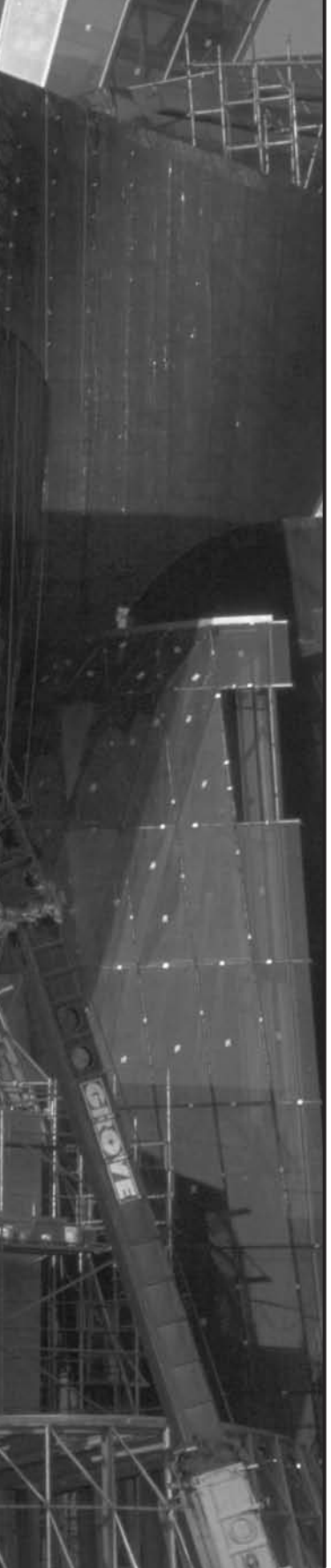

6

Diseño arquitectónico

Objetivos

El objetivo de este capítulo es introducir los conceptos de arquitectura de software y diseño arquitectónico. Al estudiar este capítulo:

- comprenderá por qué es importante el diseño arquitectónico del software;
- conocerá las decisiones que deben tomarse sobre la arquitectura de software durante el proceso de diseño arquitectónico;
- asimilará la idea de los patrones arquitectónicos, formas bien reconocidas de organización de las arquitecturas del sistema, que pueden reutilizarse en los diseños del sistema;
- identificará los patrones arquitectónicos usados frecuentemente en diferentes tipos de sistemas de aplicación, incluidos los sistemas de procesamiento de transacción y los sistemas de procesamiento de lenguaje.

Contenido

El diseño arquitectónico se interesa por entender cómo debe organizarse un sistema y cómo tiene que diseñarse la estructura global de ese sistema. En el modelo del proceso de desarrollo de software, como se mostró en el capítulo 2, el diseño arquitectónico es la primera etapa en el proceso de diseño del software. Es el enlace crucial entre el diseño y la ingeniería de requerimientos, ya que identifica los principales componentes estructurales en un sistema y la relación entre ellos. La salida del proceso de diseño arquitectónico consiste en un modelo arquitectónico que describe la forma en que se organiza el sistema como un conjunto de componentes en comunicación.

En los procesos ágiles, por lo general se acepta que una de las primeras etapas en el proceso de desarrollo debe preocuparse por establecer una arquitectura global del sistema. Usualmente no resulta exitoso el desarrollo incremental de arquitecturas. Mientras que la refactorización de componentes en respuesta a los cambios suele ser relativamente fácil, tal vez resulte costoso refactorizar una arquitectura de sistema.

Para ayudar a comprender lo que se entiende por arquitectura del sistema, tome en cuenta la figura 6.1. En ella se presenta un modelo abstracto de la arquitectura para un sistema de robot de empaquetado, que indica los componentes que tienen que desarrollarse. Este sistema robótico empaca diferentes clases de objetos. Usa un componente de visión para recoger los objetos de una banda transportadora, identifica la clase de objeto y selecciona el tipo correcto de empaque. Luego, el sistema mueve los objetos que va a empacar de la banda transportadora de entrega y coloca los objetos empacados en otro transportador. El modelo arquitectónico presenta dichos componentes y los vínculos entre ellos.

