamie (suspira):** Bien, quizá hagamos ambos prototipos.

**Vinod:** Buena idea… así dejamos que el cliente decida.

### 11.1.3 Hacer consistente la interfaz

**Cita:**

"Las cosas que parezcan diferentes deben actuar en forma diferente. Las cosas que parezcan iguales deben actuar igual."

Larry Marine

La interfaz debe presentar y obtener información en forma consistente. Esto implica: 1) que toda la información se organice de acuerdo con reglas de diseño que se respeten en todas las pantallas desplegadas, 2) que los mecanismos de entrada se limiten a un conjunto pequeño usado en forma consistente en toda la aplicación, y 3) que los mecanismos para pasar de una tarea a otra se definan e implementen de modo consistente. Mandel [Man97] define varios principios de diseño que ayudan a que la interfaz tenga consistencia:

**Permita que el usuario coloque la tarea en curso en un contexto significativo.** Muchas interfaces implementan capas complejas de interacciones con decenas de imágenes en la pantalla. Es importante dar indicadores (títulos en las ventanas, iconos gráficos, código de colores consistente, etc.) que permitan al usuario conocer el contexto del trabajo en curso. Además, debe poder determinar de dónde viene y qué alternativas hay para hacer la transición a una nueva tarea.

**Mantener la consistencia en toda la familia de aplicaciones.** Todas las aplicaciones (o productos) que hay en un grupo deben implementar las mismas reglas de diseño a fin de que se mantenga la consistencia en toda la interacción.

www.FreeLibros.me

**Si los modelos interactivos anteriores han creado expectativas en el usuario, no haga cambios a menos de que haya una razón ineludible para ello.** Una vez que una secuencia interactiva en particular se ha convertido en un estándar (como el uso de alt-G para guardar un archivo), el usuario la espera en toda aplicación que emplea. Un cambio (como utilizar alt-G para invocar la función de modificar la escala) generará confusión.

Los principios de diseño de la interfaz analizados en esta sección y en las anteriores dan una guía básica. En las que siguen, el lector aprenderá acerca del proceso de diseño de la interfaz en sí.

INFORMACIÓN

### *Usabilidad*

En un artículo útil sobre usabilidad, Larry Constantine [Con95] plantea una pregunta significativa sobre el tema: "En definitiva, ¿qué quieren los usuarios?" Y responde así:

> Lo que los usuarios desean son buenas herramientas. Todos los sistemas de software, desde sistemas operativos y lenguajes hasta aplicaciones de entrada de datos y apoyo a la toma de decisiones, sólo son herramientas. Los usuarios finales esperan lo mismo que nosotros de las herramientas que usamos. Quieren sistemas fáciles de aprender y que les ayuden a hacer su trabajo. Quieren software que no los haga más lentos, que no tenga trucos o los confunda; eso no significa que sea más fácil cometer errores o más difícil terminar el trabajo.

Constantine afirma que la usabilidad no proviene de la estética, de mecanismos de interacción avanzados o de interfaces inteligentes. En vez de eso, se obtiene cuando la arquitectura de la interfaz se ajusta a las necesidades de las personas que la emplearán.

Es ilusorio llegar a una definición formal de usabilidad. Donahue *et al*. [Don99] la definen de la manera siguiente: "La usabilidad es una medida de cuán bien un sistema de cómputo […] facilita el aprendizaje, ayuda a quienes lo emplean a recordar lo aprendido, reduce la probabilidad de cometer errores, les permite ser eficientes y los deja satisfechos con el sistema."

La única forma de determinar si existe "usabilidad" en un sistema que se construye es evaluarla o probarla. Los usuarios interactúan con el sistema y responden las preguntas siguientes [Con95]:

- ¿El sistema es utilizable sin ayuda o enseñanza continua?
- ¿Las reglas de interacción ayudan a un usuario preparado a trabajar con eficiencia?
- ¿Los mecanismos de interacción se hacen más flexibles a medida que los usuarios conocen más?
- ¿Se ha adaptado el sistema al ambiente físico y social en el que se usará?
- ¿El usuario está al tanto del estado del sistema? ¿Sabe en todo momento dónde está?
- ¿La interfaz está estructurada de manera lógica y consistente?
- ¿Los mecanismos, iconos y procedimientos de interacción son consistentes en toda la interfaz?
- ¿La interacción prevé errores y ayuda al usuario a corregirlos?
- ¿La interfaz es tolerante a los errores que se cometen?
- ¿Es sencilla la interacción?

Si cada una de estas preguntas obtiene un "sí" como respuesta, es probable que se haya logrado la usabilidad.

Entre los muchos beneficios mensurables que se obtienen de un sistema usable se encuentran los siguientes [Don99]: mayores ventas y más satisfacción del consumidor, ventaja competitiva, mejores evaluaciones en los medios, recomendaciones de boca en boca, menores costos de apoyo, más productividad del usuario final, menos costos de capacitación y documentación, y disminución de la probabilidad de litigios por parte de clientes insatisfechos.

## 11.2 ANÁLISIS Y DISEÑO DE LA INTERFAZ DE USUARIO

**WebRef**

En la dirección **www.useit.com** se encuentra una excelente fuente de información acerca del diseño de la interfaz de usuario.

El proceso general de análisis y diseño de la interfaz de usuario comienza con la creación de diferentes modelos del funcionamiento del sistema (según se percibe desde fuera). Se empieza delineando las tareas orientadas al usuario —y a la computadora— que se requieren a fin de obtener el funcionamiento del sistema, para luego considerar los aspectos que se aplican a todos los diseños de interfaz. Se emplean herramientas para hacer prototipos e implementar el modelo del diseño, y los usuarios finales evalúan la calidad.

### 11.2.1 Análisis y modelos del diseño de la interfaz

Cuando se analiza y diseña la interfaz de usuario, entran en juego cuatro diferentes modelos. Un ingeniero (o el encargado del software) establece un *modelo de usuario*, el ingeniero de soft-

www.FreeLibros.me

**Cita:**

"Si hay un 'truco' en ella, la interfaz de usuario está acabada."

**Douglas Anderson**

ware crea un *modelo del diseño*, el usuario final desarrolla una imagen mental que frecuentemente se nombra *modelo mental* o *percepción del sistema*, y los implementadores del sistema crean un *modelo de implementación*. Desafortunadamente, cada uno de estos modelos difiere en forma significativa. El papel del diseñador de la interfaz es conciliar estas diferencias y obtener una representación consistente de la interfaz.

El modelo del usuario establece el perfil de los usuarios finales del sistema. En su columna introductoria sobre el "diseño centrado en el usuario", Jeff Patton [Pat07] afirma lo siguiente:

> La verdad es que los diseñadores y desarrolladores —incluido yo— piensan con frecuencia en los usuarios finales. Sin embargo, en ausencia de un modelo mental fuerte de usuarios específicos, los sustituimos con nosotros. Esto no es centrarse en el usuario, sino en uno mismo.

Para construir una interfaz de usuario eficaz, "todo diseño debe comenzar con la comprensión de los usuarios que se busca, lo que incluye los perfiles de edad, género, condiciones físicas, educación, antecedentes culturales o étnicos, motivación, metas y personalidad" [Shn04]. Además, los usuarios se clasifican como:

**CONSEJO**

*Incluso un usuario principiante quiere atajos; aun los usuarios frecuentes conocedores en ocasiones necesitan ser guiados. Hay que darles lo que necesitan.*

*Principiantes.* Sin conocimiento sintáctico[1] del sistema y poco conocimiento semántico[2] de la aplicación o uso de la computadora en general.

*Usuarios intermitentes que saben.* Con conocimiento semántico razonable de la aplicación, pero relativamente poco recuerdo de la información sintáctica necesaria para usar la interfaz.

*Usuarios frecuentes conocedores.* Con buen conocimiento semántico y sintáctico, que con frecuencia les despierta el "síndrome del usuario poderoso"; es decir, individuos que buscan atajos y modos de interacción abreviados.

**PUNTO CLAVE**

El modelo mental del usuario conforma el modo en el que éste percibe la interfaz y señala si satisface sus necesidades.

El *modelo mental* del usuario (percepción del sistema) es la imagen del sistema que los usuarios finales llevan en la cabeza. Por ejemplo, si se pidiera a un usuario de un procesador de texto en particular que describiera su operación, lo que guiaría su respuesta sería la percepción que tuviera del sistema. La exactitud de la descripción dependerá del perfil del usuario (por ejemplo, en el mejor de los casos, los principiantes darán una respuesta esquemática) y de la familiaridad general con el software en el dominio de la aplicación. Un usuario que entienda bien los procesadores de texto, pero que haya trabajado con el procesador específico una sola vez, tal vez esté más preparado para hacer una descripción más completa de su funcionamiento que el principiante que haya pasado semanas tratando de entender el sistema.

El *modelo de implementación* combina la manifestación externa del sistema basado en computadora (la vista y sensación de la interfaz) con toda la información de apoyo (libros, manuales, videos, archivos de ayuda, etc.) que describe la sintaxis y semántica de la interfaz. Cuando el modelo de la implementación y el modelo mental del usuario coinciden, quienes utilizan el software por lo general se sienten cómodos con éste y lo usan de manera eficaz. Para lograr esta "fusión" de los modelos, el modelo del diseño debe haberse desarrollado de manera que incluya la información contenida en el modelo del usuario, y el modelo de la implementación debe reflejar de manera exacta la información sintáctica y semántica de la interfaz.

**Cita:**

"… hay que poner atención en lo que los usuarios hacen, no en lo que dicen."

**Jakob Nielsen**

Los modelos descritos en esta sección son "abstracciones de lo que el usuario hace o piensa que hace o de lo que alguien piensa que debe hacerse cuando se usa un sistema interactivo" [Mon84]. En esencia, estos modelos permiten que el diseñador de la interfaz satisfaga un ele-

1 En este contexto, *conocimiento sintáctico* se refiere a la mecánica de interacción que se requiere para usar con eficacia la interfaz.

2 El término *conocimiento semántico* se refiere al sentido subyacente de la aplicación: el entendimiento de las funciones que se realizan, el significado de la entrada y salida, y las metas y objetivos del sistema.

www.FreeLibros.me

mento clave del principio más importante del diseño de la interfaz de usuario: "Conocer al usuario, conocer las tareas."

### 11.2.2 El proceso

El proceso de análisis y diseño de interfaces de usuario es iterativo y se representa con un modelo espiral similar al que se estudió en el capítulo 2. En relación con la figura 11.1, el proceso de análisis y diseño de la interfaz de usuario comienza en el interior de la espiral e incluye cuatro actividades estructurales distintas [Man97]: 1) análisis y modelado de la interfaz, 2) diseño de ésta, 3) construcción y 4) validación. La espiral que se presenta en la figura 11.1 implica que cada una de dichas tareas tendrá lugar más de una vez y que cada recorrido del contorno de la espiral representa una elaboración mayor de los requerimientos y del diseño resultante. En la mayoría de los casos, la actividad de modelado involucra la hechura de prototipos, única forma práctica de validar lo que se haya diseñado.

**Cita:**

"Es mejor diseñar la experiencia del usuario que rectificarla."

**Jon Meads**

El *análisis de la interfaz* se centra en el perfil de los usuarios que interactuarán con el sistema. Se registra el nivel de habilidad, la comprensión del negocio y la receptividad general hacia el nuevo sistema; también se definen diferentes categorías de usuarios. Se recaban los requerimientos de cada una de éstas. En esencia, se trabaja para entender la percepción del sistema (véase la sección 11.2.1) para cada clase de usuarios.

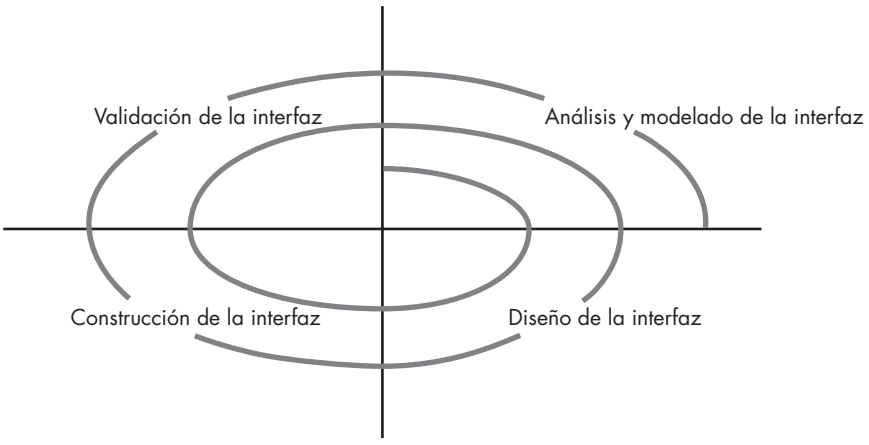
Una vez definidos los requerimientos generales, se lleva a cabo un detallado *análisis de la tarea*. Asimismo, se identifican, describen y elaboran aquellas tareas que el usuario realice para alcanzar las metas del sistema (por medio de varios recorridos de la espiral). En la sección 11.3, se estudia con más detalle el análisis de la tarea. Por último, el análisis del ambiente del usuario se centra en las características físicas del lugar de trabajo. Entre las preguntas por responder se encuentran las siguientes:

**? ¿Qué se necesita saber sobre el ambiente cuando comienza el diseño de la interfaz de usuario?**

- ¿Dónde se encontrará físicamente la interfaz?
- ¿El usuario estará sentado, de pie o haciendo otras tareas no relacionadas con la interfaz?
- ¿El hardware de la interfaz cumple las restricciones de espacio, iluminación o ruido?
- ¿Hay consideraciones especiales de factores humanos generadas por los factores ambientales?

La información recabada como parte del análisis se utiliza para crear un modelo de análisis de la interfaz. Con este modelo como base comienza la acción de diseñar.

**FIGURA 11.1**
**Proceso de diseño de la interfaz de usuario**




www.FreeLibros.me

La meta del *diseño de la interfaz* es definir un conjunto de objetos y acciones de ésta (y sus representaciones en la pantalla) que permitan al usuario efectuar todas las tareas definidas en forma tal que cumpla cada meta de la usabilidad definida para el sistema. El diseño de la interfaz se estudia con más detalle en la sección 11.4.

La *construcción de la interfaz* comienza por lo general con la creación de un prototipo que permite evaluar los escenarios de uso. A medida que avanza el proceso de diseño, se emplea un grupo de herramientas de la interfaz de usuario (véase la sección 11.5) para terminar de construirla.

La *validación de la interfaz* se centra en: 1) la capacidad de la interfaz para implementar correctamente todas las tareas del usuario, incluir todas las variaciones de éstas y alcanzar todos los requerimientos generales del usuario; 2) el grado en el que la interfaz es fácil de usar y de aprender y 3) la aceptación que tiene por parte del usuario como herramienta útil en su trabajo.

Como ya se dijo, las actividades descritas en esta sección ocurren de manera iterativa. Por esto, no es necesario intentar especificar cada uno de los detalles (del modelo de análisis o de diseño) en la primera etapa. En los pasos posteriores del proceso, se elaboran los detalles de la tarea, la información de diseño y las características de operación de la interfaz.

## 11.3 Análisis de la interfaz[3]

Un aspecto clave de todos los modelos del proceso de la ingeniería de software es éste: *entender el problema antes de tratar de diseñar una solución*. En el caso del diseño de la interfaz de usuario, entender el problema significa comprender: 1) a las personas (usuarios finales) que interactuarán con el sistema a través de la interfaz, 2) las tareas que los usuarios finales deban realizar como parte de su trabajo, 3) el contenido que se presenta como parte de la interfaz y 4) el ambiente en el que se efectuarán estas tareas. En las secciones siguientes se analizan estos elementos del análisis de la interfaz, en un intento por establecer un fundamento sólido para las tareas de diseño siguientes.

### 11.3.1 Análisis del usuario

Es probable que la frase "interfaz de usuario" sea toda la justificación que se necesita para dedicar algo de tiempo a entender al usuario antes de preocuparse por los aspectos técnicos. Ya se dijo que cada usuario tiene una imagen mental del software y que ésta puede ser diferente de la que se forman otros usuarios. Además, la imagen mental del usuario tal vez sea muy distinta de la del modelo del diseño que hizo el ingeniero de software. La única manera en la que se logra hacer converger la imagen mental y el modelo de diseño es trabajando para entender a los usuarios y la forma en la que usarán el sistema. Para ello, se utiliza información procedente de una variedad amplia de fuentes:

**¿Cómo se aprende lo que el usuario quiere de la interfaz?**

**Entrevistas.** Éste es el enfoque más directo, los miembros del equipo de software se reúnen con los usuarios para entender mejor sus necesidades, motivaciones, cultura laboral y una multitud de aspectos adicionales. Esto se logra en reuniones individuales o a través de grupos de enfoque.

**Información de ventas.** El personal de ventas habla con los usuarios de manera regular y recaba información que ayuda al equipo de software a clasificarlos y a entender mejor sus requerimientos.

3 Es razonable afirmar que esta sección debiera formar parte del capítulo 5, 6 o 7, pues en éstos se estudian aspectos del análisis de los requerimientos. Se situó aquí porque el análisis y el diseño de la interfaz se relacionan de cerca entre sí, y es frecuente que la frontera sea nebulosa.

www.FreeLibros.me

CONSEJO

*Sobre todo, dedique tiempo a hablar con los usuarios reales, pero con cuidado. Una opinión enfática no necesariamente significa que la mayoría de usuarios esté de acuerdo con ella.*

**Información de mercadotecnia.** El análisis del mercado es invaluable para la definición de segmentos del mercado y para la comprensión sobre cómo usará el software en formas sutilmente distintas cada uno de estos segmentos.

**Información de apoyo.** El equipo de apoyo habla a diario con los usuarios. Constituye la fuente de información más probable acerca de lo que funciona y lo que no, lo que les gusta a los usuarios y lo que les desagrada, qué características generan preguntas y cuáles son fáciles de usar.

Las preguntas siguientes (adaptadas de [Hac98]) ayudarán al lector a entender mejor a los usuarios de un sistema:

PUNTO CLAVE

¿Cómo se aprende sobre la demografía y las características de los usuarios finales?

- ¿Los usuarios son profesionales capacitados, técnicos, oficinistas o trabajadores de manufactura?
- ¿Qué nivel de educación formal tiene el usuario promedio?
- ¿Los usuarios son capaces de aprender mediante materiales escritos o han manifestado el deseo de recibir enseñanzas en un aula?
- ¿Los usuarios son mecanógrafos expertos o tienen fobia a los teclados?
- ¿Cuál es el rango de edades de la comunidad de usuarios?
- ¿Los usuarios estarán representados sobre todo por un género?
- ¿Cómo se compensa a los usuarios por el trabajo que realizan?
- ¿Los usuarios trabajan en un horario normal de oficina o hasta terminar el trabajo que hacen?
- ¿El software va a ser parte integral del trabajo de los usuarios o sólo lo emplearán de manera ocasional?
- ¿Cuál es el idioma principal de los usuarios?
- ¿Cuáles son las consecuencias si el usuario comete un error al emplear el sistema?
- ¿Los usuarios son expertos en el tema en el que está centrado el sistema?
- ¿Los usuarios quieren saber sobre la tecnología que hay tras la interfaz?

Una vez respondidas estas preguntas, se sabrá quiénes son los usuarios finales, qué es probable que los motive y agrade, cómo se clasificarían en distintas clases o perfiles de usuarios, cuáles son sus modelos mentales del sistema y cómo debe caracterizarse la interfaz de usuario para que satisfaga sus necesidades.

### 11.3.2 Análisis y modelado de la tarea

La meta del análisis de la tarea es responder las siguientes preguntas:

PUNTO CLAVE

La meta del usuario es cumplir una o más de las tareas a través de la interfaz. Para ello, ésta debe brindar mecanismos que le permitan realizarlas.

- ¿Qué trabajo realizará el usuario en circunstancias específicas?
- ¿Qué tareas y subtareas se efectuarán cuando el usuario haga su trabajo?
- ¿Qué dominio de problema específico manipulará el usuario al realizar su labor?
- ¿Cuál es la secuencia de las tareas (el flujo del trabajo)?
- ¿Cuál es la jerarquía de las tareas?

Para responder estas preguntas, debe recurrirse a las técnicas que ya se estudiaron en el libro, pero aplicadas en este caso a la interfaz de usuario.

**Casos de uso.** En capítulos anteriores se aprendió que el caso de uso describe la manera en la que un actor (en el contexto del diseño de la interfaz de usuario, un actor siempre es una persona) interactúa con el sistema. Cuando se utiliza como parte del análisis de la tarea, el caso

www.FreeLibros.me

de uso se desarrolla con objeto de mostrar la forma en la que un usuario final lleva a cabo alguna tarea específica relacionada con el trabajo. En la mayoría de las veces, el caso de uso se escribe en un estilo informal (un simple párrafo) en primera persona. Por ejemplo, suponga que una empresa pequeña de software quiere elaborar un sistema de diseño asistido por computadora dirigido explícitamente a diseñadores de interiores. Para entender mejor la forma en la que realizan su trabajo, se pide a diseñadores reales que describan una función específica del diseño. Cuando se preguntó a un diseñador de interiores: "¿Cómo decide si se coloca mobiliario en una habitación?", respondió con el siguiente caso de uso informal:

> Comienzo por hacer un plano de la habitación, de las dimensiones y de la ubicación de puertas y ventanas. Me preocupa mucho cómo entra la luz al cuarto, la vista que hay por las ventanas (si es bella, quiero dirigir la atención hacia ese lugar), la longitud de una pared no obstruida y el flujo del movimiento por la habitación. Después me fijo en la lista de muebles que el cliente haya elegido: mesas, sillas, sofá, gabinetes; la lista de accesorios: lámparas, tapetes, pinturas, esculturas, plantas, adornos, y tomo notas acerca de cualquier deseo que mi cliente tenga. Después dibujo cada objeto de mis listas en un formato a escala de la casa. Pongo letreros en lo que haya dibujado y utilizo lápiz porque siempre las muevo. Luego hago una perspectiva (dibujo tridimensional) del cuarto para dar a mi cliente la sensación de cómo se vería.

Este caso de uso presenta la descripción básica de una tarea importante del trabajo para el sistema de diseño asistido por computadora. De él pueden extraerse tareas, objetos y el flujo general de la interacción. Además, pueden concebirse otras características del sistema que agradarían al diseñador de interiores. Por ejemplo, podría tomarse una fotografía digital a través de cada ventana de la habitación. Cuando se elabore la perspectiva, podría representarse la vista exterior real que habrá en todas ellas.

## CasaSegura

### *Casos de uso para el diseño de la interfaz de usuario*

**La escena:** Cubículo de Vinod, cuando sigue el diseño de la interfaz de usuario.

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

**La conversación:**

**Jamie:** Vi a nuestro contacto de mercadotecnia y le pedí que escribiera un caso de uso para la interfaz de vigilancia.

**Vinod:** ¿Desde el punto de vista de quién?

**Jamie:** Del propietario, ¿de quién más?

**Vinod:** También está el papel del administrador del sistema; aun si fuera el propietario, sería un punto de vista distinto. El "administrador" prepara el sistema, lo configura, elige el plano, sitúa las cámaras…

**Jamie:** Todo lo que le pedí fue que desempeñara el papel de una propietaria que quiere ver el video.

**Vinod:** Está bien. Es uno de los comportamientos principales de la interfaz de la función de vigilancia. Pero también vamos a tener que examinar el comportamiento de la administración del sistema.

**Jamie (irritado):** Tienes razón.

[Jamie sale para encontrarse con la persona de mercadotecnia. Regresa pocas horas después.]

**Jamie:** Tuve suerte, la encontré y trabajamos juntos en el caso de uso del administrador. Básicamente vamos a definir "administración" como una función aplicable a todas las demás de *CasaSegura*. Aquí está lo que obtuvimos.

[Jamie presenta el caso de uso informal a Vinod.]

**Caso de uso informal:** Quiero poder preparar o editar la plantilla del sistema en cualquier momento. Cuando preparo el sistema, selecciono una función de administración. Ésta pregunta si deseo hacer una nueva sesión o editar una ya existente. Si selecciono una nueva, el sistema presenta una pantalla de dibujo que permite dibujar el plano de la casa en una cuadrícula. Habrá iconos para paredes, ventanas y puertas, de manera que sea fácil dibujarlas. Sólo estiro los iconos a sus longitudes apropiadas. El sistema mostrará las longitudes en pies o metros (puedo elegir el sistema de medidas). Selecciono en una biblioteca sensores y cámaras, y las coloco en el plano de la casa. Etiqueto cada una o el sistema lo hace de manera automática. Puedo establecer los valores de los sensores y cámaras desde menús apropiados. Si elijo editarlos, puedo mover sensores o cámaras, agregar otros nuevos o eliminar los ya existentes, editar el plano de la casa y editar los parámetros de sensores y cámaras. En todo caso, espero que el sistema haga una comprobación consistente y que me ayude a evitar los errores.

www.FreeLibros.me

**Vinod (después de leer el escenario):** Bien, es probable que haya algunos patrones de diseño útiles [véase el capítulo 12] o componentes reutilizables para las interfaces gráficas de usuario para los programas de dibujo. Apuesto 50 dólares a que implementamos una parte o el total de la interfaz del administrador con el uso de ellos.

**Jamie:** De acuerdo. Lo verificaré.

**Elaboración de la tarea.** En el capítulo 8 se vio la elaboración paso a paso (también llamada descomposición de funciones o refinamiento *stepwise* o por etapas) como un mecanismo para mejorar las tareas del procesamiento requeridas para que el software realice alguna función deseada. El análisis de la tarea para el diseño de la interfaz utiliza un enfoque de elaboración para ayudar a entender las actividades humanas que la interfaz de usuario debe incluir.


*La elaboración de la tarea es muy útil, pero también puede ser peligrosa. No porque haya hecho una tarea debe suponer que no existe otra forma de realizarla, y que no se intentará cuando se implemente la interfaz de usuario.*

El análisis de la tarea se aplica en dos formas. Como ya se dijo, es frecuente que un sistema interactivo basado en computadora se utilice para remplazar una actividad manual o semimanual. Para entender las tareas que deben realizarse a fin de lograr el objetivo de la actividad, deben entenderse las tareas que llevan a cabo las personas (con el enfoque manual) y luego mapearlas en un conjunto de tareas similar (pero no necesariamente idéntico) que se implementen en el contexto de la interfaz de usuario. En forma alternativa, puede estudiarse una especificación existente para obtener una solución basada en computadora y derivar un conjunto de tareas de usuario que incluirán al modelo de usuario, al del diseño y la percepción del sistema.

Sin importar el enfoque general del análisis de la tarea, primero deben definirse y clasificarse las tareas. Ya se dijo que un enfoque es la elaboración paso a paso. Por ejemplo, volvamos a considerar el sistema de diseño asistido por computadora para diseñadores de interiores. Al mirar trabajar a un diseñador de interiores, verá que su labor comprende varias actividades principales: distribución de los muebles (recuerde el caso de uso mencionado), selección de telas y materiales, elección de cubiertas de paredes y ventanas, presentación (al cliente), cotización y compras. Cada una de estas tareas principales se divide en subtareas. Por ejemplo, con el uso de la información contenida en el caso de uso, la distribución del mobiliario se desglosa en las tareas siguientes: 1) dibujar un plano con base en las dimensiones de la habitación, 2) colocar puertas y ventanas en las ubicaciones apropiadas, 3a) usar plantillas de muebles para dibujar en el plano bosquejos del mobiliario a escala, 3b) usar plantillas de moldes para dibujar en el plano formas a escala, 4) mover los bosquejos de muebles y las formas para obtener la mejor colocación, 5) poner leyendas en todos los bosquejos de muebles y formas, 6) dibujar dimensiones para mostrar la ubicación y 7) dibujar una perspectiva para el cliente. Para cada una de las demás tareas principales podría emplearse un enfoque similar.

Es posible desglosar aún más las subtareas 1 a 7. Las subtareas 1 a 6 mejorarán, manipulando la información y realizando acciones dentro de la interfaz de usuario. Por otro lado, la subtarea 7 se ejecuta en forma automática en el software y dará como resultado poca interacción directa con el usuario.[4] El modelo del diseño de la interfaz debe incluir cada una de estas tareas en forma consistente con el modelo del usuario (perfil de un diseñador de interiores "común") y con la percepción del sistema (lo que el diseñador de interiores espera de un sistema automatizado).

**Elaboración del objeto.** En vez de centrarse en las tareas que debe realizar un usuario, puede examinarse el caso de uso y la demás información obtenida del usuario para extraer los objetos físicos que usa el diseñador de interiores. Estos objetos se dividen en clases: se definen los atributos de las clases, y la evaluación de las acciones aplicadas a cada objeto proporciona

4 Sin embargo, tal vez no sea éste el caso. El diseñador de interiores podría desear especificar la perspectiva que se va a dibujar, la escala, el uso del color y otra información. El caso de uso relacionado con el dibujo de la perspectiva daría la información necesaria para efectuar esta tarea.

www.FreeLibros.me

una lista de operaciones. Por ejemplo, la plantilla de muebles se traduce en una clase llamada **Mobiliario** con atributos que incluirían **tamaño, forma y ubicación**, entre otros. El diseñador de interiores *seleccionaría* el objeto de la clase **Mobiliario**, lo *movería* a cierta posición en el plano (otro objeto, en este contexto), *dibujaría* el bosquejo del mueble, etc. Las tareas *seleccionar*, *mover* y *dibujar* son operaciones. El modelo del análisis de la interfaz de usuario no daría una implementación literal para cada una de estas operaciones. No obstante, a medida que se elabore el diseño, se definen los detalles de cada operación.

CONSEJO

*Aunque la elaboración del objeto es útil, no debe usarse como enfoque único. Durante el análisis de la tarea debe tomarse en cuenta la opinión del usuario.*

**Análisis del flujo del trabajo.** Cuando varios usuarios distintos, cada uno con diferentes roles, utilizan una interfaz de usuario, a veces es necesario ir más allá del análisis de la tarea y de la elaboración del objeto, y aplicar el *análisis del flujo del trabajo*. Esta técnica permite entender cómo se efectúa un proceso de trabajo cuando están involucradas varias personas (y roles). Considere una compañía que trate de automatizar por completo el proceso de surtir y entregar recetas de medicinas. Todo el proceso[5] giraría alrededor de una aplicación basada en web accesible para médicos (o sus asistentes), farmacéuticos y pacientes. El flujo del trabajo puede representarse con eficacia con un diagrama UML de canal (variación del diagrama de actividades).

Sólo consideraremos una pequeña parte del proceso de trabajo: la situación que ocurre cuando un paciente solicita una nueva entrega. La figura 11.2 presenta el diagrama de canal que indica las tareas y decisiones para cada uno de los tres roles ya mencionados. Esta información se obtiene de entrevistas o de casos de uso escritos por cada actor. No obstante, el flujo de eventos (mostrados en la figura) permite reconocer cierto número de características clave de la interfaz:

**Cita:**

"Es mucho mejor adaptar la tecnología al usuario que obligar a éste a adaptarse a la tecnología."

**Larry Marine**

1. Cada usuario implementa tareas a través de la interfaz: entonces, la vista y sensación de la interfaz diseñada para el paciente será distinta de la que se definió para farmacéuticos o médicos.
2. El diseño de la interfaz para farmacéuticos y médicos debe incluir acceso, y desplegar la información desde fuentes secundarias (por ejemplo, acceder al inventario para el farmacéutico y acceder a información acerca de medicaciones alternativas para el médico).
3. Muchas de las actividades anotadas en el diagrama de canal se elaboran más con el uso del análisis de la tarea o la elaboración del objeto (por ejemplo, *Surtir receta* implicaría la entrega de una orden por correo, una visita a la farmacia o visitar un centro de distribución de medicinas especiales).

**Representación jerárquica.** Al comenzar a analizar la interfaz, ocurre un proceso de elaboración. Una vez establecido el flujo de los trabajos, se define una jerarquía de tareas para cada tipo de usuario. Ésta proviene de la elaboración paso a paso de cada tarea identificada para el usuario. Por ejemplo, considere las tareas de usuario siguientes y la jerarquía de las subtareas.

**Tarea del usuario: *Solicitar que se surta una receta***

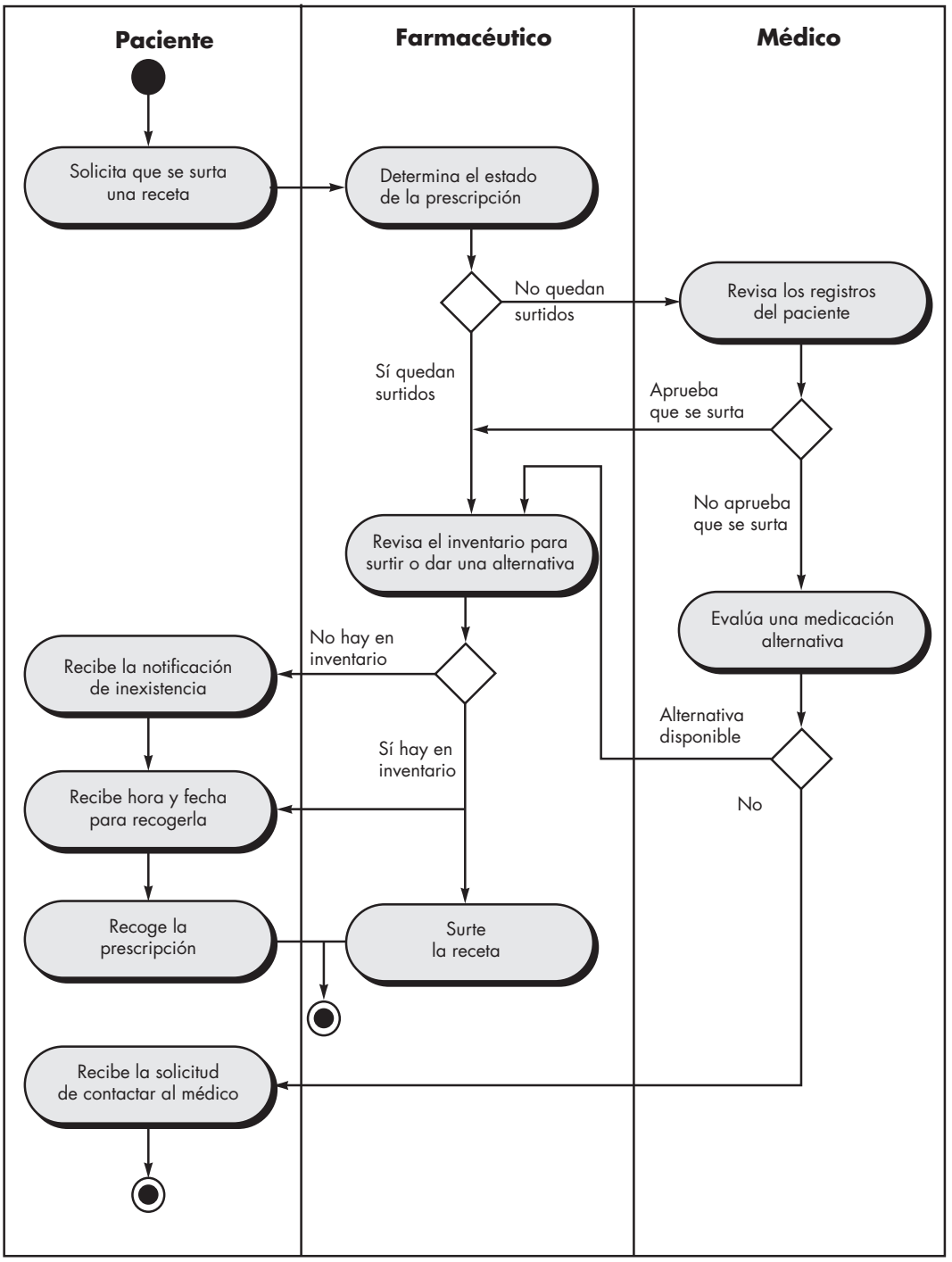
- *Dar información de identificación.*
  - *Especificar nombre.*
  - *Especificar identificación de usuario.*
  - *Especificar PIN y clave.*
- *Especificar número de prescripción.*
- *Especificar la fecha en la que se requiere surtir.*

Para terminar la tarea, se definen tres subtareas. Una de ellas, *dar información de identificación*, se elabora más en tres subsubtareas adicionales.

5 Este ejemplo se adaptó de [Hac98].

www.FreeLibros.me

**FIGURA 11.2**
**Diagrama de canal para la función de surtir receta**




### 11.3.3 Análisis del contenido de la pantalla

Las tareas identificadas en la sección 11.3.2 conducen a la presentación de varios tipos diferentes de contenido. Para las aplicaciones modernas, el contenido de la pantalla varía de reportes basados en caracteres (como una hoja de cálculo), gráficas (histograma, modelo tridimensional, fotografía de alguien) o información especializada (por ejemplo, archivos de audio o video). Las

www.FreeLibros.me

técnicas de modelado del análisis estudiadas en los capítulos 6 y 7 identifican los objetos de datos de salida producidos por una aplicación. Estos objetos de datos pueden ser: 1) generados por componentes (no relacionados con la interfaz) en otras partes de la aplicación, 2) adquiridos a partir de datos almacenados en una base accesible desde la aplicación o 3) transmitidos desde sistemas externos a la aplicación en cuestión.

Durante esta etapa del análisis de la interfaz, se toma en cuenta el formato y la estética del contenido (según la despliega la interfaz). Entre las preguntas planteadas y respondidas, se encuentran las siguientes:

**¿Cómo se determina el formato y la estética del contenido desplegado como parte de la interfaz de usuario?**

- ¿Se asignan diferentes tipos de datos a sitios consistentes en la geografía de la pantalla (por ejemplo, las fotografías aparecen siempre en la esquina superior derecha)?
- ¿El usuario puede personalizar la ubicación del contenido en la pantalla?
- ¿Se asigna una identificación apropiada a todo el contenido que hay en la pantalla?
- Si se presenta un reporte grande, ¿cómo debe dividirse para facilitar su comprensión?
- ¿Se dispondrá de mecanismos para pasar directamente a información resumida de grandes conjuntos de datos?
- ¿Las salidas gráficas estarán a escala para que se ajusten a los bordes del dispositivo de pantalla que se utilice?
- ¿Cómo se empleará el color para mejorar la comprensión?
- ¿De qué manera se presentará al usuario los mensajes de error y las advertencias?

Las respuestas a estas (y otras) preguntas ayudarán a establecer los requerimientos de la presentación del contenido.

### 11.3.4 Análisis del ambiente de trabajo

Hackos y Redish [Hac98] estudian la importancia del análisis del ambiente de trabajo cuando afirman lo siguiente:

> Las personas no realizan aisladas su trabajo. Están influidas por la actividad que las rodea, las características físicas del sitio de trabajo, el tipo de equipo que usan y las relaciones laborales que tienen con las demás personas. Si los productos que usted diseña no se ajustan al ambiente, su uso será difícil o frustrante.

En ciertas aplicaciones se coloca la interfaz de usuario de un sistema basado en computadora en una "ubicación amigable" (con iluminación adecuada, buena altura de la pantalla, acceso fácil al teclado, etc.), pero en otras (como en una fábrica o en la cabina de un avión) la iluminación es menos que buena, el ruido es notable, un teclado o ratón no son opción y la posición de la pantalla no es la ideal. El diseñador de la interfaz puede estar restringido por factores que se confabulan contra la facilidad del uso.

Además de los factores del ambiente físico, también entra en juego la cultura del sitio de trabajo. ¿Se medirá de algún modo la interacción con el sistema (por ejemplo, el tiempo de cada transacción o la exactitud de ésta)? ¿Tendrán que compartir información dos o más personas para que se dé una entrada? ¿Cómo se apoyará a los usuarios del sistema? Estas preguntas y muchas otras relacionadas deben responderse antes de que comience el diseño de la interfaz.

## 11.4 Etapas del diseño de la interfaz

Una vez concluido el análisis de la interfaz, todas las tareas (u objetos y acciones) requeridas por el usuario final habrán sido identificadas en detalle y comenzará la actividad de diseño de la interfaz. El diseño de la interfaz, como todo el de la ingeniería de software, es un proceso

www.FreeLibros.me

**Cita:**

"El diseño interactivo [es] una mezcla tersa de las artes gráficas, la tecnología y la psicología."

**Brad Wieners**

iterativo. Cada etapa del diseño de la interfaz de usuario ocurre varias veces, en las que se elabora y refina la información desarrollada en la etapa anterior.

Aunque se han propuesto muchos modelos diferentes para el modelo de la interfaz de usuario ([Nor86], [Nie00]), todos sugieren alguna combinación de las etapas siguientes:

1. Definir objetos y acciones de la interfaz (operaciones) con el uso de la información desarrollada en el análisis de la interfaz (sección 11.3).
2. Definir eventos (acciones del usuario) que harán que cambie el estado de la interfaz de usuario. Hay que modelar este comportamiento.
3. Ilustrar cada estado de la interfaz como lo vería en la realidad el usuario final.
4. Indicar cómo interpreta el usuario el estado del sistema a partir de la información provista a través de la interfaz.

En ciertos casos, se comienza con bosquejos de cada estado de la interfaz (lo que vería el usuario en diferentes circunstancias) y después se trabaja hacia atrás para definir objetos, acciones y otra información importante del diseño. Sin importar la secuencia de las tareas de diseño, debe 1) siempre apegarse a las reglas doradas estudiadas en la sección 11.1, 2) modelar la forma en la que se va a implementar la interfaz y 3) considerar el ambiente (por ejemplo, tecnología de la pantalla, sistema operativo, herramientas de desarrollo, etc.) que se empleará.

### 11.4.1 Aplicación de las etapas de diseño de la interfaz

Una etapa importante del diseño de la interfaz es la definición de objetos de la interfaz y de las acciones que se aplican a ellos. Para lograr esto, se elaboran escenarios de uso en forma muy parecida a la descrita en el capítulo 6. Es decir, se escribe un caso de uso. Se aíslan los sustantivos (objetos) y verbos (acciones) a fin de crear una lista de objetos y acciones.

Una vez definidos éstos y elaborados en forma iterativa, se clasifican por tipo. Se identifican los objetos blanco, fuente y aplicación. Un *objeto fuente* (icono de reporte, por ejemplo) se arrastra y se deja sobre un *objeto blanco* (como un icono de impresora). La consecuencia de esta acción es que se crea un informe en papel. Un *objeto de aplicación* representa datos específicos de la aplicación que no se manipulan directamente como parte de la interacción en la pantalla. Por ejemplo, se usa una lista de correo para guardar nombres. La lista en sí tal vez se ordene, fusione o purgue (acciones basadas en menús), pero no se arrastra y suelta por medio de alguna interacción del usuario.

Cuando se está satisfecho con la definición de todos los objetos y acciones importantes (para una iteración del diseño), se realiza la distribución de la pantalla. Como otras actividades del diseño de la interfaz, la distribución de la pantalla es un proceso interactivo en el que se llevan a cabo el diseño gráfico y la colocación de iconos, la definición de descripciones de texto en la pantalla, la especificación y títulos de las ventanas, y la definición de aspectos grandes y pequeños del menú. Si hubiera una metáfora apropiada del mundo real para la aplicación, éste es el momento en el que se especifica, y la distribución se organiza en forma tal que complemente a la metáfora.

Para dar una ilustración breve de las etapas de diseño mencionadas, considere un escenario de uso para el sistema *CasaSegura* (estudiado en capítulos anteriores). A continuación se presenta un caso de uso preliminar (escrito por el propietario) para la interfaz:

> **Caso de uso preliminar:** Quiero tener acceso a mi sistema de *CasaSegura* a través de internet, desde cualquier lugar remoto. Con el uso de un software de navegación que opere en mi computadora portátil (cuando estoy en el trabajo o de viaje), determino el estado del sistema de alarma, activo o desactivo el sistema, vuelvo a configurar zonas de seguridad y veo diferentes habitaciones de la casa por medio de las cámaras de video instaladas.

www.FreeLibros.me

Para acceder a *CasaSegura* desde una ubicación remota, doy una identificación y una clave. Éstas definen niveles de acceso (por ejemplo, no a todos los usuarios se les permite reconfigurar el sistema) y dan seguridad. Una vez validadas, reviso el estado del sistema y lo cambio activando o desactivando *CasaSegura*. Reconfiguro el sistema con el plano de la casa, con la vista de cada uno de los sensores de seguridad, con el despliegue de cada zona ya configurada y con la modificación de las zonas según se requiera. Puedo ver el interior de la casa por medio de cámaras de video colocadas estratégicamente. Abro el ángulo de visión y acerco la toma que da cada cámara de las diferentes vistas del interior.

Con base en este caso de uso, se identifican las siguientes tareas del propietario, objetos y datos:

- *acceder* al sistema *CasaSegura*
- *introducir* una **identificación** y **clave** que permitan el acceso remoto
- *comprobar* el **estado del sistema**
- *activar* o *desactivar* el **sistema** *CasaSegura*
- *mostrar* el **plano** y las **ubicaciones de los sensores**
- *mostrar* las **zonas** del plano
- *cambiar* las **zonas** en el plano
- *mostrar* las **ubicaciones de las cámaras de video** en el plano
- *seleccionar* una **cámara de video** para ver
- *ver* **imágenes de video** (cuatro cuadros por segundo)
- *abrir* el **ángulo** o *acercar* la **cámara de video**

De esta lista de tareas del propietario se extraen los objetos (en negritas) y las acciones (en cursiva). La mayoría de objetos anotados son de aplicación. Sin embargo, la **ubicación de cámara de video** (objeto fuente) se arrastra y suelta en **cámara de video** (objeto blanco) para crear una **imagen de video** (ventana con pantalla de video).


*Aunque las herramientas automatizadas pueden ser útiles en el desarrollo de prototipos de la distribución, en ocasiones todo lo que se necesita es lápiz y papel.*

Se crea un bosquejo preliminar de la distribución de la pantalla para la vigilancia con video (véase la figura 11.3).[6] Para invocar la imagen de video, se selecciona un icono de ubicación de cámara de video, *C*, ubicado en el plano que aparece en la ventana de vigilancia. En este caso, a continuación se arrastra una ubicación de cámara en la sala (S) y se suelta en el icono de cámara de video que aparece en la parte superior izquierda de la pantalla. Aparece la ventana de video y muestra una toma de la cámara localizada en la sala. Los controles deslizantes de acercamiento y apertura se utilizan para controlar el aumento y dirección de la imagen de video. Para seleccionar una vista de otra cámara, el usuario simplemente arrastra y suelta un icono diferente de ubicación de cámara en el correspondiente a la cámara que hay en la esquina superior izquierda de la pantalla.

El bosquejo de distribución tendría que darse con una expansión de cada objeto de menú de la barra, que indicarían cuáles acciones están disponibles en el modo de vigilancia con video (estado). Durante el diseño de la interfaz se crearía un conjunto completo de bosquejos para cada una de las tareas del propietario anotadas en el escenario de uso.

### 11.4.2 Patrones de diseño de la interfaz de usuario

Las interfaces de usuario gráficas se han vuelto tan comunes que ha surgido una amplia variedad de patrones de diseño de ellas. Como ya se dijo en este libro, un patrón de diseño es una

6 Observe que esto difiere un poco de la implementación de estas características en los capítulos anteriores. Esto tal vez se considere un primer bosquejo del diseño y represente una alternativa digna de tomarse en cuenta.

www.FreeLibros.me

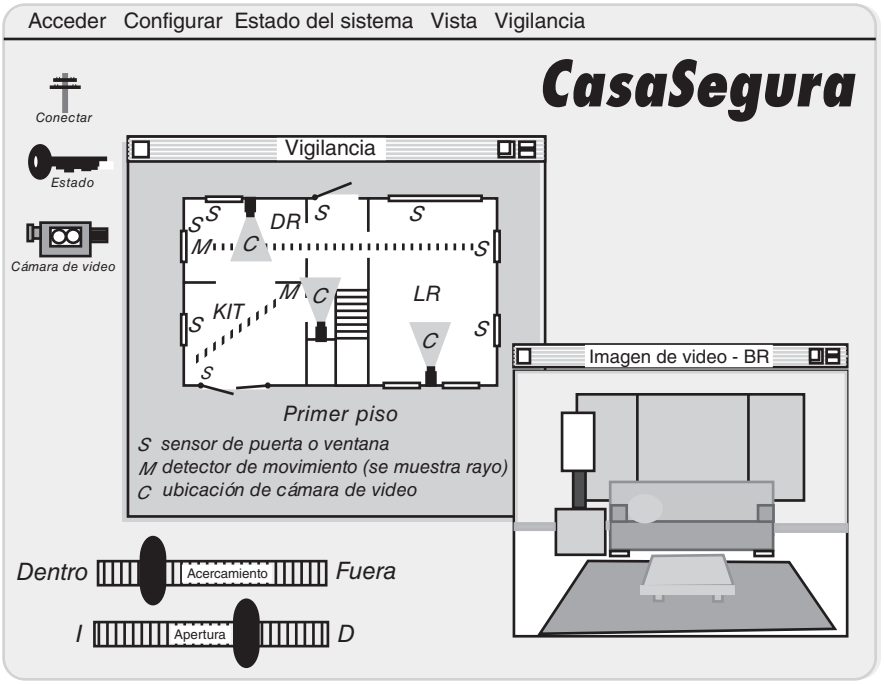



**FIGURA 11.3**
**Distribución preliminar de la pantalla**

abstracción que prescribe una solución de diseño para un problema de diseño bien delimitado.

**WebRef**

Se ha propuesto una amplia variedad de patrones de diseño de interfaces gráficas. En la dirección **www.hcipatterns.org** se encuentran varios vínculos hacia sitios de patrones.

Como ejemplo de un problema de diseño de la interfaz que es común encontrar, considere una situación en la que un usuario debe introducir una o más fechas, a veces varios meses antes. Hay muchas soluciones para este sencillo problema y se han propuesto varios patrones distintos. Laakso [Laa00] sugiere un patrón llamado **CalendarStrip** que produce un calendario continuo, giratorio, en el que se resalta la fecha actual y se eligen las futuras, tomándolas del calendario. La metáfora del calendario es bien conocida por todos los usuarios y da un mecanismo eficaz para situar en contexto una fecha futura.

En la última década se han propuesto muchos patrones de diseño de la interfaz. En el capítulo 12 se presenta un análisis más detallado de los patrones de diseño de la interfaz de usuario. Además, Erickson [Eri08] proporciona vínculos a muchos grupos basados en web.

### 11.4.3 Aspectos del diseño

A medida que evoluciona una interfaz de usuario, casi siempre surgen cuatro aspectos comunes del diseño: tiempo de respuesta del sistema, herramientas de ayuda para el usuario, manejo de información errónea y leyendas de los comandos. Desafortunadamente, son muchos los diseñadores que no enfrentan estos aspectos hasta que es relativamente tarde en el proceso de diseño (a veces sucede que la primera sospecha de que existe un problema no ocurre hasta que ya existe un prototipo operativo). Es frecuente que el resultado sean iteraciones innecesarias, retrasos en el proyecto y frustración del usuario final. Es mucho mejor establecer cada uno de ellos como un aspecto del diseño que debe tomarse en cuenta al comenzar el diseño del software, cuando es fácil y barato hacer cambios.

**Tiempo de respuesta.** El tiempo de respuesta del sistema es la queja principal en muchas aplicaciones interactivas. En general, se mide desde el momento en el que el usuario ejecuta alguna acción de control (por ejemplo, oprime la tecla de "enter" o hace clic en el ratón) y hasta que el software responde con la salida o acción deseada.

www.FreeLibros.me

El tiempo de respuesta tiene dos características importantes: longitud y variabilidad. Si el tiempo de respuesta es demasiado largo, es inevitable que el usuario sienta frustración y tensión. La *variabilidad* se refiere a la desviación del tiempo de respuesta promedio y, por muchos aspectos, es su característica más importante. La variabilidad baja permite que el usuario establezca un ritmo de interacción, aun si el tiempo de respuesta fuera relativamente largo. Por ejemplo, un tiempo de respuesta de 1 segundo para un comando resulta con frecuencia preferible a una respuesta que varíe de 0.l a 2.5 segundos. Cuando la variabilidad es significativa, el usuario siempre se sale de balance, se pregunta si tras bambalinas ha ocurrido algo "distinto".

**Cita:**

"Un error común que cometen las personas cuando tratan de diseñar algo por completo a prueba de tontos es subestimar la ingenuidad de los tontos completos."

**Douglas Adams**

**Herramientas de ayuda.** Casi siempre, todo usuario de un sistema interactivo basado en computadora requiere ayuda ocasional. En ciertos casos, una simple pregunta dirigida a un colega conocedor lo resuelve. En otros, la única opción es la búsqueda detallada en muchos "manuales del usuario". Sin embargo, en la mayor parte de los casos, el software moderno brinda herramientas de ayuda en línea que permiten al usuario obtener la respuesta para sus preguntas o resolver un problema sin tener que salir de la interfaz.

Cuando se consideren las herramientas de ayuda, deben tomarse en cuenta varios aspectos del diseño [Rub88]:

- ¿Habrá ayuda para todas las funciones del sistema y en todo momento durante la interacción con éste? Las opciones incluyen ayuda para un solo subconjunto de todas las funciones y acciones o para todas las funciones.
- ¿Cómo pedirá ayuda el usuario? Las opciones incluyen un menú, una tecla de función especial o un comando AYUDA.
- ¿Cómo se presentará la ayuda? Las opciones incluyen una ventana por separado, una referencia sobre un documento impreso (lejos de ser lo ideal) o una sugerencia de uno o dos renglones generados en una posición fija de la pantalla.
- ¿Cómo volverá el usuario a la interacción normal? Las opciones incluyen un botón de regreso que se muestre en la pantalla, una tecla de función o una secuencia de control.
- ¿Cómo ayudaría que la información estuviera estructurada? Las opciones incluyen una estructura "plana" en la que se acceda a la información por medio de un teclado, una jerarquía en capas de la información que provea cada vez más detalles a medida que el usuario avance en la estructura o el uso de hipertexto.

**Cita:**

"La interfaz del infierno: 'para corregir este error y continuar, introduzca cualquier número primo de 11 dígitos…'".

**Autor desconocido**

**Manejo de errores.** Los mensajes de error y las advertencias son "malas noticias" que llegan a los usuarios desde sistemas interactivos cuando algo sale mal. En el peor de los casos, los mensajes de error y advertencias dan información inútil o equívoca y sólo sirven para aumentar la frustración del usuario. Son pocos los usuarios de computadoras que no se han encontrado con un error del tipo siguiente: *"La aplicación XXX ha terminado porque encontró un error del tipo 1023."* En algún lado debe de existir una explicación del error 1023; de otro modo, ¿por qué habrían incluido los diseñadores dicha información? No obstante, el mensaje no da una indicación real de lo que salió mal o dónde buscar más información. Un mensaje de error presentado de esta manera no hace nada para mitigar la ansiedad del usuario o para ayudarlo a corregir el problema.

En general, todo mensaje de error o advertencia producida por un sistema interactivo debería tener las siguientes características:

**? ¿Qué características debe tener un "buen" mensaje de error?**

- El mensaje debe describir el problema en un lenguaje que entienda el usuario.
- El mensaje debe dar consejos constructivos para corregir el error.
- El mensaje debe indicar cualesquiera consecuencias negativas del error (por ejemplo, archivos de datos potencialmente corrompidos) para que el usuario pueda revisarlas a fin de asegurarse de que no hayan tenido lugar (o corregirlas si las hubo).

www.FreeLibros.me

- El mensaje debe estar acompañado de una clave audible o visual. Es decir, debe generarse un sonido que acompañe la aparición del mensaje o éste debe cintilar momentáneamente o desplegarse en un color que se reconozca con facilidad como el "color del error".
- El mensaje "no debe juzgar". Es decir, sus palabras nunca deben culpar al usuario.

Como en realidad a nadie le gustan las malas noticias, a pocos usuarios les agradará un mensaje de error, no importa lo bien diseñado que esté. Pero una filosofía eficaz de mensajes de error puede hacer mucho para mejorar la calidad de un sistema interactivo y disminuirá de manera significativa la frustración del usuario cuando sucedan los problemas.

**Leyendas del menú y de los comandos.** El comando escrito fue alguna vez el modo más común de interacción entre el usuario y el sistema de software, y era normal usarlo para aplicaciones de todo tipo. Hoy día, el uso de interfaces orientadas a las ventanas, y de apuntar y tomar, ha reducido la dependencia de los comandos escritos, pero algunos usuarios avanzados aún prefieren un modo de interacción orientado a los comandos. Cuando se dan comandos o leyendas en el menú como un modo de interactuar, surgen ciertos aspectos relacionados con el diseño:

- ¿Toda opción de menú tiene un comando correspondiente?
- ¿Qué forma tendrán los comandos? Las opciones incluyen una secuencia de control (por ejemplo, alt-P), teclas de función o palabras escritas.
- ¿Cuán difícil será aprender y recordar los comandos? ¿Qué puede hacerse si se olvida un comando?
- ¿Los comandos pueden ser personalizados o abreviados por el usuario?
- ¿Las leyendas del menú se explican por sí mismas en el contexto de la interfaz?
- ¿Son consistentes los submenús con la función implicada por un tema maestro del menú?

Como ya se dijo, en todas las aplicaciones deben establecerse convenciones para el uso de los comandos. Con frecuencia resultan confusos y se facilita que el usuario cometa errores si tiene que escribir alt-D cuando se ha de duplicar un objeto gráfico en una aplicación y alt-D cuando ha de eliminarse en otra. Resulta obvio el potencial que hay para el error.

**WebRef**

En la dirección **www3.ibm.com/able/guidelines/software/accesssoftware.html** se dan lineamientos para desarrollar software accesible.

**Accesibilidad de la aplicación.** Conforme las aplicaciones de la computación se hacen ubicuas, los ingenieros de software deben asegurarse de que el diseño de la interfaz incluya mecanismos que permitan el acceso fácil de las personas con necesidades especiales. La *accesibilidad* para los usuarios (e ingenieros de software) que tengan discapacidades físicas es un imperativo por razones éticas, legales y comerciales. Son varios los lineamientos de accesibilidad (por ejemplo [W3C03]) —muchos de ellos diseñados para aplicaciones web pero aplicables con frecuencia a todos los tipos de software— que hacen sugerencias detalladas para el diseño de interfaces que alcancen niveles variables de accesibilidad. Otras (ver [App08], [Mic08]) brindan lineamientos específicos para la "tecnología de ayuda" y se abocan a las necesidades de quienes tienen discapacidades visuales, auditivas, motrices, del habla y de aprendizaje.

**Internacionalización.** Los ingenieros de software y sus gerentes invariablemente subestiman el esfuerzo y aptitudes que se requieren para crear interfaces de usuario que incluyan las necesidades de lugares e idiomas diferentes. Con demasiada frecuencia, las interfaces se diseñan para una localidad y lenguaje y después se adaptan para funcionar en otros países. El desafío para los diseñadores de interfaces es crear un software "globalizado". Es decir, las interfa-

www.FreeLibros.me

ces de usuario deben emplearse para que incluyan un núcleo general de funcionalidad que se distribuya a todos aquellos que utilicen el software. Las características de *localización* permiten que la interfaz se personalice para un mercado específico.

Los ingenieros de software disponen de varios lineamientos para la internacionalización (consultar [IBM03]. Éstos abordan aspectos amplios del diseño (por ejemplo, las pantallas difieren en mercados distintos) y de implementación discreta (alfabetos distintos generan requerimientos de escritura y espaciamiento especializados). El estándar *Unicode* [Uni03] se desarrolló para resolver el difícil desafío de manejar decenas de idiomas naturales con cientos de caracteres y símbolos.

HERRAMIENTAS DE SOFTWARE

### *Desarrollo de la interfaz de usuario*

**Objetivo:** Estas herramientas permiten al ingeniero crear una interfaz de usuario gráfica con relativamente poco desarrollo de software especializado. Las herramientas dan acceso a componentes reutilizables y hacen que la creación de una interfaz se reduzca a seleccionar capacidades predefinidas que se ensamblan con el empleo de la herramienta.

**Mecánica:** Las interfaces modernas se construyen con el empleo de un conjunto de componentes reutilizables que se acoplan con algunos componentes personalizados desarrollados para obtener características especializadas. La mayor parte de las herramientas para el desarrollo de interfaces de usuario permiten que el ingeniero de software las cree con el empleo de la capacidad de "arrastrar y soltar". Es decir, el desarrollador selecciona entre muchas herramientas predefinidas (como constructores de formas, mecanismos de interacción, capacidad de procesamiento de comandos, etc.) y las coloca dentro del contenido de la interfaz que se va a crear.

**Herramientas representativas:**[7]

*LegalSuite GUI*, desarrollada por Seagull Software (**www.seagull-software.com**), permitió la creación de interfaces de usuario gráficas basadas en un navegador y da facilidades para hacer la reingeniería de interfaces anticuadas.

*MotifCommon Desktop Environment*, desarrollada por The Open Group (**www.osf.org/tech/desktop/cde/**), es una interfaz de usuario gráfica integrada para sistemas abiertos de computación de escritorio. Produce una interfaz gráfica única estandarizada para la administración de datos, archivos (escritorio gráfico) y aplicaciones.

*Altia Design 8.0*, desarrollada por Altia (**www.altia.com**), es una herramienta para crear interfaces gráficas de usuario en plataformas diferentes (como en un automóvil, portátiles, industriales, etcétera).

## 11.5 DISEÑO DE UNA INTERFAZ PARA WEBAPPS

Toda interfaz de usuario —diseñada para una *webapp*, aplicación de software tradicional, producto de consumo o dispositivo industrial— debe tener las características de usabilidad que se estudiaron en este capítulo. Dix [Dix99] afirma que debe diseñarse una interfaz de *webapp* de modo que responda tres preguntas principales del usuario final:

*Si es probable que los usuarios entren a la webapp por distintos lugares y niveles en la jerarquía del contenido, hay que asegurarse de diseñar cada página con características de navegación que lleven al usuario a otros puntos de interés.*

*¿Dónde estoy?* La interfaz debe: 1) dar una indicación de la *webapp* a la que se ha accedido[8] y 2) informar al usuario de su localización en la jerarquía del contenido.

*¿Qué puedo hacer ahora?* La interfaz siempre debe ayudar al usuario a entender sus opciones actuales: cuáles funciones están disponibles, qué vínculos están vivos, qué contenido es relevante, etcétera.

*¿Dónde he estado, hacia dónde voy?* La interfaz debe facilitar la navegación. Para ello, debe disponer un "mapa" (implementado en forma tal que sea fácil de entender) que indique

7 Las herramientas mencionadas aquí no son obligatorias, sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

8 Todos hemos marcado una página web y cuando regresamos tiempo después no tenemos una indicación del sitio web o del contexto para la página (ni la manera de pasar a otra ubicación dentro del sitio).

www.FreeLibros.me

dónde ha estado el usuario y las trayectorias que pueden tomarse para moverse a cualquier punto de la *webapp*.

Una interfaz eficaz de *webapp* debe dar respuestas a todas estas preguntas cuando el usuario navegue por su contenido y por sus funciones.

### 11.5.1 Principios y lineamientos del diseño de la interfaz

**Cita:**

"Si un sitio es perfectamente usable, pero carece de un estilo elegante y apropiado, entonces fallará."

**Curt Cloninger**

La interfaz de usuario de una *webapp* es la "primera impresión" que se recibe. Sin importar el valor de su contenido, ni la sofisticación de sus capacidades y servicios de procesamiento, así como el beneficio general de la *webapp* en sí, una interfaz mal diseñada decepcionará al usuario potencial y en realidad hará que éste vaya a cualquier otro sitio. Debido al enorme volumen de *webapps* competidoras en virtualmente toda área temática, la interfaz debe "atrapar" de inmediato al usuario potencial.

Bruce Tognozzi [Tog01] define un conjunto de características fundamentales que todas las interfaces deben tener y con ello establece la filosofía que todo diseñador de interfaces de *webapps* debe seguir:

**PUNTO CLAVE**

Una buena interfaz de *webapps* es entendible y benévola, lo que da al usuario la sensación de tener el control.

> Las interfaces eficaces son atractivas visualmente y perdonan los errores, lo que da a sus usuarios la sensación de tener el control. Los usuarios perciben rápidamente la totalidad de sus opciones, captan cómo lograr sus metas y cómo hacer su trabajo.
>
> Las interfaces eficaces no preocupan al usuario con el funcionamiento interno del sistema. El trabajo se guarda de manera cuidadosa y continua, con opción total para que el usuario deshaga cualquier actividad en cualquier momento.
>
> Las aplicaciones y servicios eficaces realizan un máximo de trabajo, al tiempo que requieren un mínimo de información de parte de los usuarios.

A fin de diseñar interfaces de *webapps* con estas características, Tognozzi [Tog01] identifica un conjunto de principios generales de diseño:[9]

**¿Hay un conjunto de principios básicos que se apliquen al diseño de una interfaz de usuario gráfica?**

**Previsión.** *Una webapp debe diseñarse de modo que prevea el siguiente movimiento del usuario.* Por ejemplo, considere una *webapp* de ayuda al cliente desarrollada por un fabricante de impresoras para computadora. Un usuario solicita un objeto de contenido que presenta información sobre el controlador de la impresora para un sistema operativo reciente. El diseñador de la *webapp* debe prever que el usuario tal vez pida descargar el controlador y debe brindar facilidades de navegación que lo permitan, sin requerir que el usuario busque esta capacidad.

**Comunicación.** *La interfaz debe comunicar el estado de cualquier actividad iniciada por el usuario.* La comunicación puede ser obvia (por ejemplo, un mensaje de texto) o sutil (como la imagen de una hoja de papel que se mueva a través de una impresora para indicar que hay una impresión en curso). La interfaz también debe comunicar el estado del usuario (como su identificación) y su ubicación dentro de la jerarquía del contenido de la *webapp*.

**Consistencia.** *El uso de controles de navegación, menús, iconos y estética (color, forma y distribución) debe ser consistente en la webapp*. Por ejemplo, si un texto en color azul y subrayado implica un vínculo de navegación, el contenido nunca debe incorporar texto con dichas características que no impliquen un vínculo. Además, un objeto, digamos un triángulo amarillo, que se utilice para indicar un mensaje de precaución antes de que el usuario invoque una función o acción particular, no debe usarse para otros propósitos en ningún otro lugar de la *webapp*. Por

9 Los principios originales de Tognozzi fueron adaptados y ampliados para su uso en este libro. Véase [Tog01] para un análisis más amplio de ellos.

www.FreeLibros.me

último, toda característica de la interfaz debe responder de manera consistente con las expectativas del usuario.[10]

**Autonomía controlada.** *La interfaz debe facilitar el movimiento del usuario a través de la webapp, pero lo debe hacer de manera que obligue a respetar las convenciones que se hayan establecido para la aplicación*. Por ejemplo, debe controlarse la navegación hacia partes seguras de la *webapp* por medio de la identificación y clave del usuario, y no debe haber ningún mecanismo de navegación que permita que un usuario evite dichos controles.

**Cita:**

"El mejor recorrido es aquel con el menor número de pasos. Hay que acortar la distancia entre el usuario y su meta."

**Autor desconocido**

**Eficiencia.** *El diseño de la webapp y su interfaz deben optimizar la eficiencia del trabajo del usuario, no la del desarrollador que la diseña y construye ni del ambiente cliente-servidor que la ejecuta.* Tognozzi [Tog01] aborda esto cuando escribe lo siguiente: "Esta sencilla verdad explica por qué es tan importante que todos los involucrados en un proyecto de software aprecien la importancia de hacer que la productividad del usuario sea la meta número uno, y de entender la diferencia vital entre construir un sistema eficiente y dar poder a un usuario eficiente."

**Flexibilidad.** *La interfaz debe tener flexibilidad suficiente para permitir que algunos usuarios realicen tareas directamente, y que otros exploren la webapp en forma aleatoria.* En cada caso, debe permitir al usuario entender dónde se encuentra y darle la funcionalidad para deshacer errores y volver a trazar trayectorias de navegación mal elegidas.

**WebRef**

Una búsqueda en web revelará muchas bibliotecas disponibles, por ejemplo, paquetes de aplicaciones en Java, interfaces y clases, en **java.sun.com** o COM, DCOM y Bibliotecas Tipo en **msdn.Microsoft.com**.

**Centrarse.** *La interfaz de la webapp (y el contenido que presente) debe mantenerse centrada en las tareas en curso del usuario.* En todos los hipermedios, existe la tendencia a llevar al usuario a contenido poco relacionado. ¿Por qué? Porque es muy fácil hacerlo... El problema es que el usuario puede extraviarse con rapidez en muchas capas de información de apoyo y perder de vista el contenido original que buscaba inicialmente.

**Ley de Fitt.** *"El tiempo para llegar a un objetivo está en función de la distancia que hay hasta él y del tamaño que tenga"* [Tog01]. Con base en un estudio realizado en la década de 1950 [Fit54], la Ley de Fitt "es un método eficaz para modelar movimientos rápidos e intencionados, donde un apéndice (como una mano) comienza en reposo en una posición específica de arranque y vuelve al reposo dentro de un área objetivo" [Zha02]. Si una secuencia de selecciones o entradas estandarizadas (con muchas opciones diferentes dentro de la secuencia) es definida por una tarea de usuario, la primera selección (con el ratón, por ejemplo) debe estar físicamente cerca de la siguiente selección. Por ejemplo, considere una interfaz de la página inicial de una *webapp* en un sitio de comercio electrónico que vende productos de consumo electrónicos.

Cada opción del usuario implica un conjunto de elecciones o acciones por seguir. Por ejemplo, la opción "comprar un producto" requiere que el usuario introduzca una categoría de producto seguida por el nombre de éste. La categoría del producto (equipo de audio, televisiones, reproductores DVD, etc.) aparece como menú desplegable tan pronto como se elige "comprar un producto". Entonces, la elección siguiente resulta obvia de inmediato (está cerca) y el tiempo para llegar a ella es despreciable. Por el contrario, si la elección apareciera en un menú localizado en el otro lado de la pantalla, el tiempo para que el usuario llegue (y luego elija) será mucho más largo.

**Objetos de la interfaz humana.** *Se ha desarrollado una vasta biblioteca de objetos reutilizables de interfaces humanas para webapps. Úselas.* Cualquier objeto de interfaz que pueda ser "visto, escuchado, tocado o percibido de otro modo" [Tog01] por un usuario final, puede obtenerse de alguna, entre muchas, librerías de objetos.

10 Tognozzi [Tog01] hace la observación de que la única manera de asegurarse de que se entienden bien las expectativas del usuario es realizando exhaustivamente pruebas en las que participe (véase el capítulo 20).

www.FreeLibros.me

**Reducción de la latencia.** *En vez de hacer que el usuario espere a que termine alguna operación interna (como descargar una imagen gráfica compleja), la webapp debe usar tareas múltiples, de manera que permita que el usuario continúe con su trabajo mientras finaliza la operación.* Además de reducir la latencia, los retrasos deben explicarse de modo que el usuario entienda lo que esté pasando. Esto incluye: 1) dar retroalimentación auditiva cuando una selección no dé como resultado una acción inmediata por parte de la *webapp*, 2) desplegar un reloj con animación o una barra de avance que indique que hay un procesamiento en marcha y 3) dar alguna distracción (presentación o texto animado) cuando tenga lugar un procesamiento tardado.

**Aprendizaje.** *Una interfaz de webapp debe diseñarse para minimizar el tiempo de aprendizaje y, una vez aprendida, minimizar el que se dedique a reaprender cuando se regrese a la webapp.* En general, la interfaz debe hacer énfasis en un diseño sencillo e intuitivo que organice el contenido y funcionalidad en categorías que resulten obvias para el usuario.


*Las metáforas son una idea excelente porque reflejan la experiencia del mundo real. Sólo hay que asegurarse de que la metáfora elegida resulte bien conocida para los usuarios finales.*

**Metáforas.** *Una interfaz que use una metáfora de interacción es más fácil de aprender y de usar, en la medida en la que la metáfora sea apropiada para la aplicación y el usuario.* Una metáfora debe recurrir a imágenes y conceptos salidos de la experiencia del usuario, pero no necesita ser una reproducción exacta de una experiencia del mundo real. Por ejemplo, un sitio de comercio electrónico que implemente el pago automatizado en una institución financiera, usa una metáfora de chequera (lo que no sorprende) para ayudar al usuario a especificar y programar los pagos de sus facturas. Sin embargo, cuando un usuario "escribe" un cheque, no necesita introducir el nombre completo del beneficiario, sino que elige de una lista de ellos o hace que el sistema lo seleccione con base en las primeras letras que escriba. La metáfora permanece intacta, pero el usuario recibe ayuda de la *webapp*.

**Mantener la integridad de los productos del trabajo.** *Un producto del trabajo (por ejemplo, un formato llenado por un usuario o una lista especificada por él) debe guardarse en forma automática, de modo que no se pierda si ocurriera un error.* Todos hemos experimentado la frustración que surge cuando al terminar de llenar un formato extenso en una *webapp*, se pierde su contenido debido a un error (cometido por nosotros, por la *webapp* o al transmitirlo del cliente al servidor). Para evitar esto, la *webapp* debe diseñarse para que guarde todos los datos especificados por el usuario. La interfaz debe apoyar esta función y dar al usuario un mecanismo fácil de recuperación de la información "perdida".

**Legibilidad.** *Toda la información presentada en la interfaz debe ser legible para jóvenes y viejos.* El diseñador de la interfaz debe hacer énfasis en estilos legibles para las letras, en su tamaño y en el color del fondo, que debe contrastar.

**Dar seguimiento al estado.** *Cuando resulte apropiado, debe darse seguimiento al estado de la interacción del usuario y guardarlo, de modo que éste pueda salir y volver más tarde para recuperarlo de donde lo haya dejado.* En general, las *cookies* pueden diseñarse para que guarden información del estado. Sin embargo, son una tecnología controvertida y para ciertos usuarios resultan más atractivas otras soluciones de diseño.

**Navegación visible.** *Una interfaz de webapp bien diseñada da "la ilusión de que los usuarios están en el mismo lugar, con el trabajo llevado a ellos"* [Tog01]. Cuando se emplea este enfoque, la navegación no es asunto del usuario. En vez de ello, éste recupera objetos del contenido y selecciona funciones que se despliegan y ejecutan a través de la interfaz.

Nielsen y Wagner [Nie96] sugieren algunos lineamientos prácticos para el diseño de interfaces (basados en su rediseño de una *webapp* importante), que constituyen un buen complemento de los principios ya sugeridos en esta sección:

www.FreeLibros.me

## CasaSegura

### *Revisión del diseño de la interfaz*

**La escena:** Oficina de Doug Miller.

**Participantes:** Doug Miller (gerente del grupo de ingeniería de software de *CasaSegura*) y Vinod Raman, miembro del equipo de ingeniería de software del producto de *CasaSegura*.

**La conversación:**

**Doug:** Vinod, ¿han podido revisar tú y el equipo el prototipo de la interfaz de comercio electrónico de **CasaSeguraAsegurada.com**?

**Vinod:** Sí… todos lo vimos desde un punto de vista técnico, y tengo muchas observaciones. Se las envié ayer por correo electrónico a Sharon [gerente del equipo de la *webapp* para la venta externa del sitio web de comercio electrónico de *CasaSegura*].

**Doug:** Tú y Sharon pueden reunirse y analizar eso… hazme un resumen de los aspectos importantes.

**Vinod:** Sobre todo, han hecho un buen trabajo, nada grandioso, pero es una interfaz normal de comercio electrónico, estética aceptable, distribución razonable, atendieron todas las funciones importantes…

**Doug (sonríe contrito):** ¿Pero?

**Vinod:** Bueno, hay algunas cosas…

**Doug:** Cómo…

**Vinod (presenta a Doug una secuencia de esquemas del prototipo de interfaz):** Aquí está el menú de funciones principales que aparece en la página de inicio:

**Aprenda sobre *CasaSegura*.**

**Describa su casa.**

**Recomendaciones sobre componentes de *CasaSegura*.**

**Compre un sistema de *CasaSegura*.**

**Obtenga ayuda técnica.**

El problema no está en estas funciones. Están bien, pero el nivel de abstracción no es el correcto.

**Doug:** Todas son funciones principales, ¿o no?

**Vinod:** Lo son, pero ahí hay algo… puedes comprar un sistema sin introducir una lista de componentes… no hay una necesidad real de describir la casa si no quieres hacerlo. Sugiero que sólo haya cuatro opciones de menú en la página de inicio.

**Aprenda sobre *CasaSegura*.**

**Especifique el sistema de *CasaSegura* que necesite.**

**Compre un sistema de *CasaSegura*.**

**Obtenga ayuda técnica.**

Cuando se seleccione **Especifique el sistema de *CasaSegura* que necesite**, entonces se tendrán las siguientes opciones:

**Seleccione los componentes de *CasaSegura*.**

**Recomendaciones sobre componentes de *CasaSegura*.**

Si eres un usuario conocedor, seleccionarás los componentes de un conjunto de menús desplegables clasificados en sensores, cámaras, paneles de control, etc. Si necesitaras ayuda, solicitarás una recomendación y eso requerirá que describas tu casa. Creo que es un poco más lógico.

**Doug:** Estoy de acuerdo. ¿Has hablado con Sharon de esto?

**Vinod:** No, primero quiero analizar esto con la gente de mercadotecnia; después la llamaré.

**Cita:**

"Las personas tienen muy poca paciencia con los sitios web mal diseñados."

**Jakob Nielsen y Annette Wagner**

- La lectura rápida en un monitor de computadora es aproximadamente 25 por ciento más lenta que la que se hace en un papel. Por tanto, no obligue al usuario a leer grandes cantidades de texto, en particular cuando se explique la operación de la *webapp*, o dé ayuda para la navegación.
- Evite los avisos "en construcción": un vínculo innecesario es un camino seguro a la decepción.
- Los usuarios prefieren no desplazar la pantalla. La información importante debe situarse dentro de las dimensiones de una ventana normal de navegación.
- Los menús de navegación y los encabezados deben diseñarse de manera consistente y deben estar disponibles en todas las páginas a las que tenga acceso el usuario. El diseño no debe basarse en funciones del navegador que ayuden a la navegación.
- La estética nunca debe obstaculizar la funcionalidad. Por ejemplo, un solo botón será una mejor opción de navegación que una imagen agradable pero vaga o que un icono cuyo objetivo no esté claro.
- Las opciones de navegación deben ser obvias, aun para el usuario casual. Éste no debe tener que buscar en la pantalla para determinar cómo entrar a otro contenido o servicios.

www.FreeLibros.me

Una interfaz bien diseñada mejora la percepción que tenga el usuario del contenido o de los servicios provistos por el sitio. No necesita ser espectacular, pero siempre debe estar bien estructurada y con ergonomía apropiada.

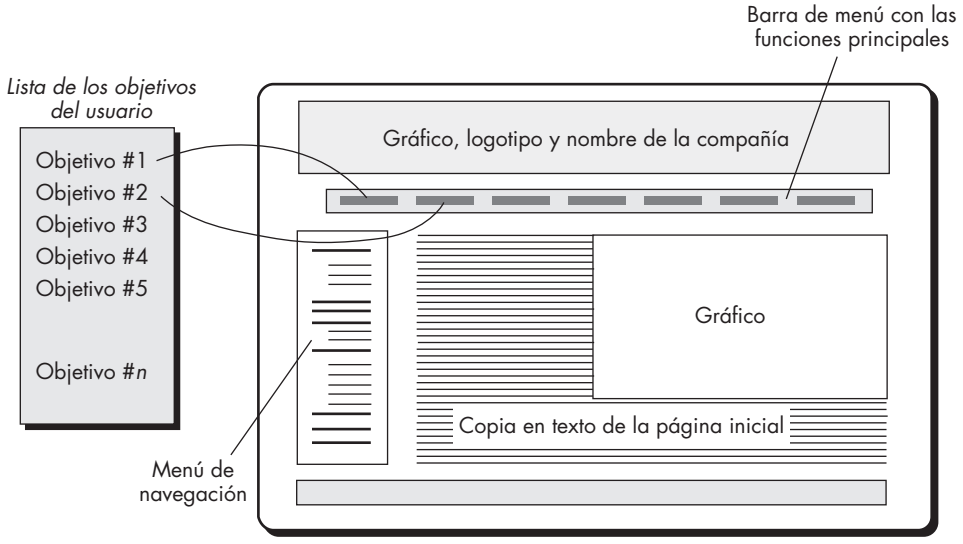
### 11.5.2 Flujo de trabajos para el diseño de la interfaz de webapp

Al comenzar este capítulo, se dijo que el diseño de la interfaz de usuario comienza con la identificación de los requerimientos del usuario, de la tarea y de los elementos ambientales. Una vez identificadas las tareas del usuario, se crean y analizan los escenarios del usuario (casos de uso) con objeto de definir un conjunto de objetos y acciones de la interfaz.

La información contenida en los formatos del modelo de requerimientos son la base para la creación de una distribución de pantalla que ilustre el diseño gráfico y la ubicación de los iconos, la definición de texto descriptivo en la pantalla, así como la especificación y apilamiento de las ventanas y de los temas mayores y menores del menú. Después se utilizan las herramientas para hacer prototipos e implementar en definitiva el modelo de diseño de la interfaz. Las tareas que siguen representan un flujo de trabajo rudimentario para diseñar una interfaz para *webapp*:

1. **Revisar la información contenida en el modelo de requerimientos y refinarla según se requiera.**
2. **Desarrollar un esquema aproximado de la distribución de la interfaz para la *webapp*.** Como parte de la actividad de modelación de los requerimientos, tal vez se haya desarrollado un prototipo de la interfaz (incluida la distribución). Si ya existe la distribución, debe revisarse y refinarse como se requiera. Si no se hubiera desarrollado la distribución de la interfaz, debe trabajarse con los participantes para hacerlo en este momento. En la figura 11.4 se presenta una primera distribución esquemática.
3. **Mapear los objetivos del usuario en acciones específicas de la interfaz.** Para la gran mayoría de *webapps*, el usuario tendrá un conjunto de objetivos primarios relativamente pequeño. Éstos deben mapearse en acciones específicas de la interfaz, como se muestra en la figura 11.4. En esencia, debe responderse la pregunta siguiente: "¿Cómo hace la interfaz para que el usuario logre cada objetivo?"
4. **Definir un conjunto de tareas de usuario asociadas con cada acción.** Cada acción de la interfaz (por ejemplo, "comprar un producto") se asocia con un conjunto de

**FIGURA 11.4**
**Mapeo de los objetivos del usuario en acciones de la interfaz**




www.FreeLibros.me

tareas de usuario. Éstas se identificaron durante la modelación de los requerimientos. Durante el diseño deben mapearse en interacciones específicas que incluyan los aspectos de navegación, objetos de contenido y funciones de la *webapp*.

5. **Elaborar un guión de las imágenes en la pantalla para cada acción de la interfaz.** A medida que se considera cada acción, debe crearse una secuencia de imágenes del guión (imágenes en la pantalla) a fin de ilustrar la manera en la que responde la interfaz a la interacción con el usuario. Deben identificarse los objetos de contenido (aunque todavía no se hayan diseñado ni desarrollado). Deben mostrarse las funciones de la *webapp* e indicarse los vínculos de navegación.
6. **Refinar la distribución de la interfaz y los guiones con entradas del diseño de la estética.** En la mayor parte de casos, el lector será el responsable de hacer la distribución aproximada y de elaborar los guiones, pero el aspecto estético y la sensación que genere un sitio comercial importante con frecuencia es desarrollada por un artista, más que por profesionales técnicos. El diseño de la estética (véase el capítulo 13) se integra con el trabajo realizado por el diseñador de la interfaz.
7. **Identificar los objetos de la interfaz de usuario requeridos para implementar la interfaz.** Esta tarea quizá requiera buscar en una biblioteca de objetos ya existentes para encontrar aquellos (clases) que sean reutilizables y apropiados para la interfaz de la *webapp*. Además, en este momento se especifican cualesquiera clases personalizadas.
8. **Desarrollar una representación del procedimiento de la interacción del usuario con la interfaz.** Esta tarea opcional utiliza diagramas UML de secuencia o diagramas de actividades (véase el apéndice 1) a fin de ilustrar el flujo de actividades (y decisiones) que tienen lugar cuando el usuario interactúa con la *webapp*.
9. **Desarrollar una representación del comportamiento de la interfaz.** Esta opción de tarea emplea diagramas de estado UML (apéndice 1) para representar transiciones de estado y los eventos que las causan. Se definen los mecanismos de control (tales como los objetos y acciones de que dispone el usuario para modificar el estado de una *webapp*).
10. **Describir la distribución de la interfaz para cada estado.** Con el uso de la información de diseño desarrollada en las tareas 2 y 5, se asocia una distribución específica o imagen de la pantalla a cada estado de la *webapp* descrito en la tarea 8.
11. **Refinar y revisar el modelo del diseño de la interfaz.** La revisión de la interfaz debe centrarse en la usabilidad.

Es importante observar que el conjunto definitivo de tareas que se elijan debe adaptarse a los requerimientos especiales de la aplicación que se va a elaborar.

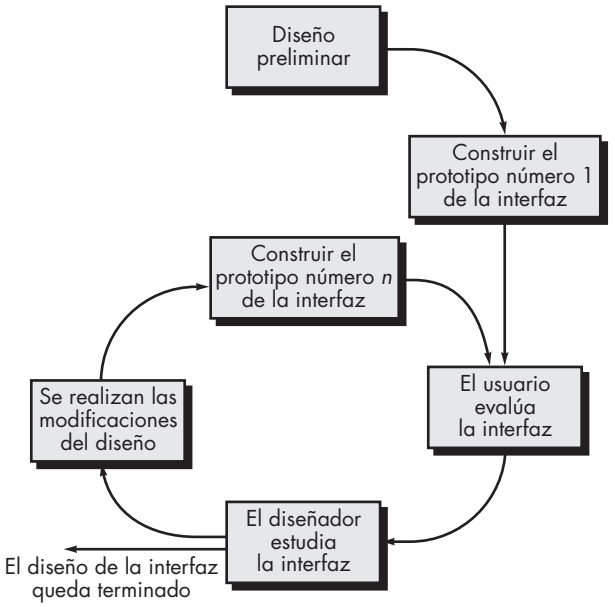
## 11.6 Evaluación del diseño

Una vez que se crea un prototipo operativo de la interfaz de usuario, debe evaluarse con objeto de determinar si satisfacen las necesidades de éste. La evaluación abarca un espectro de formalidad que va desde una "prueba de manejo" informal, en la que el usuario da retroalimentación instantánea a un estudio diseñado formalmente que utilice métodos estadísticos para evaluar cuestionarios que respondería una población de usuarios finales.

El ciclo de evaluación de la interfaz de usuario toma la forma que se aprecia en la figura 11.5. Una vez terminado el modelo del diseño, se crea un prototipo de primer nivel. Éste es evaluado

www.FreeLibros.me

**FIGURA 11.5**
**Ciclo de evaluación del diseño de la interfaz**




por el usuario,[11] quien hace comentarios directos acerca de la eficacia de la interfaz. Además, se emplean técnicas formales de evaluación (tales como cuestionarios, hojas de calificación, etc.), de las que se extrae información (por ejemplo: a 80 por ciento de todos los usuarios no le gusta el mecanismo para guardar archivos de datos). Las modificaciones del diseño se hacen con base en las aportaciones de los usuarios, y así se crea el siguiente nivel de prototipo. El ciclo de evaluación continúa hasta que ya no es necesario modificar más el diseño de la interfaz.

El enfoque del prototipo es eficaz, pero ¿es posible evaluar la calidad de una interfaz de usuario antes de que se construya el prototipo? Si se identifican y corrigen a tiempo los problemas potenciales, se reducirá el número de bucles en el ciclo de evaluación y disminuirá el tiempo de desarrollo. Si se hubiera creado un modelo del diseño de la interfaz, se aplicarían los siguientes criterios de evaluación [Mor81] durante las primeras revisiones de éste:

1. La longitud y complejidad del modelo de requerimientos o especificaciones escritas del sistema y su interfaz darán una indicación de la cantidad de aprendizaje requerido por los usuarios del sistema.
2. El número de tareas del usuario especificadas y el número promedio de acciones por tarea indicarán el tiempo de interacción de la eficiencia general del sistema.
3. El número de acciones, tareas y estados del sistema indicados por el modelo del diseño implicarán la carga de memoria para los usuarios del sistema.
4. El estilo de la interfaz, las herramientas de ayuda y el protocolo del manejo de errores darán una indicación general de la complejidad de la interfaz y de su grado de aceptación por parte del usuario.

Una vez construido el prototipo, se reúnen varios datos cualitativos y cuantitativos que ayuden a evaluar la interfaz. Para obtener los datos cualitativos, se distribuyen cuestionarios a los usuarios del prototipo. Las preguntas pueden ser: 1) de respuesta simple, sí o no, 2) de respuesta

11 Es importante observar que los expertos en ergonomía y diseño de interfaces también son quienes revisan éstas. Las revisiones se denominan *evaluaciones heurísticas* o *recorridos cognitivos*.

www.FreeLibros.me

numérica, 3) de escala (subjetiva), 4) de escalas de aprobación (completamente de acuerdo, de acuerdo), 5) de respuesta porcentual (subjetiva) o 6) de respuesta abierta.

Si se desea obtener datos cuantitativos, puede llevarse a cabo alguna forma de estudio de tiempos. Se observa a los usuarios durante la interacción y se obtienen datos —tales como número de tareas terminadas correctamente en un periodo de tiempo estándar, frecuencia de las acciones, secuencia de éstas, tiempo dedicado a "mirar" la pantalla, número y tipos de errores, tiempo de recuperación del error, tiempo dedicado a usar la ayuda y número de referencias por periodo de tiempo estándar— que se utilizan como guía para la modificación de la interfaz.

El análisis completo de los métodos de evaluación de la interfaz de usuario queda más allá del alcance de este libro. Para mayor información, consulte a [Hac98] y [Sto05].

## 11.7 Resumen

La interfaz de usuario es presumiblemente el elemento más importante de un sistema o producto basado en computadora. Si la interfaz estuviera mal diseñada, afectaría mucho la capacidad del usuario de aprovechar el poder computacional y el contenido de información de una aplicación. En realidad, una interfaz defectuosa haría que fallara incluso una aplicación bien diseñada y con buena implementación.

Son tres los principios importantes que guían el diseño de interfaces eficaces: 1) dar el control al usuario, 2) reducir la memorización del usuario y 3) hacer que la interfaz sea consistente. Para lograr que una interfaz cumpla estos principios, debe llevarse a cabo un proceso de diseño bien organizado.

El desarrollo de una interfaz de usuario comienza con una serie de tareas de análisis. El análisis del usuario define los perfiles de distintos usuarios finales y proviene de varias fuentes comerciales y técnicas. El análisis de la tarea define las tareas y acciones del usuario por medio de un enfoque de elaboración o bien otro orientado a objetos, la aplicación de casos de uso, elaboración de tareas y objetos, análisis del flujo de trabajo y representaciones jerárquicas de la tarea para entender bien la interacción humano-computadora. El análisis ambiental identifica las estructuras físicas y sociales en las que debe operar la interfaz.

Una vez definidas las tareas, se crean y analizan escenarios de usuario para definir un conjunto de objetos y acciones de la interfaz. Esto da una base para la creación de la distribución de la pantalla que ilustre el diseño gráfico y la colocación de los iconos, la definición de un texto descriptivo en la pantalla, la especificación y apilamiento de las ventanas y la especificación de los temas principales y secundarios del menú. A medida que se refina el modelo del diseño, se consideran aspectos tales como tiempo de respuesta, estructura de comandos y acciones, manejo de errores y herramientas de ayuda. Para construir un prototipo a fin de que lo evalúe el usuario, se utilizan varias herramientas de implementación.

Igual que el diseño de la interfaz para el software convencional, el correspondiente a *webapps* describe la estructura y organización de la interfaz de usuario e incluye una representación de la distribución de la pantalla, la definición de los modos de interacción y la descripción de mecanismos de navegación. Un conjunto de principios de diseño de la interfaz y del flujo de los trabajos respectivos guía al diseñador de *webapp* cuando hace la distribución y cuando diseña los mecanismos de control de la interfaz.

La interfaz de usuario es la ventana hacia el software. En muchos casos, moldea la percepción del usuario respecto de la calidad del sistema. Si la "ventana" está manchada, ondulante o rota, el usuario puede rechazar un sistema basado en computadora que, en lo demás, sería poderoso.

www.FreeLibros.me

## Problemas y puntos por evaluar

**11.1.** Describa la peor interfaz con la que haya trabajado y haga una crítica de los conceptos introducidos en este capítulo. Describa la mejor interfaz con la que haya laborado y critíquela respecto de los conceptos estudiados aquí.

**11.2.** Desarrolle dos principios de diseño adicionales que "den el control al usuario".

**11.3.** Elabore dos principios de diseño adicionales que "reduzcan la necesidad de memorización por parte del usuario".

**11.4.** Enuncie otros dos principios de diseño que "hagan consistente a la interfaz".

**11.5.** Considere una de las siguientes aplicaciones interactivas (o una aplicación asignada por su profesor):

*a*) Sistema de publicación de escritorio
*b*) Sistema de diseño asistido por computadora
*c*) Sistema de diseño de interiores (como el descrito en la sección 11.3.2)
*d*) Sistema automatizado de inscripción a materias para una universidad
*e*) Sistema de administración de biblioteca
*f*) Encuestas basadas en internet para elecciones públicas
*g*) Sistema de banca en casa
*h*) Aplicación interactiva asignada por su profesor

Desarrolle los modelos de usuario, del diseño, mental y de implementación para cualquiera de dichos sistemas.

**11.6.** Haga el análisis detallado de las tareas para alguno de los sistemas enlistados en el problema 11.5. Utilice un enfoque de elaboración u orientado a objetos.

**11.7.** Agregue al menos cinco preguntas adicionales a la lista desarrollada para el análisis del contenido de la sección 11.3.3.

**11.8.** Continúe el problema 11.5 y defina los objetos y acciones de la interfaz para la aplicación que haya elegido. Identifique cada tipo de objeto.

**11.9.** Desarrolle un conjunto de distribuciones de pantalla con la definición de los temas principales y secundarios del menú para el sistema que haya escogido en el problema 11.5.

**11.10.** Desarrolle varias distribuciones de pantalla con la definición de los temas de menú principales y secundarios para el sistema *CasaSegura*. Puede elegir un enfoque distinto del que se muestra en la figura 11.3.

**11.11.** Describa su enfoque de las herramientas de ayuda para el usuario a fin de hacer el modelo del diseño del análisis de tareas y el análisis de la tarea que haya realizado como parte de los problemas 11.5 a 11.8.

**11.12.** Dé algunos ejemplos que ilustren por qué la variabilidad del tiempo de respuesta llega a ser importante.

**11.13.** Desarrolle un enfoque que integre de manera automática los mensajes de error y las herramientas de ayuda para el usuario. Es decir, el sistema reconocerá automáticamente el tipo de error y dará una ventana de ayuda con sugerencias para corregirlo. Realice un diseño del software razonablemente completo que tome en cuenta las estructuras de los datos y algoritmos.

**11.14.** Elabore un cuestionario de evaluación de la interfaz que contenga 20 preguntas generales que se apliquen a la mayoría de interfaces. Haga que lo respondan 10 de sus compañeros para un sistema interactivo que todos utilicen. Resuma los resultados y haga un informe para su grupo.

## Lecturas adicionales y fuentes de información

Aunque este libro no trata específicamente las interfaces humano-computadora, gran parte de lo que afirma Donald Norman (*The Design of Everyday Things*, reimpresión, Currency/Doubleday, 1990) sobre la psicología del diseño eficaz se aplica a la interfaz de usuario. Se recomienda su lectura a cualquier persona que piense hacer el diseño de una interfaz de alta calidad.

www.FreeLibros.me

Las interfaces de usuario gráficas son ubicuas en el mundo moderno de la computación. Se trate de un cajero automático, teléfono inalámbrico, tablero electrónico de un automóvil, sitio web o aplicación de negocios, la interfaz de usuario constituye una ventana hacia el software. Por esta razón, abundan los libros que se abocan al diseño de la interfaz. Butow (*User Interface Design for Mere Mortals*, Addison-Wesley, 2007), Galitz (*The Essential Guide to User Interface Design*, 3a. ed., Wiley, 2007), Lehikonen *et al.* (*Personal Content Experience: Managing Digital Life in the Mobile Age*, Wiley-Interscience, 2007), Cooper *et al.* (*About Face 3: The Essentials of Interaction Design*, 3a. ed., Wiley, 2007), Ballard (*Designing the Mobile User Experience*, Wiley, 2007), Nielsen (*Coordinating User Interfaces for Consistency*, Morgan-Kauffamn, 2006), Lauesen (*User Interface Design: A Software Engineering Perspective*, Addison-Wesley, 2005), Barfield (*The User Interface: Concepts and Design*, Bosko Books, 2004) son textos que analizan la usabilidad, los conceptos de la interfaz de usuario, principios y técnicas de diseño, y que también contienen ejemplos útiles.