En la práctica, hay un significativo traslape entre los procesos de ingeniería de requerimientos y el diseño arquitectónico. De manera ideal, una especificación de sistema no debe incluir cierta información de diseño. Esto no es realista, excepto para sistemas muy pequeños. La descomposición arquitectónica es por lo general necesaria para estructurar y organizar la especificación. Por lo tanto, como parte del proceso de ingeniería de requerimientos, usted podría proponer una arquitectura de sistema abstracta donde se asocien grupos de funciones de sistemas o características con componentes o subsistemas a gran escala. Luego, puede usar esta descomposición para discutir con los participantes sobre los requerimientos y las características del sistema.

Las arquitecturas de software se diseñan en dos niveles de abstracción, que en este texto se llaman *arquitectura en pequeño* y *arquitectura en grande*:

1. La arquitectura en pequeño se interesa por la arquitectura de programas individuales. En este nivel, uno se preocupa por la forma en que el programa individual se separa en componentes. Este capítulo se centra principalmente en arquitecturas de programa.
2. La arquitectura en grande se interesa por la arquitectura de sistemas empresariales complejos que incluyen otros sistemas, programas y componentes de programa. Tales sistemas empresariales se distribuyen a través de diferentes computadoras, que diferentes compañías administran y poseen. En los capítulos 18 y 19 se cubren las arquitecturas grandes; en ellos se estudiarán las arquitecturas de los sistemas distribuidos.

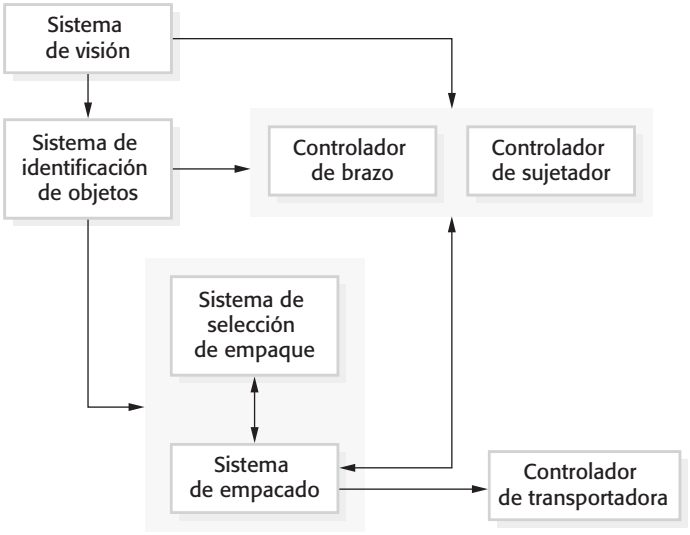

Figura 6.1 Arquitectura de un sistema de control para un robot empacador

La arquitectura de software es importante porque afecta el desempeño y la potencia, así como la capacidad de distribución y mantenimiento de un sistema (Bosch, 2000). Como afirma Bosch, los componentes individuales implementan los requerimientos funcionales del sistema. Los requerimientos no funcionales dependen de la arquitectura del sistema, es decir, la forma en que dichos componentes se organizan y comunican. En muchos sistemas, los requerimientos no funcionales están también influidos por componentes individuales, pero no hay duda de que la arquitectura del sistema es la influencia dominante.

Bass y sus colaboradores (2003) analizan tres ventajas de diseñar y documentar de manera explícita la arquitectura de software:

1. *Comunicación con los participantes* La arquitectura es una presentación de alto nivel del sistema, que puede usarse como un enfoque para la discusión de un amplio número de participantes.
2. *Análisis del sistema* En una etapa temprana en el desarrollo del sistema, aclarar la arquitectura del sistema requiere cierto análisis. Las decisiones de diseño arquitectónico tienen un efecto profundo sobre si el sistema puede o no cubrir requerimientos críticos como rendimiento, fiabilidad y mantenibilidad.
3. *Reutilización a gran escala* Un modelo de una arquitectura de sistema es una descripción corta y manejable de cómo se organiza un sistema y cómo interoperan sus componentes. Por lo general, la arquitectura del sistema es la misma para sistemas con requerimientos similares y, por lo tanto, puede soportar reutilización de software a gran escala. Como se explica en el capítulo 16, es posible desarrollar arquitecturas de línea de productos donde la misma arquitectura se reutilice mediante una amplia gama de sistemas relacionados.