Los libros más antiguos de Beyer y Hontzblatt (*Contextual Design: A Customer Centered Approach to Systems Design*, Morgan-Kauffman, 2002), Raskin (*The Human Interface*, Addison-Wesley, 2000), Constantine y Lockwood (*Software for Use*, ACM Press, 1999) y Mayhew (*The Usability Engineering LifeCycle*, Morgan-Kauffman, 1999) presentan análisis que dan lineamientos y principios de diseño adicionales, así como sugerencias para recabar requerimientos de la interfaz, modelación de su diseño, implementación y prueba.

Johnson (*GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*, Morgan-Kauffman, 2000) ofrece una guía útil para aquellos que aprenden mejor con el estudio de contraejemplos. Un libro ameno escrito por Cooper (*The Inmates Are Running the Asylum*, Sams Publising, 1999) analiza por qué los productos de alta tecnología nos desconciertan y cómo diseñarlos para que no lo hagan.

Existe una amplia variedad de fuentes de información sobre el diseño de la interfaz disponibles en internet. En el sitio web del libro, hay una lista actualizada de referencias en la red mundial que son relevantes para el diseño de la interfaz de usuario; la dirección es: **www.mhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

www.FreeLibros.me

CAPÍTULO

# 12

# DISEÑO BASADO EN PATRONES

CONCEPTOS CLAVE



Todos hemos encontrado un problema de diseño y pensamos en silencio: *me pregunto si alguien ha desarrollado una solución para esto...* La respuesta casi siempre es *sí*. El problema es encontrarla; luego, estar seguro de que en verdad se ajusta al problema en cuestión, entender las limitaciones que restringen su aplicación y, por último, traducir la solución propuesta al ambiente de diseño.

Pero, ¿qué pasaría si la solución estuviera codificada de algún modo? ¿Qué ocurriría si hubiera una manera estandarizada de describir un problema (de modo que destacara) y un método organizado para representar su solución? Con el empleo de un formato estándar, puede observarse que los problemas de software han sido codificados y descritos, igual que las soluciones (y sus restricciones) propuestas. Este método codificado se denomina *patrones de diseño*, se emplea para describir problemas y sus soluciones y permite que la comunidad de la ingeniería de software aborde el conocimiento de diseño en forma tal que es posible reutilizarlo.

La historia de los patrones de software no comienza con un científico de la computación, sino con un arquitecto constructor, Christopher Alexander, quien reconoció que siempre que se diseñaba un edificio era reconocible un conjunto de problemas recurrentes. Definió éstos y sus soluciones como *patrones*, y los describió del modo siguiente [Ale77]:

> Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente, y luego describe el núcleo de su solución en forma tal que es posible usarla un millón de veces sin elaborarla dos veces de la misma forma.

## UNA MIRADA RÁPIDA

**¿Qué es?** El diseño basado en patrones crea una aplicación nueva, encontrando un conjunto de soluciones comprobadas para un conjunto de problemas delineado con claridad. Cada problema y su solución está descrito por un patrón de diseño catalogado y analizado por otros ingenieros de software que han encontrado el problema e implantado su solución cuando diseñaban otras aplicaciones. Cada patrón de diseño provee un enfoque demostrado para una parte del problema que debe resolverse.

**¿Quién lo hace?** Un ingeniero de software estudia cada problema hallado para una nueva aplicación y después trata de encontrar una solución relevante, buscando en un depósito de patrones.

**¿Por qué es importante?** ¿Ha escuchado el lector la frase "reinventar la rueda"? Eso pasa todo el tiempo en el desarrollo de software y representa una pérdida de tiempo y energía. Al utilizar patrones de diseño existentes, se adquiere una solución probada para un problema específico. A medida que se aplica cada patrón, las soluciones se integran y la aplicación que se va a elaborar se acerca más al diseño final.

**¿Cuáles son los pasos?** El modelo de requerimientos se estudia con objeto de despejar el conjunto jerárquico de problemas por resolver. Se divide el espacio de problemas de modo que sea posible identificar subconjuntos de problemas asociados con funciones específicas del software. Los problemas también pueden organizarse por tipo: arquitectónicos, en el nivel de componentes, algorítmicos, de interfaz de usuario, etcétera. Una vez definido un subconjunto de problemas, se busca uno o más depósitos de patrones a fin de determinar si existe un patrón de diseño previo representado en un nivel de abstracción apropiado. Los patrones aplicables se adaptan a las necesidades específicas del software que se va a elaborar. La solución específica de problemas se aplica en situaciones en las que no se detectan patrones.

**¿Cuál es el producto final?** Se desarrolla un modelo del diseño que ilustra la arquitectura del software, la interfaz de usuario y los detalles en el nivel de componentes.

**¿Cómo me aseguro de que lo hice bien?** A medida que cada patrón de diseño se traduce en cierto elemento del modelo del diseño, se revisan los productos del trabajo respecto de su claridad, corrección, completitud y consistencia con los requerimientos y con los demás patrones.



www.FreeLibros.me

Las ideas de Alexander fueron traducidas por vez primera al mundo del software en los libros de Gamma [Gam95], Buschmann [Bus96] y muchos de sus colegas.[1] Actualmente, existen decenas de depósitos de patrones y el diseño basado en patrones se emplea en muchos dominios diferentes de aplicación.

## 12.1 Patrones de diseño

**PUNTO CLAVE**

Las fuerzas son aquellas características del problema y los atributos de la solución que restringen la forma en la que puede desarrollarse el diseño.

Un *patrón de diseño* se caracteriza como "una regla de tres partes que expresa una relación entre cierto contexto, un problema y una solución" [Ale79]. Para el diseño de software, el *contexto* permite al lector entender el ambiente en el que reside el problema y qué solución sería apropiada en dicho ambiente. Un conjunto de requerimientos, incluidas limitaciones y restricciones, actúan como *sistema de fuerzas* que influyen en la manera en la que puede interpretarse el problema en este contexto y en cómo podría aplicarse con eficacia la solución.

Para entender mejor estos conceptos, piense en una situación[2] en la que una persona debe viajar entre Nueva York y Los Ángeles. En dicho contexto, el viaje tendrá lugar dentro de un país industrializado (Estados Unidos), con el empleo de una infraestructura de transporte (como carreteras, líneas aéreas y ferrocarriles). El sistema de fuerzas que afecta la forma en la que se resuelve el problema de transporte incluye lo siguiente: cuán rápido quiere ir la persona desde Nueva York hasta Los Ángeles, si el viaje incluye paradas en miradores, cuánto dinero puede gastar la persona, si el viaje está previsto para lograr un propósito específico y los vehículos personales que tiene a su disposición el viajero. Dadas estas fuerzas, puede definirse mejor el problema (viajar de Nueva York a Los Ángeles). Por ejemplo, las investigaciones (recabar requerimientos) indican que la persona tiene muy poco dinero, posee sólo una bicicleta (y es un ciclista entusiasta), quiere hacer el viaje a fin de reunir dinero para sus obras de caridad favoritas y dispone de mucho tiempo. La solución al problema, dado el contexto y el sistema de fuerzas, es un viaje en bicicleta a campo traviesa. Si las fuerzas fuesen distintas (debe minimizarse el tiempo de recorrido y su propósito es asistir a una reunión de negocios), será más apropiada otra solución.

**Cita:**

"Nuestra responsabilidad es hacer lo que podamos, aprender lo que se pueda, mejorar las soluciones y transmitirlas."

**Richard P. Feynman**

Es razonable afirmar que la mayoría de problemas tienen muchas soluciones, pero sólo es eficaz aquella que resulta apropiada en el contexto del problema existente. Es el sistema de fuerzas el que hace que un diseñador elija una solución específica. El objetivo es proporcionar la que satisfaga mejor al sistema de fuerzas, aun cuando éstas sean contradictorias. Por último, toda solución tiene consecuencias que afectan otros aspectos del software y que se vuelven parte del sistema de fuerzas de otros problemas por resolver en el sistema mayor.

Coplien [Cop05] caracteriza un patrón de diseño eficaz del modo siguiente:

- *Resuelve un problema:* los patrones entrañan soluciones, no sólo principios o estrategias abstractas.
- *Es un concepto probado:* los patrones incluyen soluciones con un historial, no teorías o especulaciones.
- *La solución no es obvia:* muchas técnicas de solución de problemas (como los paradigmas o métodos de diseño de software) tratan de obtener soluciones a partir de sus principios originales. Los mejores patrones *generan* indirectamente una solución a un problema, enfoque necesario para los problemas más difíciles del diseño.
- *Describe una relación:* los patrones no sólo describen módulos, sino estructuras y mecanismos más profundos del sistema.

1 Existen análisis anteriores de los patrones de software, pero estos dos libros clásicos fueron los primeros que dieron un tratamiento cohesivo al tema.

2 Este ejemplo se adaptó de [Cor98].

www.FreeLibros.me

- *El patrón tiene un componente humano significativo (minimiza la intervención humana).* Todo el software sirve para el confort humano o la calidad de vida; los mejores patrones recurren explícitamente a la estética y a la utilidad.

Dicho en forma más clara, un buen patrón de diseño incorpora el conocimiento de diseño pragmático, ganado con dificultad, en una forma que permite que otros lo reutilicen "un millón de veces sin elaborarla dos veces de la misma forma". Un patrón de diseño evita "reinventar la rueda" o, peor aún, inventar una "nueva rueda" que sea un poco menos redonda, demasiado pequeña para el uso que se pretende y muy angosta para el terreno en el que rodará. Si se usan con eficacia, los patrones de diseño invariablemente harán del lector un buen diseñador de software.

### 12.1.1 Clases de patrones

Una de las razones por las que los ingenieros de software están interesados (e intrigados) por los patrones de diseño es que los seres humanos son inherentemente buenos para reconocer patrones. Si no fuera así, estaríamos congelados en el tiempo y el espacio: seríamos incapaces de aprender de nuestras experiencias, sin voluntad para ir más lejos debido a que nuestra incapacidad de reconocer situaciones nos haría correr grandes riesgos, estaríamos desarticulados por un mundo que parecería no tener regularidades ni consistencia lógica. Por suerte, nada de esto sucede debido a que reconocemos patrones en virtualmente todos los aspectos de nuestras vidas.

En el mundo real, los patrones que reconocemos los aprendemos en el tiempo que dura nuestra vida. Los reconocemos al instante y comprendemos inherentemente lo que significan y cómo podemos usarlos. Algunos de estos patrones nos permiten detectar fenómenos recurrentes. Por ejemplo, imagine que va camino a casa desde su trabajo, por la autopista, cuando su sistema de navegación (o el radio) le informa que en dirección opuesta ocurrió un serio accidente. Se encuentra a siete kilómetros del accidente, pero ya vio que el tráfico se hace lento, reconoce un patrón que llamamos **CuellodeBotella**. Las personas que están en los carriles en la dirección que sigue usted disminuyen su velocidad para ver lo que pasó en el lado opuesto de la carretera. El patrón **CuellodeBotella** produce resultados predecibles (embotellamiento), pero no hace nada más que describir un fenómeno. En el habla de los patrones, éste sería un patrón *no generativo* debido a que describe un contexto y un problema, pero no ofrece ninguna solución clara.

PUNTO CLAVE

Un patrón "generativo" describe el problema, el contexto y las fuerzas, y también una solución práctica para el problema.

Cuando se toman en cuenta los patrones de diseño de software, se intenta identificar los patrones *generativos*. Es decir, se identifica un patrón que describa un aspecto importante y repetitivo de un sistema, y que provea una manera de construir dicho aspecto dentro de un sistema de fuerzas que son únicas en un contexto determinado. Idealmente, podría usarse un conjunto de patrones de diseño generativos para "generar" una aplicación o sistema basado en computadora cuya arquitectura permita adaptarlo al cambio. En ocasiones se llama *generatividad* a "la aplicación sucesiva de varios patrones, cada uno de los cuales incluye su propio problema y fuerzas, y que despliega una solución más grande que emerge indirectamente como resultado de soluciones más pequeñas" [App00].

Los patrones de diseño abarcan un amplio espectro de abstracción y aplicaciones. Los *patrones arquitectónicos* describen problemas de diseño de base amplia que se resuelven con el empleo de un enfoque estructural. Los *patrones de datos* describen problemas recurrentes orientados a datos y las soluciones de modelado de datos que pueden emplearse para resolverlos. Los *patrones de componentes* (también llamados *patrones de diseño*) se enfocan a problemas asociados con el desarrollo de subsistemas y componentes, así como a la manera en la que se comunican entre sí y su ubicación dentro de una arquitectura mayor. Los *patrones de diseño de la interfaz* describen problemas comunes de interfaz de usuario y su solución con un sistema de fuerzas que incluye las características específicas de los usuarios finales. Los *patrones de webapp*

www.FreeLibros.me

enfrentan un conjunto de problemas que surgen cuando se elaboran *webapps* y es frecuente que incorporen muchas de las otras categorías de patrones mencionadas. En un nivel de abstracción más bajo, los *idiomas* describen la forma de implementar todo un algoritmo específico o una parte de él, o bien una estructura de datos, para un componente de software en el contexto de un lenguaje de programación específico.

**¿Hay alguna forma de clasificar los tipos de patrones?**

En su libro fundamental sobre patrones de diseño, Gamma *et al.*[3] [Gam95] se centran en tres tipos de patrones de relevancia especial para el diseño orientado a objetos: patrones creacionales, estructurales y conductuales.

Los *patrones creacionales* se centran en la "creación, composición y representación" de objetos. Gamma *et al.* [Gam95] hacen la observación de que los patrones creacionales "encierran conocimiento acerca de cuáles son las clases concretas que usa el sistema", pero al mismo tiempo "ocultan la forma en la que las instancias de dichas clases se crean y agrupan". Los patrones creacionales ofrecen mecanismos que hacen más fácil la formación de las instancias de los objetos dentro de un sistema y establecen "restricciones en el tipo y número de objetos que es posible crear dentro de un sistema" [Maa07].

INFORMACIÓN

### *Patrones creacionales, estructurales y conductuales*

En la red mundial hay una amplia variedad de patrones de diseño que se ajustan a las categorías creacional, estructural y conductual. En Wikipedia (**www.wikipedia.org**) se encuentran los siguientes ejemplos:

**Patrones creacionales**

- **Patrón de fábrica abstracta:** centraliza la decisión acerca de para qué fábrica deben hacerse instancias.
- **Patrón de método de fabricación:** centraliza la creación de un objeto de tipo específico para elegir una entre varias implementaciones.
- **Patrón constructor:** separa la construcción de un objeto complejo a partir de su representación, de modo que el mismo proceso de construcción pueda crear representaciones distintas.
- **Patrón prototipo:** se usa cuando el costo inherente que implica crear un nuevo objeto en la forma estándar (como con el empleo de una "nueva" clave) es prohibitivo para una aplicación dada.
- **Patrón de instancia única:** restringe la formación de instancias de una clase a un objeto.

**Patrones estructurales**

- **Patrón adaptador:** "adapta" una interfaz para una clase en otra que espera un cliente.
- **Patrón agregado:** es una versión del patrón compuesto con métodos para agregar hijos.
- **Patrón de puente:** desacopla una abstracción de su implementación, de modo que las dos puedan variar en forma independiente.
- **Patrón compuesto:** estructura de árbol de objetos en los que cada uno tiene la misma interfaz.
- **Patrón contenedor:** crea objetos con el único propósito de que sostengan a otros y los administren.
- **Patrón de proximidad:** clase que funciona como interfaz respecto de otra cosa.
- **Tubos y filtros:** cadena de procesos en los que la salida de cada uno es la entrada del siguiente.

**Patrones conductuales**

- **Cadena de patrones de responsabilidad:** objetos de comando que son manejados o pasados a otros objetos por medio de otros que contienen procesamiento lógico.
- **Patrón de comando:** objetos de comando que encierran una acción y sus parámetros.
- **Escucha de eventos:** se distribuyen datos a objetos registrados para recibirlos.
- **Patrón intérprete:** implementa un lenguaje de computadora especializado para resolver con rapidez un conjunto específico de problemas.
- **Patrón iterador:** los iteradores se utilizan para acceder en forma secuencial a los elementos de un agregado sin exponer su representación subyacente.
- **Patrón mediador:** proporciona una interfaz unificada a un conjunto de interfaces en un subsistema.
- **Patrón visitante:** forma de separar un algoritmo de un objeto.
- **Patrón visitante de un solo servicio:** optimiza la implementación de un visitante que se haya asignado, utilizado sólo una vez y luego eliminado.
- **Patrón visitante jerárquico:** brinda una forma de visitar cada nodo en una estructura jerárquica de datos, como un árbol.

Las descripciones detalladas de estos patrones pueden obtenerse por medio de vínculos en **www.wikipedia.org**.

3 En la bibliografía sobre patrones, Gamma *et al.* [Gam95] son llamados con frecuencia "la banda de los cuatro" (GoF, por sus siglas en inglés).

www.FreeLibros.me

Los *patrones estructurales* se centran en problemas y soluciones asociados con la manera en la que se organizan e integran las clases y objetos para construir una estructura más grande. En esencia, ayudan a establecer relaciones entre entidades dentro de un sistema. Por ejemplo, los patrones estructurales que se centran en aspectos orientados a clases proporcionan mecanismos de herencia que conducen a interfaces de programa más eficaces. Los patrones estructurales que se centran en objetos sugieren técnicas para combinar objetos dentro de otros objetos o para integrarlos en una estructura más amplia.

Los *patrones conductuales* se enfocan a problemas asociados con la asignación de responsabilidad entre los objetos y a la manera en la que se efectúa la comunicación entre ellos.

### 12.1.2 Estructuras


Una estructura es una "miniarquitectura" reutilizable que sirve como base desde la que se pueden aplicar otros patrones de diseño.

Los patrones en sí mismos podrían no ser suficientes para desarrollar un diseño completo. En ciertos casos, tal vez sea necesario dar el esqueleto de una infraestructura específica para la implementación, llamada *estructura*, para el trabajo de diseño. Es decir, puede seleccionarse una "*miniarquitectura reutilizable* que provea la estructura y comportamiento generales para una familia de abstracciones de software, así como un contexto [...] que especifique su colaboración y uso en un dominio determinado" [Amb98].

Una estructura no es un patrón arquitectónico, sino un esqueleto con varios "puntos de conexión" (también llamados *ganchos* o *ranuras*) que permiten adaptarlo a un dominio de problema específico. Los puntos de conexión permiten integrar clases o funciones específicas de un problema dentro del esqueleto. En un contexto orientado a objetos, una estructura es un conjunto de clases que cooperan.

Gamma *et al.* [Gam95] describen de la manera siguiente las diferencias entre los patrones de diseño y las estructuras:

1. *Los patrones de diseño son más abstractos que las estructuras.* Las estructuras están incrustadas en el código, pero en éste sólo es posible incrustar ejemplos de patrones. Una ventaja de las estructuras es que se escriben en lenguajes de programación y no sólo son estudiadas, sino ejecutadas y reutilizadas directamente [...]
2. *Los patrones de diseño son elementos arquitectónicos más pequeños que las estructuras.* Una estructura normal contiene varios patrones de diseño, pero lo contrario nunca se cumple.
3. *Los patrones de diseño están menos especializados que las estructuras.* Las estructuras siempre tienen un dominio particular de aplicación. En contraste, los patrones de diseño se usan en casi cualquier tipo de aplicación. Si bien es posible tener patrones de diseño más especializados, incluso éstos no imponen la arquitectura de una aplicación.

En esencia, el diseñador de una estructura afirmará que una miniestructura reutilizable es aplicable a todo software por desarrollarse en un dominio limitado de aplicación. Para ser más eficaces, las estructuras se aplican sin cambio. Pueden agregarse otros elementos de diseño, pero sólo a través de los puntos de conexión que permiten que el diseñador modifique el esqueleto de la estructura.

### 12.1.3 Descripción de un patrón

El diseño basado en patrones comienza con el reconocimiento de patrones en la aplicación que se trata de construir, continúa con una búsqueda para determinar si otros han usado el patrón y termina con la aplicación de un patrón apropiado para el problema de que se trate. Es frecuente que la segunda sea la tarea más difícil. ¿Cómo se encuentran patrones que se ajusten a las necesidades?

Una respuesta a esta pregunta debe basarse en la comunicación eficaz del problema al que se dirige el patrón, el contexto en el que reside ése, el sistema de fuerzas que moldean el con-

www.FreeLibros.me

texto y la solución propuesta. Para comunicar esta información sin ambigüedades, se requiere un formato o plantilla estándar del documento. Aunque se han propuesto varios formatos diferentes de patrones, casi todos contienen un conjunto importante del contenido sugerido por Gamma *et al.* [Gam95]. En el recuadro se presenta un formato simplificado de patrón.

**INFORMACIÓN**

### *Formato de diseño del patrón*

*Nombre del patrón*: describe la esencia del patrón con un nombre corto pero expresivo
*Problema*: describe el problema al que se dirige el patrón
*Motivación*: proporciona un ejemplo del problema
*Contexto*: describe el ambiente en el que reside el problema, incluido el dominio de aplicación
*Fuerzas*: lista el sistema de fuerzas que afectan la manera en la que debe resolverse el problema; incluye el análisis de las limitaciones y restricciones que deben ser tomadas en cuenta
*Solución*: hace la descripción detallada de la solución propuesta para el problema
*Objetivo*: describe el patrón y lo que hace
*Colaboraciones*: describe la manera en la que otros patrones contribuyen a la solución
*Consecuencias*: describe los intercambios potenciales que deben de ser considerados cuando se implementa el patrón, y las consecuencias de usar éste
*Implementación*: identifica los aspectos especiales que deben considerarse cuando se implemente el patrón
*Usos conocidos*: da ejemplos de usos reales del patrón de diseño en aplicaciones reales
*Patrones relacionados*: menciona referencias de patrones de diseño relacionados

**Cita:**

"Los patrones siempre están a medio cocinar, lo que significa que siempre deben ser terminados y adaptados al ambiente específico por el usuario."

**Martin Fowler**

Los nombres de los patrones de diseño deben escogerse con cuidado. Uno de los problemas técnicos clave en el diseño basado en patrones es la incapacidad de encontrar los ya existentes entre cientos o miles de candidatos. La búsqueda del patrón "correcto" se simplifica muchísimo con un nombre significativo para el patrón.

El formato del patrón es un medio estandarizado para describir un patrón de diseño. Cada una de sus entradas representa características del patrón de diseño que pueden ser buscadas (en una base de datos, por ejemplo) a fin de encontrar el que sea apropiado.

### 12.1.4 Lenguajes y repositorios de patrones

Cuando se utiliza el término *lenguaje*, lo primero que viene a la mente es un lenguaje natural (como inglés, español o chino) o uno de programación (como C++ o Java). En ambos casos, el lenguaje tiene una sintaxis y semántica que se utiliza para comunicar ideas o instrucciones de procedimiento en forma eficaz.

Cuando se emplea el término *lenguaje* en el contexto de los patrones de diseño, adopta un significado un poco distinto. Un *lenguaje de patrón* agrupa un conjunto de patrones, cada uno de los cuales se describe con el uso de un formato estandarizado (véase la sección 12.1.3) e interrelacionado para mostrar cómo colaboran los patrones para resolver problemas en un dominio de aplicación.[4]

En un lenguaje natural, las palabras están organizadas en oraciones que dan el significado. La estructura de las oraciones es descrita por la sintaxis del lenguaje. En un lenguaje de patrón, los patrones de diseño están organizados en forma tal que proporcionan un "método estructurado para describir las buenas prácticas de diseño en un dominio particular".[5]

4 Christopher Alexander originalmente propuso lenguajes de patrón para construir arquitecturas y hacer planeación urbana. Actualmente, se han desarrollado lenguajes de patrón para todo, desde las ciencias sociales hasta proceso de la ingeniería de software.

5 Esta descripción es de Wikipedia y se encuentra en la dirección http://en.wikipedia.org/wiki/Pattern_language.

www.FreeLibros.me


*Si no puede encontrar un lenguaje de patrones que se adapte al dominio de su problema, busque analogías en otro conjunto de patrones.*

En cierta forma, un lenguaje de patrones es análogo a un manual de instrucciones de hipertexto para resolver problemas en un dominio específico de aplicaciones. El dominio del problema en cuestión primero se describe de manera jerárquica, comenzando con problemas de diseño amplio asociados con el dominio, y luego se refina cada uno de ellos en niveles de abstracción más bajos. En un contexto de software, los problemas de diseño amplio tienden a ser de naturaleza arquitectónica y se abocan a la estructura general de la aplicación y a los datos o contenido que le dan servicio. Los problemas arquitectónicos se mejoran hacia niveles más bajos de abstracción, lo que conduce a patrones de diseño que resuelven los subproblemas y que colaboran entre sí en el nivel de componentes (o clases). En vez de que un lenguaje de patrones represente una lista secuencial de patrones, lo hace con un conjunto interconectado en el que el usuario comienza con un problema de diseño amplio y "lo presta" a problemas específicos no descubiertos, así como a sus soluciones.

**WebRef**

Para ver una lista de lenguajes de patrones útiles, consulte la dirección **c2.com/ppr/titles.html**. Además, en **hillside.net/patterns/** se encuentra información adicional al respecto.

Se han propuesto decenas de lenguajes de patrones para el diseño de software [Hil08]. En la mayor parte de casos, los patrones de diseño que forman parte de un lenguaje de patrones se almacenan en un repositorio de patrones accesible a través de la web (consulte [Boo08], [Cha03] y [HPR02]). El repositorio proporciona un índice de todos los patrones de diseño y contiene vínculos de hipermedios que permiten al usuario entender las colaboraciones entre patrones.

## 12.2 DISEÑO DE SOFTWARE BASADO EN PATRONES

Los mejores diseñadores en cualquier campo tienen una aptitud asombrosa para ver los patrones que caracterizan un problema y los que pueden combinarse para generar una solución. Los desarrolladores de software de Microsoft [Mic04] lo dicen del siguiente modo:

> Si bien el diseño basado en patrones es relativamente nuevo en el campo del desarrollo de software, la tecnología industrial lo ha utilizado durante décadas, quizá siglos. Los catálogos de mecanismos y configuraciones estándar proporcionan elementos de diseño utilizados para hacer la ingeniería de automóviles, aviones, máquinas herramienta y robots. La aplicación del diseño basado en patrones al desarrollo de software promete a éste los mismos beneficios que tiene la tecnología industrial: ser predecible, disminuir el riesgo y aumentar la productividad.

A través del proceso de diseño, debe buscarse toda oportunidad para aplicar los patrones de diseño existentes (cuando cumplan las necesidades del diseño), en vez de crear otros nuevos.

### 12.2.1 El diseño basado en patrones, en contexto

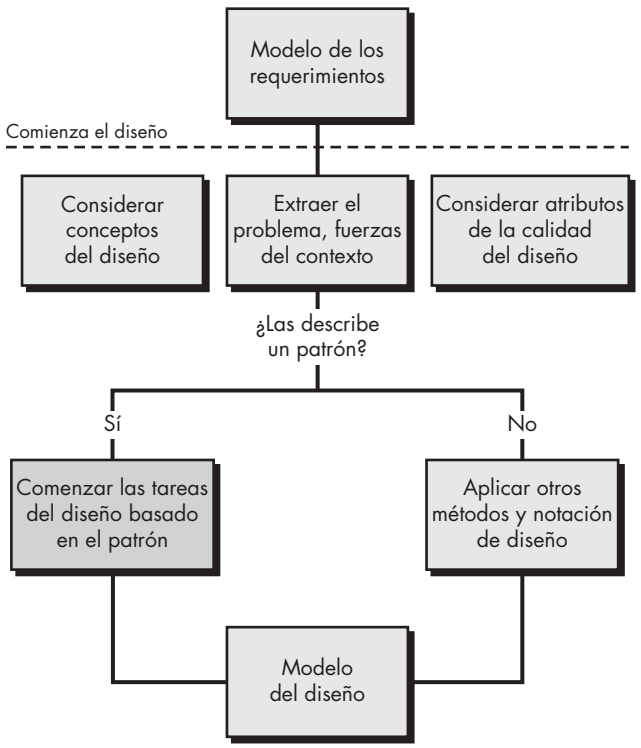
El diseño basado en patrones no se utiliza en el vacío. Los conceptos y técnicas analizados para el diseño arquitectónico, en el nivel de componentes y de la interfaz de usuario (capítulos 9 a 11), se utilizan junto con un enfoque basado en patrones.

En el capítulo 8 se dijo que un conjunto de lineamientos y atributos de la calidad se emplean como la base para tomar todas las decisiones del diseño de software. Éstas reciben influencia de un conjunto de conceptos fundamentales del diseño (como la separación de problemas, mejora por etapas, independencia funcional, etc.) que se logran con el uso de heurísticos que han evolucionado a lo largo de muchas décadas, así como de las mejores prácticas (tales como las técnicas de notación de modelos) que se han propuesto para hacer que el diseño sea más fácil de realizar y que tenga más eficacia como base para la construcción.

En la figura 12.1 se ilustra el papel que juega en todo esto el diseño basado en patrones. Un diseñador de software comienza con un modelo de requerimientos (explícito o implícito) que muestra una representación abstracta del sistema. El modelo de requerimientos describe el conjunto problema, establece el contexto e identifica el sistema de fuerzas que actúan. Tal vez implique al sistema en forma abstracta, pero el modelo de requerimientos hace poco para representar al diseño explícitamente.

www.FreeLibros.me

**FIGURA 12.1**
**El diseño basado en patrones, en contexto**




Cuando el lector inicie su trabajo como diseñador, es importante que recuerde los atributos de la calidad. Éstos (por ejemplo, un diseño debe implementar todos los requerimientos explícitos establecidos en el modelo de los requerimientos) fijan una forma de evaluar la calidad del software, pero no son de mucha ayuda para lograrlos en la realidad. El diseño que se cree debe tener los conceptos fundamentales del diseño analizados en el capítulo 8. Entonces, deben aplicarse técnicas probadas para traducir las abstracciones contenidas en el modelo de requerimientos en una forma más concreta, que es el diseño del software. Para lograr esto, se usarán los métodos y herramientas de modelado disponibles para el diseño arquitectónico, en el nivel de componentes y de la interfaz, pero sólo cuando se enfrente un problema, contexto y sistema de fuerzas que no se haya resuelto antes. Si ya existe una solución, ¡úsela¡ Y esto quiere decir aplicar un enfoque del diseño basado en patrones.

### 12.2.2 Pensar en patrones

En un libro excelente sobre diseño basado en patrones, Shalloway y Trott [Sha05] comentan acerca de la "nueva forma de pensar" cuando se utilizan patrones como parte de la actividad de diseño:

> Tuve que abrir mi mente a una nueva forma de pensar. Y cuando lo hice, escuché a [Christopher] Alexander decir que "el buen diseño de software no se logra sólo poniendo juntas las partes ejecutoras".

El buen diseño comienza con la consideración del contexto: el panorama. Cuando se evalúa el contexto, se extrae una jerarquía de problemas que deben resolverse. Algunos de éstos serán de naturaleza global, mientras que otros se abocarán a características y funciones específicas del software. Todo será afectado por las fuerzas del sistema que influirán en la naturaleza de la solución propuesta.

www.FreeLibros.me

Shalloway y Trott [Sha05] sugieren el siguiente enfoque,[6] que permite que un diseñador piense en patrones:

**? El diseño basado en patrones parece interesante para el problema que debo resolver. Pero, ¿por dónde comienzo?**

1. Asegurarse de entender el panorama: el contexto en el que se encuentra el software que se va a elaborar. El modelo de requerimientos debe transmitir esa comprensión.
2. Estudiar el panorama, identificar los patrones presentes en ese nivel de abstracción.
3. Comenzar el diseño con patrones del "panorama" que establezcan un contexto o esqueleto para el trabajo de diseño adicional.
4. "Trabajar dentro del contexto" [Sha05] en busca de patrones en niveles más bajos de abstracción que contribuyan a la solución del diseño.
5. Repetir los pasos 1 a 4 hasta que el diseño esté completo.
6. Mejorar el diseño, adaptando cada patrón a las especificidades del software que se trata de elaborar.

Es importante observar que los patrones no son entidades independientes. Los patrones de diseño presentes en un nivel alto de abstracción invariablemente influirán en la manera en la que otros patrones se aplican en niveles más bajos de abstracción. Además, es frecuente que los patrones colaboren entre sí. Esto implica que, cuando se selecciona un patrón arquitectónico, muy bien puede influir en los patrones de diseño en el nivel de componentes que se elijan. Del mismo modo, cuando se seleccione un patrón de diseño de la interfaz, a veces es obligatorio usar patrones que colaboren con él.

Para ilustrar lo anterior, considere la *webapp* **CasaSeguraAsegurada.com**. Si se considera el panorama, la *webapp* debe abordar cierto número de problemas fundamentales:

- Cómo dar información acerca de los productos y servicios de *CasaSegura*
- De qué manera vender los productos y servicios de *CasaSegura* a los clientes
- En qué forma establecer la vigilancia y control de un sistema de seguridad instalado que se base en internet.

Cada uno de estos problemas fundamentales puede desglosarse aún más en un conjunto de subproblemas. Por ejemplo, *Cómo vender por internet* implica un patrón de **comercio electrónico** que en sí mismo acarrea un gran número de patrones en niveles de abstracción más bajos. El patrón **ComercioElectrónico** (probablemente un patrón arquitectónico) implica mecanismos para abrir una cuenta de un cliente, mostrar los productos que se van a vender, seleccionar los que se van a adquirir, etcétera. Entonces, si se piensa en patrones, es importante determinar si existe un patrón para abrir una cuenta. Si se dispone de **AbrirCuenta** como patrón viable para el contexto del problema, puede colaborar con otros patrones, tales como **ElaborarFormatodeEntrada, AdministrarFormatosdeEntrada** y **ValidarFormatosdeEntrada**. Cada uno delinea problemas por resolver y las soluciones que se aplicarán.

### 12.2.3 Tareas de diseño

Cuando se utiliza una filosofía de diseño basado en patrones, se llevan a cabo las siguientes tareas:

1. **Examinar el modelo de requerimientos y desarrollar una jerarquía del problema.** Describir cada problema y subproblema aislando el problema, el contexto y el sistema de fuerzas que se aplican. Hay que trabajar desde los problemas amplios (nivel

6 Basado en el trabajo de Christopher Alexander [Ale79].

www.FreeLibros.me

**¿Cuáles son las tareas requeridas para crear un diseño basado en patrones?**

de abstracción elevado) hasta los subproblemas más pequeños (niveles más bajos de abstracción).

2. **Determinar si se ha desarrollado un lenguaje del patrón confiable para el dominio del problema.** Como se dijo en la sección 12.1.4, un lenguaje del patrón se dirige a problemas asociados con un dominio específico de la aplicación. El equipo de software *CasaSegura* buscaría un lenguaje de patrón desarrollado específicamente para productos para la seguridad del hogar. Si no se encontrara ese nivel de especificidad del lenguaje del patrón, el equipo dividiría el problema del software de *CasaSegura* en una serie de dominios generales del problema (de vigilancia de dispositivos digitales, de interfaz de usuario, de administración de video digital) y en la búsqueda de lenguajes de patrón apropiados.
3. **A partir de un problema amplio, determinar si para el mismo se dispone de uno o más patrones arquitectónicos.** Si existe un patrón arquitectónico, hay que asegurarse de estudiar todos los patrones colaboradores. Si el patrón es apropiado, debe adaptarse la solución del diseño propuesta y elaborar un elemento del modelo del diseño que lo represente en forma adecuada. Como se dijo en la sección 12.2.2, un problema amplio para la *webapp* **CasaSeguraAsegurada.com** se aborda con un patrón **ComercioElectrónico**. Éste sugerirá una arquitectura específica para enfrentar los requerimientos del comercio electrónico.
4. **Con el uso de colaboraciones provistas para el patrón arquitectónico, deben estudiarse los problemas en el nivel de subsistema o componente, y buscar los patrones más apropiados para enfrentarlos.** Tal vez sea necesario buscar en varios depósitos de patrones, así como en la lista de aquellos que correspondan a la solución arquitectónica. Si se encuentra un patrón apropiado, hay que adaptar la solución del diseño propuesta y construir un elemento del modelo del diseño que lo represente de manera adecuada. Hay que asegurarse de aplicar el paso 7.
5. **Repetir los pasos 2 a 5 hasta que se hayan resuelto todos los problemas amplios.** Esto implica comenzar con el panorama general y elaborarlo para resolver problemas en niveles cada vez más detallados.
6. **Si los problemas de diseño de la interfaz de usuario han sido aislados (éste es el caso casi siempre), buscar los muchos depósitos de patrones de diseño de la interfaz de usuario para encontrar patrones apropiados.** Se procede en forma similar a los pasos 3, 4 y 5.
7. **Sin importar su nivel de abstracción, si resulta promisorio un lenguaje de patrón o un depósito de patrones o un patrón individual, hay que comparar el problema por resolver con el patrón o patrones presentados.** Debe asegurarse de estudiar el contexto y las fuerzas para garantizar que el patrón, en efecto, da una solución factible para el problema.
8. **Asegurarse de refinar el diseño a medida que se obtiene de los patrones, con el empleo de criterios de calidad como guía.**

Aunque este enfoque del diseño es de arriba abajo, las soluciones del diseño en la vida real a veces son más complejas. Gillis [Gil06] comenta al respecto lo siguiente:

> Los patrones de diseño que se manejan en la ingeniería de software deben usarse en forma deductiva y racionalista. De modo que cuando se tiene el problema o requerimiento general X y el patrón de diseño Y resuelve X, entonces utilice Y. Pero cuando reflexiono en mi propio proceso —y tengo razones para pensar que no soy el único—, observo que es más orgánico que eso, más inductivo que deductivo, más de abajo hacia arriba que de arriba abajo.

www.FreeLibros.me

> Es obvio que debe lograrse un balance. Cuando un proyecto se encuentra en la fase inicial y trato de dar el salto de los requerimientos abstractos a una solución concreta del diseño, es frecuente que tome aire para realizar una búsqueda [...] he hallado que los patrones de diseño son útiles, lo que me permite enmarcar con rapidez el problema de diseño en términos concretos.

Además, el enfoque basado en patrones debe usarse junto con otros conceptos y técnicas de diseño de software.

### 12.2.4 Construcción de una tabla para organizar el patrón


*Las entidades que aparecen en la tabla pueden darse con una indicación de la aplicabilidad relativa del patrón.*

A medida que avanza el diseño basado en patrones, quizá se encuentren problemas para organizar y clasificar los candidatos que surjan de múltiples lenguajes y repositorios de patrones. Para ayudar a organizar la evaluación de los patrones candidatos, Microsoft [Mic04] sugiere crear una *tabla de organización de patrones* que tenga la forma general que se ilustra en la figura 12.2.

Una tabla organizadora de patrones puede implementarse como modelo de hoja de cálculo con el uso del formato de la figura. La columna de la izquierda (sombreada) está organizada por datos/contenido, arquitectura, nivel de componentes y aspectos de la interfaz de usuario. En el renglón superior se enlistan cuatro tipos de patrón: base de datos, aplicación, implementación e infraestructura. En las celdas de la tabla se anotan los nombres de los patrones que son candidatos.

Para obtener las entradas de la tabla organizadora, se busca en lenguajes y repositorios de patrones que aborden un enunciado particular del problema. Cuando se encuentra uno o más patrones candidatos, se introducen en el renglón correspondiente al enunciado del problema y en la columna que corresponda al tipo de patrón. El nombre del patrón se introduce como hipervínculo hacia la URL o dirección web que contenga la descripción completa del patrón.

### 12.2.5 Errores comunes en el diseño

El diseño basado en patrones lo hará un mejor diseñador de software, pero *no* es una panacea. Igual que todos los métodos de diseño, debe comenzarse con los primeros principios, con én-

**FIGURA 12.2**

**Tabla de organización de patrones**
*Fuente:* Adaptada de [Mic04].

| | Base de datos | Aplicación | Implementación | Infraestructura |
|---|---|---|---|---|
| **Datos/Contenido** | | | | |
| Enunciado del problema ... | Nombre(s) del patrón | | Nombre(s) del patrón | |
| Enunciado del problema ... | | Nombre(s) del patrón | | Nombre(s) del patrón |
| Enunciado del problema ... | Nombre(s) del patrón | | | Nombre(s) del patrón |
| **Arquitectura** | | | | |
| Enunciado del problema ... | | Nombre(s) del patrón | | |
| Enunciado del problema ... | | Nombre(s) del patrón | | Nombre(s) del patrón |
| Enunciado del problema ... | | | | |
| **Nivel de componentes** | | | | |
| Enunciado del problema ... | | Nombre(s) del patrón | Nombre(s) del patrón | |
| Enunciado del problema ... | | | | Nombre(s) del patrón |
| Enunciado del problema ... | | Nombre(s) del patrón | Nombre(s) del patrón | |
| **Interfaz de usuario** | | | | |
| Enunciado del problema ... | | Nombre(s) del patrón | Nombre(s) del patrón | |
| Enunciado del problema ... | | Nombre(s) del patrón | Nombre(s) del patrón | |
| Enunciado del problema ... | | Nombre(s) del patrón | Nombre(s) del patrón | |

www.FreeLibros.me


*No fuerce a un patrón, incluso si se aboca al problema en cuestión. Si el contexto y las fuerzas son los equivocados, busque otro patrón.*

fasis en los fundamentos de la calidad del software y el aseguramiento de que el diseño sí satisface las necesidades expresadas por el modelo de los requerimientos.

Cuando se emplea el diseño basado en patrones, suelen ocurrir varios errores comunes. En ciertos casos, no se dedica el tiempo suficiente a entender el problema subyacente, su contexto y fuerzas, y en consecuencia se elige un patrón que parece correcto, pero es inapropiado para llegar a la solución que se requiere. Una vez seleccionado el patrón equivocado, se es renuente a reconocer el error y se fuerza el patrón para que se ajuste. En otros casos, el problema tiene fuerzas que no son consideradas por el patrón escogido, lo que da como resultado un ajuste deficiente o erróneo. En ocasiones se aplica demasiado literalmente y no se implementan las adaptaciones requeridas para el espacio del problema.

¿Es posible evitar estos errores? En la mayoría de los casos la respuesta es *sí*. Todo buen diseñador busca una segunda opinión y ve con buenos ojos la revisión de su trabajo. Las técnicas de revisión que se estudian en el capítulo 15 ayudan a garantizar que el diseño basado en patrones que se haya obtenido dé como resultado una solución de alta calidad para el problema de software que debe resolverse.

## 12.3 Patrones arquitectónicos


Una arquitectura de software puede tener cierto número de patrones arquitectónicos que se aboquen a aspectos tales como concurrencia, persistencia y distribución.

Si un constructor de vivienda decide edificar una de estilo colonial con vestíbulo al centro, sólo hay un estilo arquitectónico que puede aplicarse. Los detalles de éste (como el número de chimeneas, fachada, ubicación de puertas y ventanas, etcétera) variarán en forma considerable; pero, una vez tomada la decisión de la arquitectura general, el estilo será impuesto por el diseño.[7]

Los patrones arquitectónicos son un poco distintos. Por ejemplo, toda vivienda (y todo estilo arquitectónico para ellas) emplea un patrón **Cocina**. Éste y aquéllos con los que colabora abordan problemas asociados con el almacenamiento y preparación de comida, las herramientas que se necesitan para realizar estas tareas y las reglas para situarlas en relación con el flujo de trabajo en dicho espacio. Además, el patrón tal vez se enfrente a problemas asociados con barras, iluminación, interruptores eléctricos, isla central, pisos, etc. Es obvio que hay más de un diseño para la cocina, el que es dictado con frecuencia por el contexto y el sistema de fuerzas. Pero todo diseño se concibe en el contexto de la "solución" sugerida por el patrón **Cocina**.

**¿Cuáles son algunos dominios comunes del patrón arquitectónico?**

Como ya se dijo, los patrones arquitectónicos para el software definen un enfoque específico para el manejo de algunas características del sistema. Bosch [Bos00] y Booch [Boo08] definen cierto número de dominios del patrón arquitectónico. En los párrafos siguientes se describen ejemplos representativos:

**Control de acceso.** Hay muchas situaciones en las que el acceso a datos, características y funciones realizadas por una aplicación está limitado a usuarios finales definidos específicamente. Desde un punto de vista arquitectónico, el acceso a cierta parte de la arquitectura del software debe controlarse de manera rigurosa.

**Concurrencia.** Muchas aplicaciones deben manejar tareas múltiples de manera que simule paralelismo (esto ocurre, por ejemplo, siempre que un solo procesador administra varias tareas o componentes "paralelos"). Hay varias formas distintas en las que una aplicación maneja la concurrencia y cada una puede presentarse con un patrón arquitectónico diferente. Por ejemplo, un enfoque consiste en usar un patrón de **AdministracióndeProcesosdelSistemaOperativo**

7 Esto implica que habrá un atrio y vestíbulo central, que las habitaciones estarán situadas a izquierda y derecha del atrio, que la casa tendrá dos (o más) plantas, que las recámaras de la casa se ubicarán en la planta alta, entre otras características. Estas "reglas" son obligatorias una vez que se toma la decisión de usar un estilo colonial con vestíbulo central.

www.FreeLibros.me

que proporcione características incrustadas para el SO que permitan que los componentes se ejecuten de manera concurrente. El patrón también incorpora funcionalidad del SO que administra la comunicación entre procesos, programación de tareas y otras capacidades que se requieren para lograr concurrencia. Otro enfoque tal vez defina un programador de tareas en el nivel de la aplicación. Un patrón **ProgramadordeTareas** contiene un conjunto de objetos activos en el que cada uno de estos incluye una operación *tick ()* [Bos00]. El programador invoca en forma periódica *tick ()* para cada objeto, que luego realiza las funciones que debe antes de devolver el control al programador, el que después hace la invocación de la operación *tick ()* para el siguiente objeto concurrente.

**Distribución.** El problema de la distribución se aboca a la manera en la que los sistemas o componentes de los sistemas se comunican entre sí en un ambiente distribuido. Se toman en cuenta dos subproblemas: 1) la forma en la que se conectan las entidades una con la otra y 2) la naturaleza de la comunicación que tiene lugar. El patrón arquitectónico más común establecido para enfrentar el problema de distribución es el **Negociador**. Un negociador actúa como un "mediador" entre el componente del cliente y el del servidor. El cliente envía un mensaje al negociador (que contiene toda la información apropiada para que se efectúe la comunicación) y éste finaliza la conexión.

**Persistencia.** Los datos persisten si sobreviven a la ejecución del proceso que los creó. Los datos persistentes se almacenan en una base de datos o archivos que pueden ser leídos o modificados por otros procesos en un momento posterior. En los ambientes orientados a objetos, la idea de un objeto persistente extiende un poco más el concepto de persistencia. Los valores de todos los atributos del objeto, el estado general de éste y otra información complementaria se almacenan para su recuperación y uso futuro. En general, se emplean dos patrones arquitectónicos para lograr la persistencia: un patrón de **SistemadeAdministracióndeBasedeDatos** que aplica la capacidad de almacenamiento y recuperación de un SABD a la arquitectura y un patrón **PersistenciaaNiveldeAplicación** que construye las características de la persistencia

INFORMACIÓN

### *Repositorios de patrones de diseño*

En la red mundial hay muchas fuentes disponibles de patrones de diseño. Algunos se obtienen de lenguajes de patrón publicados en forma individual, mientras que otros forman parte de un portal o repositorio de patrones. Es recomendable consultar los siguientes recursos en web:

*Hillside.net* **http://hillside.net/patterns/**
Una de las colecciones más amplias de patrones y lenguajes de patrón disponibles en web.

*Repositorio de Patrones Portland*
**http://c2.com/ppr/index.html** Contiene apuntadores hacia una amplia variedad de recursos y colecciones de patrones.

*Índice de patrones* **http://c2.com/cgi/wiki?PatternIndex**
"Colección ecléctica de patrones".

*Manual de patrones arquitectónicos de Booch* **www.booch.com/architecture/index.jsp** Referencias bibliográficas a cientos de patrones arquitectónicos y de diseño de componentes.

**Colecciones de patrones UI**

*Patrones UI/HCI* **www.hcipatterns.org/patterns.html**

*Patrones UI de Jennifer Tidwell*
**www.time-tripper.com/uipatterns/**

*Patrones de diseño UI móviles*
**http://patterns.littlespringsdesign.com/wikka.php?wakka=mobile**

**Patrones**

*Lenguaje de patrones para el diseño UI*
**www.maplefish.com/todd/papers/Experiences.html**

*Biblioteca de diseño interactivo para juegos*
**www.eelke.com/research/usability.html**

*Patrones de diseño UI*
**www.cs.helsinki.fi/u/salaakso/patterns/**

**Patrones de diseño especializado**

*Aviónica* **http://g.oswego.edu/dl/acs/acs/acs.html**

*Sistemas de información para negocios*
**www.objectarchitects.de/arcus/cookbook/**

*Procesamiento distribuido* **www.cs.wustl.edu/~schmidt/**

*Patrones IBM para comercio electrónico*
**www.128.ibm.com/developerworks/patterns/**

*Biblioteca de patrones de diseño Yahoo!*
**http://developer.yahoo.com/ypatterns/**

*WebPatterns.org* **http://webpatterns.org/**

www.FreeLibros.me

en la arquitectura de la aplicación (por ejemplo, un software de procesamiento de textos que administre su propia estructura de documento).

Antes de que pueda elegirse cualquiera de los patrones arquitectónicos representativos mencionados en los párrafos anteriores, debe evaluarse lo apropiado que es para la aplicación y el estilo arquitectónico general, así como el contexto y sistema de fuerzas que especifiquen.

## 12.4 PATRONES DE DISEÑO EN EL NIVEL DE COMPONENTES

Los patrones de diseño en el nivel de componentes brindan soluciones comprobadas que se abocan a uno o más subproblemas extraídos del modelo de requerimientos. En muchos casos, los patrones de diseño de este tipo se centran en algún elemento funcional de un sistema. Por ejemplo, la aplicación **CasaSeguraAsegurada.com** debe resolver el siguiente subproblema de diseño: *¿Cómo pueden obtenerse especificaciones del producto e información acerca de cualquier dispositivo de CasaSegura?*

Después de enunciar el subproblema que afecta a la solución, debe considerarse el contexto y el sistema de fuerzas que también la afecten. Al estudiar el modelo de requerimientos apropiados del caso de uso, se observa que el consumidor utiliza la especificación de un dispositivo de *CasaSegura* (como un sensor de seguridad o cámara) con propósitos de información. Sin embargo, cuando se selecciona la función de comercio electrónico, quizá se requiera otro tipo de información relacionada con la especificación (por ejemplo, el precio).

La solución del subproblema involucra una búsqueda. Como buscar es un problema muy común, no es sorprendente que haya muchos patrones relacionados con dicha tarea. Al investigar en varios repositorios de patrones, se encuentran los siguientes, así como el problema que resuelve cada uno:

**AdvancedSearch.** Los usuarios deben encontrar un objeto específico en una gran colección de ellos.

**HelpWizard.** Los usuarios necesitan ayuda acerca de cierto tema relacionado con el sitio web o necesitan encontrar una página específica dentro del sitio.

**SearchArea.** Los usuarios deben encontrar una página.

**SearchTips.** Los usuarios requieren saber cómo controlar el motor de búsqueda.

**SearchResults.** Los usuarios tienen que procesar una lista de resultados de una búsqueda.

**SearchBox.** Los usuarios tienen que encontrar un objeto o información específicos.

Para **CasaSeguraAsegurada.com**, el número de productos es particularmente grande y cada uno tiene una clasificación relativamente sencilla, por lo que es probable que no sean necesarios **AdvanceSearch** y **HelpWizard**. De manera similar, la búsqueda es lo bastante simple como para no requerir **SearchTips**. Sin embargo, la descripción de **SearchBox** se da (en parte) como sigue:

---

**Search Box**

*(Adaptado de www.welie.com/patterns/showPattern.php?patternID=search.)*

**Problema:** Los usuarios necesitan encontrar un objeto o información específica.

**Motivación:** Cualquier situación en la que se aplique una búsqueda por medio de una palabra clave a través de una colección de objetos de contenido organizada como páginas web.

**Contexto:** En vez de navegar para obtener información o contenido, el usuario quiere hacer una búsqueda directa a través del contenido de múltiples páginas web.

www.FreeLibros.me

Cualquier sitio web que ya tenga navegación primaria. El usuario tal vez desee buscar un objeto en cierta categoría. El usuario quizá quiera especificar una consulta adicional.

**Fuerzas:** El sitio web ya cuenta con navegación primaria. Los usuarios quieren buscar un objeto en cierta categoría. Los usuarios desean especificar una consulta más profunda con el empleo de operadores booleanos sencillos.

**Solución:** Ofrecer funciones de búsqueda que consisten en una etiqueta de búsqueda, campo de palabra clave, filtro aplicable y botón de "ir". Oprimir la tecla *return* tiene el mismo efecto que seleccionar el botón *ir*. Asimismo, proveer Sugerencias de Búsqueda y ejemplos en una página distinta. Junto a la función de búsqueda se coloca un vínculo hacia la página. El cuadro de edición para el término de la búsqueda es suficientemente grande como para dar acomodo a tres consultas normales de usuario (alrededor de 20 caracteres). Si el número de filtros es más de 2, se usa un cuadro mayor para seleccionar los filtros, o bien un botón de radio.

Los resultados de la búsqueda se presentan en una página nueva con una etiqueta clara que contiene al menos "resultados de la búsqueda" o algo similar. La función de búsqueda se repite en la parte superior de la página con las palabras clave ingresadas, de modo que los usuarios sepan cuáles fueron.

*La descripción continúa con otras entradas, como se describe en la sección 12.1.3.*

---

El patrón continúa para describir cómo acceder, presentar, ajustar, etc. los resultados de la búsqueda. Con base en esto, el equipo de **CasaSeguraAsegurada.com** puede diseñar los componentes requeridos para implementar la búsqueda o (lo que es más probable) adquirir los componentes reutilizables existentes.

## CasaSegura

### *Aplicación de patrones*

**La escena:** Plática informal durante el diseño de un incremento de software que implementa el control de sensores por internet para **CasaSeguraAsegurada.com**

**Participantes:** Jamie (responsable del diseño) y Vinod (arquitecto jefe del sistema de **CasaSeguraAsegurada.com**).

**La conversación:**

**Vinod:** Entonces, ¿cómo va el diseño de la interfaz de control de la cámara?

**Jamie:** No va mal. Ya diseñé la mayoría de las capacidades para conectarla a los sensores reales sin demasiados problemas. También comencé a pensar acerca de la interfaz para que los usuarios en verdad muevan, abran y acerquen las cámaras desde una página web remota, pero no estoy seguro de haber terminado ya.

**Vinod:** ¿Qué es lo que tienes?

**Jamie:** Bueno, los requerimientos son que el control de la cámara sea muy interactivo: cuando el usuario mueva el control, la cámara debe moverse tan pronto como sea posible. Entonces, pensaba en disponer varios botones, como en una cámara normal, para que cuando el usuario controle la cámara haga clic en ellos.

**Vinod:** Mmm. Sí, eso funcionaría, pero no estoy seguro de que esté bien: cada vez que se haga clic en el control, se necesitará esperar para que ocurra toda la comunicación entre el cliente y el servidor, por lo que no habrá una buena percepción de retroalimentación rápida.

**Jamie:** Eso es lo que he pensado y por ello no estoy muy conforme con el enfoque, pero no estoy seguro de qué más hacer.

**Vinod:** Bueno, ¿por qué no usar el patrón **ControldeDispositivoInteractivo**?

**Jamie:** Mmm… ¿qué es eso? Nunca lo he oído.

**Vinod:** Básicamente es un patrón para el problema exacto que describes. La solución que propone es crear una conexión de control entre el servidor y el dispositivo, a través del cual pueden enviarse los comandos de control. De esa forma, no necesitas mandar solicitudes HTTP normales. El patrón incluso muestra cómo puedes implementar esto con el uso de algunas técnicas AJAX sencillas. En el lado del cliente tienes algunos scripts de Java que se comunican directamente con el servidor y que envían los comandos tan pronto como el usuario está sin hacer nada.

www.FreeLibros.me

**Jamie:** Genial… Eso es justo lo que necesito para resolver el problema. ¿Dónde lo encuentro?

**Vinod:** Está disponible en un repositorio en línea. Ésta es la URL.

**Jamie:** Iré a revisarlo.

**Vinod:** Sí, pero recuerda revisar el campo de consecuencias para el patrón. Creo recordar que había algo acerca de tener cuidado con ciertos aspectos de seguridad. Pienso que es porque se crea un canal de control separado y de ese modo se evaden los mecanismos normales de seguridad de web.

**Jamie:** Buena observación. Es probable que no se me hubiera ocurrido… Gracias.

## 12.5 Patrones de diseño de la interfaz de usuario

En los últimos años, se han propuesto cientos de patrones de interfaz de usuario (IU). La mayoría se ubican en una de las siguientes 10 categorías (se estudian con un ejemplo representativo)[8] según los describen Tidwell [Tid02] y VanWelie[Wel01]:

**Whole UI.** Proporciona una guía para diseñar la estructura y navegación de alto nivel a través de toda la interfaz.

**Patrón: NavegacióndeAltoNivel (TopLevelNavigation)**

**Descripción breve:** Se usa cuando un sitio o aplicación implementa cierto número de funciones importantes. Da un menú de alto nivel, acoplado con frecuencia con un logotipo o gráfico identificador que permite la navegación directa hacia cualquiera de las funciones principales del sistema.

**Detalles:** Las funciones importantes (por lo general limitadas a tener de cuatro a siete nombres de función) se enlistan en la parte superior de la pantalla (también es posible tener formatos en columnas verticales) en un renglón de texto horizontal. Cada nombre da un vínculo hacia la función o fuente de información apropiada. Es frecuente usarla con el patrón **MigajasdePan** (**BreadCrumbs)** que se estudia más adelante.

**Elementos de navegación:** Cada nombre de función o contenido representa un vínculo hacia la función o contenido apropiados.

**Distribución de la página.** Se aboca a la organización general de páginas (para sitios web) o de distintas pantallas (para aplicaciones interactivas).

**Patrón: ApilarTarjetas (CardStack)**

**Descripción breve:** Se utiliza cuando deben seleccionarse aleatoriamente cierto número de subfunciones o categorías de contenido específicas relacionadas con una característica o función. Tiene la apariencia de una pila de tarjetas con "pestaña", cada una seleccionable con un clic del ratón, que representan subfunciones o categorías de contenido específicas.

**Detalles:** Las tarjetas con pestañas son una metáfora bien entendida y son fáciles de manipular por parte del usuario. Cada tarjeta con su pestaña (divisor) tiene un formato ligeramente diferente. Algunas requieren de entradas y tienen botones u otros mecanismos de navegación; otras más son informativas. Pueden combinarse con otros patrones, tales como **ListaDesplegable** y **LlenarLosEspacios**, entre otros.

**Elementos de navegación:** Un clic en una pestaña hace que aparezca la tarjeta apropiada. También están presentes características de navegación dentro de la tarjeta, pero en general éstas deben iniciar una función relacionada con los datos de la tarjeta, no establecer un vínculo real hacia otra pantalla.

8 Aquí se utiliza un formato del patrón abreviado. En [Tid02] y [Wel01] se encuentran descripciones del patrón completo (así como decenas de otros patrones).

www.FreeLibros.me

**Formatos y entrada.** Considera varias técnicas de diseño para llenar las entradas en el nivel de formato.

**Patrón: Llenar los espacios**

**Descripción breve:** Permite introducir datos alfanuméricos en un "cuadro de texto".

**Detalles:** Los datos entran en un cuadro de texto. En general, se validan y procesan después de pulsar algún indicador de texto o gráfico (como un botón que diga "ir", "enviar", "siguiente", etc.). En muchos casos, este patrón se combina con una lista desplegable u otros patrones (por ejemplo, BUSCAR <*lista desplegable*> PARA <*llenar los espacios* del cuadro de texto>).

**Elementos de navegación:** Indicador de texto o gráfico que inicia la validación y el procesamiento.

**Tablas.** Dan una guía para el diseño a fin de crear y manipular datos tabulares de todo tipo.

**Patrón: OrdenarTabla (SortableTable)**

**Descripción breve:** Despliega una lista larga de registros que pueden ordenarse por medio de un mecanismo de cambio para cualquier etiqueta de columna.

**Detalles:** Cada renglón de la tabla representa un registro completo. Cada columna representa un campo del registro. Cada encabezado de columna en realidad es un botón seleccionable que se pulsa para iniciar un ordenamiento ascendente o descendente en el campo asociado con la columna para todos los registros desplegados. Por lo general, la tabla es ajustable y tiene algún mecanismo de desplazamiento para el caso de que el número de registros sea más grande que el espacio disponible en la ventana.

**Elementos de navegación:** Cada encabezado de columna inicia el ordenamiento de todos los registros. No se da otro elemento de navegación, aunque en ciertos casos cada registro contiene vínculos de navegación hacia otro contenido o funciones.

**Manipulación directa de los datos.** Se aboca a la edición, modificación y transformación de los datos.

**Patrón: MigajasdePan (BreadCrumbs)**

**Descripción breve:** Brinda una ruta completa de navegación cuando el usuario trabaja con una jerarquía compleja de páginas o pantallas.

**Detalles:** Se da a cada página o pantalla un identificador único. La ruta de navegación hacia la ubicación actual se especifica en una ubicación predefinida para cada pantalla. La ruta tiene la forma siguiente: **página inicial>página del tema principal>página del subtema>página específica>página actual.**

**Elementos de navegación:** Cualquiera de las entradas en la pantalla de las *migajas de pan* puede usarse como un apuntador hacia el vínculo de regreso hacia un nivel más alto de la jerarquía.

**Navegación.** Ayuda al usuario a navegar a través de menús jerárquicos, páginas web y pantallas interactivas.

**Patrón: Editar (EditInPlace)**

**Descripción breve:** Brinda capacidades de edición de texto sencillo para ciertos tipos de contenido en la ubicación que se muestra en la pantalla. No es necesario que el usuario introduzca explícitamente alguna función de edición de texto o algún modo.

**Detalles:** El usuario observa en la pantalla el contenido que debe modificarse. Con un doble clic en el contenido, se indica al sistema que se desea editar. El contenido se resalta para indicar que el modo de edición está activado para que el usuario haga los cambios apropiados.

**Elementos de navegación:** Ninguno.

www.FreeLibros.me

**Búsqueda.** Permite hacer búsquedas de contenido específico a través de información conservada en un sitio web o que está contenida en almacenamientos persistentes de datos accesibles a través de una aplicación interactiva.

**Patrón: BúsquedaSimple (SimpleSearch)**

**Descripción breve:** Da la capacidad de buscar un sitio web o fuente persistente de datos para un concepto simple de datos descritos por una cadena alfanumérica.

**Detalles:** Brinda la capacidad de hacer una búsqueda local (una página o un archivo) o global (todo el sitio o la base de datos completa) para la cadena de búsqueda. Genera una lista de "aciertos" ordenados según su probabilidad de satisfacer las necesidades del usuario. No hace búsquedas de conceptos múltiples o con operaciones booleanas especiales (véase *patrón de búsqueda avanzada*).

**Elementos de navegación:** Cada entrada de la lista representa un vínculo de navegación hacia los datos a los que se hace referencia con la entrada.

**Elementos de página.** Implanta elementos específicos de una página web o de una pantalla del monitor.

**Patrón: Mago (Wizard)**

**Descripción breve:** Lleva al usuario paso a paso a través de una tarea compleja y lo guía para que la termine por medio de ventanas sencillas en la pantalla.

**Detalles:** El ejemplo clásico es un proceso de registro de cuatro etapas. El patrón mago genera una ventana en cada una de ellas, en las que solicita información del usuario paso a paso.

**Elementos de navegación:** La navegación hacia delante y atrás permite que el usuario vuelva a cada etapa en el proceso mago.

**Comercio electrónico.** Específicos para sitios web, estos patrones implementan elementos recurrentes de las aplicaciones de comercio electrónico.

**Patrón: CarritodeCompras (ShoppingCart)**

**Descripción breve:** Da una lista de artículos seleccionados para compra.

**Detalles:** Enlista artículos, cantidad, código del producto, disponibilidad (en inventario, fuera de inventario), precio, información para la entrega, costos de envío y otra información relevante para la compra. También da la facilidad de editar (por ejemplo, retirar, cambiar la cantidad, etcétera).

**Elementos de navegación:** Contiene la capacidad de continuar la compra o salir.

**Varios.** Son patrones que no se ajustan fácilmente a ninguna de las categorías anteriores. En ciertos casos, dependen del dominio u ocurren sólo para clases específicas de usuarios.

**Patrón: IndicadordeAvance (ProgressIndicator)**

**Descripción breve:** Proporciona una indicación del avance cuando una operación dura más de *n* segundos.

**Detalles:** Se representa con un icono animado o cuadro de mensaje que contiene alguna indicación visual (por ejemplo, una "barra de peluquero", barra de avance con indicador de porcentaje, etc.) de que el procesamiento está en curso. También contiene una indicación de texto acerca del estado del procesamiento.

**Elementos de navegación:** Es frecuente que contenga un botón que permita al usuario hacer una pausa o cancelar el procesamiento.

Cada uno de los ejemplos de patrones anteriores (y todos los de cada categoría) también pueden tener un diseño completo en el nivel de componentes, incluso clases de diseño, atributos, operaciones e interfaces.

www.FreeLibros.me

El estudio exhaustivo de los patrones de interfaz de usuario se encuentra más allá del alcance de este libro. Si el lector está interesado, se le recomienda consultar a [Duy02], [Bor01], [Tid02] y [Wel01].

## 12.6 PATRONES DE DISEÑO DE WEBAPP

En este capítulo aprendimos que hay cuatro tipos diferentes de patrones y muchas formas de clasificarlos. Cuando se consideran los problemas de diseño que deben resolverse para construir una *webapp*, es bueno considerar categorías de patrones en dos dimensiones: centrarse en el diseño del patrón y en el nivel de granularidad. *Centrarse en el diseño* identifica cuál aspecto del modelo del diseño es relevante (por ejemplo, arquitectura de la información, navegación e interacción). La *granularidad* determina el nivel de abstracción que se considera (¿el patrón se aplica a toda la *webapp*, a una sola página web, a un subsistema o a un componente individual de la *webapp*?)

### 12.6.1 Centrarse en el diseño


La atención se hace "más fina" a medida que se avanza en el diseño.

En los capítulos anteriores se hizo énfasis en un avance del diseño que comienza por tomar en cuenta la arquitectura, aspectos en el nivel del componente y representaciones de la interfaz de usuario. En cada paso se consideran los problemas y soluciones propuestos para comenzar en un nivel alto de abstracción a fin de pasar poco a poco a otro más detallado y específico. En otras palabras, el diseño se "angosta" a medida que avanza. Los problemas (y soluciones) que se encontrarán cuando se diseñe una arquitectura de información para una *webapp* serán diferentes de aquellos que aparecen cuando se diseña una interfaz. Por tanto, no debe sorprender que los patrones para el diseño de *webapps* se desarrollen para distintos niveles de atención, de modo que se aborden los problemas (y sus soluciones) únicos que se encuentren en cada nivel. Los patrones de *webapps* se clasifican con el empleo de los siguientes niveles de atención en el diseño:

- **Patrones de arquitectura de la información**: se relacionan con la estructura general del espacio de información y con las formas en las que los usuarios interactúan con ésta.
- **Patrones de navegación:** definen estructuras de los vínculos de navegación, tales como jerarquías, anillos, recorridos, etcétera.
- **Patrones de interacción**: contribuyen al diseño de la interfaz de usuario. Los patrones en esta categoría se enfrentan al modo en el que la interfaz informa al usuario de las consecuencias de una acción específica, cómo expande el usuario el contenido con base en el empleo del contexto y sus deseos, la mejor manera de describir el destino implícito por un vínculo, la manera de informar al usuario acerca del estado de una interacción en curso y aspectos relacionados con la interfaz.
- **Patrones de presentación**: ayudan a presentar el contenido al usuario a través de la interfaz. Los patrones en esta categoría se abocan al modo de organizar las funciones de control de la interfaz de usuario para mejorar su uso, a mostrar la relación entre una acción de la interfaz y los objetos de contenido a los que afecta y a la forma de establecer jerarquías eficaces del contenido.
- **Patrones funcionales**: definen los flujos de trabajo, comportamientos, procesamiento, comunicación y otros elementos algorítmicos dentro de una *webapp*.

En la mayoría de casos, sería inútil explorar la colección de patrones de arquitectura de la información cuando se encuentra un problema en el diseño de la interacción. Se estudiarían los

www.FreeLibros.me

patrones de interacción porque es la atención en el diseño lo que es relevante para el trabajo que se está ejecutando.

### 12.6.2 Granularidad del diseño

Cuando un problema involucra aspectos del "panorama", debe tratarse de desarrollar soluciones (y los patrones de uso relevantes) que se centren en éste. A la inversa, cuando la atención es muy estrecha (como cuando se selecciona únicamente un aspecto de un conjunto reducido de cinco o menos de ellos), la solución (y el patrón correspondiente) se busca con más estrechez. En términos del nivel de granularidad, los patrones se describen en los niveles siguientes:

- **Patrones arquitectónicos.** Este nivel de abstracción se relacionará por lo común con patrones que definen la estructura general de la *webapp*, que indican las relaciones entre diferentes componentes o incrementos y que definen las reglas para especificar las relaciones entre los elementos (páginas, paquetes, componentes y subsistemas) de la arquitectura.
- **Patrones de diseño.** Éstos se abocan a un elemento específico del diseño, como un agrupamiento de componentes, a fin de resolver algún problema de diseño, relaciones entre los elementos de una página, o mecanismos para efectuar la comunicación entre componentes. Ejemplo de esto sería el patrón **Broadsheet** para la distribución de la página inicial de una *webapp*.
- **Patrones de componentes.** Este nivel de abstracción se relaciona con elementos individuales de pequeña escala de una *webapp*. Algunos ejemplos son los elementos de interacción individual (botones de radio), de navegación (¿cómo dar formato a los vínculos?) o funcionales (algoritmos específicos).

También es posible definir la relevancia de distintos patrones para diferentes clases de aplicaciones o dominios. Por ejemplo, una colección de patrones (en diferentes niveles de atención al diseño y granularidad) puede tener relevancia particular para los negocios electrónicos (*e-business)*.

INFORMACIÓN

#### *Repositorios de patrones para el diseño de hipermedios*

El sitio web IAWiki (**http://iawiki.net/WebsitePatterns**) es un espacio de colaboración para los arquitectos de la información y contiene muchos recursos útiles. Entre ellos hay vínculos hacia varios catálogos y repositorios útiles de patrones de hipermedios. Ahí están representados cientos de patrones de diseño:

*Repositorios de patrones de diseño de hipermedios*
**www.designpattern.lu.unisi.ch/**
*Patrones de interacción, por Tom Erickson*
**www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html**
*Patrones de diseño web, por Martijn van Welie*
**www.welie.com/patterns/**
*Patrones web para el diseño de la IU*
**http://harbinger.sims.berkeley.edu/ui_designpatterns/webpatterns2/webpatterns/home.php**
*Patrones para sitios web personales*
**www.rdrop.com/%7ehalf/Creations/Writings/Web.patterns/index.html**
*Mejora de sistemas de información web con patrones de navegación*
**http://www8.org/w8-papers/5b-hypertext-media/improving/improving.html**
*Un lenguaje de patrón HTML 2.0*
**www.anamorph.com/docs/patterns/default.html**
*Campos comunes. Un lenguaje de patrón para el diseño HCI*
**www.mit.edu/~jtidwell/interaction_patterns.html**
*Patrones para sitios web personales*
**www.rdrop.com/~half/Creations/Writings/Web.patterns/index.html**
*Lenguaje de patrón de indización*
**www.cs.brown.edu/~rms/InformationStructures/Indexing/Overview.html**

www.FreeLibros.me

## 12.7 Resumen

Los patrones de diseño dan un mecanismo codificado para describir problemas y su solución en forma tal que permiten que la comunidad de ingeniería de software diseñe el conocimiento para que sea reutilizado. Un patrón describe un problema, indica el contexto y permite que el usuario entienda el ambiente en el que sucede el problema, y enlista un sistema de fuerzas que indican cómo puede interpretarse el problema en su contexto, y el modo en el que se aplica la solución. En el trabajo de la ingeniería de software se identifican y documentan patrones generativos que describen un aspecto importante y repetible de un sistema para después proporcionar la manera de elaborar dicho aspecto dentro de un sistema de fuerzas que es único para un contexto determinado.

Los patrones arquitectónicos describen problemas de diseño amplios que se resuelven con un enfoque estructural. Los patrones de datos describen problemas recurrentes orientados a datos y las soluciones para modelar éstos que se utilizan para resolverlos. Los patrones de componentes (también conocidos como patrones de diseño) se abocan a problemas asociados con el desarrollo de subsistemas y componentes, la manera en la que se comunican entre sí y su ubicación en una arquitectura mayor. Los patrones de diseño de interfaces describen problemas comunes de la interfaz de usuario y su solución con un sistema de fuerzas que incluye las características específicas de los usuarios finales. Los patrones de *webapps* se enfrentan a un conjunto de problemas que surgen cuando se construyen *webapps*, y es frecuente que incorporen muchas categorías de los patrones mencionados. Una estructura proporciona el marco en el que residen los patrones y los idiomas describen los detalles de implementación específicos del lenguaje de programación para un algoritmo, o parte de él, o para una estructura de datos específica. Para hacer las descripciones de patrones, se emplea un formato o plantilla estándar. Un lenguaje de patrón incluye un conjunto de patrones, cada uno de los cuales es descrito con el empleo de una plantilla estándar e interrelacionada para que muestre cómo colaboran los patrones para resolver problemas en el dominio de la aplicación.

El diseño basado en patrones se utiliza junto con métodos de diseño arquitectónico, en el nivel de componentes y de interfaz de usuario. El enfoque del diseño comienza con el estudio del modelo de requerimientos a fin de detectar los problemas, definir el contexto y describir el sistema de fuerzas. A continuación se buscan los lenguajes de patrón para el dominio del problema con objeto de determinar si existen patrones para los problemas detectados. Una vez que se han encontrado los patrones apropiados, se usan como guía para el diseño.

## Problemas y puntos por evaluar

**12.1.** Analice las tres "partes" de un patrón de diseño y dé un ejemplo concreto de cada uno en algún campo distinto del software.

**12.2.** ¿Cuál es la diferencia entre un patrón no generativo y uno generativo?

**12.3.** ¿En qué difieren los patrones arquitectónicos de los patrones de componentes?

**12.4.** ¿Qué es estructura y en qué difiere de un patrón? ¿Qué es un idioma y en qué se diferencia de un patrón?

**12.5.** Con el uso de la plantilla de diseño de patrones presentada en la sección 12.1.3, desarrolle la descripción completa de un patrón sugerido por su profesor.

**12.6.** Desarrolle un lenguaje de esqueleto de patrón para un deporte con el que esté familiarizado. Comience por abordar el contexto, el sistema de fuerzas y los problemas amplios que deban resolver un entrenador y su equipo. Sólo necesita especificar los nombres de los patrones y hacer la descripción frase por frase de cada uno.

www.FreeLibros.me

**12.7.** Encuentre cinco repositorios de patrones y presente la descripción abreviada de los tipos de patrones que contenga cada uno.

**12.8.** Cuando Christopher Alexander afirma que "un buen diseño no se logra con sólo reunir las partes ejecutantes", ¿qué cree usted que quiere decir?

**12.9.** Con el uso de las tareas de diseño basado en patrones mencionadas en la sección 12.2.3, desarrolle un diseño de esqueleto para el "sistema de diseño de interiores" descrito en la sección 11.3.2.

**12.10.** Elabore una tabla de organización de patrones para aquellos que haya utilizado en el problema 12.9.

**12.11.** Con el uso de la plantilla para diseñar patrones que se presentó en la sección 12.1.3, desarrolle la descripción completa para el patrón **Cocina** mencionado en la sección 12.3.

**12.12.** La banda de los cuatro [Gam95] ha propuesto varios patrones de componentes aplicables a sistemas orientados a objetos. Seleccione uno de ellos (disponibles en web) y analícelo.

**12.13.** Encuentre tres repositorios de patrones de interfaz de usuario. Seleccione uno de cada repositorio y haga una descripción breve de él.

**12.14.** Encuentre tres repositorios de patrones para *webapps*. Seleccione uno de cada repositorio y descríbalos brevemente.

## Lecturas adicionales y fuentes de información

En la última década se han escrito muchos libros sobre el diseño de patrones destinados a los ingenieros de software. Gamma *et al.* [Gam95] escribieron un libro fundamental sobre dicho tema. Las contribuciones más recientes incluyen los textos de Lasater (*Design Patterns*, Wordware Publishing, Inc., 2007), Holzner (*Design Patterns for Dummies*, For Dummies, 2006), Freeman *et al.* (*Head First Design Patterns*, O'Reilly Media, Inc., 2005) y Shalloway y Trott (*Design Patterns Explained*, 2a. ed., Addison-Wesley, 2004). Una edición especial de *IEEE Software* (julio/agosto, 2007) estudia una amplia variedad de temas sobre patrones de software. Kent Beck (*Implementation Patterns*, Addison-Wesley, 2008) estudia patrones para codificar e implementar aspectos que se encuentran durante la actividad de construcción.

Otros libros se centran en patrones de diseño según se encuentran en el desarrollo de aplicaciones específicas y ambientes de lenguajes. Las contribuciones en esta área incluyen las siguientes: Bowers (*Pro CSS and HTML Design Patterns*, Apress, 2007), Tropashko y Burleson (*SQL Design Patterns: Expert Guide to SQL Programming*, Rampant Techpress, 2007), Mahemoff (*Ajax Design Patterns*, O'Reilly Media, Inc., 2006), Metsker y Wake (*Design Patterns in Java*, Addison-Wesley, 2006), Nilsson (*Applying Domain-Driven Design and Patterns: With Examples en C# and .NET*, Addison-Wesley, 2006), Sweat (*PHPArchitect's Guide to PHP Design Patterns*, Marco Tabini & Associates, Inc., 2005), Metsker (*Design Patterns C#*, Addison-Wesley, 2004), Grand y Merrill (*Visual Basic .NET Design Patterns*, Wiley, 2003), Crawford y Kaplan (*J2EE Design Patterns*, O'Reilly Media, Inc., 2003), Juric *et al*. (*J2EE Design Patterns Applied*, Wrox Press, 2002), y Marinescu y Roman (*EJB Design Patterns*, Wiley, 2002).

Otros libros se abocan a dominios de aplicaciones específicas. Entre éstos están las contribuciones de Kuchana (*Software Architecture Design Patterns in Java*, Auerbach, 2004), Joshi (C++ *Design Patterns and Derivatives Pricing*, Cambridge University Press, 2004), Douglass (*Real-Time Design Patterns*, Addison-Wesley, 2002) y Schmidt y Rising (*Design Patterns in Communication Software*, Cambridge University Press, 2001).

La lectura de los libros clásicos escritos por el arquitecto Christopher Alexander (*Notes on the Synthesis or Form,* Harvard University Press, 1964, y *A Pattern Language: Towns, Builidings, Construction*, Oxford University Press, 1977) es útil para el diseñador de software que trata de entender por completo los patrones de diseño.

En internet hay una amplia variedad de fuentes de información sobre el diseño de patrones. En el sitio web del libro se encuentra la lista actualizada de referencias en la red mundial que son relevantes para el diseño basado en patrones: **www.mhhe.com/engsc/compsci/pressman/professional/olc/ser.htm**.

www.FreeLibros.me

CAPÍTULO

# 13

# DISEÑO DE WEBAPPS



En su autorizado libro sobre diseño web, Jakob Nielsen [Nie00] afirma lo siguiente: "En esencia, hay dos enfoques fundamentales del diseño: el ideal artístico de expresarse a sí mismo y el ideal de la ingeniería de resolver un problema para un cliente." En la primera década del desarrollo de la web, el enfoque que elegían muchos diseñadores era el ideal artístico. El diseño se desarrollaba de manera *ad hoc* y por lo general se efectuaba a medida que se generaba HTML. Después evolucionó a partir de la visión artística que surgió de la construcción de *webapps*.

Incluso hoy, muchos desarrolladores web utilizan *webapps* como cartel infantil para un "diseño limitado". Afirman que la inmediatez y volatilidad de una *webapp* palidecen ante el diseño formal, que éste evoluciona conforme se elabora (se codifica) una aplicación y que debe dedicarse relativamente poco tiempo a crear un modelo detallado del diseño. Este argumento tiene algo de verdad, pero sólo para *webapps* relativamente sencillas. Cuando el contenido y las funciones son complejos o cuando el tamaño de la *webapp* incluye cientos o miles de objetos de contenido, funciones y clases de análisis y cuando el éxito de la *webapp* tenga influencia directa en el éxito del negocio, el diseño no puede ni debe tomarse a la ligera.

Esta realidad conduce al segundo enfoque de Nielsen: "el ideal de la ingeniería es resolver un problema para un cliente". La ingeniería web[1] adopta esta filosofía, y un enfoque más riguroso del diseño de *webapps* permite que los desarrolladores la hagan realidad.

## UNA MIRADA RÁPIDA

**¿Qué es?** El diseño de *webapps* incluye actividades técnicas y no técnicas que incluyen lo siguiente: establecer la vista y sensación de la *webapp*, creando la distribución estética de la interfaz de usuario, definiendo la estructura arquitectónica general, desarrollando el contenido y la funcionalidad que residen en la arquitectura y planeando la navegación que ocurre dentro de la *webapp*.

**¿Quién lo hace?** En la creación del modelo del diseño de una *webapp*, intervienen ingenieros web, diseñadores gráficos, desarrolladores de contenido y varios participantes más.

**¿Por qué es importante?** El diseño permite crear un modelo que se evalúe respecto de su calidad para mejorarlo antes de la generación de contenido y código, de la realización de las pruebas y del involucramiento de un gran número de usuarios. El diseño es el lugar donde se establece la calidad de la *webapp*.

**¿Cuáles son los pasos?** El diseño de una *webapp* incluye seis etapas principales que son orientadas por la información obtenida durante la modelación de los requerimientos. El diseño del contenido utiliza el contenido del modelo (desarrollado durante el análisis) como la base para establecer el diseño de los objetos del contenido. El diseño estético (también llamado diseño gráfico) establece la vista y sensación que el usuario final percibe. El diseño arquitectónico se centra en la estructura general de hipermedios de todos los objetos y funciones del contenido. El diseño de la interfaz establece la distribución y mecanismos de distribución que definen a la interfaz de usuario. El diseño de la navegación define la forma en la que el usuario final navega a través de la estructura de hipermedios. Y el diseño de los componentes representa la estructura interna detallada de los elementos funcionales de la *webapp*.

**¿Cuál es el producto final?** El principal producto que se genera durante el diseño de la *webapp* es un modelo del diseño que incluye aspectos del diseño del contenido, de la estética, de la arquitectura, de la interfaz, de la navegación y en el nivel de componentes.

**¿Cómo me aseguro de que lo hice bien?** Cada elemento del modelo del diseño se revisa para descubrir errores, inconsistencias u omisiones. Además, se toman en cuenta soluciones alternativas y se evalúa el grado en el que el modelo actual del diseño llevará a una implementación eficaz.

1 La *ingeniería web* [Pre08] es una versión adaptada del enfoque de ingeniería de software que se presenta en todo este libro. Propone una estructura ágil, pero disciplinada, para construir sistemas y aplicaciones basados en web con calidad industrial.



www.FreeLibros.me

## 13.1 Calidad del diseño de webapps

El diseño es la actividad de la ingeniería que genera un producto de alta calidad. Esto lleva a una pregunta recurrente que surge en todas las disciplinas de la ingeniería: ¿qué es calidad? En esta sección estudiaremos la respuesta en el contexto del desarrollo de *webapps*.

Toda persona que haya navegado en la red mundial o que haya utilizado una intranet corporativa se ha formado una opinión sobre lo que constituye una "buena" *webapp*. Los puntos de vista individuales varían mucho. A algunos usuarios les gustan los gráficos brillantes, otros prefieren el texto simple, algunos más demandan mucha información, mientras los hay que desean una presentación abreviada. A algunos les agradan las herramientas analíticas sofisticadas o tener acceso a bases de datos y a otros les gusta lo sencillo. En realidad, la percepción del usuario acerca de lo que es "bueno" (y en consecuencia la aceptación o rechazo de la *webapp*) puede ser un aspecto más importante que cualquier otro de índole técnica sobre la calidad de las *webapps*.

**Cita:**

"Si los productos se diseñan para que se ajusten mejor a las tendencias naturales del comportamiento humano, entonces las personas estarán más satisfechas, más complacidas y serán más productivas."

**Susan Weinschenk**

Pero, ¿cómo se percibe la calidad de una *webapp*? ¿Qué atributos debe tener para que haya bondad ante los ojos de los usuarios finales y a la vez existan las características técnicas de calidad que permitan corregir, adaptar, mejorar y dar apoyo a la aplicación en el largo plazo?

En realidad, a las *webapps* se aplican todas las características técnicas de calidad del diseño estudiadas en el capítulo 8 y los atributos generales de calidad que se vieron en el capítulo 14. Sin embargo, los más relevantes de estos atributos generales —usabilidad, funcionalidad, confiabilidad, eficiencia y susceptibilidad de recibir mantenimiento— brindan una base útil para evaluar la calidad de los sistemas basados en web.

Olsina *et al.* [Ols99] han preparado un "árbol de requerimientos de calidad" que identifica un conjunto de atributos técnicos —usabilidad, funcionalidad, confiabilidad, eficiencia y susceptibilidad de recibir mantenimiento— que generan la calidad en las *webapps*.[2] La figura 13.1 resume su trabajo. Los criterios que ahí aparecen tienen interés especial si el lector diseña, construye y da mantenimiento a *webapps* a largo plazo.

Offutt [Off02] agrega los siguientes a los cinco atributos principales de la calidad que se mencionan en la figura 13.1:

**¿Cuáles son los principales atributos de la calidad de las *webapps*?**

**Seguridad.** Las *webapps* se han integrado mucho con bases de datos críticas, corporativas y gubernamentales. Las aplicaciones de comercio electrónico extraen y después almacenan información delicada para el cliente. Por estas y muchas otras razones, la seguridad de las *webapps* tiene importancia capital en muchas situaciones. La medida clave de la seguridad de una *webapp* y de su ambiente de servidor es su capacidad para rechazar los accesos no autorizados o para detener un ataque proveniente del exterior. El análisis detallado de la seguridad de las *webapps* está más allá del alcance de este libro. Si el lector está interesado en este tema, puede consultar [Vac06], [Kiz05] o [Kal03].

**Disponibilidad.** Aun la mejor *webapp* será incapaz de satisfacer las necesidades de los usuarios si no se encuentra disponible. En sentido técnico, la disponibilidad es la medida porcentual del tiempo que una *webapp* puede utilizarse. El usuario final común espera que las webapps se hallen disponibles las 24 horas de los 365 días del año. Algo menos que eso es tomado como inaceptable.[3] Pero el "tiempo arriba" no es el único indicador de la disponibilidad. Offutt [Off02] sugiere que "el empleo de características que sólo se encuentren disponibles en un navegador o plataforma" hace que quienes tengan una configuración diferente de navegador o plataforma no puedan utilizar la *webapp*. Invariablemente, el usuario irá a otro sitio.

2 Estos atributos de la calidad son similares a los que se mencionan en los capítulos 8 y 14. Esto implica que las características de la calidad son universales para todo el software.

3 Por supuesto, esta expectativa no es realista. Para las *webapps* importantes, deben programarse "tiempos fuera" a fin de que reciban arreglos y actualizaciones.

www.FreeLibros.me


**FIGURA 13.1**
**Árbol de requerimientos de la calidad.**
*Fuente:* [Ols99].

**Escalabilidad.** ¿Una *webapp* y su ambiente de servidor pueden crecer para que manejen 100, 1 000, 10 000 o 100 000 usuarios? ¿La *webapp* y los sistemas con los que tiene interfaz son capaces de manejar una variación significativa del volumen o su respuesta se desplomará (o cesará)? No basta construir una *webapp* exitosa. También es importante que pueda asimilar la carga del éxito (muchos más usuarios) y que tenga aún más éxito.

**Tiempo para llegar al mercado.** Aunque el tiempo que toma llegar al mercado en realidad no es un atributo de la calidad en el sentido técnico, sí lo es desde el punto de vista de la empresa. Es frecuente que la primera *webapp* que llega a un segmento específico del mercado obtenga un número desproporcionado de usuarios finales.

**INFORMACIÓN**

### *Diseño de una webapp. Lista de revisión de la calidad*

La siguiente es una lista adaptada de la información contenida en **Webreference.com**, donde se plantean preguntas que ayudarán a los diseñadores y a los usuarios finales de la *webapp* a evaluar la calidad de la aplicación:

- ¿Es posible adaptar las opciones de contenido, función o navegación a las preferencias del usuario?
- ¿Puede personalizarse el contenido o la funcionalidad al ancho de banda con el que se comunica el usuario?
- ¿Se han utilizado de manera apropiada las imágenes y otros medios distintos del texto? ¿Es posible adaptar el tamaño de los archivos de imagen para mejorar la eficiencia de la pantalla?
- ¿Las tablas están organizadas y tienen un tamaño tal que se entienden y despliegan de modo eficiente?
- ¿El HTML está optimizado a fin de eliminar las ineficiencias?
- ¿El diseño general de la página tiene lectura y navegación fáciles?
- ¿Todos los vínculos llevan a información de interés para los usuarios?
- ¿Es probable que la mayor parte de vínculos persistan en la red mundial?
- ¿La *webapp* tiene herramientas de administración del sitio, tales como historial del uso, prueba de vínculos, búsqueda local y seguridad?

Aquellos que buscan información disponen de miles de millones de páginas web. Aun las búsquedas bien dirigidas en la red mundial generan una avalancha de contenidos. Con tantas fuentes de información entre las cuales elegir, ¿cómo evalúa el usuario la calidad (es decir, la veracidad, exactitud, completitud, oportunidad, etc.) del contenido que presenta una *webapp*? Tillamn [Til00] sugiere el siguiente conjunto de criterios útiles para ello:

www.FreeLibros.me

**¿Qué debe considerarse cuando se evalúa la calidad del contenido?**

- ¿Es fácil determinar el alcance y la profundidad del contenido a fin de estar seguros de que satisface las necesidades del usuario?
- ¿Puede identificarse fácilmente la formación y la autoridad de los autores del contenido?
- ¿Es posible determinar la actualidad del contenido, la fecha de su última actualización y en qué consistió ésta?
- ¿El contenido y su ubicación son estables (permanecerán en la URL de referencia)?

Además de estas preguntas relacionadas con el contenido, deben agregarse las siguientes:

- ¿Es creíble el contenido?
- ¿El contenido es único?, es decir, ¿la *webapp* brinda algún beneficio único a quienes la emplean?
- ¿Es valioso el contenido para la comunidad de usuarios a la que se dirige?
- ¿Está bien organizado el contenido? ¿Está indizado? ¿Se accede a él con facilidad?

Las listas de comprobación citadas en esta sección representan sólo una muestra pequeña de los aspectos que deben estudiarse a medida que el diseño de la *webapp* evoluciona.

## 13.2 METAS DEL DISEÑO

En su columna periódica sobre el diseño web, Jean Kaiser [Kai02] sugiere un conjunto de metas para el diseño que son aplicables virtualmente a toda *webapp*, sin importar su dominio de aplicación, tamaño o complejidad.

**Cita:**

"Que algo pueda hacerse, no significa que deba hacerse."

**Jean Kaiser**

**Simplicidad:** Aunque parezca algo pasado de moda, el aforismo "todo con moderación" es aplicable a las *webapps*. Existe una tendencia entre ciertos diseñadores a dar al usuario final "demasiado": contenido exhaustivo, extremos visuales, animaciones intrusas, páginas web enormes... y la lista sigue. Es mejor moderación y simplicidad.

El contenido debe ser informativo pero sucinto y debe utilizar un modo de entrega (texto, imágenes, video o audio) que resulte apropiado para la información que se envíe. La estética debe ser agradable pero no abrumadora (demasiados colores tienden a distraer al usuario en vez de mejorar la interacción). La arquitectura debe lograr los objetivos de la *webapp* de la manera más sencilla posible. La navegación debe ser directa y sus mecanismos, obvios para la intuición del usuario final. Las funciones deben ser fáciles de utilizar y más fáciles de entender.

**Cita:**

"Para ciertas personas, el diseño web se centra en el aspecto visual y en la percepción… Para otras, se trata de estructurar la información y la navegación a través del espacio del documento. Otras más consideran incluso que el diseño web es tecnología… En realidad, el diseño incluye todo esto y tal vez más."

**Thomas Powell**

**Consistencia.** Esta meta del diseño se aplica virtualmente a todo elemento del modelo del diseño. El contenido debe construirse de modo congruente (formato y tipografía del texto deben ser los mismos en todos los documentos de texto; las imágenes deben tener coherencia en su aspecto, color y estilo). El diseño gráfico (estética) debe presentar una vista consistente en todas las partes de la *webapp*. El diseño arquitectónico debe establecer plantillas que generen una estructura de hipermedios constante. El diseño de la interfaz debe definir modos consistentes de interacción, navegación y despliegue del contenido. Los mecanismos de navegación deben usarse de manera consistente en todos los elementos de la *webapp*. Como dice Kaiser [Kai02]: "recuerde que para un visitante, un sitio web es un lugar físico. Si sus páginas no tienen un diseño consistente, son fuente de confusión".

**Identidad.** El diseño de la estética, la interfaz y la navegación de una *webapp* deben ser consistentes con el dominio de la aplicación para la que se va a elaborar. Un sitio web para un grupo de *hip-hop* sin duda tendrá un aspecto y sensación distintos que una *webapp* diseñada para una compañía de servicios financieros. La arquitectura de la *webapp* será diferente por completo, las

www.FreeLibros.me

interfaces se construirán para que reciban a distintas categorías de usuarios, la navegación se organizará para que cumpla objetivos diferentes. Usted (y quienes contribuyan al diseño) debe trabajar para establecer la identidad de la *webapp* por medio del diseño.

**Robustez.** Con base en la identidad que se haya establecido, es frecuente que una *webapp* haga una "promesa" implícita al usuario. Éste espera contenido y funciones robustas que sean relevantes para sus necesidades. Si no existen o son insuficientes, es probable que la *webapp* fracase.

**Navegabilidad.** Ya se dijo que la navegación debe ser sencilla y consistente. También debe estar diseñada en forma tal que sea intuitiva y predecible. Es decir, el usuario debe comprender cómo moverse por la *webapp* sin tener que buscar vínculos o instrucciones para la navegación. Por ejemplo, si un campo de iconos gráficos o de imágenes contiene algunos que serán usados como mecanismos de navegación, deben identificarse visualmente. Nada es más frustrante que intentar encontrar el vínculo vivo apropiado entre muchas imágenes.

También es importante colocar los vínculos hacia el contenido y las funciones de la *webapp* en una ubicación predecible en cada página web. Si se requiere desplazar la página (lo que sucede con frecuencia), los vínculos situados en las partes superior e inferior de la página hacen que las tareas de navegación del usuario sean más fáciles.

**Atractivo visual.** De todas las categorías de software, las aplicaciones web son indiscutiblemente las más visuales, dinámicas y estéticas. La belleza (atractivo visual) radica sin lugar a dudas en los ojos del espectador, pero muchas características del diseño (aspecto y sensación del contenido, distribución de la interfaz, coordinación del color, balance del texto, imágenes y otros medios) aumentan el atractivo visual.

**Compatibilidad.** Una *webapp* se usará en varios ambientes (distinto hardware, tipos de conexión, sistemas operativos, navegadores, etcétera) y debe diseñarse para que sea compatible con cada uno.

## 13.3 PIRÁMIDE DEL DISEÑO DE WEBAPPS

**Cita:**

"Si un sitio es perfectamente utilizable, pero carece de un estilo elegante y apropiado, fracasará."

Curt Cloninger

¿Qué es el diseño de una *webapp*? Esta sencilla pregunta es más difícil de responder de lo que se creería. En nuestro libro [Pre08] de ingeniería web, David Lowe y yo lo analizamos del modo siguiente:

> La creación de un diseño eficaz requerirá por lo general de un conjunto diversificado de aptitudes. En ocasiones, para proyectos pequeños, un desarrollador único necesitará tener varias habilidades. Para los proyectos grandes, es aconsejable o factible confiar en la experiencia de especialistas: ingenieros web, diseñadores gráficos, desarrolladores de contenido, programadores, especialistas de bases de datos, arquitectos de la información, ingenieros de redes, expertos en seguridad y probadores. Depender de estas distintas aptitudes permite la creación de un modelo cuya calidad puede evaluarse a fin de mejorar su contenido y su código *antes* de que se generen contenido y código, de que se realicen pruebas y de que se involucre un gran número de usuarios. Si el análisis reside en donde *se establece la calidad de la webapp*, entonces el diseño está donde *la calidad está en verdad incrustada*.

La mezcla apropiada de habilidades de diseño variará en función de la naturaleza de la *webapp*. La figura 13.2 ilustra la pirámide del diseño de las *webapps*. Cada nivel representa una acción del diseño que se describe en las siguientes secciones.

## 13.4 DISEÑO DE LA INTERFAZ DE LA WEBAPP

Cuando el usuario interactúa con un sistema basado en computadora, se aplica un conjunto de principios fundamentales y lineamientos generales de diseño. Éstos se estudiaron en el capítulo

www.FreeLibros.me

**FIGURA 13.2**
**Pirámide del diseño de las webapps**


11.[4] Aunque las *webapps* plantean algunas dificultades especiales en el diseño de la interfaz de usuario, los principios y lineamientos básicos son aplicables.

Uno de los retos del diseño de la interfaz de las *webapps* es la naturaleza indeterminada del punto en el que entra el usuario. Es decir, éste puede ingresar por una ubicación "inicial" de la *webapp* (la página de arranque, por ejemplo) o por algún vínculo en cierto nivel inferior de la arquitectura de aquélla. En algunos casos, la *webapp* se diseña de modo que redirija al usuario a una ubicación inicial, pero si esto es algo indeseable, entonces el diseño debe dar características de navegación en la interfaz que acompañen a todos los objetos de contenido y de las cuales se disponga sin importar el modo en el que el usuario ingrese al sistema.

Los objetivos de la interfaz de una *webapp* son los siguientes: 1) establecer una ventana congruente en el contenido y las funciones que brinda, 2) guiar al usuario a través de una serie de interacciones con la *webapp* y 3) organizar las opciones de navegación y contenido disponibles para el usuario. Para lograr una interfaz consistente, primero debe usarse un diseño estético (véase la sección 13.5) a fin de establecer un "aspecto" coherente. Esto incluye muchas características, pero debe ponerse énfasis en la distribución y la forma de los mecanismos de navegación. Para guiar la interacción del usuario, debe establecerse una metáfora[5] apropiada que permita al usuario tener una comprensión intuitiva de la interfaz. A fin de implementar las opciones de navegación, puede seleccionarse alguno de los siguientes mecanismos:

**? ¿De qué mecanismos de interacción disponen los diseñadores de *webapps*?**

- *Menús de navegación*: contienen palabras clave (organizadas en forma vertical u horizontal) que enlistan contenido o funciones clave. Estos menús se implementan de modo que el usuario pueda elegir entre una jerarquía de subtemas que se despliegan al seleccionar la opción principal en el menú.
- *Iconos gráficos*: botones, interruptores y otras imágenes similares que permiten que el usuario seleccione alguna propiedad o que especifique una decisión.
- *Imágenes*: cierta representación gráfica que el usuario selecciona para establecer un vínculo hacia un objeto de contenido o función de la *webapp*.

4 La sección 11.5 está dedicada al diseño de la interfaz de la *webapp*. Si aún no se ha leído, es el momento de hacerlo.

5 En este contexto, una metáfora es la representación (establecida por la experiencia del usuario con el mundo real) que se modela en el contexto de la interfaz. Un ejemplo sencillo sería un interruptor deslizable que se utilice para controlar el volumen auditivo de un archivo .mpg.

www.FreeLibros.me

Es importante observar que en cada nivel de la jerarquía del contenido debe proporcionarse uno o varios de estos mecanismos de control.

## 13.5 DISEÑO DE LA ESTÉTICA

**? No todo ingeniero web (o de software) tiene talento artístico (estético). Si el lector se encuentra en esta categoría, contrate un diseñador gráfico experimentado para que haga el trabajo de diseño estético.**

El diseño estético, también llamado *diseño gráfico*, es una actividad artística que complementa los aspectos técnicos del diseño de las *webapps*. Sin estética, una *webapp* tal vez sea funcional pero no atractiva. Con estética, una *webapp* lleva a sus usuarios a un mundo que los sitúa en un nivel tanto visceral como intelectual.

Pero, ¿qué es la estética? Hay un viejo refrán que dice que "la belleza está en los ojos del espectador". Esto es particularmente cierto cuando se trata del diseño estético de las *webapps*. Para llevar a cabo éste con eficacia, hay que volver a la jerarquía del usuario desarrollada como parte del modelo de requerimientos (véase el capítulo 5) y preguntar: *¿quiénes son los usuarios de la webapp y qué "vista" desean tener?*

### 13.5.1 Aspectos de la distribución

Toda página web tiene una cantidad limitada de "superficie" que se utiliza para dar apoyo a la estética no funcional, características de navegación, contenido de información y funciones dirigidas al usuario. El desarrollo de dicha superficie se planea durante el diseño estético.

**Cita:**

"Descubrimos que las personas evalúan rápidamente un sitio tan sólo por su diseño visual."

*Lineamientos Stanford para la credibilidad en web*

Igual que todos los temas de la estética, cuando se diseña la distribución de la pantalla no hay reglas absolutas. Sin embargo, es útil considerar varios lineamientos de la distribución general:

**No tema al espacio en blanco.** No es aconsejable ocupar con información cada centímetro cuadrado de una página web. El amasijo resultante hará difícil que el usuario identifique la información o las características que necesita y creará un caos visual que no será agradable a los ojos.

**Hacer énfasis en el contenido.** Después de todo, ésta es la razón de que el usuario esté ahí. Nielsen [Nie00] sugiere que la página web común debe tener 80 por ciento de contenido y destinar el resto a la navegación y otras características.

**Organizar los elementos con una distribución que vaya desde arriba a la izquierda hacia abajo a la derecha.** La gran mayoría de usuarios de una página web la recorrerán en forma muy parecida a como lo hacen con las hojas de un libro: desde arriba a la izquierda hacia abajo a la derecha.[6] Si los elementos de la distribución tienen prioridades específicas, aquellos que sean prioritarios deben colocarse en la parte superior izquierda de la superficie de la página.

**Agrupar la navegación, el contenido y la función en forma geográfica dentro de la página.** Los humanos buscamos patrones virtualmente en todas las cosas. Si en una página web no hay patrones discernibles, es probable que la frustración del usuario aumente (debido a la búsqueda innecesaria de la información requerida).

**CONSEJO**

*Los usuarios tienden a tolerar el desplazamiento vertical mejor que el horizontal. Evite los formatos anchos para la página.*

**No aumente la superficie con la barra de desplazamiento.** Aunque es frecuente que se necesite el desplazamiento, la mayor parte de estudios indican que los usuarios preferirían no hacerlo. Es mejor reducir el contenido de la página o presentar en varias páginas el que sea necesario.

**Cuando se diseñe la distribución hay que considerar la resolución y tamaño de la ventana del navegador.** En vez de definir tamaños fijos dentro de una plantilla, el diseño debe especificar todos los parámetros en términos de porcentaje del espacio disponible [Nie00].

6 Hay excepciones culturales y lingüísticas, pero esta regla se aplica a la mayor parte de usuarios.

www.FreeLibros.me

### 13.5.2 Aspectos del diseño gráfico

El diseño gráfico toma en cuenta cada aspecto de la vista y sensación de la *webapp*. El proceso de diseño gráfico comienza con la distribución (véase la sección 13.5.1) y avanza hacia la consideración de los esquemas de color globales; tipos, tamaños y estilos del texto; uso de medios complementarios (audio, video y animación) y todos los demás elementos estéticos de una aplicación.

El análisis exhaustivo de los temas del diseño gráfico de *webapps* está más allá del alcance de este libro. El lector puede obtener recomendaciones y lineamientos en muchos sitios web dedicados a dicha materia (como **www.graphic-design.com**, **www.grantasticdesigns.com** y **www.wpdfd.com**) o en uno o más documentos impresos (como [Roc06] y [Gor02]).

**INFORMACIÓN**

***Sitios web bien diseñados***

A veces, la mejor manera de entender lo que es un buen diseño de *webapps* es ver algunos ejemplos. En su artículo "Las veinte mejores recomendaciones para el diseño web", Marcelle Toor (**www.graphic-design.com/Web/feature/tips.html**) recomienda los siguientes sitios como ejemplos de lo que constituye un buen diseño gráfico:

**www.creativepro.com/designresource/home/787.html:** empresa de diseño dirigida por Primo Angeli

**www.workbook.com:** este sitio presenta los portafolios de varios ilustradores y diseñadores

**www.pbs.org/riverofsong:** serie de televisión y radio públicas acerca de la música estadounidense

**www.RKDINC.com:** empresa de diseño con un portafolio en línea y buenas recomendaciones de diseño

**www.creativehotlist.com/index.html:** buena fuente de sitios bien diseñados desarrollados por agencias, empresas de artes gráficas y otros especialistas de la comunicación

**www.btdnyc.com:** compañía de diseño encabezada por Beth Toudreau

## 13.6 DISEÑO DEL CONTENIDO

**Cita:**

"Los buenos diseñadores generan la regularidad a partir del caos; comunican sus ideas con claridad, organizando y manipulando palabras e imágenes."

**Jeffery Veen**

El diseño del contenido se centra en dos tareas diferentes del diseño, cada una de las cuales es dirigida por individuos que poseen habilidades distintas. En primer lugar, se desarrolla una representación del diseño para los objetos del contenido y los mecanismos requeridos para establecer una relación entre ellos. Además, se crea la información dentro de un objeto de contenido específico. El trabajo posterior es llevado a cabo por escritores, diseñadores gráficos y otros actores que generan el contenido que se usará en la *webapp*.

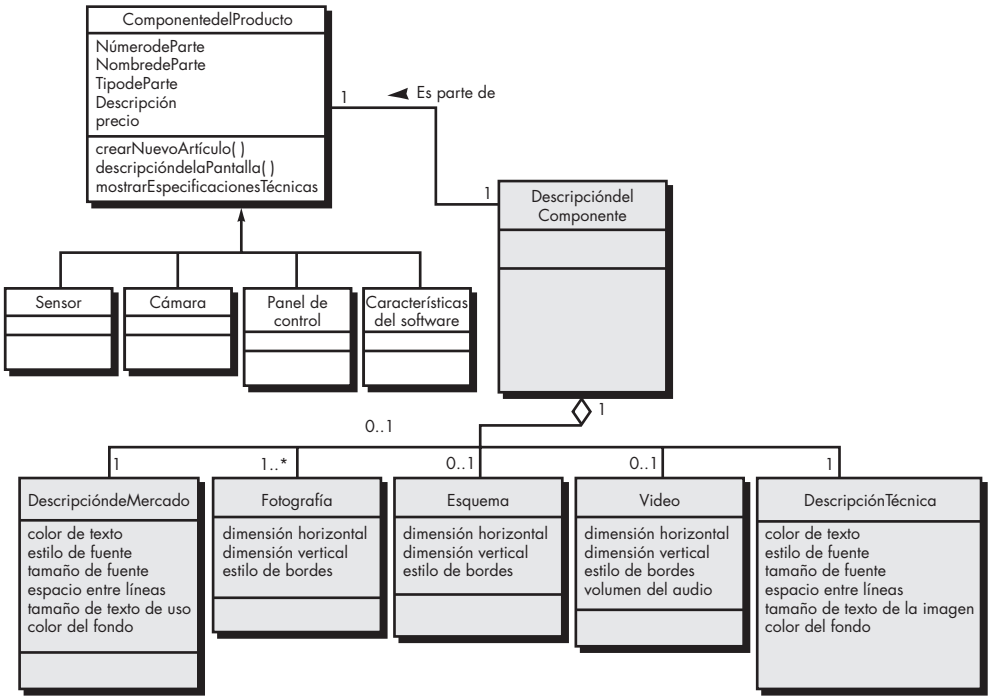
### 13.6.1 Objetos de contenido

La relación entre los objetos de contenido definidos como parte del modelo de requerimientos para la *webapp* y los objetos de diseño que representan el contenido es análoga a la relación que existe entre las clases de análisis y los componentes del diseño que se describió en capítulos anteriores. En el contexto del diseño de la *webapp*, un objeto de contenido se parece más a un objeto de datos del software tradicional. Un objeto de contenido tiene atributos que incluyen información de contenido específico (normalmente definido durante el modelado de los requerimientos de la *webapp*) y atributos de implementación específica que se establecen como parte del diseño.

Por ejemplo, piense en una clase de análisis, **ComponentedelProducto**, desarrollada para el sistema de comercio electrónico de *CasaSegura*. El atributo de la clase de análisis, **descripción**, se representa como clase de análisis llamada **DescripcióndeComponente** y está compuesta por cinco objetos de contenido: **DescripcióndelMercado, Fotografía, DescripciónTécnica, Esquema** y **Video**, que se muestran como objetos sombreados en la figura 13.3. La información

www.FreeLibros.me

**FIGURA 13.3**
**Representación del diseño de objetos de contenido**




contenida dentro del objeto de contenido se etiqueta como atributos. Por ejemplo, **Fotografía** (imagen de tipo .jpg) tiene los atributos **dimensión horizontal, dimensión vertical** y **estilo de bordes**.

Puede usarse una asociación UML y un agregado[7] para representar relaciones entre los objetos de contenido. Por ejemplo, la asociación UML que se ilustra en la figura 13.3 indica que se usa una **DescripcióndeComponente** para cada instancia de la clase **ComponentedelProducto**. La **DescripcióndeComponente** está integrada sobre los cinco objetos de contenido ilustrados. Sin embargo, la notación de multiplicidad que se aprecia indica que **Esquema** y **Video** son opcionales (son posibles 0 ocurrencias), que se requiere una **DescripcióndelMercado** y una **DescripciónTécnica**, y que se emplean una o varias instancias de **Fotografía**.

### 13.6.2 Aspectos de diseño del contenido

Una vez modelados los objetos del contenido, la información que va a entregar cada objeto debe registrar al autor y después editarse para que satisfaga del mejor modo posible las necesidades del consumidor. La autoría del contenido es trabajo de especialistas en el área relevante de quien diseña el objeto de contenido, dando un bosquejo de la información que se va a entregar y una indicación de los tipos de objetos de contenido general (por ejemplo, texto descriptivo, imágenes, fotografías, etc.) que se usarán para entregar la información. El diseño estético (véase la sección 13.5) también puede aplicarse para representar la vista y sensación apropiadas para el contenido.

Los objetos se "cortan" [Pow02] a medida que se diseñan para formar las páginas de la *webapp*. El número de objetos de contenido incorporado en una página individual está en función de las necesidades del usuario, de las restricciones impuestas por la velocidad de descarga de la conexión de internet y de las restricciones impuestas por la cantidad de desplazamiento vertical que el usuario tolerará.

7 En el apéndice 1 se estudian ambas representaciones.

www.FreeLibros.me

## 13.7 Diseño arquitectónico

**Cita:**

"…la estructura arquitectónica de un sitio bien diseñado no siempre es visible para el usuario: no debe serlo."

**Thomas Powell**

El diseño arquitectónico está ligado con las metas establecidas para una *webapp*, con el contenido que se va a presentar, con los usuarios que la visitarán y con la filosofía de navegación adoptada. Como diseñador de la arquitectura, el lector debe identificar la arquitectura del contenido y la de la *webapp*. La *arquitectura del contenido*[8] se centra en la manera en la que los objetos de contenido (o compuestos, como páginas web) se estructuran para la presentación y la navegación. La *arquitectura de la webapp* se aboca a la forma en la que la aplicación queda estructurada para administrar la interacción con el usuario, manejar tareas de procesamiento interno, navegar con eficacia y presentar el contenido.

En la mayoría de los casos, el diseño arquitectónico se lleva a cabo en paralelo con el de la interfaz, el estético y el del contenido. Como la arquitectura de la *webapp* tal vez esté muy influida por la navegación, las decisiones que se tomen durante esta acción del diseño influirán en el trabajo realizado durante el diseño de aquélla.

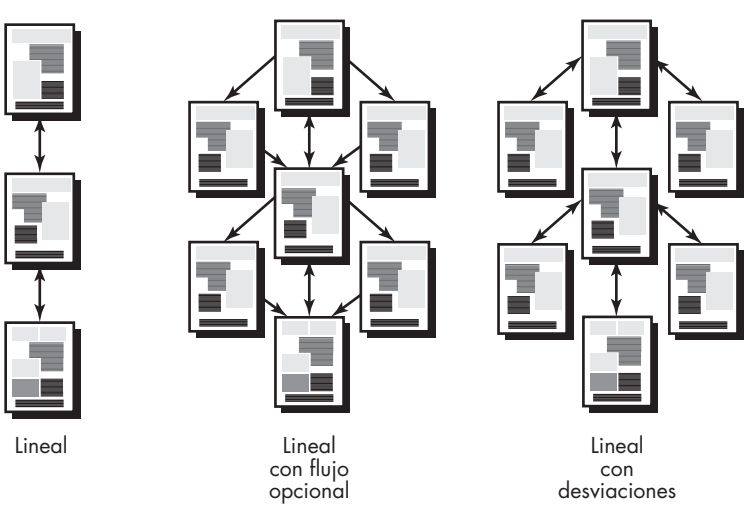
### 13.7.1 Arquitectura del contenido

El diseño del contenido se centra en la definición de la estructura general de los hipermedios de la *webapp*. Aunque en ocasiones se crean arquitecturas personalizadas, siempre se tiene la opción de elegir entre cuatro distintas estructuras de contenido [Pow00]:

**? ¿Qué tipos de arquitectura del contenido es común encontrar?**

Las *estructuras lineales* (véase la figura 13.4) se encuentran cuando es común una secuencia predecible de interacciones (con cierta variación o diferencia). Un ejemplo clásico es la presentación de *tutoriales* en los que se despliegan páginas de información junto con imágenes relacionadas, videos cortos o audio, sólo después de haber mostrado cierta información de prerrequisitos. La secuencia de la presentación del contenido es predefinida y por lo general es lineal. Otro ejemplo sería una secuencia de entrada para ordenar un producto en la que debe proporcionarse información específica en un orden determinado. En tales casos, resultan apropiadas las estructuras mostradas en la figura 13.4. A medida que el contenido y el procesamiento se hacen más complejos, el flujo exclusivamente lineal que se aprecia a la izquierda de la figura da origen a estructuras lineales más complejas en las que puede invocarse contenido alternativo o

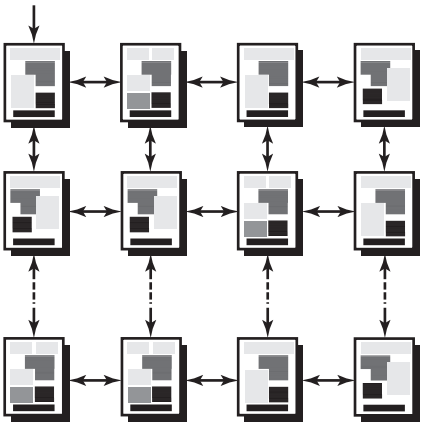
**Figura 13.4** Estructuras lineales




8 El término *arquitectura de la información* también se utiliza para denotar estructuras que produzcan una mejor organización, etiquetado, navegación y búsqueda de objetos de contenido.

www.FreeLibros.me

**FIGURA 13.5**
**Estructura de malla**

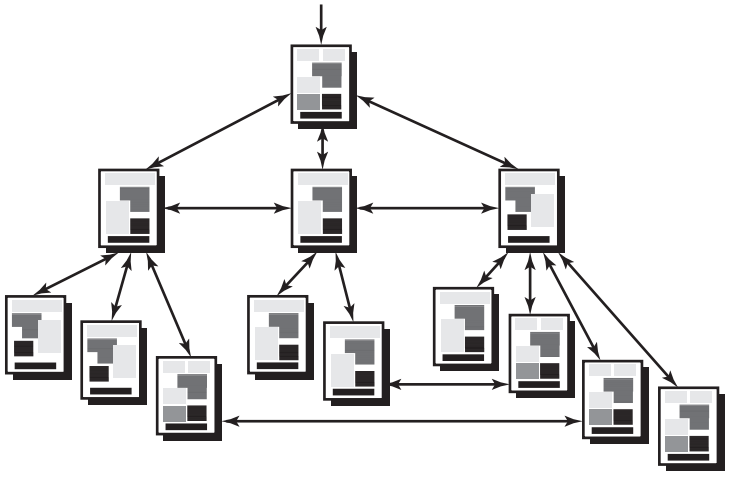


en las que sucede una desviación para adquirir contenido complementario (estructura que aparece en el lado derecho de la figura 13.4).

Las *estructuras de malla* (véase la figura 13.5) son una opción arquitectónica que se aplica cuando es posible organizar el contenido de una *webapp* en forma categórica en dos (o más) dimensiones. Por ejemplo, considere una situación en la que un sitio de comercio electrónico vende palos de golf. La dimensión horizontal de la malla representa el tipo de palo (madera, metal, cuña, mazo, etc.). La dimensión vertical representa las ofertas que hacen los distintos fabricantes de palos de golf. Entonces, un usuario podría navegar por la malla en forma horizontal a fin de encontrar la columna de mazos y después en forma vertical para examinar las ofertas de los fabricantes que los venden. Esta arquitectura de *webapps* es útil sólo cuando se encuentra contenido muy regular [Pow00].

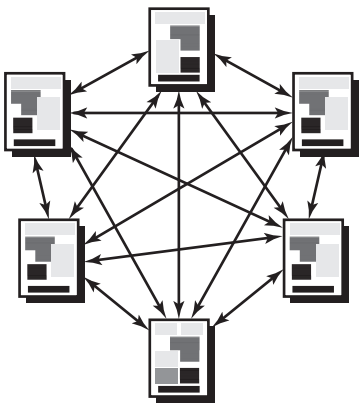
Las *estructuras jerárquicas* (véase la figura 13.6) son sin duda la arquitectura más común de las *webapps*. A diferencia de las jerarquías de software divididas que se estudiaron en el capítulo 9 y que motivan a controlar el flujo sólo a lo largo de las ramas verticales de la jerarquía, la estructura jerárquica de las *webapps* se diseña en forma tal que permite (por medio de la ramificación del hipertexto) que el flujo del control sea en el sentido horizontal a través de las ramas verticales de la estructura. Así, el contenido presentado en la última rama del lado izquierdo de la jerarquía puede tener vínculos de hipertexto que llevan directamente al contenido que existe en la parte media de la rama del lado derecho de la estructura. Sin embargo, debe observarse

**FIGURA 13.6**
**Estructura jerárquica**



www.FreeLibros.me

**FIGURA 13.7**
**Estructura de red**



que aunque dicha ramificación permite una navegación rápida por el contenido de la *webapp*, genera confusión para el usuario.

Una *estructura de red o "telaraña pura"* (véase la figura 13.7) es similar en muchos sentidos a la arquitectura que evoluciona a partir de sistemas orientados a objetos. Los componentes arquitectónicos (páginas web, en este caso) se diseñan de modo que pasan virtualmente el control (por medio de vínculos de hipertexto) a cada componente del sistema. Este enfoque permite una flexibilidad considerable de navegación, pero al mismo tiempo confunde al usuario.

Las estructuras arquitectónicas estudiadas en los párrafos anteriores se combinan para formar *estructuras compuestas*. La arquitectura general de una *webapp* puede ser jerárquica, pero una parte de la estructura puede tener características lineales y otra, forma de red. La meta del diseñador arquitectónico es ajustar la estructura de la *webapp* con el contenido que va a presentarse y con el procesamiento que va a efectuarse.

### 13.7.2 Arquitectura de las webapps

La arquitectura de una *webapp* describe una infraestructura que permite que un sistema o aplicación basados en web alcance sus objetivos empresariales. Jacyntho *et al.* [Jac02b] describe las características básicas de esta infraestructura del modo siguiente:

> Las aplicaciones deben construirse con el empleo de capas en las que se tomen en cuenta distintas preocupaciones; en particular, deben separarse los datos de la aplicación de los contenidos de ésta (nodos de navegación), y éstos, a su vez, deben separarse con toda claridad del aspecto y la sensación de la interfaz (páginas).

Los autores sugieren una arquitectura del diseño en tres capas que desacopla la interfaz de la navegación y del comportamiento de la aplicación. Plantean que mantener separadas la interfaz, la aplicación y la navegación, simplifica la implementación y mejora la reutilización.

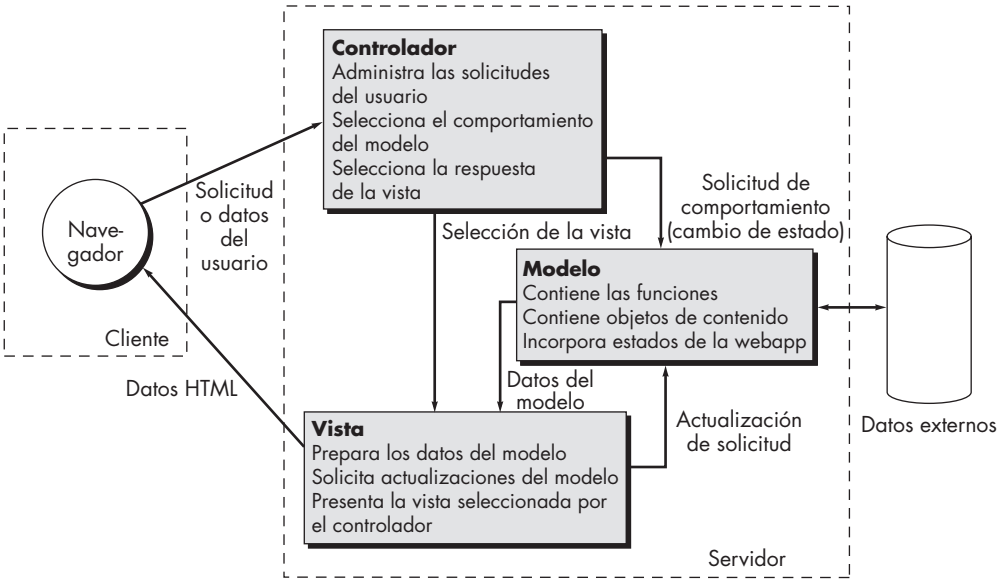
**PUNTO CLAVE**
La arquitectura CVM desacopla la interfaz de usuario de las funciones de la *webapp* y del contenido de información.

La arquitectura de *controlador de la vista del modelo* (CVM) [Kra88] [9] es uno de varios modelos sugeridos para la infraestructura de *webapps* que desacoplan la interfaz de usuario de sus funciones y contenido informativo. El *modelo* (a veces denominado "objeto de modelo") contiene todo el contenido y la lógica de procesamiento específicos de la aplicación, incluso todos los objetos de contenido, acceso a fuentes de datos o información externos y todas las funciones de procesamiento que son específicas de la aplicación. La *vista* contiene todas las funciones específicas de la interfaz y permite la presentación de contenido y lógica de procesamiento, incluidos

9 Debe observarse que el CVM es en realidad un patrón de diseño arquitectónico desarrollado para el ambiente Smalltalk (véase **www.cetus-links.org/oo_smalltalk.html**) que puede usarse para cualquier aplicación interactiva.

www.FreeLibros.me

**FIGURA 13.8**
**La arquitectura CVM**
*Fuente:* Adaptado de [Jac02].




todos los objetos de contenido, el acceso a fuentes de datos o información del exterior y todas las funciones de procesamiento que requiere el usuario final. El *controlador* administra el acceso al modelo y la vista, y coordina el flujo de datos entre ellos. En una *webapp*, "la vista es actualizada por el controlador con datos del modelo, basándose en las entradas que da el usuario" [WMT02]. En la figura 13.8 se muestra una representación de la arquitectura CVM.

En relación con la figura, el controlador maneja las solicitudes o datos del usuario. El controlador también selecciona el objeto de vista que sea aplicable con base en la solicitud del usuario. Una vez determinado el tipo de solicitud, se transmite al modelo un pedido de comportamiento, que implementa la funcionalidad o recupera el contenido requerido para dar acomodo a la solicitud. El objeto de modelo accede a los datos almacenados en una base de datos corporativa, como parte de un almacén de datos locales o como una colección de archivos independientes. El objeto de vista apropiado debe dar formato y organizar los datos desarrollados por el modelo para luego trasmitirlos desde el servidor de la aplicación hacia el navegador del cliente para que se desplieguen en la máquina de éste.

En muchos casos, la arquitectura de la *webapp* se define en el contexto del ambiente de desarrollo en el que va a implementarse la aplicación. Si el lector está interesado, puede consultar en [Fow03] el análisis de los ambientes de desarrollo y el papel que juegan en el diseño de arquitecturas de aplicaciones web.

## 13.8 DISEÑO DE LA NAVEGACIÓN

**Cita:**

"Gretel, sólo espera que salga la luna y veremos las migajas del pan que desmenucé; ellas nos mostrarán el camino de regreso a casa."

***Hansel y Gretel***

Una vez que la arquitectura de la *webapp* ha sido establecida y se han identificado sus componentes (páginas, textos, subprogramas y otras funciones de procesamiento), deben definirse las rutas de navegación que permitan a los usuarios acceder al contenido y a las funciones de la *webapp*. Para lograr esto, debe hacerse lo siguiente: 1) identificar la semántica de navegación para los distintos usuarios del sitio y 2) definir la mecánica (sintaxis) para efectuar la navegación.

### 13.8.1 Semántica de la navegación

Como muchas acciones del diseño de *webapps*, el diseño de la navegación comienza con la consideración de la jerarquía del usuario y los casos de uso relacionados (véase el capítulo 5),

www.FreeLibros.me

desarrollados para cada categoría de usuario (actor). Cada actor puede usar la *webapp* en forma algo diferente, por lo que tendrán distintos requerimientos de navegación. Además, los casos de uso desarrollados por cada actor definirán un conjunto de clases que incluirán uno o más objetos de contenido o funciones de la webapp. A medida que cada usuario interactúe con la *webapp*, encuentra una serie de *unidades semánticas de navegación* (USN): "conjunto de estructuras de información y navegación relacionadas que colaboran para el cumplimiento de un subconjunto de requerimientos del usuario relacionados" [Cac02].

**PUNTO CLAVE**

Una USN describe los requerimientos de navegación para cada caso de uso. En esencia, la USN muestra la forma en la que un actor avanza entre los objetos de contenido o entre las funciones de una *webapp*.

Una USN está compuesta por un conjunto de elementos de navegación llamados [Gna99] *formas de navegar* (FdN). Una FdN representa la mejor ruta de navegación a fin de lograr una meta para un tipo de usuario específico. Cada FdN está organizada como un conjunto de *nodos de navegación* (NN) conectados por vínculos. En ciertos casos, un vínculo navegable es otra USN. Entonces, la estructura de navegación general de una *webapp* está organizada como jerarquía de USN.

Para ilustrar el desarrollo de una USN, considere el caso de uso **SeleccionarComponentes deCasaSegura:**

> **Caso de uso: Seleccionar Componentes de CasaSegura**
>
> La *webapp* recomendará componentes del producto (como paneles de control, sensores, cámaras, etc.) y otras características (como funciones con base en PC implementadas en el software) para cada habitación y entrada exterior. Si se piden alternativas, la *webapp* las dará, en caso de que existan. Podré obtener información descriptiva y de precios de cada componente del producto. La *webapp* creará y mostrará una cuenta de los materiales conforme seleccione distintos componentes. Podré dar un nombre a la cuenta de los materiales y guardarla para futuras referencias (véase el caso de uso **Guardar Configuración**).

Los conceptos subrayados en la descripción del caso de uso representan clases y objetos de contenido que se incorporarán en una o más USN que permitirán que un cliente experimente el escenario descrito en el caso de uso **Seleccionar Componentes de CasaSegura**.

**Cita:**

"El problema de la navegación en un sitio web es conceptual, técnico, espacial, filosófico y logístico. En consecuencia, las soluciones tienden a reclamar combinaciones complejas e improvisadas de arte, ciencia y psicología organizacional."

**Tim Horgan**

La figura 13.9 ilustra un análisis parcial de la semántica de la navegación que implica el caso de uso **Seleccionar Componentes de CasaSegura**. Con el empleo de la terminología ya mencionada, la figura también representa una forma de navegación (FdeN) para la *webapp* **CasaSeguraAsegurada.com**. Las clases importantes de dominio del problema se muestran junto con objetos seleccionados de contenido (en este caso, el paquete de objetos de contenido llamado **DescripcióndeComponentes**, atributo de la clase **ComponentedelProducto**). Estos conceptos son nodos de navegación. Cada flecha representa un vínculo de navegación[10] y tiene la leyenda con la acción iniciada por el usuario que hace que el vínculo tenga lugar.

Es posible crear una USN para cada caso de uso asociado con cada rol de usuario. Por ejemplo, un **cliente nuevo** de **CasaSeguraAsegurada.com** puede tener tres diferentes casos de uso, los cuales dan como resultado el acceso a distintas funciones de información y de *webapp*. Se crea entonces una USN para cada meta.

Durante las etapas iniciales del diseño de la navegación, se evalúa la arquitectura del contenido de la *webapp* a fin de determinar una o más FdN para cada caso de uso. Como ya se dijo, una FdN identifica los nodos de navegación (por ejemplo, contenido) y después los vínculos que permiten navegar entre ellos. Entonces, las FdN están organizadas en USN.

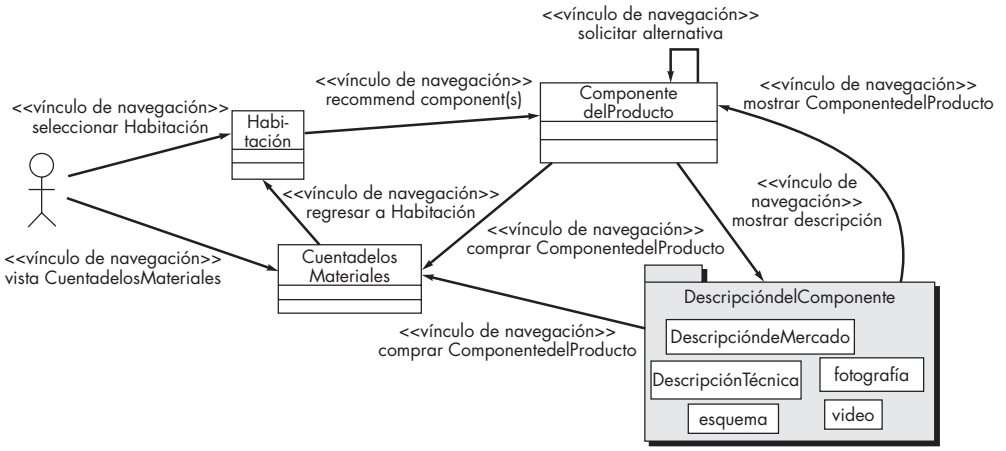
### 13.8.2 Sintaxis de navegación

Al avanzar en el diseño, la tarea siguiente es definir la mecánica de la navegación. Se dispone de varias opciones para desarrollar un enfoque de implementación para cada USN:

10 Éstas se denominan a veces *vínculos semánticos de navegación* (VSN) [Cac02].

www.FreeLibros.me

**FIGURA 13.9**
**Creación de una USN**




**CONSEJO**

*En la mayoría de situaciones, elija mecanismos de navegación horizontales o verticales, pero no ambos.*

- *Vínculo de navegación individual*: incluye vínculos basados en texto, iconos, botones e interruptores, así como metáforas gráficas. Deben elegirse vínculos que sean apropiados para el contenido y consistentes con la heurística que conduzca al diseño de una interfaz de alta calidad.
- *Barra de navegación horizontal*: enlista las categorías principales de contenido o de funciones en una barra que contiene vínculos apropiados. En general, se enlistan de cuatro a siete categorías.
- *Columna de navegación vertical*: 1) enlista las principales categorías de contenido o funciones o 2) enlista virtualmente todos los principales objetos de contenido que hay dentro de la *webapp*. Si se elige la segunda opción, las columnas de navegación pueden "expandirse" para que presenten objetos de contenido como parte de una jerarquía (seleccionar una entrada en la columna original ocasiona una expansión que enlista una segunda capa de objetos de contenido relacionados).

**CONSEJO**

*El mapa del sitio debe ser accesible desde cualquier página. El mapa mismo debe estar organizado de modo que la estructura de la información de la webapp se vea fácilmente.*

- *Pestañas*: metáfora que no es más que una variación de la barra o columna de navegación y representa categorías de contenido o funciones como pestañas que se seleccionan cuando se requiere un vínculo.
- *Mapas del sitio*: dan una tabla de contenido que incluye todo el contenido a fin de navegar hacia todos los objetos y funciones contenidas dentro de la *webapp*.

Además de elegir la mecánica de navegación, también deben establecerse las convenciones y ayudas apropiadas para navegar. Por ejemplo, los iconos y vínculos gráficos deben invitar a hacer "clic" en ellos, desvaneciendo las aristas a fin de darles una apariencia tridimensional. Debe diseñarse retroalimentación auditiva o visual con objeto de dar al usuario una indicación de que se ha escogido cierta opción de navegación. Para la navegación basada en texto debe utilizarse color que indique los vínculos de navegación y que señale aquéllos ya recorridos. Éstas son unas cuantas convenciones entre las decenas que hay para el diseño y que hacen que la navegación sea amigable para el usuario.

## 13.9 DISEÑO EN EL NIVEL DE COMPONENTES

Las *webapps* modernas dan funciones de procesamiento cada vez más complejas que: 1) realizan un procesamiento localizado para generar contenido y capacidad de navegación en forma dinámica, 2) proporcionan capacidad de cómputo o de procesamiento de datos que resultan apropiados para el dominio del negocio de la *webapp*, 3) dan consulta y acceso complejos a

www.FreeLibros.me

bases de datos y 4) establecen interfaces de datos con sistemas corporativos externos. Para lograr estas capacidades (y muchas otras) deben diseñarse y construirse componentes de programas con forma idéntica a los componentes del software tradicional.

Los métodos de diseño estudiados en el capítulo 10 se aplican a los componentes de las *webapps* con poca, o ninguna, modificación. El ambiente de implementación, los lenguajes de programación, los patrones de diseño, estructuras y software, tal vez varíen un poco, pero el enfoque general del diseño es el mismo.

## 13.10 MÉTODO DE DISEÑO DE HIPERMEDIOS ORIENTADO A OBJETOS (MDHOO)

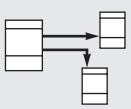
En la última década, se han propuesto varios métodos de diseño para aplicaciones web. Hasta hoy, ninguno de ellos es el dominante.[11] En esta sección se presenta un panorama breve de uno de los métodos de diseño de *webapps* más estudiado.

Daniel Schwabe *et al.* [Sch95, Sch98b] propusieron por primera vez el *Método de Diseño de Hipermedios Orientado a Objetos* (MDHOO), que está compuesto de cuatro distintas actividades de diseño: diseño conceptual, diseño de navegación, diseño abstracto de la interfaz e implementación. En la figura 13.10 se presenta un resumen de estas actividades de diseño y en las secciones que siguen se analizan brevemente.

### 13.10.1 Diseño conceptual del MDHOO

El *diseño conceptual* del MDHOO genera una representación de los subsistemas, clases y relaciones que definen el dominio de aplicación para la *webapp*. Se puede utilizar UML[12] para crear diagramas de clase apropiados, agregaciones y representaciones compuestas de clase, diagramas de colaboración y otra clase de información que describa el dominio de la aplicación.

**FIGURA 13.10**
**Resumen del MDHOO**
*Fuente:* Adaptado de [Sch95].

| |  Diseño conceptual |  Diseño de la navegación |  Diseño abstracto de la interfaz |  Implementación |
|---|---|---|---|---|
| Productos del trabajo | Clases, subsistemas, relaciones, atributos | Vínculos de nodos, estructuras de acceso, contextos de navegación, transformaciones de navegación | Objetos abstractos de la interfaz, respuestas a eventos externos, transformaciones | Webapp ejecutable |
| Mecanismos de diseño | Clasificación, composición, agregación, generalización, especialización | Mapeo entre objetos conceptuales y de navegación | Mapeo entre la navegación y los objetos perceptibles | Recurso proporcionado por el ambiente meta |
| Preocupaciones del diseño | Semántica de modelado del dominio de la aplicación | Toma en cuenta el perfil del usuario y la tarea. Hace énfasis en aspectos cognitivos | Modelado de los objetos perceptibles, implementación de las metáforas escogidas. Descripción de la interfaz para objetos de navegación | Corrección; desempeño de la aplicación; completitud |

11 En realidad, son relativamente pocos los desarrolladores web que usan un método específico cuando trabajan en una *webapp*. Hay la esperanza de que este enfoque ad-hoc del diseño cambie a medida que transcurra el tiempo.

12 El MDHOO no prescribe una notación específica; sin embargo, su empleo es común cuando se aplica este método.

www.FreeLibros.me

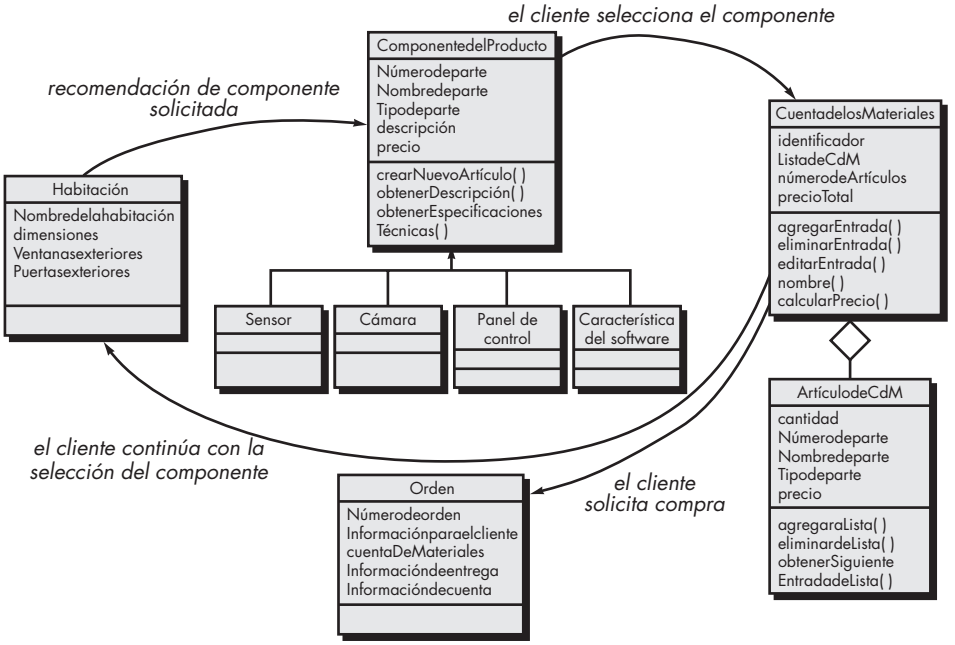



**FIGURA 13.11**
**Esquema conceptual parcial para CasaSeguraAsegurada.com**

Como ejemplo sencillo del diseño conceptual del MDHOO, piense en la aplicación de comercio electrónico **CasaSeguraAsegurada.com**. En la figura 13.11, se presenta un "esquema conceptual". Durante el diseño conceptual se reutilizan los diagramas de clase, agregaciones e información desarrollada como parte del análisis de la *webapp*, con objeto de representar las relaciones entre clases.

### 13.10.2 Diseño de la navegación para el MDHOO

El *diseño de la navegación* identifica un conjunto de "objetos" que se derivan de las clases definidas en el diseño conceptual. Para incluir éstos, se define una serie de "clases de navegación" o "nodos". Se utiliza UML para crear casos de uso, tablas de estado y diagramas de secuencia apropiados; todas éstas son representaciones que ayudan a entender mejor los requerimientos de la navegación. Además, conforme se desarrolla el diseño, se utilizan patrones para el diseño de la navegación. El MDHOO emplea un conjunto predefinido de clases de navegación: nodos, vínculos, anclas y estructuras de acceso [Sch98b]. Estas últimas son más elaboradas e incluyen mecanismos tales como un índice de la *webapp*, mapa del sitio o recorrido guiado.

Una vez definidas las clases de navegación, el MDHOO "estructura el espacio de navegación, agrupando los objetos de navegación en conjuntos llamados contextos" [Sch98b]. Un *contexto* incluye la descripción de la estructura de navegación local, la restricción impuesta al acceso de los objetos de contenido y los métodos (operaciones) requeridos para acceder a los objetos de contenido. Se desarrolla una plantilla contextual (análoga a las tarjetas CRC estudiadas en el capítulo 6) que se emplea para dar seguimiento a los requerimientos de navegación de cada categoría de usuario a través de los distintos contextos definidos en el MDHOO. Al hacer esto, surgen trayectorias específicas de navegación (que en la sección 13.8.1 llamamos FdN).

### 13.10.3 Diseño abstracto de la interfaz y su implementación

La acción de *diseño abstracto de la interfaz* especifica los objetos de la interfaz que el usuario ve cuando ocurre una interacción con la *webapp*. Se emplea un modelo formal de objetos de inter-

www.FreeLibros.me

faz, llamado *vista de datos abstractos* (VDA), para representar la relación entre objetos de interfaz y de navegación, así como las características de comportamiento de los objetos de interfaz.

El modelo VDA define una "plantilla estática" [Sch98b] que representa la metáfora de la interfaz e incluye una representación de los objetos de navegación dentro de la interfaz y la especificación de los objetos de ésta (como menús, botones e iconos) que ayudan a la navegación y a la interacción. Además, el modelo VDA contiene un componente de comportamiento (similar al diagrama de estado UML) que indica la forma en la que los eventos "disparan la navegación y cuáles son las transformaciones de la interfaz que ocurren cuando el usuario interactúa con la aplicación" [Sch01a].

La actividad de *implementación* del MDHOO representa una iteración del diseño específica del ambiente en el que opera la *webapp*. Las clases, navegación e interfaz se caracterizan cada una en forma tal que pueden construirse para el ambiente cliente-servidor, sistemas operativos, software de apoyo, lenguajes de programación, y otras características ambientales que son relevantes para el problema.

## 13.11 Resumen

La calidad de una *webapp* —definida en términos de usabilidad, funcionalidad, confiabilidad, eficiencia, facilidad de mantenimiento, seguridad, escalabilidad y tiempo para llegar al mercado— se introduce durante la etapa de diseño. Para lograr estos atributos de calidad, un buen diseño de la *webapp* debe tener las siguientes características: sencillez, consistencia, identidad, robustez, navegabilidad y atractivo visual. Para lograrlo, la actividad de diseño de la *webapp* se centra en seis distintos elementos del diseño.

El diseño de la interfaz describe la estructura y organización de la interfaz de usuario e incluye una representación de la distribución de la pantalla, una definición de los modos de interacción y una descripción de los mecanismos de navegación. Un conjunto de principios de diseño de la interfaz y el flujo de trabajo del diseño guían el trabajo de diseño de la distribución y los mecanismos de control de la interfaz.

El diseño estético, llamado también *diseño gráfico*, describe el "aspecto y sensación" de la *webapp*, e incluye esquemas de color; distribución geométrica; tamaño del texto, de las fuentes y su colocación; empleo de imágenes y otras decisiones relacionadas con la estética. Un conjunto de lineamientos de diseño gráfico da la base para el enfoque de diseño.

El diseño del contenido define distribución, estructura y bosquejo de todo el contenido que se presenta como parte de la *webapp*, y establece las relaciones entre los objetos del contenido. El diseño del contenido comienza con la representación de sus objetos, así como las asociaciones y relaciones entre ellos. Un conjunto de primitivas de navegación establece la base para el diseño de ésta.

El diseño arquitectónico identifica la estructura general de los hipermedios para la *webapp*, e incluye la arquitectura del contenido y de la *webapp*. Los estilos arquitectónicos para el contenido incluyen estructuras lineales, de malla, jerárquicas y de red. La arquitectura de la *webapp* describe una infraestructura que permite que un sistema o aplicación basado en web cumpla con sus objetivos de negocios.

El diseño de la navegación representa el flujo de ésta entre los objetos de contenido y todas las funciones de la *webapp*. La semántica de la navegación se define, describiendo un conjunto de unidades semánticas de navegación. Cada unidad está compuesta por formas de navegación, así como vínculos y nodos para ello. La sintaxis de navegación ilustra los mecanismos utilizados para navegar descritos como parte de la semántica.

El diseño de los componentes desarrolla la lógica de procesamiento detallada que se requiere para implementar componentes funcionales que desarrollen una función completa de la *web-*

www.FreeLibros.me

*app*. Las técnicas de diseño descritas en el capítulo 10 son aplicables para la ingeniería de los componentes de la *webapp*.

El Método de Diseño de Hipermedios Orientado a Objetos (MDHOO) es una de varias propuestas para hacer el diseño de *webapps*. El MDHOO sugiere un proceso que incluye diseño conceptual, diseño de la navegación, diseño abstracto de la interfaz y la implementación.

## Problemas y puntos por evaluar

**13.1.** ¿Por qué es insuficiente para elaborar *webapps* la filosofía de diseño del "ideal artístico"? ¿Hay algún caso en el que ésa sea la filosofía por seguir?

**13.2.** En este capítulo se selecciona un conjunto amplio de atributos de la calidad de las *webapps*. Seleccione las tres que crea que son las más importantes y construya un argumento que explique por qué debe hacerse énfasis en cada una durante el trabajo de diseño de *webapps*.

**13.3.** Agregue al menos cinco preguntas adicionales a la Lista de Verificación del Diseño de *webapps* que se presentó en la sección 13.1.

**13.4.** El lector es un diseñador de *webapps* de *Corporación de Aprendizaje del Futuro*, compañía de aprendizaje a distancia. Trata de implementar un "motor de aprendizaje" basado en internet que permita entregar el contenido de un curso a los estudiantes. El motor de aprendizaje brinda la infraestructura básica para entregar el contenido del aprendizaje sobre cualquier tema (los diseñadores del contenido prepararán el que sea apropiado). Desarrolle el diseño de un prototipo de interfaz para el motor de aprendizaje.

**13.5.** ¿Cuál es el sitio web de estética más agradable que usted haya visitado y por qué?

**13.6.** Considere el objeto de contenido **Orden**, generado una vez que un usuario de **CasaSeguraAsegurada.com** haya terminado la selección de todos los componentes y esté listo para finalizar su compra. Desarrolle una descripción UML para **Orden**, así como todas las representaciones del diseño que sean apropiadas.

**13.7.** ¿Cuál es la diferencia entre la arquitectura del contenido y la de una *webapp*?

**13.8.** Reconsidere el "motor de aprendizaje" de *Aprendizaje del Futuro* que se describió en el problema 13.4, seleccione una arquitectura del contenido que resulte apropiada para la *webapp*. Analice el porqué de su selección.

**13.9.** Utilice UML para desarrollar tres o cuatro representaciones del diseño de objetos de contenido que se encontrarían al diseñar el "motor de aprendizaje" descrito en el problema 13.4.

**13.10.** Investigue un poco acerca de la arquitectura de controlador de vista del modelo (CVM) y decida si sería apropiada para la *webapp* del "motor de aprendizaje" del problema 13.4.

**13.11.** ¿Cuál es la diferencia entre la sintaxis de navegación y la semántica de ésta?

**13.12.** Defina dos o tres USN para la *webapp* **CasaSeguraAsegurada.com**. Describa con detalle cada una.

**13.13.** Escriba un texto breve sobre un método de diseño de hipermedios que no sea MDHOO.

## Lecturas adicionales y fuentes de información

Van Duyne *et al.* (*The Design of Sites*, 2a. ed., Prentice Hall, 2007) escribieron un libro exhaustivo que cubre la mayoría de aspectos importantes del proceso de diseño de *webapps*. Cubre con detalle los modelos del proceso de diseño y los patrones de diseño. Wodtke (*Information Architecture*, New Riders Publishing, 2003), Rosenfeld y Morville (*Information Architecture for the World Wide Web*, O'Reilly & Associates, 2002), y Reiss (*Practical Information Architecture*, Addison-Wesley, 2000) abordan la arquitectura del contenido y otros temas.

Aunque se han escrito cientos de libros sobre el "diseño web", son muy pocos los que estudian métodos técnicos significativos para hacer el trabajo de diseño. En el mejor de los casos, presentan varios lineamientos útiles para el diseño de *webapps*, dan ejemplos de páginas web y de programación en Java y analizan los detalles técnicos importantes para implementar *webapps* modernas. Entre los representantes de esta cate-

www.FreeLibros.me

goría están los libros de Sklar (*Principles of Web Design*, 4a. ed., Course Technology, 2008), McIntire (*Visual Design for the Modern Web*, New Riders Press, 2007), Niederst (*Web Design in a Nutshell*, 3a. ed., O-Reilly, 2006), Eccher (*Advanced Professional Web Design*, Charles River Media, 2006), Cederholm (*Bulletproof Web Design*, New Riders Press, 2005) y Shelly *et al.* (*Web Design*, 2a. ed., Course Technology, 2005). El estudio enciclopédico de Powell [Pow02] y el profundo análisis de Nielsen [Nie00] sobre el diseño también son útiles en cualquier biblioteca.

Los libros de Beaird (*The Principles of Beautiful Web Design*, SitePoint, 2007), Clarke y Holzschlag (*Transcending CSS: The Fine Art of Web Design*, New Riders Press, 2006) y Golbeck (*Art Theory for Web Design*, Addison Wesley, 2005), hacen énfasis en el diseño estético y son una lectura benéfica para los profesionales con poca experiencia en el tema.

El punto de vista ágil del diseño (y otros temas) de *webapps* es presentado por Wallace *et al.* (*Extreme Programming for Web Projects*, Addison-Wesley, 2003). Conallen (*Building Web Applications with UML,* 2a. ed., Addison-Wesley, 2002) y Rosenberg y Scott (*Applying Use-Case Driven Object Modeling with UML*, Addison-Wesley, 2001) presentan ejemplos detallados de *webapps* modeladas con el empleo de UML.

En el contexto de libros escritos acerca de ambientes específicos de desarrollo, también se mencionan técnicas de diseño. Los lectores interesados en ello deben estudiar textos sobre HTML, CSS, J2EE, Java, .NET, XML, Perl, Ruby on Rails, Ajax y varias aplicaciones empleadas para crear *webapps* (*Dreamweaver, HomePage, Frontpage, GoLive, MacroMedia Flash,* etc.) con trucos de diseño útiles.

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

www.FreeLibros.me

PARTE

# Tres

# ADMINISTRACIÓN DE LA CALIDAD

En esta parte de *Ingeniería del software*, aprenderá los principios, conceptos y técnicas que se aplican para administrar y controlar la calidad del software. En los próximos capítulos se responderán preguntas como las siguientes:

- ¿Cuáles son las características generales del software de alta calidad?
- ¿Cómo se revisa la calidad y de qué manera se llevan a cabo revisiones eficaces?
- ¿En qué consiste el aseguramiento de la calidad del software?
- ¿Qué estrategias son aplicables para probar el software?
- ¿Qué métodos se utilizan para diseñar casos de prueba eficaces?
- ¿Hay métodos realistas que aseguren que el software es correcto?
- ¿Cómo pueden administrarse y controlarse los cambios que siempre ocurren cuando se elabora el software?
- ¿Qué medidas y unidades de medición se usan para evaluar la calidad de los modelos de requerimientos y diseño, código fuente y casos de prueba?

Una vez respondidas estas preguntas, el lector estará mejor preparado para asegurar que se ha producido software de alta calidad.



www.FreeLibros.me

CAPÍTULO

# 14 CONCEPTOS DE CALIDAD



El redoble de tambores para mejorar la calidad del software comenzó tan luego que éste empezó a integrarse en cada faceta de nuestras vidas. En la década de 1990, las principales corporaciones reconocieron que cada año se desperdiciaban miles de millones de dólares en software que no tenía las características ni la funcionalidad que se habían prometido. Lo que era peor, tanto el gobierno como la industria se preocupaban por la posibilidad de que alguna falla de software pudiera afectar infraestructura importante y provocara pérdidas de decenas de miles de millones de dólares. Al despuntar el nuevo siglo, *CIO Magazine* [Lev01] dio la alerta: "Dejemos de desperdiciar $78 mil millones de dólares al año", y lamentaba el hecho de que "las empresas estadounidenses gastan miles de millones de dólares en software que no hace lo que se supone que debe hacer". *InformationWeek* [Ric01] se hizo eco de la misma preocupación:

> A pesar de las buenas intenciones, el código defectuoso sigue siendo el duende de la industria del software, es responsable hasta de 45% del tiempo que están fuera los sistemas basados en computadoras y costó a las empresas estadounidenses alrededor de $100 mil millones de dólares el último año en pérdidas de productividad y reparaciones, afirma Standish Group, empresa de investigación de mercados. Eso no incluye el costo que implica perder a los clientes disgustados. Como los productores de tecnologías de información escriben aplicaciones que se basan en software empacado en infraestructura, el código defectuoso también puede inutilizar aplicaciones personalizadas...
>
> Pero, ¿cuán malo es el software defectuoso? Las respuestas varían, mas los expertos dicen que sólo se requiere de tres a cuatro defectos por cada 1 000 líneas de código para que un programa tenga mal desempeño. Hay que pensar que la mayoría de los programadores cometen un error en cada 10 líneas de código que escriben, lo que, multiplicado por los millones de líneas que hay en muchos productos comerciales, permite imaginar que la corrección de los errores cuesta a los vendedores de software al menos la mitad de sus presupuestos de desarrollo durante las pruebas. ¿Comprende lo que esto significa?

## UNA MIRADA RÁPIDA

**¿Qué es?** La respuesta no es tan fácil como quizá se piense. La calidad se reconoce cuando se ve, por lo que puede ser algo difícil de definir. Pero para el software de computadora, la calidad es algo que debe definirse, y eso es lo que haremos en este capítulo.

**¿Quién lo hace?** Los involucrados en el proceso del software —ingenieros, gerentes y todos los participantes— son los responsables de la calidad.

**¿Por qué es importante?** Puede hacerse bien o puede repetirse. Si un equipo de software hace énfasis en la calidad de todas las actividades de la ingeniería de software, se reduce el número de repeticiones que deben hacerse. Esto da como resultado menores costos y, lo que es más importante, mejora el tiempo de llegada al mercado.

**¿Cuáles son los pasos?** Para lograr software de alta calidad, deben ocurrir cuatro actividades: usar procesos y prácticas probados de la ingeniería de software, administrar bien el proyecto, realizar un control de calidad exhaustivo y contar con infraestructura de aseguramiento de la calidad.

**¿Cuál es el producto final?** Software que satisface las necesidades del consumidor, con un desempeño apropiado y confiable, y que agrega valor para todos los que lo utilizan.

**¿Cómo me aseguro de que lo hice bien?** Hay que dar seguimiento a la calidad, estudiando los resultados de todas las actividades de control de calidad y midiendo ésta con el estudio de los errores antes de la entrega y de los defectos detectados en el campo.



www.FreeLibros.me

En 2005, *ComputerWorld* [Hil05] se quejaba de que "el mal software es una plaga en casi todas las organizaciones que emplean computadoras, lo que ocasiona horas de trabajo perdidas por el tiempo que están fuera de uso las máquinas, por datos perdidos o corrompidos, oportunidades de venta perdidas, costos elevados de apoyo y mantenimiento, y poca satisfacción del cliente. Un año después, *InfoWorld* [Fos06] escribió acerca del "lamentable estado de la calidad del software" e informaba que el problema de la calidad no había mejorado.

Actualmente, la calidad del software es preocupante, pero, ¿de quién es la culpa? Los clientes culpan a los desarrolladores, pues afirman que sus prácticas descuidadas producen software de mala calidad. Los desarrolladores culpan a los clientes (y a otros participantes) con la afirmación de que las fechas de entrega irracionales y un flujo continuo de cambios los obligan a entregar software antes de haber sido validado por completo. ¿Quién tiene la razón? *Ambos*, y ése es el problema. En este capítulo se analiza el concepto de calidad del software y por qué es útil estudiarlo con seriedad siempre que se apliquen prácticas de ingeniería de software.

## 14.1 ¿Qué es calidad?

En su libro místico, *El zen y el arte del mantenimiento de la motocicleta*, Robert Persig [Per74] comenta lo siguiente acerca de lo que llamamos *calidad*:

> Calidad... sabes lo que es, pero no sabes lo que es. Pero eso es una contradicción. Algunas cosas son mejores que otras; es decir, tienen más calidad. Pero cuando tratas de decir lo que es la calidad, además de las cosas que la tienen, todo se desvanece... No hay nada de qué hablar. Pero si no puede decirse qué es Calidad, ¿cómo saber lo que es, o incluso saber que existe? Si nadie sabe lo que es, entonces, para todos los propósitos prácticos, no existe en absoluto. Pero para todos los propósitos prácticos, en realidad sí existe. ¿En qué otra cosa se basan las calificaciones? ¿Por qué paga fortunas la gente por algunos artículos y tira otros a la basura? Es obvio que algunas cosas son mejores que otras... pero, ¿en qué son mejores? Y así damos vueltas y más vueltas, ruedas de metal que patinan sin nada en lo que hagan tracción. ¿Qué demonios es la Calidad? ¿Qué es?

Es cierto: ¿qué es?


*¿Cuáles son las diferentes maneras en las que puede verse la calidad?*

En un nivel algo pragmático, David Garvin [Gar84], de Harvard Business School, sugiere que "la calidad es un concepto complejo y de facetas múltiples" que puede describirse desde cinco diferentes puntos de vista. El *punto de vista trascendental* dice (como Persig) que la calidad es algo que se reconoce de inmediato, pero que no es posible definir explícitamente. El *punto de vista del usuario* concibe la calidad en términos de las metas específicas del usuario final. Si un producto las satisface, tiene calidad. El *punto de vista del fabricante* la define en términos de las especificaciones originales del producto. Si éste las cumple, tiene calidad. El *punto de vista del producto* sugiere que la calidad tiene que ver con las características inherentes (funciones y características) de un producto. Por último, el *punto de vista basado en el valor* la mide de acuerdo con lo que un cliente está dispuesto a pagar por un producto. En realidad, la calidad incluye todo esto y más.

**Cita:**

"La gente olvida cuán rápido hiciste un trabajo, pero siempre recuerda cuán bien lo realizaste."

**Howard Newton**

La *calidad del diseño* se refiere a las características que los diseñadores especifican para un producto. El tipo de materiales, tolerancias y especificaciones del desempeño, todo contribuye a la calidad del diseño. Si se utilizan mejores materiales, tolerancias más estrictas y se especifican mayores niveles de desempeño, la calidad del diseño de un producto se incrementa si se fabrica de acuerdo con las especificaciones.

En el desarrollo del software, la *calidad del diseño* incluye el grado en el que el diseño cumple las funciones y características especificadas en el modelo de requerimientos. La *calidad de la conformidad* se centra en el grado en el que la implementación se apega al diseño y en el que el sistema resultante cumple sus metas de requerimientos y desempeño.

www.FreeLibros.me

Pero, ¿son la calidad del diseño y de la conformidad los únicos aspectos que deben considerar los ingenieros de software? Robert Glass [Gla98] afirma que es mejor plantear una relación más intuitiva:

**satisfacción del usuario = producto que funciona + buena calidad + entrega dentro del presupuesto y plazo**

En última instancia, Glass sostiene que la calidad es importante, pero que si el usuario no está satisfecho, nada de lo demás importa. DeMarco [DeM98] refuerza esta opinión al decir que "la calidad de un producto está en función de cuánto cambia al mundo para bien". Este punto de vista de la calidad afirma que si un producto de software beneficia mucho a los usuarios finales, éstos se mostrarán dispuestos a tolerar problemas ocasionales de confiabilidad o desempeño.

## 14.2 Calidad del software

Incluso los desarrolladores de software más experimentados estarán de acuerdo en que obtener software de alta calidad es una meta importante. Pero, ¿cómo se define la calidad del *software*? En el sentido más general se define[1] como: *Proceso eficaz de software que se aplica de manera que crea un producto útil que proporciona valor medible a quienes lo producen y a quienes lo utilizan.*

Hay pocas dudas acerca de que la definición anterior podría modificarse o ampliarse en un debate sin fin. Para propósitos de este libro, la misma sirve a fin de enfatizar tres puntos importantes:

1. Un proceso *eficaz de software* establece la infraestructura que da apoyo a cualquier esfuerzo de elaboración de un producto de software de alta calidad. Los aspectos de administración del proceso generan las verificaciones y equilibrios que ayudan a evitar que el proyecto caiga en el caos, contribuyente clave de la mala calidad. Las prácticas de ingeniería de software permiten al desarrollador analizar el problema y diseñar una solución sólida, ambas actividades críticas de la construcción de software de alta calidad. Por último, las actividades sombrilla, tales como administración del cambio y revisiones técnicas, tienen tanto que ver con la calidad como cualquier otra parte de la práctica de la ingeniería de software.
2. Un *producto útil* entrega contenido, funciones y características que el usuario final desea; sin embargo, de igual importancia es que entrega estos activos en forma confiable y libre de errores. Un producto útil siempre satisface los requerimientos establecidos en forma explícita por los participantes. Además, satisface el conjunto de requerimientos (por ejemplo, la facilidad de uso) con los que se espera que cuente el software de alta calidad.
3. Al *agregar valor para el productor y para el usuario* de un producto, el software de alta calidad proporciona beneficios a la organización que lo produce y a la comunidad de usuarios finales. La organización que elabora el software obtiene valor agregado porque el software de alta calidad requiere un menor esfuerzo de mantenimiento, menos errores que corregir y poca asistencia al cliente. Esto permite que los ingenieros de software dediquen más tiempo a crear nuevas aplicaciones y menos a repetir trabajos mal hechos. La comunidad de usuarios obtiene valor agregado porque la aplicación provee una capacidad útil en forma tal que agiliza algún proceso de negocios. El resultado final es 1) mayores utilidades por el producto de software, 2) más rentabilidad cuando una

1 Esta definición ha sido adaptada de [Bes04] y sustituye aquélla más orientada a la manufactura presentada en ediciones anteriores de este libro.

www.FreeLibros.me

aplicación apoya un proceso de negocios y 3) mejor disponibilidad de información, que es crucial para el negocio.

### 14.2.1 Dimensiones de la calidad de Garvin

David Garvin [Gar87] sugiere que la calidad debe tomarse en cuenta, adoptando un punto de vista multidimensional que comience con la evaluación de la conformidad y termine con una visión trascendental (estética). Aunque las ocho dimensiones de Garvin de la calidad no fueron desarrolladas específicamente para el software, se aplican a la calidad de éste:

**Calidad del desempeño.** ¿El software entrega todo el contenido, las funciones y las características especificadas como parte del modelo de requerimientos, de manera que da valor al usuario final?

**Calidad de las características.** ¿El software tiene características que sorprenden y agradan la primera vez que lo emplean los usuarios finales?

**Confiabilidad.** ¿El software proporciona todas las características y capacidades sin fallar? ¿Está disponible cuando se necesita? ¿Entrega funcionalidad libre de errores?

**Conformidad.** ¿El software concuerda con los estándares locales y externos que son relevantes para la aplicación? ¿Concuerda con el diseño *de facto* y las convenciones de código? Por ejemplo, ¿la interfaz de usuario está de acuerdo con las reglas aceptadas del diseño para la selección de menú o para la entrada de datos?

**Durabilidad.** ¿El software puede recibir mantenimiento (cambiar) o corregirse (depurarse) sin la generación inadvertida de eventos colaterales? ¿Los cambios ocasionarán que la tasa de errores o la confiabilidad disminuyan con el tiempo?

**Servicio.** ¿Existe la posibilidad de que el software reciba mantenimiento (cambios) o correcciones (depuración) en un periodo de tiempo aceptablemente breve? ¿El equipo de apoyo puede adquirir toda la información necesaria para hacer cambios o corregir defectos? Douglas Adams [Ada93] hace un comentario irónico que parece pertinente: "La diferencia entre algo que puede salir mal y algo que posiblemente no salga mal es que cuando esto último sale mal, por lo general es imposible corregirlo o repararlo."

**Estética.** No hay duda de que todos tenemos una visión diferente y muy subjetiva de lo que es estético. Aun así, la mayoría de nosotros estaría de acuerdo en que una entidad estética posee cierta elegancia, un flujo único y una "presencia" obvia que es difícil de cuantificar y que, no obstante, resulta evidente. El software estético tiene estas características.

**Percepción.** En ciertas situaciones, existen prejuicios que influirán en la percepción de la calidad por parte del usuario. Por ejemplo, si se introduce un producto de software elaborado por un proveedor que en el pasado ha demostrado mala calidad, se estará receloso y la percepción de la calidad del producto tendrá influencia negativa. De manera similar, si un vendedor tiene una reputación excelente se percibirá buena calidad, aun si ésta en realidad no existe.

Las dimensiones de la calidad de Garvin dan una visión "suave" de la calidad del software. Muchas de estas dimensiones (aunque no todas) sólo pueden considerarse de manera subjetiva. Por esta razón, también se necesita un conjunto de factores "duros" de la calidad que se clasifican en dos grandes grupos: 1) factores que pueden medirse en forma directa (por ejemplo, defectos no descubiertos durante las pruebas) y 2) factores que sólo pueden medirse indirectamente (como la usabilidad o la facilidad de recibir mantenimiento). En cada caso deben hacerse mediciones: debe compararse el software con algún dato para llegar a un indicador de la calidad.

www.FreeLibros.me

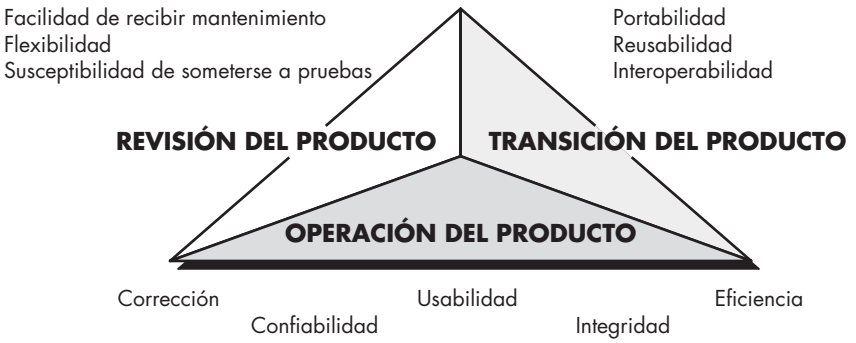



**FIGURA 14.1** Factores de la calidad de McCall

### 14.2.2 Factores de la calidad de McCall

McCall, Richards y Walters [McC77] proponen una clasificación útil de los factores que afectan la calidad del software. Éstos se ilustran en la figura 14.1 y se centran en tres aspectos importantes del producto de software: sus características operativas, su capacidad de ser modificado y su adaptabilidad a nuevos ambientes.

En relación con los factores mencionados en la figura 14.1, McCall *et al.*, hacen las descripciones siguientes:

*Corrección.* Grado en el que un programa satisface sus especificaciones y en el que cumple con los objetivos de la misión del cliente.

*Confiabilidad.* Grado en el que se espera que un programa cumpla con su función y con la precisión requerida [debe notarse que se han propuesto otras definiciones más completas de la confiabilidad (véase el capítulo 25)].

*Eficiencia.* Cantidad de recursos de cómputo y de código requeridos por un programa para llevar a cabo su función.

*Integridad.* Grado en el que es posible controlar el acceso de personas no autorizadas al software o a los datos.

*Usabilidad.* Esfuerzo que se requiere para aprender, operar, preparar las entradas e interpretar las salidas de un programa.

*Facilidad de recibir mantenimiento.* Esfuerzo requerido para detectar y corregir un error en un programa (ésta es una definición muy limitada).

*Flexibilidad.* Esfuerzo necesario para modificar un programa que ya opera.

*Susceptibilidad de someterse a pruebas.* Esfuerzo que se requiere para probar un programa a fin de garantizar que realiza la función que se pretende.

*Portabilidad.* Esfuerzo que se necesita para transferir el programa de un ambiente de sistema de hardware o software a otro.

*Reusabilidad.* Grado en el que un programa (o partes de uno) pueden volverse a utilizar en otras aplicaciones (se relaciona con el empaque y el alcance de las funciones que lleva a cabo el programa).

*Interoperabilidad.* Esfuerzo requerido para acoplar un sistema con otro.

**Cita:**

"La amargura de la mala calidad permanece mucho tiempo después de que ya se ha olvidado la dulzura de haber cumplido el plazo programado."

**Karl Weigers (cita sin acreditación)**

Es difícil — y, en ciertos casos, imposible— desarrollar mediciones directas[2] de estos factores de la calidad. En realidad, muchas de las unidades de medida definidas por McCall *et al.*, sólo

2 Una *medición directa* implica que hay un solo valor cuantificable que da una indicación directa del atributo en estudio. Por ejemplo, el "tamaño" de un programa se mide directamente, contando el número de sus líneas de código.

www.FreeLibros.me

pueden obtenerse de manera indirecta. Sin embargo, la evaluación de la calidad de una aplicación por medio de estos factores dará un indicio sólido de ella.

### 14.2.3 Factores de la calidad ISO 9126

El estándar ISO 9126 se desarrolló con la intención de identificar los atributos clave del software de cómputo. Este sistema identifica seis atributos clave de la calidad:

**Funcionalidad.** Grado en el que el software satisface las necesidades planteadas según las establecen los atributos siguientes: adaptabilidad, exactitud, interoperabilidad, cumplimiento y seguridad.

**Confiabilidad.** Cantidad de tiempo que el software se encuentra disponible para su uso, según lo indican los siguientes atributos: madurez, tolerancia a fallas y recuperación.

**Usabilidad.** Grado en el que el software es fácil de usar, según lo indican los siguientes subatributos: entendible, aprendible y operable.

**Eficiencia.** Grado en el que el software emplea óptimamente los recursos del sistema, según lo indican los subatributos siguientes: comportamiento del tiempo y de los recursos.

**Facilidad de recibir mantenimiento.** Facilidad con la que pueden efectuarse reparaciones al software, según lo indican los atributos que siguen: analizable, cambiable, estable, susceptible de someterse a pruebas.

**Portabilidad.** Facilidad con la que el software puede llevarse de un ambiente a otro según lo indican los siguientes atributos: adaptable, instalable, conformidad y sustituible.


*Aunque resulta tentador desarrollar mediciones cuantitativas para los factores de calidad mencionados aquí, también puede crearse una lista de comprobación de atributos que den una indicación sólida de la presencia del factor.*

Igual que otros factores de la calidad del software estudiados en las subsecciones anteriores, los factores ISO 9126 no necesariamente conducen a una medición directa. Sin embargo, proporcionan una base útil para hacer mediciones indirectas y una lista de comprobación excelente para evaluar la calidad del sistema.

### 14.2.4 Factores de calidad que se persiguen


"Cualquier actividad se vuelve creativa cuando a quien la realiza le importa hacerla bien, o mejor."

**John Updike**

Las dimensiones y factores de la calidad presentados en las secciones 14.2.1 y 14.2.2 se centran en el software como un todo y pueden utilizarse como indicación general de la calidad de una aplicación. Un equipo de software puede desarrollar un conjunto de características de la calidad y las preguntas asociadas correspondientes que demuestren[3] el grado en el que se satisface cada factor. Por ejemplo, McCall identifica la *usabilidad* como un factor importante de la calidad. Si se pidiera revisar una interfaz de usuario para evaluar su usabilidad, ¿cómo se haría? Se comenzaría con los subatributos propuestos por McCall —entendible, aprendible y operable— pero en un sentido práctico: ¿qué significan éstos?

Para hacer la evaluación, se necesita determinar atributos específicos y medibles (o al menos reconocibles) de la interfaz. Por ejemplo [Bro03]:

**Intuitiva.** Grado en el que la interfaz sigue patrones esperados de uso, de modo que hasta un novato la pueda utilizar sin mucha capacitación.

- ¿La interfaz lleva hacia una comprensión fácil?
- ¿Todas las operaciones son fáciles de localizar e iniciar?
- ¿La interfaz usa una metáfora reconocible?
- ¿La entrada está especificada de modo que economiza el uso del teclado o del ratón?

3 Estas características y preguntas se plantearían como parte de la revisión del software (véase el capítulo 15).

www.FreeLibros.me

- ¿La entrada sigue las tres reglas de oro? (véase el capítulo 11)
- ¿La estética ayuda a la comprensión y uso?

**Eficiencia.** Grado en el que es posible localizar o iniciar las operaciones y la información.

- ¿La distribución y estilo de la interfaz permite que un usuario introduzca con eficiencia las operaciones y la información?
- ¿Una secuencia de operaciones (o entrada de datos) puede realizarse con economía de movimientos?
- ¿Los datos de salida o el contenido están presentados de modo que se entienden de inmediato?
- ¿Las operaciones jerárquicas están organizadas de manera que minimizan la profundidad con la que debe navegar el usuario para hacer que alguna se ejecute?

**Robustez.** Grado en el que el software maneja entradas erróneas de datos o en el que se presenta interacción inapropiada por parte del usuario.

- ¿El software reconocerá el error si entran datos en el límite de lo permitido o más allá y, lo que es más importante, continuará operando sin fallar ni degradarse?
- ¿La interfaz reconocerá los errores cognitivos o de manipulación y guiará en forma explícita al usuario de vuelta al camino correcto?
- ¿La interfaz da un diagnóstico y guía útiles cuando se descubre una condición de error (asociada con la funcionalidad del software)?

**Riqueza.** Grado en el que la interfaz provee un conjunto abundante de características.

- ¿Puede personalizarse la interfaz según las necesidades específicas del usuario?
- ¿La interfaz tiene gran capacidad para permitir al usuario identificar una secuencia de operaciones comunes con una sola acción o comando?

A medida que se desarrolla el diseño de la interfaz, el equipo del software revisa el prototipo del diseño y plantea las preguntas anteriores. Si la respuesta a la mayor parte de éstas es "sí", es probable que la interfaz de usuario sea de buena calidad. Para cada factor de la calidad que se desee evaluar se desarrollan preguntas similares.

### 14.2.5 Transición a un punto de vista cuantitativo

En las subsecciones anteriores se presentaron varios factores cualitativos para la "medición" de la calidad del software. La comunidad de la ingeniería de software trata de obtener mediciones precisas de la calidad de éste y a veces se ve frustrada por la naturaleza subjetiva de la actividad. Cavano y McCall [Cav78] analizan esta situación:

> La determinación de la calidad es un factor clave en los eventos cotidianos: concursos para catar vinos, eventos deportivos [como gimnasia], competencias de talento, etc. En estas situaciones se juzga la calidad del modo más fundamental y directo: la comparación directa de objetos en condiciones idénticas y con conceptos predeterminados. El vino se juzga de acuerdo con su claridad, color, buqué, sabor, etc. Sin embargo, este tipo de juicio es muy subjetivo; para que tenga algún valor, debe ser hecho por un experto.
>
> La subjetividad y la especialización también se aplican a la determinación de la calidad del software. Para ayudar a resolver este problema, es necesario tener una definición más precisa de la calidad del software, así como una forma de realizar mediciones cuantitativas de la calidad a fin de hacer análisis objetivos... Como no existe algo parecido al conocimiento absoluto, no debe esperarse medir con toda exactitud la calidad del software, porque toda medición es imperfecta. Jacob Bronkowski

www.FreeLibros.me

> describió esta paradoja del conocimiento del modo siguiente: "Año con año desarrollamos instrumentos más precisos para observar la naturaleza con más nitidez. Y cuando vemos las observaciones, nos decepcionamos porque son borrosas y sentimos que son tan inciertas como siempre".

En el capítulo 23 se presenta un conjunto de unidades de medida aplicables a la evaluación cuantitativa de la calidad del software. En todos los casos, las unidades representan mediciones indirectas, es decir, nunca miden realmente la *calidad*, sino alguna manifestación de ella. El factor que complica todo es la relación precisa entre la variable que se mide y la calidad del software.

## 14.3 El dilema de la calidad del software


*Cuando se enfrente al dilema de la calidad (y todos lo hacen en un momento u otro), trate de alcanzar el balance: suficiente esfuerzo para producir una calidad aceptable sin que sepulte al proyecto.*

En una entrevista [Ven03] publicada en la web, Bertrand Meyer analiza lo que se denomina el *dilema de la calidad:*

> Si produce un sistema de software de mala calidad, usted pierde porque nadie lo querrá comprar. Por otro lado, si dedica un tiempo infinito, demasiado esfuerzo y enormes sumas de dinero para obtener un elemento perfecto de software, entonces tomará tanto tiempo terminarlo y será tan caro de producir que de todos modos quedará fuera del negocio. En cualquier caso, habrá perdido la ventana de mercado, o simplemente habrá agotado sus recursos. De modo que las personas de la industria tratan de situarse en ese punto medio mágico donde el producto es suficientemente bueno para no ser rechazado de inmediato, no en la evaluación, pero tampoco es un objeto perfeccionista ni con demasiado trabajo que lo convierta en algo que requiera demasiado tiempo o dinero para ser terminado.

Es correcto afirmar que los ingenieros de software deben tratar de producir sistemas de alta calidad. Es mejor aplicar buenas prácticas al intento de lograrlo. Pero la situación descrita por Meyer proviene de la vida real y representa un dilema incluso para las mejores organizaciones de ingeniería de software.

### 14.3.1 Software "suficientemente bueno"

En palabras sencillas, si damos por válido el argumento de Meyer, ¿es aceptable producir software "suficientemente bueno"? La respuesta a esta pregunta debe ser "sí", porque las principales compañías de software lo hacen a diario. Crean software con errores detectados y lo distribuyen a una gran población de usuarios finales. Reconocen que algunas de las funciones y características de la versión 1.0 tal vez no sean de la calidad más alta y planean hacer mejoras en la versión 2.0. Hacen esto, sabiendo que algunos clientes se quejarán; reconocen que el tiempo para llegar al mercado actúa contra la mejor calidad, y liberan el software, siempre y cuando el producto entregado sea "suficientemente bueno".

Exactamente, ¿qué significa "suficientemente bueno"? El software suficientemente bueno contiene las funciones y características de alta calidad que desean los usuarios, pero al mismo tiempo tiene otras más oscuras y especializadas que contienen errores conocidos. El vendedor de software espera que la gran mayoría de usuarios finales perdone los errores gracias a que estén muy contentos con la funcionalidad de la aplicación.

Esta idea resulta familiar para muchos lectores. Si usted es uno de ellos, le pido que considere algunos de los argumentos contra lo "suficientemente bueno".

Es verdad que lo "suficientemente bueno" puede funcionar en ciertos dominios de aplicación y para unas cuantas compañías grandes de software. Después de todo, si una empresa tiene un presupuesto enorme para mercadotecnia y convence a suficientes personas de que compren la versión 1.0, habrá tenido éxito en capturarlos. Como ya se dijo, puede sostener que en las versiones posteriores mejorará la calidad. Al entregar la versión 1.0 suficientemente buena, habrá capturado al mercado.

www.FreeLibros.me

Si el lector trabaja para una compañía pequeña, debe tener cuidado con esta filosofía. Al entregar un producto suficientemente bueno (defectuoso), corre el riesgo de causar un daño permanente a la reputación de su compañía. Tal vez nunca tenga la oportunidad de entregar una versión 2.0 porque los malos comentarios quizá ocasionen que las ventas se desplomen y que la empresa desaparezca.

Si trabaja en ciertos dominios de aplicación (por ejemplo, software incrustado en tiempo real) o si construye software de aplicación integrado con hardware (como el software automotriz o de telecomunicaciones), entregar software con errores conocidos es una negligencia y deja expuesta a su compañía a litigios costosos. En ciertos casos, incluso, puede ser un delito. ¡Nadie quiere tener software suficientemente bueno en los aviones!

Así que proceda con cautela si piensa que lo "suficientemente bueno" es un atajo que puede resolver los problemas de calidad de su software. Tal vez funcione, pero sólo para unos cuantos y en un conjunto limitado de dominios de aplicación.[4]

### 14.3.2 El costo de la calidad

El argumento es algo parecido a esto: *sabemos que la calidad es importante, pero cuesta tiempo y dinero —demasiado tiempo y dinero— lograr el nivel de calidad en el software que en realidad queremos.* Visto así, este argumento parece razonable (véanse los comentarios anteriores de Meyer en esta sección). No hay duda de que la calidad tiene un costo, pero la mala calidad también lo tiene —no sólo para los usuarios finales que deban vivir con el software defectuoso, sino también para la organización del software que lo elaboró y que debe darle mantenimiento—. La pregunta real es ésta: *¿por cuál costo debemos preocuparnos?* Para responder a esta pregunta debe entenderse tanto el costo de tener calidad como el del software de mala calidad.

El *costo de la calidad* incluye todos los costos en los que se incurre al buscar la calidad o al realizar actividades relacionadas con ella y los costos posteriores de la falta de calidad. Para entender estos costos, una organización debe contar con unidades de medición que provean el fundamento del costo actual de la calidad, que identifiquen las oportunidades para reducir dichos costos y que den una base normalizada de comparación. El costo de la calidad puede dividirse en los costos que están asociados con la prevención, la evaluación y la falla.


*No tema incurrir en costos significativos por la prevención. Esté seguro de que su inversión tendrá un rendimiento excelente.*

Los *costos de prevención* incluyen lo siguiente: 1) el costo de las actividades de administración requeridas para planear y coordinar todas las actividades de control y aseguramiento de la calidad, 2) el costo de las actividades técnicas agregadas para desarrollar modelos completos de los requerimientos y del diseño, 3) los costos de planear las pruebas y 4) el costo de toda la capacitación asociada con estas actividades.

Los *costos de evaluación* incluyen las actividades de investigación de la condición del producto la "primera vez" que pasa por cada proceso. Algunos ejemplos de costos de evaluación incluyen los siguientes:

- El costo de efectuar revisiones técnicas (véase el capítulo 15) de los productos del trabajo de la ingeniería de software.
- El costo de recabar datos y unidades de medida para la evaluación (véase el capítulo 23)
- El costo de hacer las pruebas y depurar (véanse los capítulos 18 a 21)

Los *costos de falla* son aquellos que se eliminarían si no hubiera errores antes o después de enviar el producto a los consumidores. Los costos de falla se subdividen en internos y externos. Se incurre en *costos internos de falla* cuando se detecta un error en un producto antes del envío. Los costos internos de falla incluyen los siguientes:

4 Un análisis útil de los pros y contras del software "suficientemente bueno" se encuentra en [Bre02].

www.FreeLibros.me

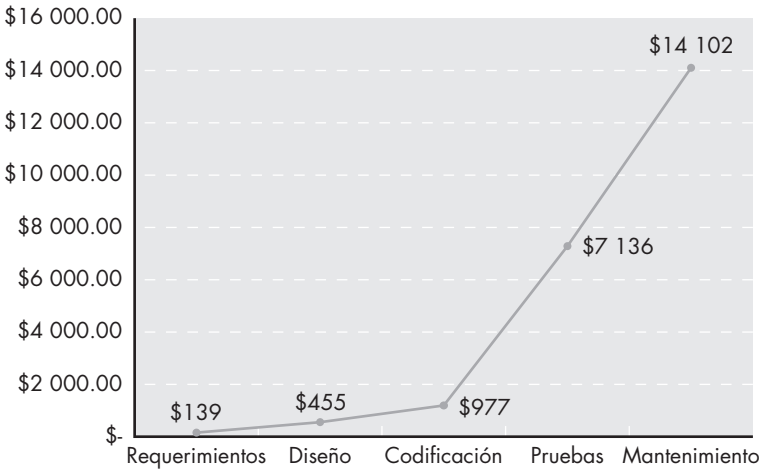



**FIGURA 14.2**
**Costo relativo de corregir errores y defectos (cifras en dólares estadounidenses)**
*Fuente:* Adaptado de [Boe01b].

- El costo requerido por efectuar repeticiones (reparaciones para corregir un error).
- El costo en el que se incurre cuando una repetición genera inadvertidamente efectos colaterales que deban mitigarse.
- Los costos asociados con la colección de las unidades de medida de la calidad que permitan que una organización evalúe los modos de la falla.

**Cita:**

"Toma menos tiempo hacer algo bien que explicar por qué se hizo mal."

**H. W. Longfellow**

Los *costos externos de falla* se asocian con defectos encontrados después de que el producto se envió a los consumidores. Algunos ejemplos de costos externos de falla son los de solución de quejas, devolución y sustitución del producto, ayuda en línea y trabajo asociado con la garantía. La mala reputación y la pérdida resultante de negocios es otro costo externo de falla que resulta difícil de cuantificar y que, sin embargo, es real. Cuando se produce software de mala calidad, suceden cosas malas.

En lo que constituye una acusación contra los desarrolladores de software que se rehúsan a considerar los costos de falla externos, Cem Kaner [Kan95] afirma lo siguiente:

> Muchos de los costos de falla externos, tales como los fondos de comercialización, son difíciles de cuantificar, por lo que muchas compañías los ignoran cuando calculan sus relaciones costo-beneficio. Otros costos externos de falla pueden reducirse (al dar un apoyo barato debido a la mala calidad después de hacer la venta, o al cobrar el apoyo a los consumidores) sin que se incremente la satisfacción del cliente. Al ignorar los costos que los malos productos generan a nuestros compradores, los ingenieros de la calidad estimulan una toma de decisiones que los hace víctimas en lugar de satisfacerlos.

Como es de esperar, los costos relacionados con la detección y la corrección de errores o defectos se incrementan en forma abrupta cuando se pasa de la prevención a la detección, a la falla interna y a la externa. La figura 14.2, basada en datos obtenidos por Boehm y Basili [Boe01b] y elaborada por Cigital, Inc. [Cig07], ilustra este fenómeno.

El costo promedio de la industria por corregir un defecto durante la generación de código es aproximadamente de US$977 por error. El promedio del costo en el que incurre la industria por corregir el mismo error si se descubre durante las pruebas del sistema es de US$7 136. Cigital, Inc. [Cig07] tome en cuenta que una aplicación grande contiene 200 errores introducidos durante la codificación.

> De acuerdo con datos promedio, el costo de encontrar y corregir defectos durante la fase de codificación es de US$977 por defecto. Entonces, el costo total por corregir los 200 errores "críticos" durante esta fase es de (200 × US$977) US$195 400, aproximadamente.

www.FreeLibros.me

> Los datos promedio de la industria indican que el costo de encontrar y corregir defectos durante la fase de pruebas del sistema es de US$7 136 por cada uno. En este caso, si se supone que en dicha fase se descubren aproximadamente 50 defectos críticos (tan sólo 25% de los descubiertos por Cigital en la fase de codificación), el costo de encontrarlos y corregirlos (50 × US$7 136) sería aproximadamente de US$356 800. Esto también habría resultado en 150 errores críticos no detectados ni corregidos. El costo de encontrar y corregir estos 150 defectos en la fase de mantenimiento (150 × US$14 102) habría sido de US$2 115 300. Entonces, el costo total de encontrar y corregir los 200 defectos (US$2 115 300 + US$356 800) después de la fase de codificación habría sido de US$2 472 100.

Aun si la organización de software tuviera costos que fueran la mitad del promedio de la industria (la mayor parte de compañías no tiene idea de cuáles son sus costos), los ahorros asociados con el control de calidad temprano y las actividades para su aseguramiento (efectuadas durante el análisis de los requerimientos y el diseño) serían notables.

### 14.3.3 Riesgos

En el capítulo 1 de este libro se dijo que "la gente basa su trabajo, confort, seguridad, entretenimiento, decisiones y su propia vida, en software de cómputo. Más vale que esté bien hecho". La implicación es que el software de mala calidad aumenta los riesgos tanto para el desarrollador como para el usuario final. En la subsección anterior se analizó uno de dichos riesgos (el costo). Pero lo perjudicial de las aplicaciones mal diseñadas e implementadas no siempre se mide en dólares y tiempo. Un ejemplo extremo [Gag04] servirá para ilustrar esto.

En el mes de noviembre de 2000, en un hospital de Panamá, 28 pacientes recibieron dosis masivas de rayos gama durante su tratamiento contra diversos tipos de cáncer. En los meses que siguieron, 5 de estos pacientes murieron por envenenamiento radiactivo y 15 más sufrieron complicaciones serias. ¿Qué fue lo que ocasionó esta tragedia? Un paquete de software, desarrollado por una compañía estadounidense, que fue modificado por técnicos del hospital para calcular las dosis de radiación para cada paciente.

Los tres médicos panameños que "pellizcaron" el software para que diera capacidad adicional fueron acusados de asesinato en segundo grado. La empresa de Estados Unidos enfrentó litigios serios en los dos países. Gage y McCormick comentan lo siguiente:

> Éste no es un relato para prevenir a los médicos, aun cuando luchen por estar fuera de la cárcel si no entienden o hacen mal uso de la tecnología. Tampoco es la narración de cómo pueden salir heridos, o algo peor, los seres humanos a causa del software mal diseñado o poco explicado, aunque hay muchos ejemplos al respecto. Ésta es la alerta para cualquier creador de programas de cómputo: la calidad del software importa, las aplicaciones deben ser a prueba de tontos y el código mal desplegado —ya sea incrustado en el motor de un automóvil, un brazo robótico o un dispositivo de curación en un hospital— puede matar.

La mala calidad conlleva riesgos, algunos muy serios.

### 14.3.4 Negligencia y responsabilidad

La historia es muy común. Una entidad gubernamental o corporativa contrata a una compañía importante de desarrollo de software o a una consultoría para que analice los requerimientos y luego diseñe y construya un "sistema" basado en software para apoyar alguna actividad de importancia. El sistema debe auxiliar a una función corporativa principal (como la administración de pensiones) o a alguna función gubernamental (por ejemplo, la administración del cuidado de la salud o los créditos hipotecarios).

El trabajo comienza con las mejores intenciones por ambas partes, pero en el momento en el que el sistema se entrega, las cosas han marchado mal. El sistema va retrasado, no da los resultados y funciones deseadas, comete errores y no cuenta con la aprobación del cliente. Comienzan los litigios.

www.FreeLibros.me

En la mayor parte de los casos, el cliente afirma que el desarrollador ha sido negligente (en cuanto a la manera en la que aplicó las prácticas del software), por lo que no merece el pago. Es frecuente que el desarrollador diga que el cliente ha cambiado repetidamente sus requerimientos y trastornado de diversas maneras los acuerdos para el trabajo. En cualquier caso, es la calidad del sistema lo que está en entredicho.

### 14.3.5 Calidad y seguridad

A medida que aumenta la importancia crítica de los sistemas y aplicaciones basados en web, la seguridad de las aplicaciones se ha vuelto más importante. En pocas palabras, el software que no tiene alta calidad es fácil de penetrar por parte de intrusos y, en consecuencia, el software de mala calidad aumenta indirectamente el riesgo de la seguridad, con todos los costos y problemas que eso conlleva.

En una entrevista para *ComputerWorld*, el autor y experto en seguridad Gary McGraw comenta lo siguiente [Wil05]:

> La seguridad del software se relaciona por completo con la calidad. Debe pensarse en seguridad, confiabilidad, disponibilidad y dependencia, en la fase inicial, en la de diseño, en la de arquitectura, pruebas y codificación, durante todo el ciclo de vida del software [proceso]. Incluso las personas conscientes del problema de la seguridad del software se centran en las etapas finales del ciclo de vida. Entre más pronto se detecte un problema en el software, mejor. Y hay dos clases de problemas. Uno son los errores, que son problemas de implementación. El otro son las fallas del software: problemas de arquitectura en el diseño. La gente presta demasiada atención a los errores pero no la suficiente a las fallas.

Para construir un sistema seguro hay que centrarse en la calidad, y eso debe comenzar durante el diseño. Los conceptos y métodos analizados en la parte 2 del libro llevan a una arquitectura del software que reduce las "fallas". Al eliminar las fallas de arquitectura (con lo que mejora la calidad del software) será más difícil que intrusos penetren en el software.

### 14.3.6 El efecto de las acciones de la administración

Es frecuente que la calidad del software reciba influencia tanto de las decisiones administrativas como de las tecnológicas. Incluso las mejores prácticas de la ingeniería de software pueden ser arruinadas por malas decisiones gerenciales y por acciones cuestionables de la administración del proyecto.

En la parte 4 de este libro se analiza la administración del proyecto en el contexto del proceso del software. Al iniciar toda tarea del proyecto, el líder de éste tomará decisiones que tienen un efecto significativo en la calidad del producto.

**Decisiones de estimación.** Como se dice en el capítulo 26, un equipo de software rara vez puede darse el lujo de dar una estimación para el proyecto *antes* de que se hayan establecido las fechas de entrega y especificado un presupuesto general. En vez de ello, el equipo realiza un "filtro sanitario" para garantizar que las fechas de entrega y puntos de revisión son racionales. En muchos casos, hay una presión enorme del tiempo para entrar al mercado que fuerza al equipo a aceptar fechas de entrega irreales. En consecuencia, se toman atajos, se pasan por alto las actividades que elevan la calidad del software y disminuye la calidad del producto. Si una fecha de entrega es irracional, es importante poner los pies sobre la tierra. Explique por qué se necesita más tiempo o, alternativamente, sugiera un subconjunto de funciones que puedan entregarse (sin demasiada calidad) en el tiempo programado.

**Decisiones de programación.** Cuando se establece un programa de desarrollo de un proyecto de software (véase el capítulo 27), se establece la secuencia de las tareas con base en dependencias. Por ejemplo, como el componente **A** depende del procesamiento que ocurra

www.FreeLibros.me

dentro de los componentes **B**, **C** y **D**, el componente **A** no puede programarse para ser probado hasta que los componentes **B**, **C** y **D** no hayan sido probados por completo. La programación del proyecto reflejaría esto. Pero si el tiempo es demasiado escaso y debe disponerse de **A** para realizar pruebas de importancia crítica, puede decidirse a probar **A** sin sus componentes subordinados (que están un poco retrasados) a fin de que esté disponible para otras pruebas que se realicen antes de la entrega. Después de todo, el plazo final se acerca. En consecuencia, **A** podría tener defectos ocultos que sólo se descubrirían mucho tiempo después. La calidad bajaría.

**Decisiones orientadas al riesgo.** La administración del riesgo (véase el capítulo 28) es uno de los atributos clave de un proyecto exitoso de software. En realidad se necesita saber lo que puede salir mal y establecer un plan de contingencia para ese caso. Demasiados equipos de software prefieren un optimismo ciego y establecen un programa de desarrollo con la suposición de que nada saldrá mal. Lo que es peor, no tienen manera de manejar las cosas que salgan mal. En consecuencia, cuando un riesgo se convierte en realidad, reina el caos y aumenta el grado de locuras que se cometen, con lo que invariablemente la calidad se desploma.

El dilema de la calidad del software se resume mejor con el enunciado de la Ley de Meskimen: *Nunca hay tiempo para hacerlo bien, pero siempre hay tiempo para hacerlo otra vez.* Mi consejo es: tomarse el tiempo para hacerlo bien casi nunca es la decisión equivocada.

## 14.4 Lograr la calidad del software

La calidad del software no sólo se ve. Es el resultado de la buena administración del proyecto y de una correcta práctica de la ingeniería de software. La administración y práctica se aplican en el contexto de cuatro actividades principales que ayudan al equipo de software a lograr una alta calidad en éste: métodos de la ingeniería de software, técnicas de administración de proyectos, acciones de control de calidad y aseguramiento de la calidad del software.

### 14.4.1 Métodos de la ingeniería de software

**? ¿Qué necesito hacer para influir en la calidad de manera positiva?**

Si espera construir software de alta calidad, debe entender el problema que se quiere resolver. También debe ser capaz de crear un diseño que esté de acuerdo con el problema y que al mismo tiempo tenga características que lleven al software a las dimensiones y factores de calidad que se estudiaron en la sección 14.2.

En la parte 2 de este libro se presentó una amplia variedad de conceptos y métodos que conducen a una comprensión razonablemente completa del problema y al diseño exhaustivo que establece un fundamento sólido para la actividad de construcción. Si el lector aplica estos conceptos y adopta métodos apropiados de análisis y diseño, se eleva sustancialmente la probabilidad de crear software de alta calidad.

### 14.4.2 Técnicas de administración de proyectos

El efecto de las malas decisiones de administración sobre la calidad del software se estudió en la sección 14.3.6. Las implicaciones son claras: si 1) un gerente de proyecto usa estimaciones para verificar que las fechas pueden cumplirse, 2) se comprenden las dependencias de las actividades programadas y el equipo resiste la tentación de usar atajos, 3) la planeación del riesgo se lleva a cabo de manera que los problemas no alienten el caos, entonces la calidad del software se verá influida de manera positiva.

Además, el plan del proyecto debe incluir técnicas explícitas para la administración de la calidad y el cambio. Las técnicas que llevan a buenas prácticas de administración de proyectos se estudian en la parte 4 de este libro.

www.FreeLibros.me

### 14.4.3 Control de calidad

**? ¿Qué es el control de calidad del software?**

El control de calidad incluye un conjunto de acciones de ingeniería de software que ayudan a asegurar que todo producto del trabajo cumpla sus metas de calidad. Los modelos se revisan para garantizar que están completos y que son consistentes. El código se inspecciona con objeto de descubrir y corregir errores antes de que comiencen las pruebas. Se aplica una serie de etapas de prueba para detectar los errores en procesamiento lógico, manipulación de datos y comunicación con la interfaz. La combinación de mediciones con retroalimentación permite que el equipo del software sintonice el proceso cuando cualquiera de estos productos del trabajo falla en el cumplimiento de las metas de calidad. Las actividades de control de calidad se estudian en detalle en lo que resta de la parte 3 de este libro.

### 14.4.4 Aseguramiento de la calidad

**WebRef**

Pueden encontrarse vínculos útiles acerca de técnicas de aseguramiento de la calidad en la dirección **www.niwotridge.com/Resources/PM-SWEResources/SoftwareQualityAssurance.htm**

El aseguramiento de la calidad establece la infraestructura de apoyo a los métodos sólidos de la ingeniería de software, la administración racional de proyectos y las acciones de control de calidad, todo de importancia crucial si se trata de elaborar software de alta calidad. Además, el aseguramiento de la calidad consiste en un conjunto de funciones de auditoría y reportes para evaluar la eficacia y completitud de las acciones de control de calidad. La meta del aseguramiento de la calidad es proveer al equipo administrativo y técnico los datos necesarios para mantenerlo informado sobre la calidad del producto, con lo que obtiene perspectiva y confianza en que las acciones necesarias para lograr la calidad del producto funcionan. Por supuesto, si los datos provistos a través del aseguramiento de la calidad identifican los problemas, es responsabilidad de la administración enfrentarlos y aplicar los recursos necesarios para resolver los correspondientes a la calidad. En el capítulo 16 se estudia en detalle el aseguramiento de la calidad del software.

## 14.5 Resumen

La preocupación por la calidad de los sistemas basados en software ha aumentado a medida que éste se integra en cada aspecto de nuestras vidas cotidianas. Pero es difícil hacer la descripción exhaustiva de la calidad del software. En este capítulo se define la calidad como un proceso eficaz del software aplicado de modo que crea un producto útil que da un valor medible a quienes lo generan y a quienes lo utilizan.

Con el tiempo se han propuesto varias dimensiones y factores de calidad del software. Todos ellos tratan de definir un conjunto de características que, si se logran, llevarán a un software de alta calidad. McCall y los factores de calidad de la norma ISO 9126 establecen características tales como confiabilidad, usabilidad, facilidad de dar mantenimiento, funcionalidad y portabilidad, como indicadores de la existencia de calidad.

Toda organización de software se enfrenta al dilema de la calidad del software. En esencia, todos quieren elaborar sistemas de alta calidad, pero en un mundo dirigido por el mercado, sencillamente no se dispone del tiempo y el esfuerzo requeridos para producir software "perfecto". La cuestión es la siguiente: ¿debe elaborarse software que sea "suficientemente bueno"? Aunque muchas compañías hacen eso, hay una desventaja notable que debe tomarse en cuenta.

Sin importar el enfoque que se elija, la calidad tiene un costo que puede estudiarse en términos de prevención, evaluación y falla. Los costos de prevención incluyen todas las acciones de la ingeniería de software diseñadas para prevenir los defectos. Los costos de evaluación están asociados con aquellas acciones que evalúan los productos del trabajo de software para determinar su calidad. Los costos de falla incluyen el precio interno de fallar y los efectos externos que precipitan la mala calidad.

www.FreeLibros.me

La calidad del software se consigue por medio de la aplicación de métodos de ingeniería de software, prácticas adecuadas de administración y un control de calidad exhaustivo, todo lo cual es apoyado por la infraestructura de aseguramiento de la calidad. En los capítulos que siguen se estudian con cierto detalle el control y aseguramiento de la calidad.

## Problemas y puntos por evaluar

**14.1.** Describa cómo evaluaría la calidad de una universidad antes de inscribirse. ¿Cuáles factores serían importantes? ¿Cuáles tendrían importancia crítica?

**14.2.** Garvin [Gar84] describe cinco puntos de vista distintos sobre la calidad. Dé un ejemplo de cada uno con el uso de uno o más productos electrónicos conocidos con los que esté familiarizado.

**14.3.** Con el uso de la definición de calidad del software propuesta en la sección 14.2, diga si cree posible crear un producto útil que genere valor medible sin el uso de un proceso eficaz. Explique su respuesta.

**14.4.** Agregue dos preguntas adicionales a cada una de las dimensiones de la calidad de Garvin presentadas en la sección 14.2.1.

**14.5.** Los factores de calidad de McCall se desarrollaron en la década de 1970. Casi todos los aspectos de la computación han cambiado mucho desde entonces, no obstante lo cual aún se aplican al software moderno. ¿Qué conclusiones saca con base en ello?

**14.6.** Con el empleo de los subatributos mencionados en la sección 14.2.3 para el factor de calidad llamado "facilidad de recibir mantenimiento", de la ISO 9126, desarrolle preguntas que exploren si estos atributos existen o no. Continúe el ejemplo presentado en la sección 14.2.4.

**14.7.** Describa con sus propias palabras el dilema de la calidad del software.

**14.8.** ¿Qué es un software "suficientemente bueno"? Mencione una compañía dada y productos específicos que crea que fueron desarrollados con el uso de la filosofía de lo suficientemente bueno.

**14.9.** Considere cada uno de los cuatro aspectos de la calidad y diga cuál piensa que es el más caro y por qué.

**14.10.** Haga una búsqueda en web y encuentre otros tres ejemplos de "riesgos" para el público que puedan atribuirse directamente a la mala calidad de un software. Comience la búsqueda en **http://catless.ncl.ac.uk/risks.**

**14.11.** ¿Son lo mismo *calidad* y *seguridad*? Explique su respuesta.

**14.12.** Explique por qué es que muchos de nosotros utilizamos la ley de Meskimen. ¿Qué ocurre con el software de negocios que causa esto?

## Lecturas y fuentes de información adicionales

Los conceptos básicos de calidad del software se estudian en los libros de Henry y Hanlon (*Software Quality Assurance*, Prentice-Hall, 2008), Kahn *et al.* (*Software Quality: Concepts and Practice*, Alpha Science International, Ltd., 2006), O'Regan (*A Practical Approach to Software Quality*, Springer, 2002) y Daughtrey (*Fundamental Concepts for the Software Quality Engineer*, ASQ Quality Press, 2001).

Duvall *et al.* (*Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley, 2007), Tian (*Software Quality Engineering*, Wiley-IEEE Computer Society Press, 2005), Kandt (*Software Engineering Quality Practices*, Auerbach, 2005), Godbole (*Software Quality Assurance: Principles and Practice*, Alpha Science International, Ltd., 2004) y Galin (*Software Quality Assurance: From Theory to Implementation*, Addison-Wesley, 2003) presentan estudios detallados del aseguramiento de la calidad del software. Stamelos y Sfetsos (*Agile Software Development Quality Assurance*, IGI Global, 2007) estudian el aseguramiento de la calidad en el contexto del proceso ágil.

El diseño sólido conduce a una alta calidad del software. Jayasawal y Patton (*Design for Trustworthy Software*, Prentice-Hall, 2006) y Ploesch (*Contracts, Scenarios and Prototypes*, Springer, 2004) analizan las herramientas y técnicas para desarrollar software "robusto".

La medición es un componente importante de la ingeniería de calidad del software. Ejiogu (*Software Metrics: The Discipline of Software Quality*, BookSurge Publishing, 2005), Kan (*Metrics and Models in Software*

www.FreeLibros.me

*Quality Engineering*, Addison-Wesley, 2002) y Nance y Arthur (*Managing Software Quality*, Springer, 2002) estudian unidades de medida y modelos importantes relacionados con la calidad. Los aspectos de la calidad del software orientados al equipo los estudia Evans (*Achieving Software Quality through Teamwork*, Artech House Publishers, 2004).

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

www.FreeLibros.me

CAPÍTULO

# 15 TÉCNICAS DE REVISIÓN



CONSEJO

*Las revisiones son como filtros en el flujo del trabajo del proceso de software. Si son muy pocas, el flujo queda "sucio". Si son demasiadas, se hace lento hasta detenerse. Utilice métricas para determinar cuáles son las revisiones que funcionan y haga énfasis en ellas. Elimine del flujo las revisiones ineficaces, con objeto de acelerar el proceso.*

Las revisiones del software son un "filtro" para el proceso del software. Es decir, se aplican en varios puntos durante la ingeniería de software y sirven para descubrir errores y defectos a fin de poder eliminarlos. Las revisiones del software "purifican" los productos del trabajo de la ingeniería de software, incluso los modelos de requerimientos y diseño, código y datos de prueba. Freedman y Weinberg [Fre90] analizan del modo siguiente la necesidad de hacer revisiones:

> El trabajo técnico necesita las revisiones por la misma razón que los lápices necesitan borradores: *errar es humano*. La segunda razón por la que son necesarias las revisiones técnicas es porque, si bien las personas son buenas para detectar algunos de sus propios errores, muchas clases de ellos pasan desapercibidos con más facilidad para quien los comete que para otras personas. Por tanto, este proceso de revisión es la respuesta a la oración de Robert Burns:
>
> *Oh, quiera algún Dios el regalo darnos*
>
> *de vernos a nosotros como los demás nos ven*
>
> Una revisión —cualquiera — es una forma de utilizar la diversidad de un grupo para lo siguiente:
>
> 1. Resaltar las mejoras necesarias en el producto que elaboró una sola persona o equipo;
> 2. Confirme aquellas partes de un producto en las que no se desea o no se necesita hacer una mejora;
> 3. Realice el trabajo técnico de calidad más uniforme, o al menos más predecible, que pueda lograrse sin hacer revisiones, a fin de que el trabajo técnico sea más manejable.

Como parte de la ingeniería de software, pueden realizarse muchos diferentes tipos de revisiones. Cada uno tiene su lugar. Una reunión informal alrededor de la máquina del café es una forma de revisión si se analizan problemas técnicos. La presentación formal de la arquitectura del software a un público de clientes, administradores y técnicos también es una forma de revi-

## UNA MIRADA RÁPIDA

**¿Qué es?** Conforme se desarrollen los productos del trabajo de la ingeniería de software se cometerán errores. No es vergonzoso, mientras se trate de detectarlos y corregirlos con ahínco —con mucho ahínco— antes de que lleguen a los usuarios finales. Las revisiones técnicas son el mecanismo más eficaz para detectar los errores en una etapa temprana del proceso de software.

**¿Quién lo hace?** Son los ingenieros de software quienes realizan una revisión técnica, también llamada revisión de pares, con sus colegas.

**¿Por qué es importante?** Si encuentra un error al principio del proceso, es menos caro corregirlo. Además, los errores tienen un modo de amplificarse a medida que avanza el proceso. Por ello, un error relativamente pequeño que se deje sin atender al comenzar el proceso se amplifica en un conjunto más grande de errores en una etapa posterior del proyecto. Finalmente, las revisiones ahorran tiempo, reduciendo la cantidad de repeticiones que se requerirán hacia el final del proyecto.

**¿Cuáles son los pasos?** El enfoque de las revisiones variará en función del grado de formalidad que se elija. En general, se utilizan seis etapas, aunque no todas se emplean siempre: planeación, preparación, estructurar la reunión, resaltar los errores, hacer las correcciones (fuera de la revisión) y verificar que las correcciones se hayan hecho en forma apropiada.

**¿Cuál es el producto final?** El resultado de una revisión es una lista de conceptos o errores descubiertos. Además, también se indica el estado técnico del producto final.

**¿Cómo me aseguro de que lo hice bien?** En primer lugar, seleccione el tipo de revisión que sea apropiada para su cultura de desarrollo. Siga los lineamientos que lleven a ejecutar revisiones exitosas. Si éstas conducen a un software de alta calidad, lo habrá hecho bien.



www.FreeLibros.me

sión. Sin embargo, en este libro nos centramos en las *revisiones técnicas o por pares*, ejemplificadas por las *revisiones casuales,* walkthroughs e *inspecciones*. Desde el punto de vista del control de calidad, una revisión técnica (RT) es el filtro más eficaz. Realizado por ingenieros de software (y de otro tipo) para ingenieros de software, la RT es un medio eficaz para detectar errores y mejorar la calidad.

## 15.1 Efecto de los defectos del software en el costo

En el contexto del proceso del software, los términos *defecto* y *falla* son sinónimos. Los dos implican un problema de calidad descubierto *después* de haberse liberado el software a los usuarios finales (o a otra actividad estructural del proceso del software). En capítulos anteriores se empleó el término *error* para denotar un problema de calidad descubierto por ingenieros de software (o de otra clase) *antes* de entregar el software al usuario final (o a alguna actividad estructural del proceso del software).

**Información**

### *Equivocaciones, errores y defectos*

La meta del control de calidad del software, y en un sentido más amplio de la administración de la calidad en general, es eliminar los problemas de calidad que se encuentren en el software. Se hace referencia a estos problemas con diferentes nombres: *equivocaciones*, *fallas*, *errores* o *defectos*, por mencionar algunos. ¿Son sinónimos estos términos o hay diferencias sutiles entre ellos?

En este libro se hace una distinción clara entre un error (problema de calidad que se detecta antes de que el software se entregue a los usuarios finales) y un defecto (problema de calidad que se encuentra después de haber entregado el software a los usuarios finales[1]). Esta distinción se hace porque los errores y defectos tienen muy distinto efecto económico, empresarial, sicológico y humano. Como ingenieros de software, queremos encontrar y corregir tantos errores como sea posible antes de que el consumidor o el usuario final los encuentren. Queremos evitar los defectos porque hacen (justificadamente) que el personal de software se vea mal.

Sin embargo, es importante observar que la distinción temporal entre errores y defectos que se hace en este libro no constituye la principal forma de pensar. El consenso general de la comunidad de ingeniería de software es que defectos, errores, fallas y equivocaciones son sinónimos. Es decir, el punto en el tiempo en el que se encontró el problema no tiene que ver con el término que se usa para describirlo. Parte de la argumentación a favor de este punto de vista es que en ocasiones es difícil hacer una distinción clara entre el antes y el después de la liberación (por ejemplo, considere un proceso incremental en un desarrollo ágil).

Sin que importe el modo en el que se elija interpretar estos términos, hay que reconocer que el momento en el que se descubre un problema sí importa, y que los ingenieros de software deben tratar de detectar con ahínco —con mucho ahínco— los problemas antes de que sus clientes y usuarios finales los encuentren. Si el lector está más interesado en este tema, puede hallar un análisis razonablemente completo de la terminología acerca de las "equivocaciones" en la dirección **www.softwaredevelopment.ca/bugs.shtml**

*El objetivo principal de una revisión técnica formal es detectar los errores antes de que pasen a otra actividad de la ingeniería de software o de que se entreguen al usuario final.*

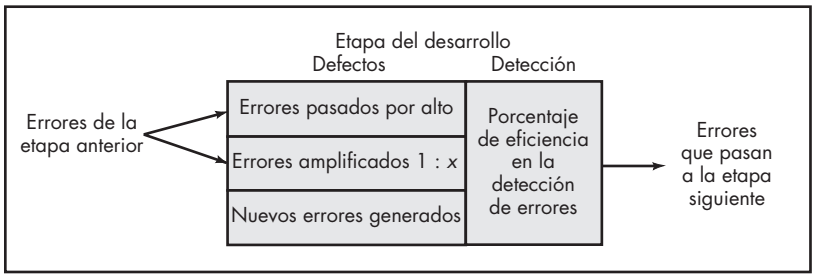
El objetivo principal de las revisiones técnicas es encontrar errores durante el proceso a fin de que no se conviertan en defectos después de liberar el software. El beneficio obvio de las revisiones técnicas es el descubrimiento temprano de los errores, de modo que no se propaguen a la siguiente etapa del proceso del software.

Varios estudios de la industria indican que las actividades de diseño introducen de 50 a 65 por ciento de todos los errores (y en realidad de todos los defectos) durante el proceso del software. Sin embargo, las técnicas de revisión han demostrado tener una eficacia de hasta 75 por ciento [Jon86] para descubrir fallas del diseño. Al detectar y eliminar un gran porcentaje de estos

1 Si se considera una mejora en el proceso del software, un problema de calidad que se propague de una actividad estructural del proceso (como el **modelado**) a otra (como la **construcción**) también se llama "defecto", porque debe encontrarse el problema antes de que un producto del trabajo (como un modelo del diseño) se "libere" a la siguiente actividad.

www.FreeLibros.me

**FIGURA 15.1**
**Modelo de amplificación del defecto**




errores, el proceso de revisión reduce de manera sustancial el costo de las actividades posteriores en el proceso del software.

## 15.2 AMPLIFICACIÓN Y ELIMINACIÓN DEL DEFECTO

**Cita:**

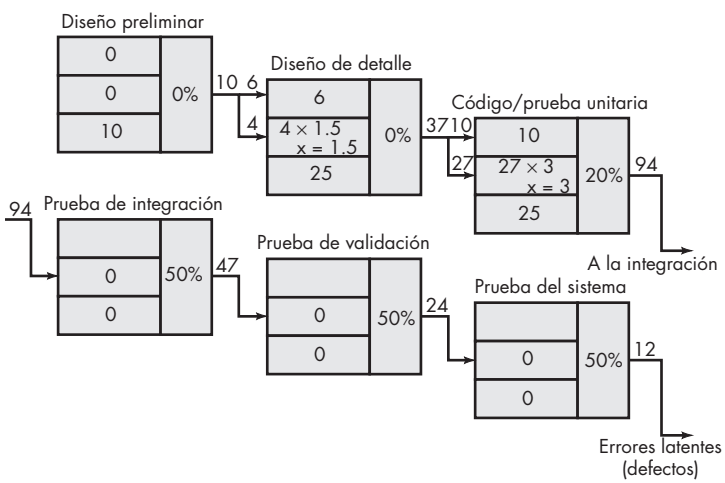
"Dicen los médicos que en sus inicios algunas enfermedades son fáciles de curar pero difíciles de reconocer… mas con el paso del tiempo, si no se detectaron y trataron al principio, se vuelven fáciles de reconocer pero difíciles de curar."

**Nicolás Maquiavelo**

Para ilustrar la generación y detección de errores durante las acciones de diseño y generación de código de un proceso de software, puede usarse un *modelo de amplificación del defecto* [IBM81]. En la figura 15.1 se ilustra esquemáticamente el modelo. Un cuadro representa una acción de la ingeniería de software. Durante la acción, los errores se generan de manera inadvertida. La revisión puede fracasar en descubrir los errores nuevos que se generan y los cometidos en etapas anteriores, lo que da como resultado cierto número de errores pasados por alto. En ciertos casos, los errores de etapas anteriores ignorados son amplificados (en un factor *x* de amplificación) por el trabajo en curso. Las subdivisiones de los cuadros representan a cada una de estas características y al porcentaje de eficiencia de la detección de errores, que es una función de la profundidad de la revisión.

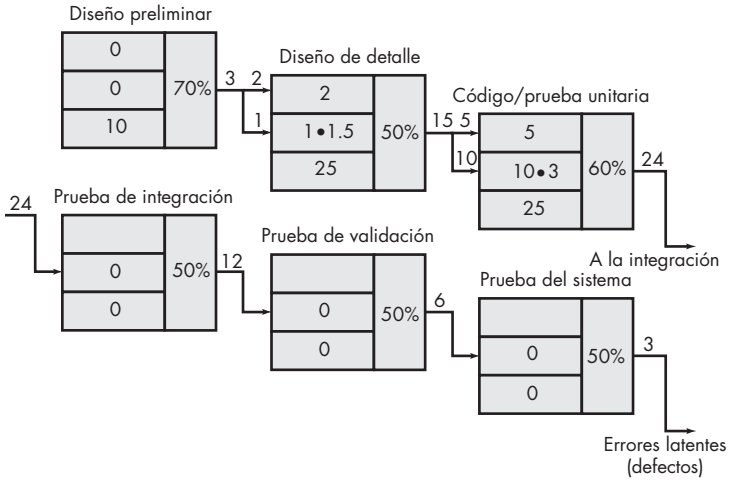
La figura 15.2 ilustra un ejemplo hipotético de amplificación del defecto para un proceso de software en el que no se hacen revisiones. En la figura, se supone que en cada etapa de prueba se detecta y corrige 50 por ciento de todos los errores de entrada sin que se introduzcan nuevos errores (suposición optimista). Diez defectos preliminares de diseño se amplifican a 94 errores antes de que comiencen las pruebas. Se liberan al campo 12 errores latentes (defectos). La figura 15.3 considera las mismas condiciones, excepto porque se efectúan revisiones del diseño y código como parte de cada acción de la ingeniería de software. En este caso, son 10 los errores

**FIGURA 15.2**
**Amplificación del defecto. Sin revisiones**




www.FreeLibros.me

**FIGURA 15.3**
**Amplificación del defecto. Se efectúan revisiones**




iniciales de diseño preliminar (arquitectura) que se amplifican a 24 antes de comenzar las pruebas. Sólo existen tres errores latentes. Pueden establecerse los costos relativos asociados con el descubrimiento y corrección de errores, así como el costo general (con y sin revisión para nuestro ejemplo hipotético). El número de errores detectados durante cada una de las etapas citadas en las figuras 15.2 y 15.3 se multiplica por el costo que implica eliminar un error (1.5 unidades de costo para el diseño, 6.5 unidades de costo antes de las pruebas, 15 unidades de costo durante las pruebas y 67 unidades de costo después de la entrega).[2] Con estos datos, el costo total del desarrollo y mantenimiento cuando se efectúan revisiones es de 783 unidades de costo. Cuando no se hacen revisiones, el costo total es de 2 177 unidades, casi tres veces más caro.

Debe dedicarse tiempo y esfuerzo a la realización de revisiones y su organización de desarrollo debe destinar el dinero para ello. Sin embargo, los resultados del ejemplo anterior dejan pocas dudas acerca de lo que puede pagar ahora o de que después deberá pagar mucho más.

## 15.3 MÉTRICAS DE REVISIÓN Y SU EMPLEO

Las revisiones técnicas son una de las muchas acciones que se requieren como parte de las buenas prácticas de la ingeniería de software. Cada acción requiere un esfuerzo humano dirigido. Como el esfuerzo disponible para el proyecto es finito, es importante que una organización de software comprenda la eficacia de cada acción, definiendo un conjunto de métricas (véase el capítulo 23) que puedan utilizarse para evaluar esa eficacia.

Aunque se han definido muchas métricas para las revisiones técnicas, un conjunto relativamente pequeño da una perspectiva útil. Las siguientes métricas para la revisión pueden obtenerse conforme se efectúe ésta:

- *Esfuerzo de preparación, $E_p$*: esfuerzo (en horas-hombre) requerido para revisar un producto del trabajo antes de la reunión de revisión real.
- *Esfuerzo de evaluación, $E_a$*: esfuerzo requerido (en horas-hombre) que se dedica a la revisión real.
- *Esfuerzo de la repetición, $E_r$*: esfuerzo (en horas-hombre) que se dedica a la corrección de los errores descubiertos durante la revisión.

2 Estos multiplicadores son algo diferentes de los datos presentados en la figura 14.2, que es más actual. Sin embargo, sirven para ilustrar los costos de la amplificación del defecto.

www.FreeLibros.me

- *Tamaño del producto del trabajo, TPT*: medición del tamaño del producto del trabajo que se ha revisado (por ejemplo, número de modelos UML o número de páginas de documento o de líneas de código).
- *Errores menores detectados,* $Err_{menores}$: número de errores detectados que pueden clasificarse como menores (requieren menos de algún esfuerzo especificado para corregirse).
- *Errores mayores detectados,* $Err_{mayores}$: número de errores encontrados que pueden clasificarse como mayores (requieren más que algún esfuerzo especificado para corregirse).

Estas métricas pueden mejorarse, asociando el tipo de producto del trabajo que se revisó con las métricas obtenidas.

### 15.3.1 Análisis de las métricas

Antes de comenzar el análisis deben hacerse algunos cálculos sencillos. El esfuerzo total de revisión y el número total de errores descubiertos se definen como sigue:

$$E_{\text{revisión}} = E_p + E_a + E_r$$

$$\text{Err}_{\text{tot}} = \text{Err}_{\text{menores}} + \text{Err}_{\text{mayores}}$$

La *densidad del error* representa los errores encontrados por unidad de producto del trabajo revisada.

$$\text{Densidad del error} = \frac{\text{Err}_{\text{tot}}}{\text{TPT}}$$

Por ejemplo, si se revisa un modelo de requerimientos con objeto de encontrar errores, inconsistencias y omisiones, es posible calcular la densidad del error en varias formas diferentes. El modelo de requerimientos contiene 18 diagramas UML como parte de 32 páginas de materiales descriptivos. La revisión detecta 18 errores menores y 4 mayores. Por tanto, $\text{Err}_{\text{tot}} = 22$. La densidad del error es 1.2 errores por diagrama UML o 0.68 errores por página del modelo de requerimientos.

Si las revisiones se llevan a cabo para varios tipos distintos de productos del trabajo (por ejemplo, modelo de requerimientos, modelo del diseño, código, casos de prueba, etc.), el porcentaje de errores no descubiertos por cada revisión se confronta con el número total de errores detectados en todas las revisiones. Además, puede calcularse la densidad del error para cada producto del trabajo.

Una vez recabados los datos para muchas revisiones efectuadas en muchos proyectos, los valores promedio de la densidad del error permiten estimar el número de errores por hallar en un nuevo documento (aún no revisado). Por ejemplo, si la densidad promedio de error para un modelo de requerimientos es de 0.6 errores por página, y un nuevo modelo de requerimientos tiene una longitud de 32 páginas, una estimación gruesa sugiere que el equipo de software encontrará alrededor de 19 o 20 errores durante la revisión del documento. Si sólo encuentra 6 errores, habrá hecho un trabajo extremadamente bueno al desarrollar el modelo de requerimientos o su enfoque de la revisión no fue tan profundo.

Una vez llevada a cabo la prueba (véanse los capítulos 17 a 20), es posible obtener datos adicionales del error, incluso el esfuerzo requerido para detectar y corregir errores no descubiertos durante las pruebas y la densidad del error del software. Los costos asociados con la detección y corrección de un error durante las pruebas pueden compararse con los de las revisiones. Esto se analiza en la sección 15.3.2.

### 15.3.2 Eficacia del costo de las revisiones

Es difícil medir en tiempo real la eficacia del costo de cualquier revisión técnica. Una organización de ingeniería de software puede evaluar la eficacia de las revisiones y su relación costo-

www.FreeLibros.me

beneficio sólo después de que éstas han terminado, de que las unidades de medida de la revisión se han recabado, de que los datos promedio han sido calculados y de que la calidad posterior del software ha sido medida (mediante pruebas).

Si regresamos al ejemplo presentado en la sección 15.3.1, se determinó que la densidad promedio del error para los modelos de requerimientos era de 0.6 errores por página. Se reveló que el esfuerzo requerido para corregir un error menor en el modelo era de 4 horas-hombre. Se vio que el esfuerzo necesario para un error mayor en los requerimientos era de 18 horas-hombre. Al estudiar los datos recabados se observa que los errores menores ocurrieron con una frecuencia cercana a 6 veces más que los errores mayores. Por tanto, puede estimarse que el esfuerzo promedio para detectar y corregir un error en los requerimientos durante la revisión es alrededor de 6 horas-hombre.

Los errores relacionados con los requerimientos no detectados durante las pruebas requieren un promedio de 45 horas-hombre para encontrarse y corregirse (no hay datos disponibles acerca de la severidad relativa del error). Con estos promedios se obtiene lo siguiente:

$$\text{Esfuerzo ahorrado por error} = E_{\text{pruebas}} - E_{\text{revisiones}}$$
$$45 - 6 = 30 \text{ horas-hombre/error}$$

Como durante la revisión del modelo de requerimientos se encontraron 22 errores, se tendrá un ahorro cercano a 660 horas-hombre en el esfuerzo dedicado a las pruebas. Y esto se refiere sólo a los errores relacionados con los requerimientos. Al beneficio general se suman aquellos asociados con el diseño y el código. El esfuerzo total conduce a ciclos de entrega más cortos y a un mejor tiempo para llegar al mercado.

En su libro sobre la revisión por pares, Karl Wiegers [Wie02] analiza datos procedentes de anécdotas de compañías grandes que han utilizado *inspecciones* (un tipo relativamente formal de revisión técnica) como parte de sus actividades de control de calidad del software. Hewlett Packard reportó un rendimiento de 10 a 1 sobre la inversión gracias a las inspecciones y afirmó que la entrega real del producto se aceleró en un promedio de 1.8 meses-calendario. AT&T indicaba que las inspecciones habían reducido el costo general de los errores de software en un factor de 10, que la calidad había mejorado en un orden de magnitud y que la productividad se había incrementado 14 por ciento. Otras empresas reportaban beneficios similares. Las revisiones técnicas (en diseño y otras actividades) generan una buena relación costo-beneficio y en verdad ahorran tiempo.

Pero para muchos profesionales del software, esta afirmación va contra la intuición. "Las revisiones toman tiempo", dicen, "y no tenemos tiempo que perder…". Afirman que el tiempo es precioso en cada proyecto de software y que la actividad de revisar "todo producto del trabajo con detalle" absorbe demasiado.

Los ejemplos presentados en esta sección indican otra cosa. Lo más importante es que los datos de la industria sobre revisiones del software se han recabado durante más de dos décadas y se resumen cualitativamente en las gráficas que aparecen en la figura 15.4

En la figura, el trabajo efectuado cuando se utilizan revisiones se refleja pronto en el desarrollo de un incremento de software, pero esta inversión temprana paga dividendos debido a que se reduce el esfuerzo necesario para hacer pruebas y correcciones. De igual importancia es que la fecha de entrega del desarrollo con revisiones ocurre antes que la que se hace sin revisiones. ¡Las revisiones no quitan tiempo, lo ahorran!

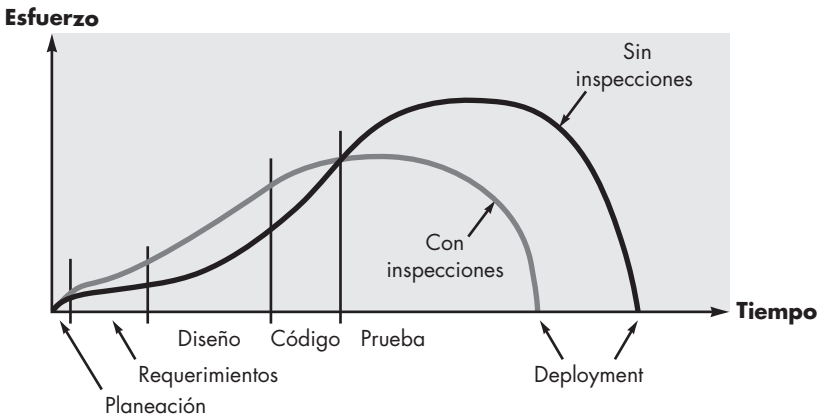
## 15.4 Revisiones: espectro de formalidad

Las revisiones técnicas deben aplicarse con un nivel de formalidad apropiado para el producto que se va a elaborar, para el plazo que tiene el proyecto y para el personal que realice el trabajo.

www.FreeLibros.me

**FIGURA 15.4**

**Esfuerzo realizado, con y sin revisiones**
*Fuente:* adaptado de [Fog86].




La figura 15.5 ilustra un modelo de referencia para las revisiones técnicas [Lai02] que identifica cuatro características que contribuyen a la formalidad con la que se efectúa una revisión.

Cada una de las características del modelo de referencia ayuda a definir el nivel de formalidad de la revisión. La formalidad de una revisión se incrementa cuando: 1) se definen explícitamente roles distintos para los revisores, 2) hay suficiente cantidad de planeación y preparación para la revisión, 3) se define una estructura distinta para la revisión (incluso tareas y productos internos del trabajo) y 4) el seguimiento por parte de los revisores tiene lugar para cualesquiera correcciones que se efectúen.

Para entender el modelo de referencia, supongamos que el lector decidió revisar el diseño de la interfaz para **CasaSeguraAsegurada.com**. Esto puede hacerse de varias maneras diferentes, que van de lo relativamente casual a lo riguroso en extremo. Si decide que el enfoque casual es más apropiado, se pide a algunos colegas (pares) que examinen el prototipo de la interfaz en un esfuerzo por descubrir problemas potenciales. Todos deciden que no habrá preparación previa, pero que evaluarán el prototipo en una forma razonablemente estructurada: primero verán la distribución, luego la estética, después las opciones de navegación, etc. Como diseñador que es, el lector decide tomar algunas notas, pero nada formales.

Pero, ¿qué pasa si la interfaz es crucial para el éxito de todo el proyecto? ¿Qué sucede si de lo acertado de su ergonomía dependen vidas humanas? Debió concluirse que era necesario un enfoque más riguroso. Se forma entonces el equipo de revisión. Cada integrante de éste tendrá

**FIGURA 15.5**

**Modelo de referencia para hacer revisiones técnicas**


www.FreeLibros.me

un rol específico: dirigir el equipo, registrar las reuniones, presentar el material, etc. Cada revisor tendrá acceso al producto del trabajo (en este caso, el prototipo de la interfaz) antes de que la revisión tenga lugar y dedicará tiempo a la búsqueda de errores, inconsistencias y omisiones. Se realizará un conjunto de tareas específicas con base en una agenda que se desarrollará antes de que ocurra la revisión. Los resultados de ésta serán registrados de manera formal y el equipo decidirá sobre el estado del producto del trabajo con base en el resultado de la revisión. Los miembros del equipo también verificarán que las correcciones se hagan de manera adecuada.

En este libro se consideran dos grandes categorías de revisiones técnicas: revisiones informales y revisiones técnicas más formales. Dentro de cada una de ellas se escogen varios enfoques diferentes. Éstos se presentan en las secciones que siguen.

## 15.5 Revisiones informales

Las revisiones informales incluyen una simple verificación de escritorio de un trabajo de ingeniería de software, hecha con algún colega, o una reunión casual (con más de dos personas) con objeto de revisar un producto o aspectos orientados a la revisión de programación por pares (véase el capítulo 3).

Una *verificación de escritorio* simple o una *reunión casual* realizada con un colega constituye una revisión. Sin embargo, como no hay una planeación o preparación por adelantado, ni agenda o estructura de la reunión, y no se da seguimiento a los errores descubiertos, la eficacia de tales revisiones es mucho menor que la de los enfoques más formales. Pero una verificación de escritorio sencilla descubre errores que de otro modo se propagarían en el proceso del software.

Una forma de mejorar la eficacia de una verificación de escritorio es desarrollar un conjunto de listas de revisión para cada producto grande del trabajo generado por el equipo de software. Las preguntas que se plantean en la lista son generales, pero servirán para guiar a los revisores en la verificación del producto. Por ejemplo, veamos una verificación de escritorio del prototipo de la interfaz de **CasaSeguraAsegurada.com**. En vez de sólo jugar con el prototipo en la estación de trabajo del diseñador, éste y un colega lo examinan con el empleo de una lista para interfaces:

- ¿La distribución está diseñada con el empleo de convenciones estándar? ¿De izquierda a derecha? ¿De arriba abajo?
- ¿La presentación necesita ser desplazada verticalmente?
- ¿Se usan con eficacia el color y la ubicación, la tipografía y el tamaño?
- ¿Todas las opciones o funciones de navegación están representadas en el mismo nivel de abstracción?
- ¿Están etiquetadas con claridad todas las elecciones de navegación?

y así sucesivamente. Cualesquiera errores o aspectos señalados por los revisores son registrados por el diseñador para resolverlos tiempo después. Las verificaciones de escritorio se programan en forma *ad hoc* o son obligatorias como parte de las buenas prácticas de la ingeniería de software. En general, la cantidad de material por revisar es relativamente pequeña y el tiempo total dedicado a una revisión de escritorio es de poco más de una hora o dos.

En el capítulo 3 se describió la *programación por pares* en la forma siguiente: "La XP recomienda que dos personas trabajen juntas en una estación de trabajo con objeto de crear el código de una narración. Esto proporciona un mecanismo para resolver problemas y asegurar la calidad en tiempo real (dos cabezas piensan más que una)."

La programación por pares se caracteriza por una verificación de escritorio continua. En vez de programar una revisión en algún momento dado, la programación por pares invita a hacer

www.FreeLibros.me

una revisión continua a medida que se crea el producto (diseño o código). El beneficio es el inmediato descubrimiento de los errores y, en consecuencia, la mejora de la calidad del producto.

En su estudio sobre la eficacia de la programación por pares, Williams y Kessler [Wil00] afirman lo siguiente:

> Las evidencias anecdóticas e iniciales señalan que la programación por pares es una técnica poderosa para generar productivamente trabajos de software de alta calidad. Los elementos de la pareja laboran y comparten sus ideas para resolver las complejidades del desarrollo del software. Realizan de manera continua inspecciones de lo que hace cada quien, lo que conduce a una forma de eliminación de defectos más rápida y eficiente. Además, se mantienen centrados intensamente en la tarea uno del otro.

Algunos ingenieros de software dicen que la redundancia inherente construida en la programación por parejas es un desperdicio de recursos. Después de todo, ¿por qué asignar dos personas a un trabajo que podría ejecutar sólo una? La respuesta a esta pregunta se encuentra en la sección 15.3.2. Si la calidad del producto del trabajo generado como consecuencia de la programación en parejas es mucho mejor que el trabajo de un individuo, los ahorros relacionados con la calidad justifican de sobra la "redundancia" implícita en la programación por parejas.

**INFORMACIÓN**

### *Listas de verificación para revisión*

Aun cuando las revisiones estén bien organizadas y se lleven a cabo de manera apropiada, no es mala idea dar a los revisores una "criba". Es decir, es útil tener una lista de verificación que dé a cada revisor las preguntas que debe plantear acerca del producto específico del trabajo que se revisa.

Una de las listas más completas es la desarrollada por la NASA en el Centro Goddard de Vuelos Espaciales, disponible en la dirección **http://sw-assurance.gsfc.nasa.gov/disciplines/quality/index.php**

Hay otras listas útiles de revisión técnica que han sido propuestas por las siguientes entidades:

*Process Impact* (**www.processimpact.com/pr_goodies.shtml**)

*Software Dioxide* (**www.softwaredioxide.com/Channels/ConView.asp?id=6309**)

*Macadamian* (**www.macadamian.com**)

*The Open Group Architecture Review Checklist* (**www.opengroup.org/architecture/togaf7-doc/arch/p4/comp/clists/syseng.htm**)

*DFAS* (puede descargarse, **www.dfas.mil/technology/pal/ssps/docstds/spm036.doc**)

## 15.6 REVISIONES TÉCNICAS FORMALES

**Cita:**

"No hay nada más urgente para alguien que corregir el trabajo de los demás."

**Mark Twain**

Una *revisión técnica formal* (RTF) es una actividad del control de calidad del software realizada por ingenieros de software (y otras personas). Los objetivos de una RTF son: 1) descubrir los errores en funcionamiento, lógica o implementación de cualquier representación del software; 2) verificar que el software que se revisa cumple sus requerimientos; 3) garantizar que el software está representado de acuerdo con estándares predefinidos; 4) obtener software desarrollado de manera uniforme y 5) hacer proyectos más manejables. Además, la RTF sirve como método de capacitación, pues permite que los ingenieros principiantes observen distintos enfoques de análisis, diseño e implementación del software. La RTF también funciona para estimular el respaldo y la continuidad debido a que varias personas se familiarizan con software que de otra manera no hubieran visto.

La RTF en realidad es una clase que incluye *walkthroughs* e *inspecciones*. Cada RTF se realiza como una reunión y tendrá éxito sólo si se planea, controla y ejecuta en forma apropiada. En las secciones que siguen se presentan lineamientos similares a aquellos usados para un walkthrough, como representativos de la revisión técnica formal. Si el lector tiene interés en las ins-

www.FreeLibros.me

pecciones de software y en obtener más información sobre walkthroughs, consulte [Rad02], [Wie02] o [Fre90].

### 15.6.1 La reunión de revisión

**WebRef**

El documento *Formal Inspection Guidebook*, de la NASA, puede descargarse del sitio **satc.gsfc.nasa.gov/Documents/fi/gdb/fi.pdf**

Sin importar cuál formato de RTF se elija, cualquiera de ellos debe cumplir las restricciones siguientes:

- En la revisión deben involucrarse de tres a cinco personas (normalmente).
- Debe haber preparación previa, pero no debe exigir más de dos horas de trabajo de cada persona.
- La duración de la reunión de revisión debe ser de al menos dos horas.

Dadas estas restricciones, debe resultar obvio que una RTF se centra en una parte específica (y pequeña) del software general. Por ejemplo, en vez de tratar de revisar todo el diseño, se hacen walkthrougs para cada componente o grupo pequeño de componentes. Al reducir el alcance, la RTF tiene mayor probabilidad de detectar errores.


Una RTF se centra en una parte relativamente pequeña de un producto del trabajo.

La atención de la RTF se dirige a un producto (por ejemplo, una parte del modelo de requerimientos, el diseño detallado de un componente o su código fuente, etc.). El individuo que haya desarrollado el producto —el *productor*— informa al líder del proyecto que ha terminado y que se requiere hacer una revisión. El líder del proyecto contacta al *líder de la revisión*, quien evalúa el producto en cuanto a su conclusión, genera copias de los materiales del producto y las distribuye a dos o tres *revisores* para la preparación previa. Se espera que cada revisor dedique de una a dos horas a la inspección del producto, tome notas y se familiarice con el trabajo. Al mismo tiempo, el líder del proyecto también revisa el producto y establece una agenda para la reunión de revisión, que por lo general se programa para el día siguiente.


*En ciertas situaciones, es buena idea que alguien distinto del productor haga el walkthroug del producto que se revisa. Esto lleva a una interpretación literal del producto y a mejorar el reconocimiento de errores.*

A la reunión de revisión acuden el líder de ésta, todos los revisores y el productor. Uno de los revisores adopta el rol de *secretario*, es decir, quien registra (por escrito) todos los acontecimientos importantes que surjan durante la revisión. La RTF comienza con el análisis de la agenda y una introducción breve por parte del productor. Después, éste procede a "recorrer" el producto del trabajo, explicando el material, mientras los revisores hacen sus comentarios con base en la preparación que hicieron. Cuando se descubren problemas o errores válidos, el secretario toma nota de ellos.

Al terminar la revisión, todos los asistentes deben decidir si: 1) aceptan el producto sin modificaciones, 2) lo rechazan debido a errores graves (una vez corregidos, se realiza otra revisión) o 3) aceptan el producto de manera provisional (se encontraron errores menores que deben corregirse, pero no se necesita otra revisión). Una vez tomada la decisión, todos los asistentes a la RTF firman el acta que indica su participación y su acuerdo con los descubrimientos del equipo de revisión.

### 15.6.2 Reporte y registro de la revisión

Durante la RTF, un revisor (el secretario) registra activamente todos los asuntos que se planteen. Éstos se resumen al final de la reunión y se produce la *lista de pendientes de la revisión*. Además se elabora un *reporte técnico formal de la revisión*. Éste responde tres preguntas:

1. ¿Qué fue lo que se revisó?
2. ¿Quién lo revisó?
3. ¿Cuáles fueron los descubrimientos y las conclusiones?

El resumen del reporte de la revisión es una sola página (quizá con anexos) que se vuelve parte del registro histórico del proyecto y se entrega al líder del proyecto y a otras partes interesadas.

www.FreeLibros.me

La lista de pendientes de la revisión tiene dos propósitos: 1) identificar áreas de problemas en el producto y 2) servir como lista de verificación de acciones que guíe al productor cuando se hagan las correcciones. La lista de pendientes normalmente se anexa al reporte técnico.

Debe establecerse un procedimiento de seguimiento para garantizar que los pendientes de la lista se corrijan de manera apropiada. A menos que esto se haga, es posible que los pendientes anotados "se pierdan en el camino". Un enfoque consiste en asignar la responsabilidad del seguimiento al líder del proyecto.

### 15.6.3 Lineamientos para la revisión


*No señale los errores en forma grosera. Una manera amable de hacerlo es plantear preguntas que lleven al productor a descubrir el error.*

Los lineamientos para efectuar revisiones técnicas formales deben establecerse por adelantado, distribuirse a todos los revisores, llegar al consenso y, después, seguirse. Una revisión sin control con frecuencia es peor que si no se hiciera ninguna. Los siguientes representan un conjunto mínimo de lineamientos para hacer revisiones técnicas formales:

1. *Revise el producto, no al productor.* Una RTF involucra personas y sus egos. Si se lleva a cabo en forma adecuada, la RTF debe dejar en todos los participantes una sensación de logro. Si se efectúa de modo inapropiado, adopta un aire inquisitorial. Los errores deben señalarse en forma amable; el tono de la reunión debe ser relajado y constructivo; el trabajo no debe apenar o menospreciar a nadie. El líder de la revisión debe conducir la reunión en tono y actitud apropiados y debe detenerla de inmediato si se sale de control.
2. *Establezca una agenda y sígala*. Una de las fallas clave de las reuniones de todo tipo es la dispersión. Una RTF debe mantenerse encarrilada y dentro del programa. El líder de la revisión tiene la responsabilidad de que así sea y no debe sentir temor de llamar al orden a las personas cuando se dispersen.
3. *Limite el debate y las contestaciones.* Cuando el revisor plantee un asunto, quizá no haya acuerdo universal acerca de su efecto. En vez de perder tiempo en debatir la cuestión, ésta debe registrarse para discutirla después.
4. *Enuncie áreas de problemas, pero no intente resolver cada uno.* Una revisión no es una sesión para resolver problemas. Es frecuente que la solución de un problema la obtenga el productor, solo o con ayuda de otra persona. La solución de los problemas debe posponerse para después de la reunión de revisión.
5. *Tome notas por escrito.* A veces es buena idea que el secretario tome notas en un pizarrón a fin de que la redacción y prioridades sean evaluadas por los demás revisores a medida que la información se registra. De manera alternativa, pueden tomarse notas directamente en una computadora.
6. *Limite el número de participantes e insista en la preparación previa.* Dos cabezas piensan más que una, pero 14 no son necesariamente mejor que 4. Mantenga limitado el número de personas involucradas. Sin embargo, todos los miembros del equipo de revisión deben prepararse. El líder de la revisión tiene que solicitar comentarios por escrito (lo que proporciona un indicador de que el revisor ha inspeccionado el material).
7. *Desarrolle una lista de verificación para cada producto que sea probable que se revise.* Una lista de verificación ayuda al líder del proyecto a estructurar la RTF y a cada revisor a centrarse en los aspectos importantes. Deben desarrollarse listas para los productos del análisis, el diseño, el código e incluso las pruebas.
8. *Asigne recursos y programe tiempo para las RTF.* Para que las revisiones sean eficaces, deben programarse como tareas del proceso de software. Además, debe programarse tiempo para hacer las inevitables modificaciones que ocurrirán como resultado de la RTF.

**Cita:**

"Una reunión es muy frecuentemente un evento en el cual los minutos son tomados y las horas son gastadas."

**Autor desconocido**

**Cita:**

"Es una de las más hermosas compensaciones de la vida, que ningún hombre pueda sinceramente ayudar a otro sin ayudarse a sí mismo."

**Ralph Waldo Emerson**

www.FreeLibros.me

9. *Dé una capacitación significativa a todos los revisores.* Para que una revisión sea eficaz, todos los revisores deben recibir cierta capacitación formal. Ésta debe hacer énfasis tanto en aspectos relacionados con el proceso como en el lado de la sicología humana de la revisión. Freedman y Weinberg [Fre90] estiman en un mes la curva de aprendizaje para que 20 personas participen de modo eficaz en una revisión.
10. *Revise las primeras revisiones.* Volver a revisar puede ser benéfico para descubrir problemas con el proceso de revisión en sí mismo. El primer producto por revisar deben ser los lineamientos de la revisión.

Debido a que son muchas las variables (número de participantes, tipo de productos del trabajo, tiempo y duración, enfoque específico de la revisión, etc.) que influyen en que una revisión sea exitosa, la organización de software debe experimentar para determinar el enfoque que mejor funcione en el contexto local.

### 15.6.4 Revisiones orientadas al muestreo

Idealmente, todo producto del trabajo de la ingeniería de software debe pasar por una revisión técnica. En el mundo real de los proyectos de software, los recursos son limitados y el tiempo, escaso. En consecuencia, es frecuente que las revisiones se omitan, aun cuando se reconozca su valor como un mecanismo de control de calidad.

Thelin *et al.* [The01] sugieren un proceso de revisión orientado al muestreo en el que se toman muestras de todos los productos del trabajo de ingeniería de software a fin de inspeccionarlos para determinar cuáles son más susceptibles de tener errores. Después se enfocan todos los recursos de la RTF sólo en aquellos productos en los que sea muy probable encontrar errores (con base en los datos obtenidos durante el muestreo).


*Las revisiones toman tiempo, y éste estará bien invertido. Sin embargo, si hay poco tiempo y no hay otra opción, no omita las revisiones. En vez de ello, aplique la revisión orientadas al muestreo.*

Para que sea eficaz, el proceso de revisión orientada al muestreo debe tratar de identificar aquellos productos del trabajo que sean objetivos principales para hacer la RTF. Para lograrlo se sugiere seguir las etapas siguientes [The01]:

1. Inspeccionar una fracción $a_i$ de cada producto del trabajo $i$. Registrar el número de fallas $f_i$ encontradas dentro de $a_i$.
2. Desarrollar una estimación gruesa del número de fallas en el producto del trabajo $i$, con la multiplicación de $f_i$ por $1/a_i$.
3. Ordenar los productos del trabajo en orden descendente de acuerdo con la estimación gruesa del número de fallas que hay en cada uno.
4. Dedicar los recursos disponibles para la revisión a aquellos productos que tengan el número estimado más grande de fallas.

La fracción del producto del trabajo de la que se tomen muestras debe ser representativa del producto del trabajo total y suficientemente grande a fin de que tenga significado para todos los revisores que no hagan muestreo. A medida que aumenta $a_i$ se incrementa la probabilidad de que la muestra sea una representación válida del producto del trabajo. Sin embargo, los recursos requeridos para hacer el muestreo también aumentan. El equipo de ingeniería de software debe establecer el mejor valor de $a_i$ para los tipos de productos generados.[3]

3 Thelin *et al.*, realizaron una simulación detallada que puede ayudar a hacer esto. Consulte [The01] para mayores detalles.

www.FreeLibros.me

**CasaSegura**

### *Aspectos de la calidad*

**La escena:** La oficina de Doug Miller, al comenzar el proyecto de software *CasaSegura*.

**Participantes:** Doug Miller (gerente del equipo de ingeniería de software de *CasaSegura*) y otros miembros del equipo.

**La conversación:**

**Doug:** Sé que no hemos dedicado tiempo a desarrollar un plan de calidad para este proyecto, pero ya estamos en él y tenemos que tomar en cuenta la calidad… ¿Correcto?

**Jamie:** Seguro. Decidimos que conforme desarrollemos el modelo de requerimientos [capítulos 6 y 7], Ed hará un procedimiento para probar cada uno de ellos.

**Doug:** Eso es realmente bueno, pero no vamos a esperar a las pruebas para evaluar la calidad, ¿verdad?

**Vinod:** No, por supuesto que no… Hemos programado revisiones en el plan del proyecto para este incremento de software. Comenzaremos el control de calidad con las revisiones.

**Jamie:** Me preocupa un poco que no tengamos tiempo suficiente para hacer todas las revisiones. En realidad, sé que no lo tendremos.

**Doug:** Mmm… ¿Qué proponen?

**Jamie:** Que seleccionemos aquellos elementos del modelo de requerimientos y diseño que tengan más importancia crítica para *CasaSegura* y que los revisemos.

**Vinod:** Pero, ¿qué pasa si hay algo mal en una parte que no hayamos revisado?

**Shakira:** Leí algo sobre una técnica de muestreo [sección 15.6.4] que podría ayudarnos a determinar candidatos a la revisión (Shakira explica este enfoque.)

**Jamie:** Quizá… pero no estoy seguro de que tengamos tiempo incluso para tomar muestras de cada elemento de los modelos.

**Vinod:** Doug, ¿qué quieres que hagamos?

**Doug:** Tomemos algo de la programación extrema [véase el capítulo 3]. Desarrollaremos los elementos de cada modelo en parejas —dos personas— y haremos una revisión informal de cada una conforme avancemos. Entonces nos abocaremos a los elementos "críticos" para hacer una revisión más formal en equipo, pero hay que mantener esas revisiones al mínimo. De ese modo, todo será observado por más de un par de ojos y también cumpliremos nuestras fechas de entrega.

**Jamie:** Eso significa que vamos a tener que revisar la programación de actividades.

**Doug:** Así es. La calidad altera la programación de este proyecto.

## 15.7 Resumen

El objetivo de toda revisión técnica es detectar errores y descubrir aspectos que tendrían un efecto negativo en el software que se va a desarrollar. Entre más pronto se descubra y corrija un error, menos probable es que se propague a otros productos del trabajo de la ingeniería de software y que se amplifique, lo que provocaría un mayor esfuerzo para corregirlo.

A fin de determinar si las actividades de control de calidad funcionan, deben determinarse varias métricas. Éstas se centran en el esfuerzo requerido para realizar la revisión y los tipos y severidad de errores descubiertos durante la revisión. Una vez recabadas las métricas, se usan para evaluar la eficacia de las revisiones que se efectúen. Los datos de la industria indican que las revisiones tienen un rendimiento elevado sobre la inversión.

Un modelo de referencia para la formalidad de la revisión identifica roles de las personas, planeación y preparación, estructura de la reunión, enfoque de corrección y verificación como las características que indican el grado de formalidad con el que se realiza una revisión. Las revisiones informales son de naturaleza casual, pero pueden usarse con eficacia para detectar errores. Las revisiones formales son más estructuradas y tienen una probabilidad mayor de dar como resultado un software de alta calidad.

Las revisiones informales se caracterizan por tener una planeación y preparación mínimas y poco registro de su desarrollo. Las verificaciones de escritorio y la programación por parejas forman parte de esta categoría de revisión.

Una revisión técnica formal es una reunión estilizada que ha demostrado ser extremadamente eficaz para detectar errores. Los walkthrougs y las inspecciones establecen roles definidos para cada revisor, estimulan la planeación y la preparación previa, requieren la aplicación de lineamientos de revisión definidos y ordenan llevar registros y hacer reportes. Las revisiones

www.FreeLibros.me

por muestreo se utilizan cuando no es posible efectuar revisiones técnicas formales para todos los productos del trabajo.

## Problemas y puntos por evaluar

**15.1.** Explique la diferencia entre un *error* y un *defecto*.

**15.2.** ¿Por qué no puede esperarse a las pruebas para encontrar y corregir todos los errores del software?

**15.3.** Suponga que en el modelo de requerimientos se han cometido 10 errores y que cada uno se amplificará en un factor de 2:1 en el diseño, y que se cometerán otros 20 errores de diseño adicionales que luego se amplificarán en un factor de 1.5:1 en el código, donde se cometerán otros 30 errores adicionales. Suponga que todas las pruebas unitarias encontrarán 30 por ciento de todos los errores, que la integración descubrirá 30 por ciento de los restantes y que las pruebas de validación hallarán 50 por ciento de los que queden. No se efectuarán revisiones. ¿Cuántos errores saldrán al público?

**15.4.** Vuelva a considerar la situación descrita en el problema 15.3, pero ahora suponga que se realizan revisiones en los requerimientos, diseño y código, con 60 por ciento de eficacia en el descubrimiento de todos los errores en esa etapa. ¿Cuántos errores saldrán al público?

**15.5.** Estudie de nuevo la situación descrita en los problemas 15.3 y 15.4. Si cada uno de los errores que salen al público tiene un costo de $4 800 por ser detectado y corregido, y hacer lo mismo para cada error descubierto en la revisión cuesta $240, ¿cuánto dinero se ahorra por efectuar revisiones?

**15.6.** En sus propias palabras, describa el significado de la figura 15.4.

**15.7.** ¿Cuál de las características del modelo de referencia piensa usted que tiene el mayor efecto en la formalidad de la revisión? Explique por qué.

**15.8.** ¿Se le ocurren algunos casos en los que una verificación de escritorio genere problemas en lugar de beneficios?

**15.9.** Una revisión técnica formal es eficaz sólo si cada quien se prepara por adelantado. ¿Cómo se reconoce a un participante que no se haya preparado? ¿Qué haría si usted fuera el líder de la revisión?

**15.10.** Al considerar todos los lineamientos para la revisión presentados en la sección 15.6.3, ¿cuál piensa que sea el más importante y por qué?

## Lecturas y fuentes de información adicionales

Se han escrito relativamente pocos libros sobre las revisiones de software. Algunas de las ediciones recientes que dan una guía útil incluyen los textos de Wong (*Modern Software Review*, IRM Press, 2006), Radice (*High Quality, Low Cost Software Inspections*, Paradoxicon Publishers, 2002), Wiegers (*Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2001) y Gilb y Graham (*Software Inspection*, Addison-Wesley, 1993). El de Freedman y Weinberg (*Handbook of Walkthroughs, Inspections and Technical Reviews*, Dorset House, 1990) sigue siendo un texto clásico y todavía proporciona información útil acerca de este tema tan importante.

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

www.FreeLibros.me

CAPÍTULO

# 16 ASEGURAMIENTO DE LA CALIDAD DEL SOFTWARE

CONCEPTOS CLAVE



El enfoque de la ingeniería de software descrito en este libro se dirige a una sola meta: producir software a tiempo y de alta calidad. Pero muchos lectores se preguntarán: "¿Qué es calidad del software?".

Philip Crosby [Cro79], en su libro clásico sobre calidad, da una respuesta irónica a esta pregunta:

> El problema de la administración de la calidad no es lo que la gente ignora de ella. El problema es lo que piensan que saben...
>
> En ese sentido, la calidad tiene mucho en común con el sexo. Todo mundo lo busca (en ciertas condiciones, por supuesto). Todos creen que lo entienden (aunque no querrían explicarlo). Todos piensan que su ejecución sólo consiste en seguir las inclinaciones naturales (después de todo, lo hacemos de algún modo). Y, por supuesto, la mayoría de la gente siente que los problemas en esta área los causan las demás personas (si sólo se dieran el tiempo de hacer las cosas bien).

En realidad, la calidad es un concepto difícil (se abordó con cierto detalle en el capítulo 14).[1]

Algunos desarrolladores de software todavía creen que la calidad del software es algo por lo que hay que empezar a preocuparse una vez generado el código. Nada podría estar más lejos de la verdad... El *aseguramiento de la calidad del software* (con frecuencia llamado *administración*

UNA MIRADA RÁPIDA

**¿Qué es?** No basta hablar por hablar para decir que la calidad del software es importante. Tiene que 1) definirse explícitamente lo que quiere decir "calidad del software", 2) crearse un conjunto de actividades que ayuden a garantizar que todo producto de la ingeniería de software tenga alta calidad, 3) desarrollarse el control de calidad y las actividades para asegurar ésta en todo proyecto de software, 4) usarse métricas para desarrollar estrategias a fin de mejorar el proceso del software y, en consecuencia, la calidad del producto final.

**¿Quién lo hace?** Todos los involucrados en el proceso de ingeniería de software son los responsables de la calidad.

**¿Por qué es importante?** Las cosas pueden hacerse bien o pueden volverse a hacer. Si un equipo de software pone el énfasis en la calidad en todas las actividades de la ingeniería de software, se reduce la cantidad de repeticiones que debe hacer. Eso da como resultado costos más bajos y, lo que es más importante, un mejor tiempo para llegar al mercado.

**¿Cuáles son las etapas?** Antes de iniciar las actividades de aseguramiento de la calidad del software (ACS), es importante definir la *calidad del software* en varios niveles diferentes de abstracción. Una vez que se entiende lo que es la calidad, el equipo de software debe identificar un conjunto de actividades de ACS que filtren los errores de los productos del trabajo antes de que se aprueben.

**¿Cuál es el producto final?** Se crea un Plan de Aseguramiento de la Calidad del Software para definir una estrategia de ACS del equipo. Durante la modelación y codificación, el producto principal del ACS es la salida de las revisiones técnicas (véase el capítulo 15). Durante las pruebas (capítulos 17 a 20), se generan los planes y procedimientos de prueba, así como otros productos del trabajo asociados con el proceso de mejora.

**¿Cómo me aseguro de que lo hice bien?** Hay que encontrar los errores antes de que se vuelvan defectos... Es decir, debe trabajarse para mejorar la eficiencia en la eliminación de defectos (capítulo 23), a fin de reducir la cantidad de repeticiones que tenga que hacer el equipo del software.

1 Si no ha leído el capítulo 14, debe leerlo ahora.



www.FreeLibros.me

*de la calidad*) es una actividad sombrilla (véase el capítulo 2) que se aplica en todo el proceso del software.

El aseguramiento de la calidad del software (ACS) incluye lo siguiente: 1) un proceso de ACS, 2) tareas específicas de aseguramiento y control de la calidad (incluidas revisiones técnicas y una estrategia de pruebas relacionadas entre sí), 3) prácticas eficaces de ingeniería de software (métodos y herramientas), 4) control de todos los productos del trabajo de software y de los cambios que sufren (véase el capítulo 22), 5) un procedimiento para garantizar el cumplimiento de los estándares del desarrollo de software (cuando sea aplicable) y 6) mecanismos de medición y reporte.

Este capítulo se centra en aspectos de la administración y en las actividades específicas del proceso que permiten a una organización de software garantizar que hace "las cosas correctas en el momento correcto y de la forma correcta".

## 16.1 Antecedentes

El control y aseguramiento de la calidad son actividades esenciales para cualquier negocio que genere productos que utilicen otras personas. Antes del siglo XX, el control de calidad era responsabilidad única del artesano que elaboraba el producto. Cuando pasó el tiempo y las técnicas de la producción en masa se hicieron comunes, el control de calidad se convirtió en una actividad ejecutada por personas diferentes de aquellas que elaboraban el producto.

**Cita:**

"Cometes demasiados errores equivocados."

**Yogi Berra**

La primera función formal de aseguramiento y control de la calidad se introdujo en los laboratorios Bell en 1916 y se difundió con rapidez al resto del mundo de la manufactura. Durante la década de 1940, sugirieron enfoques más formales del control de calidad. Éstos se basaban en la medición y en el proceso de la mejora continua [Dem86] como elementos clave de la administración de la calidad.

Actualmente, toda compañía tiene mecanismos para asegurar la calidad en sus productos. En realidad, en las últimas décadas, las afirmaciones explícitas del compromiso de una compañía con la calidad se han vuelto un mantra de la mercadotecnia.

La historia del aseguramiento de la calidad en el desarrollo del software corre de manera paralela con la historia de la calidad en la manufactura del hardware. En los primeros días de la computación (décadas de 1950 y 1960), la calidad era responsabilidad única del programador. Los estándares para asegurar la calidad del software se introdujeron en los contratos para desarrollar software militar en la década de 1970 y se extendieron con rapidez al desarrollo de software en el mundo comercial [IEE93]. Si se amplía la definición presentada al principio, el aseguramiento de la calidad del software es un "patrón planeado y sistemático de acciones" [Sch98c] que se requieren para garantizar alta calidad en el software. El alcance de la responsabilidad del aseguramiento de la calidad se caracteriza mejor si se parafrasea un comercial de un automóvil popular: "La calidad es el empleo número 1." La implicación para el software es que muchas entidades diferentes tienen responsabilidad en el aseguramiento de la calidad del software: ingenieros de software, gerentes de proyecto, clientes, vendedores y los individuos que trabajan en el grupo de ACS.

El grupo de ACS funciona como representante del cliente en el interior de la empresa. Es decir, la gente que realiza el ACS debe ver al software desde el punto de vista del cliente. ¿El software cumple adecuadamente los factores de calidad mencionados en el capítulo 14? ¿El desarrollo del software se condujo de acuerdo con estándares preestablecidos? ¿Las disciplinas técnicas han cumplido con sus roles como parte de la actividad de ACS? El grupo de ACS trata de responder éstas y otras preguntas para garantizar que se mantenga la calidad del software.

www.FreeLibros.me

## 16.2 ELEMENTOS DE ASEGURAMIENTO DE LA CALIDAD DEL SOFTWARE

**WebRef**

En la dirección **www.swqual.com/newsletter/vol2/no1/vol2no1.html**, se encuentra un análisis profundo del ACS, que incluye una amplia variedad de definiciones.

El aseguramiento de la calidad del software incluye un rango amplio de preocupaciones y actividades que se centran en la administración de la calidad del software. Éstas se resumen como sigue [Hor03]:

**Estándares.** El IEEE, ISO y otras organizaciones que establecen estándares han producido una amplia variedad de ellos para ingeniería de software y documentos relacionados. Los estándares los adopta de manera voluntaria una organización de software o los impone el cliente u otros participantes. El trabajo del ACS es asegurar que los estándares que se hayan adoptado se sigan, y que todos los productos del trabajo se apeguen a ellos.

**Revisiones y auditorías.** Las revisiones técnicas son una actividad del control de calidad que realizan ingenieros de software para otros ingenieros de software (véase el capítulo 15). Su objetivo es detectar errores. Las auditorías son un tipo de revisión efectuada por personal de ACS con objeto de garantizar que se sigan los lineamientos de calidad en el trabajo de la ingeniería de software. Por ejemplo, una auditoría del proceso de revisión se efectúa para asegurar que las revisiones se lleven a cabo de manera que tengan la máxima probabilidad de descubrir errores.

**Pruebas.** Las pruebas del software (capítulos 17 a 20) son una función del control de calidad que tiene un objetivo principal: detectar errores. El trabajo del ACS es garantizar que las pruebas se planeen en forma apropiada y que se realicen con eficiencia, de modo que la probabilidad de que logren su objetivo principal sea máxima.

**Colección y análisis de los errores.** La única manera de mejorar es medir cómo se está haciendo algo. El ACS reúne y analiza errores y datos acerca de los defectos para entender mejor cómo se cometen los errores y qué actividades de la ingeniería de software son más apropiadas para eliminarlos.

**Administración del cambio.** El cambio es uno de los aspectos que más irrumpe en cualquier proyecto de software. Si no se administra en forma adecuada, lleva a la confusión y ésta casi siempre genera mala calidad. El ACS asegura que se hayan instituido prácticas adecuadas de administración del cambio (véase el capítulo 22).

**Educación.** Toda organización de software quiere mejorar sus prácticas de ingeniería de software. Un contribuyente clave de la mejora es la educación de los ingenieros de software, de sus gerentes y de otros participantes. La organización de ACS lleva el liderazgo en la mejora del proceso de software (capítulo 30) y es clave para proponer y patrocinar programas educativos.

**Cita:**

"La excelencia es la capacidad ilimitada de mejorar la calidad de lo que se tenga para ofrecer."

**Rick Petin**

**Administración de los proveedores.** Son tres las categorías de software que se adquieren a proveedores externos: *paquetes contenidos en una caja* (por ejemplo, *Office*, de Microsoft); *un shell personalizado* [Hor03], que da una estructura básica, tipo esqueleto, que se adapta de manera única a las necesidades del comprador; y *software contratado*, que se diseña y construye especialmente a partir de especificaciones provistas por la organización cliente. El trabajo de la organización de ACS es garantizar que se obtenga software de alta calidad a partir de las sugerencias de prácticas específicas de calidad que el proveedor debe seguir (cuando sea posible) y de la incorporación de cláusulas de calidad como parte de cualquier contrato con un proveedor externo.

**Administración de la seguridad.** Con el aumento de los delitos cibernéticos y de las nuevas regulaciones gubernamentales respecto de la privacidad, toda organización de software debe instituir políticas para proteger los datos en todos los niveles, establecer cortafuegos de protección para las *webapps* y asegurar que el software no va a ser vulnerado in-

www.FreeLibros.me

ternamente. El ACS garantiza que para lograr la seguridad del software, se utilicen el proceso y la tecnología apropiados.

**Seguridad.** Debido a que el software casi siempre es un componente crucial de los sistemas humanos (como aplicaciones automotrices o aeronáuticas), la consecuencia de defectos ocultos puede ser catastrófica. El ACS es responsable de evaluar el efecto de las fallas del software y de dar los pasos que se requieren para disminuir el riesgo.

**Administración de riesgos.** Aunque el análisis y la mitigación de riesgos (véase el capítulo 28) es asunto de los ingenieros de software, la organización del ACS garantiza que las actividades de administración de riesgos se efectúen en forma apropiada y que se establezcan planes de contingencia relacionados con los riesgos.

Además de cada una de estas preocupaciones y actividades, el ACS tiene como preocupación dominante asegurar que las actividades de apoyo del software (como mantenimiento, líneas de ayuda, documentación y manuales) se lleven a cabo o se produzcan con calidad.

INFORMACIÓN

### *Recursos para la administración de la calidad*

En la red mundial existen decenas de recursos para la administración de la calidad, incluidas sociedades profesionales, organizaciones emisoras de estándares y fuentes de información general. Los sitios siguientes constituyen un buen punto de partida:

American Society for Quality (ASQ), División de Software, **www.asq.org/software**
Association for Computer Machinery, **www.acm.org**, Centro de Datos y Análisis del Software (DACS), **www.dacs.dtic.mil/**
International Organization for Standardization (ISO), **www.iso.ch**
ISO SPIECE, **www.isospiece.com**
Malcolm Baldridge National Quality Award, **www.quality.nist.gov**
Software Engineering Institute, **www.sei.cmu.edu/**
Software Testing and Quality Engineering, **www.stickyminds.com**
Six Sigma Resources, **www.isixsigma.com/** **www.asq.org/sixsigma/**
TickIT International, temas sobre certificación de la calidad, **www.tickit.org/international.htm**
Total Quality Management (TQM)
- Información general: **www.gslis.utexas.edu/~rpollock/tqm.html**
- Artículos: **www.work911.com/tqmarticles.htm**
- Glosario: **www.quality.org/TQM-MSI/TQM-glossary.html**

## 16.3 TAREAS, METAS Y MÉTRICAS DEL ACS

El aseguramiento de la calidad del software se compone de varias tareas asociadas con dos entidades diferentes: los ingenieros de software que hacen el trabajo técnico y un grupo de ACS que tiene la responsabilidad de planear, supervisar, registrar, analizar y hacer reportes acerca de la calidad.

Los ingenieros de software abordan la calidad (y ejecutan actividades para controlarla), aplicando métodos y medidas técnicas sólidos, realizando revisiones técnicas y haciendo pruebas de software bien planeadas.

### 16.3.1 Tareas del ACS

El objetivo del grupo de ACS es auxiliar al equipo del software para lograr un producto final de alta calidad. El Instituto de Ingeniería de Software recomienda un conjunto de acciones de ACS que se dirigen a la planeación, supervisión, registro, análisis y elaboración de reportes para el aseguramiento de la calidad. Estas acciones son realizadas (o facilitadas) por un grupo independiente de ACS que hace lo siguiente:

www.FreeLibros.me

**¿Cuál es el rol de un grupo de ACS?**

**Prepara el plan de ACS para un proyecto.** El plan se desarrolla como parte de la preparación del proyecto y es revisado por todos los participantes. Las acciones de aseguramiento de la calidad efectuadas por el equipo de ingeniería de software y por el grupo de ACS son dirigidas por el plan. Éste identifica las evaluaciones que se van a realizar, las auditorías y revisiones por efectuar, los estándares aplicables al proyecto, los procedimientos para reportar y dar seguimiento a los errores, los productos del trabajo que genera el grupo de ACS y la retroalimentación que se dará al equipo del software.

**Participa en el desarrollo de la descripción del software del proyecto.** El equipo de software selecciona un proceso para el trabajo que se va a realizar. El grupo de ACS revisa la descripción del proceso a fin de cumplir con la política organizacional, los estándares internos para el software, los estándares impuestos desde el exterior (como la norma ISO-9001) y otras partes del plan del proyecto de software.

**Revisa las actividades de la ingeniería de software a fin de verificar el cumplimiento mediante el proceso definido para el software.** El grupo de ACS identifica, documenta y da seguimiento a las desviaciones del proceso y verifica que se hayan hecho las correcciones pertinentes.

**Audita los productos del trabajo de software designados para verificar que se cumpla con aquellos definidos como parte del proceso de software.** El grupo de ACS revisa productos del trabajo seleccionados; identifica, documenta y da seguimiento a las desviaciones; verifica que se hayan hecho las correcciones necesarias y reporta periódicamente los resultados de su trabajo al gerente del proyecto.

**Asegura que las desviaciones en el trabajo de software y sus productos se documenten y manejen de acuerdo con un procedimiento documentado.** Las desviaciones pueden encontrarse en el plan del proyecto, la descripción del proceso, los estándares aplicables o los productos del trabajo de la ingeniería de software.

**Registra toda falta de cumplimiento y la reporta a la alta dirección.** Se da seguimiento a los incumplimientos hasta que son resueltos.

Además de estas acciones, el grupo de ACS coordina el control y administración del cambio (véase el capítulo 22) y ayuda a recabar y analizar métricas para el software.

### 16.3.2 Metas, atributos y métricas

**Cita:**

"La calidad nunca es un accidente; siempre es el resultado de una intención clara, un esfuerzo sincero, una dirección inteligente y una ejecución hábil; representa la elección sabia de muchas alternativas".

**William A. Foster**

Las acciones de ACS descritas en la sección anterior se realizan con objeto de alcanzar un conjunto de metas pragmáticas:

**Calidad de los requerimientos.** La corrección, completitud y consistencia del modelo de requerimientos tendrá una gran influencia en la calidad de todos los productos del trabajo que sigan. El ACS debe garantizar que el equipo de software ha revisado en forma apropiada el modelo de requerimientos a fin de alcanzar un alto nivel de calidad.

**Calidad del diseño.** Todo elemento del modelo del diseño debe ser evaluado por el equipo del software para asegurar que tenga alta calidad y que el diseño en sí se apegue a los requerimientos. El ACS busca atributos del diseño que sean indicadores de la calidad.

**Calidad del código.** El código fuente y los productos del trabajo relacionados (por ejemplo, otra información descriptiva) deben apegarse a los estándares locales de codificación y tener características que faciliten darle mantenimiento. El ACS debe identificar aquellos atributos que permitan hacer un análisis razonable de la calidad del código.

**Eficacia del control de calidad.** Un equipo de software debe aplicar recursos limitados, en forma tal que tenga la máxima probabilidad de lograr un resultado de alta calidad. El

www.FreeLibros.me

**FIGURA 16.1** **Metas atributos y métricas de la calidad del software**
*Fuente:* Adaptado de [Hya96].

| Meta | Atributo | Métrica |
|---|---|---|
| **Calidad de los requerimientos** | Ambigüedad | Número de modificadores ambiguos (por ejemplo, muchos, grande, amigable, etc.) |
| | Completitud | Número de TBA y TBD |
| | Comprensibilidad | Número de secciones y subsecciones |
| | Volatilidad | Número de cambios por requerimiento |
| | | Tiempo (por actividad) cuando se solicita un cambio |
| | Trazabilidad | Número de requerimientos no trazables hasta el diseño o código |
| | Claridad del modelo | Número de modelos UML |
| | | Número de páginas descriptivas por modelo |
| | | Número de errores de UML |
| **Calidad del diseño** | Integridad arquitectónica | Existencia del modelo arquitectónico |
| | Completitud de componentes | Número de componentes que se siguen hasta el modelo arquitectónico |
| | | Complejidad del diseño del procedimiento |
| | Complejidad de la interfaz | Número promedio de pasos para llegar a una función o contenido normal |
| | | Distribución apropiada |
| | Patrones | Número de patrones utilizados |
| **Calidad del código** | Complejidad | Complejidad ciclomática |
| | Facilidad de mantenimiento | Factores de diseño (capítulo 8) |
| | Comprensibilidad | Porcentaje de comentarios internos |
| | | Convenciones variables de nomenclatura |
| | Reusabilidad | Porcentaje de componentes reutilizados |
| | Documentación | Índice de legibilidad |
| **Eficacia del control de calidad** | Asignación de recursos | Porcentaje de personal por hora y por actividad |
| | Tasa de finalización | Tiempo de terminación real *versus* lo planeado |
| | Eficacia de la revisión | Ver medición de la revisión (capítulo 14) |
| | Eficacia de las pruebas | Número de errores de importancia crítica encontrados |
| | | Esfuerzo requerido para corregir un error |
| | | Origen del error |

ACS analiza la asignación de recursos para las revisiones y pruebas a fin de evaluar si se asignan en la forma más eficaz.

La figura 16.1 (adaptada de [Hya96]) identifica los atributos que son indicadores de la existencia de la calidad para cada una de las metas mencionadas. También se presentan las métricas que se utilizan para indicar la fortaleza relativa de un atributo.

## 16.4 ENFOQUES FORMALES AL ACS

En las secciones anteriores, se dijo que la calidad del software es el trabajo de cada quien y que puede lograrse por medio de una práctica competente de la ingeniería de software, así como de la aplicación de revisiones técnicas, de una estrategia de pruebas con relaciones múltiples, de un mejor control de los productos del trabajo de software y de los cambios efectuados sobre

www.FreeLibros.me

**WebRef**

En la dirección **www.gslis.utexas.edu/~rpollock/tqm.html**, se encuentra información útil acerca del ACS y de los métodos formales de la calidad.

ellos, así como de la aplicación de estándares aceptados de la ingeniería de software. Además, la calidad se define en términos de una amplia variedad de atributos de la calidad y se mide (indirectamente) con el empleo de varios índices y métricas.

En las últimas tres décadas, un segmento pequeño pero sonoro de la comunidad de la ingeniería de software ha afirmado que se requiere un enfoque más formal para el ACS. Puede decirse que un programa de cómputo es un objeto matemático. Para cada lenguaje de programación, es posible definir una sintaxis y semántica rigurosas, y se dispone de un enfoque igualmente riguroso para la especificación de los requerimientos del software (véase el capítulo 21). Si el modelo de los requerimientos (especificación) y el lenguaje de programación se representan en forma rigurosa, debe ser posible usar una demostración matemática para la corrección, de modo que se confirme que un programa se ajusta exactamente a sus especificaciones.

Los intentos de demostrar la corrección de un programa no son nuevos. Dijkstra [Dij76a] y Linger, Mills y Witt [Lin79], entre otros, han invocado pruebas de la corrección de programas y las han relacionado con el uso de conceptos de programación estructurada (véase el capítulo 10).

## 16.5 Aseguramiento estadístico de la calidad del software

El aseguramiento estadístico de la calidad del software refleja una tendencia creciente en la industria para que se vuelva más cuantitativo respecto de la calidad. Para el software, el aseguramiento estadístico de la calidad implica los pasos siguientes:

**? ¿Qué pasos se requieren para efectuar el ACS estadístico?**

1. Se recaba y clasifica la información acerca de errores y defectos del software.
2. Se hace un intento por rastrear cada error y defecto hasta sus primeras causas (por ejemplo, no conformidad con las especificaciones, error de diseño, violación de los estándares, mala comunicación con el cliente, etc.).
3. Con el uso del Principio de Pareto (80 por ciento de los defectos se debe a 20 por ciento de todas las causas posibles), se identifica 20 por ciento de las causas de errores y defectos (las *pocas vitales*).
4. Una vez identificadas las pocas causas vitales, se corrigen los problemas que han dado origen a los errores y defectos.

Este concepto relativamente simple representa un paso importante hacia la creación de un proceso adaptativo del software en el que se hacen cambios para mejorar aquellos elementos del proceso que introducen errores.

### 16.5.1 Ejemplo general

**Cita:**

"Un análisis estadístico, si se realiza en forma apropiada, es una disección delicada de las incertidumbres, una cirugía de las suposiciones."

**M. J. Moroney**

A fin de ilustrar el uso de los métodos estadísticos para el trabajo de ingeniería de software, suponga que una organización de ingeniería de software recaba información sobre los errores y defectos cometidos en un periodo de un año. Algunos de dichos errores se descubren a medida que se desarrolla el software. Otros (defectos) se encuentran después de haber liberado el software a sus usuarios finales. Aunque se descubren cientos de problemas diferentes, todos pueden rastrearse hasta una (o más) de las causas siguientes:

- Especificaciones erróneas o incompletas (EEI)
- Mala interpretación de la comunicación con el cliente (MCC)
- Desviación intencional de las especificaciones (DIE)
- Violación de los estándares de programación (VEP)
- Error en la representación de los datos (ERD)

www.FreeLibros.me

**FIGURA 16.2**
**Colección de datos para hacer ACS estadístico**

| | Total | | Serio | | Moderado | | Menor | |
|---|---|---|---|---|---|---|---|---|
| **Error** | **No.** | **%** | **No.** | **%** | **No.** | **%** | **No.** | **%** |
| IES | 205 | 22% | 34 | 27% | 68 | 18% | 103 | 24% |
| MCC | 156 | 17% | 12 | 9% | 68 | 18% | 76 | 17% |
| IDS | 48 | 5% | 1 | 1% | 24 | 6% | 23 | 5% |
| VPS | 25 | 3% | 0 | 0% | 15 | 4% | 10 | 2% |
| EDR | 130 | 14% | 26 | 20% | 68 | 18% | 36 | 8% |
| ICI | 58 | 6% | 9 | 7% | 18 | 5% | 31 | 7% |
| EDL | 45 | 5% | 14 | 11% | 12 | 3% | 19 | 4% |
| IET | 95 | 10% | 12 | 9% | 35 | 9% | 48 | 11% |
| IID | 36 | 4% | 2 | 2% | 20 | 5% | 14 | 3% |
| PLT | 60 | 6% | 15 | 12% | 19 | 5% | 26 | 6% |
| HCI | 28 | 3% | 3 | 2% | 17 | 4% | 8 | 2% |
| MIS | 56 | 6% | 0 | 0% | 15 | 4% | 41 | 9% |
| Totales | 942 | 100% | 128 | 100% | 379 | 100% | 435 | 100% |

- Interfaz componente inconsistente (ICI)
- Error en el diseño lógico (EDL)
- Pruebas incompletas o erróneas (PIE)
- Documentación inexacta o incompleta (DII)
- Error en la traducción del lenguaje de programación del diseño (LPD)
- Interfaz humano/computadora ambigua o inconsistente (IHC)
- Varios (V)

**Cita:**

"20 por ciento del código tiene 80 por cierto de los errores. Encuéntrelos, corríjalos".

**Lowell Arthur**

Para aplicar el ACS estadístico, se elabora la tabla de la figura 16.2. La tabla indica que EEI, MCC y ERD son las pocas causas vitales que originan 53 por ciento de todos los errores. Sin embargo, debe notarse que EEI, ERD, LPD y EDL se habrían seleccionado como las pocas causas vitales si se consideran sólo errores serios. Una vez que las pocas causas vitales han sido determinadas, la organización de ingeniería de software comienza su acción correctiva. Por ejemplo, a fin de corregir el MCC, deben implementarse técnicas para recabar requerimientos (capítulo 5) que mejoren la calidad de la comunicación y las especificaciones con el cliente. Para mejorar el ERD, deben adquirirse herramientas para desarrollar la modelación de casos y realizar datos y revisiones del diseño más significativos.

Es importante notar que la acción correctiva se centra sobre todo en las pocas causas vitales. En tanto éstas se corrigen, nuevas candidatas se van a la cumbre de la pila.

Las técnicas para el aseguramiento correctivo han sido propuestas para dar una mejora sustancial de la calidad [Art97]. En ciertos casos, las organizaciones de software han tenido una reducción anual de 50 por ciento en defectos después de aplicar esta técnica.

La aplicación del ACS estadístico y el Principio de Pareto se resumen en una sola oración: *Pasa tu tiempo viendo las cosas que realmente importan, pero primero asegúrate de que entiendes lo que realmente importa...*

### 16.5.2 Seis Sigma para la ingeniería de software

*Seis Sigma* es la estrategia más ampliamente usada hoy para el aseguramiento estadístico de la calidad en la industria. La estrategia Seis Sigma fue popularizada originalmente por Motorola en la década de 1980 y "es una metodología rigurosa y disciplinada que usa datos y análisis estadísticos para medir y mejorar el desempeño operativo de una compañía, identificando y eliminando defectos en procesos de manufactura y servicios" [ISI08]. El término Seis Sigma se deriva

www.FreeLibros.me

de seis desviaciones estándar —3.4 casos (defectos) por millón de ocurrencias—, lo que implica un estándar de calidad extremadamente alto. La metodología Seis Sigma define tres etapas fundamentales:

**¿Cuáles son las etapas fundamentales de la metodología Seis Sigma?**

- *Definir* los requerimientos del cliente y los que se le entregan, así como las metas del proyecto a través de métodos bien definidos de comunicación con el cliente.
- *Medir* el proceso existente y su resultado para determinar el desempeño actual de la calidad (recabar métricas para los defectos).
- *Analizar* las métricas de los defectos y determinar las pocas causas vitales.

Si se trata de un proceso de software existente que se requiere mejorar, Seis Sigma sugiere dos etapas adicionales:

- *Mejorar* el proceso, eliminando las causas originales de los defectos.
- *Controlar* el proceso para asegurar que el trabajo futuro no vuelva a introducir las causas de los defectos.

Estas etapas fundamentales y adicionales en ocasiones son conocidas como método DMAMC (definir, medir, analizar, mejorar y controlar).

Si una organización va a desarrollar un proceso de software (en vez de mejorar uno existente), a las etapas fundamentales se agregan las siguientes:

- *Diseñar* el proceso para 1) evitar las causas originales de los defectos y 2) cumplir los requerimientos del cliente.
- *Verificar* que el modelo del proceso en realidad evite los defectos y cumpla los requerimientos del cliente.

Esta variación en ocasiones es denominada método DMADV (definir, medir, analizar, diseñar y verificar).

El estudio detallado de Seis Sigma se deja a fuentes dedicadas a ese tema. Si el lector tiene interés al respecto, consulte [ISI08], [Pyz303] y [Sne03].

## 16.6 Confiabilidad del software

**Cita:**

"El precio inevitable de la confiabilidad es la simplicidad."

**C. A. R. Hoare**

No hay duda de que la confiabilidad de un programa de cómputo es un elemento importante de su calidad general. Si un programa falla repetida y frecuentemente en su desempeño, importa poco si otros factores de la calidad del software son aceptables.

La confiabilidad del software, a diferencia de muchos otros factores de la calidad, se mide y estima directamente mediante el uso de datos históricos del desarrollo. La *confiabilidad del software* se define en términos estadísticos como "la probabilidad que tiene un programa de cómputo de operar sin fallas en un ambiente específico por un tiempo específico" [Mus87]. Para ilustrar lo anterior, digamos que se estima que el programa *X* tiene una confiabilidad de 0.999 durante ocho horas de procesamiento continuo. En otras palabras, si el programa *X* fuera a ejecutarse 1 000 veces y requiriera un total de ocho horas de tiempo de procesamiento continuo (tiempo de procesamiento), es probable que operara correctamente (sin fallas) 999 veces.

Siempre que se trate de la confiabilidad del software, surge una pregunta crucial: ¿qué significa el término *falla*? En el contexto de cualquier análisis de la calidad y confiabilidad del software, la falla significa la falta de conformidad con los requerimientos del software. Pero, incluso con esta definición, hay gradaciones. Las fallas pueden ser leves o catastróficas. Una falla podría corregirse en segundos, mientras que otra tal vez requiera de varias semanas o meses de trabajo para ser corregida. Para complicar más el asunto, la corrección de una falla quizá dé como resultado la introducción de otros errores que a su vez originen otras fallas.

www.FreeLibros.me

### 16.6.1 Mediciones de la confiabilidad y disponibilidad

PUNTO CLAVE

Los problemas de confiabilidad del software casi siempre pueden seguirse hasta encontrar defectos en el diseño o en la implementación.

Los primeros trabajos sobre confiabilidad del software trataban de extrapolar la teoría matemática de la confiabilidad del hardware a la predicción de la confiabilidad del software. La mayor parte de modelos relacionados con el hardware se abocan a la falla debida al uso, en lugar de a la que tiene su origen en los defectos de diseño. En el hardware, las fallas debidas al uso físico (por ejemplo, los efectos de temperatura, corrosión y golpes) son más probables que las debidas al diseño. Desafortunadamente, con el software ocurre lo contrario. En realidad, todas las fallas del software pueden rastrearse en problemas de diseño o de implementación; el uso (véase el capítulo 1) no entra en el escenario.

Ha habido un debate permanente acerca de la relación que existe entre los conceptos clave en la confiabilidad del hardware y su aplicabilidad al software. Aunque es posible establecer un vínculo irrefutable, es útil considerar algunos conceptos sencillos que se aplican a ambos elementos del sistema.

PUNTO CLAVE

Es importante observar que el tiempo medio entre fallas y otras medidas relacionadas se basa en tiempo del CPU, no en tiempo de reloj.

Si se considera un sistema basado en computadora, una medida sencilla de su confiabilidad es el *tiempo medio entre fallas* (TMEF):

$$\text{TMEF} = \text{TMPF} + \text{TMPR}$$

donde las siglas TMPF y TMPR significan *tiempo medio para la falla* y *tiempo medio para la reparación*,[2] respectivamente.

Muchos investigadores afirman que el TMEF es una medición más útil que otras relacionadas con la calidad del software que se estudian en el capítulo 23. En pocas palabras, a un usuario final le preocupan las fallas, no la cuenta total de defectos. Como cada defecto contenido en un programa no tiene la misma tasa de fallas, la cuenta total de defectos indica muy poco acerca de la confiabilidad del sistema. Por ejemplo, considere un programa que haya estado en operación durante 3 000 horas de procesador sin falla. Muchos defectos de este programa estarían sin detectar durante decenas de miles de horas antes de ser descubiertos. El TMEF de tales errores oscuros podría ser de 30 000 o hasta 60 000 horas de procesador. Otros defectos, no descubiertos, podrían tener una tasa de fallas de 4 000 a 5 000 horas. Aun si cada uno de los errores en esta categoría (los que tienen un TMEF largo) se eliminara, el efecto que tendrían sobre el software sería despreciable.

CONSEJO

*Algunos aspectos de la disponibilidad (que no se estudian aquí) no tienen que ver con las fallas. Por ejemplo, la programación del tiempo fuera de operación (para funciones de apoyo) hace que el software no esté disponible.*

Sin embargo, el TMEF puede ser problemático por dos razones: 1) proyecta un tiempo entre fallas, pero no da una tasa de fallas proyectada y 2) puede interpretarse mal, como la vida promedio, cuando *no* es esto lo que implica.

Una medición alternativa de confiabilidad es la de las *fallas en el tiempo* (FET): medición estadística de cuántas fallas tendrá un componente en mil millones de horas de operación. Por tanto, 1 FET es equivalente a una falla en cada mil millones de horas de operación.

Además de una medida de la confiabilidad, también debe desarrollarse otra para la disponibilidad. La *disponibilidad del software* es la probabilidad de que un programa opere de acuerdo con los requerimientos en un momento determinado de tiempo, y se define así:

$$\text{Disponibilidad} = \frac{\text{TMPF}}{\text{TMPF} + \text{TMPR}} \times 100\%$$

La medición del TMEF para la confiabilidad es igualmente sensible al TMPF y al TMPR. La medición de la disponibilidad es un poco más sensible al TMPR, que es una medición indirecta de la facilidad que tiene el software para recibir mantenimiento.

2 Aunque tal vez se requiera depurar (y hacer otras correcciones relacionadas) como consecuencia de la falla, en muchos casos el software funcionará de manera apropiada después de reiniciar, sin ningún otro cambio.

www.FreeLibros.me