Hofmeister y sus colaboradores (2000) proponen que una arquitectura de software sirve en primer lugar como un plan de diseño para la negociación de requerimientos de sistema y, en segundo lugar, como un medio para establecer discusiones con clientes, desarrolladores y administradores. También sugieren que es una herramienta esencial para la administración de la complejidad. Oculta los detalles y permite a los diseñadores enfocarse en las abstracciones clave del sistema.

Las arquitecturas de sistemas se modelan con frecuencia usando diagramas de bloques simples, como en la figura 6.1. Cada recuadro en el diagrama representa un componente. Los recuadros dentro de recuadros indican que el componente se dividió en subcomponentes. Las flechas significan que los datos y/o señales de control pasan de un componente a otro en la dirección de las flechas. Hay varios ejemplos de este tipo de modelo arquitectónico en el catálogo de arquitectura de software de Booch (Booch, 2009).

Los diagramas de bloque presentan una imagen de alto nivel de la estructura del sistema e incluyen fácilmente a individuos de diferentes disciplinas que intervienen en el proceso de desarrollo del sistema. No obstante su amplio uso, Bass y sus colaboradores (2003) no están de acuerdo con los diagramas de bloque informales para describir una arquitectura. Afirman que tales diagramas informales son representaciones arquitectónicas deficientes, pues no muestran ni el tipo de relaciones entre los componentes del sistema ni las propiedades externamente visibles de los componentes.

Las aparentes contradicciones entre práctica y teoría arquitectónica surgen porque hay dos formas en que se utiliza un modelo arquitectónico de un programa:

1. *Como una forma de facilitar la discusión acerca del diseño del sistema* Una visión arquitectónica de alto nivel de un sistema es útil para la comunicación con los participantes de un sistema y la planeación del proyecto, ya que no se satura con detalles. Los participantes pueden relacionarse con él y entender una visión abstracta del sistema. En tal caso, analizan el sistema como un todo sin confundirse por los detalles. El modelo arquitectónico identifica los componentes clave que se desarrollarán, de modo que los administradores pueden asignar a individuos para planear el desarrollo de dichos sistemas.
2. *Como una forma de documentar una arquitectura que se haya diseñado* La meta aquí es producir un modelo de sistema completo que muestre los diferentes componentes en un sistema, sus interfaces y conexiones. El argumento para esto es que tal descripción arquitectónica detallada facilita la comprensión y la evolución del sistema.

Los diagramas de bloque son una forma adecuada para describir la arquitectura del sistema durante el proceso de diseño, pues son una buena manera de soportar las comunicaciones entre las personas involucradas en el proceso. En muchos proyectos, suele ser la única documentación arquitectónica que existe. Sin embargo, si la arquitectura de un sistema debe documentarse ampliamente, entonces es mejor usar una notación con semántica bien definida para la descripción arquitectónica. No obstante, tal como se estudia en la sección 6.2, algunas personas consideran que la documentación detallada ni es útil ni vale realmente la pena el costo de su desarrollo.

6.1 Decisiones en el diseño arquitectónico

El diseño arquitectónico es un proceso creativo en el cual se diseña una organización del sistema que cubrirá los requerimientos funcionales y no funcionales de éste. Puesto que se trata de un proceso creativo, las actividades dentro del proceso dependen del tipo de sistema que se va a desarrollar, los antecedentes y la experiencia del arquitecto del sistema, así como de los requerimientos específicos del sistema. Por lo tanto, es útil pensar en el diseño arquitectónico como un conjunto de decisiones a tomar en vez de una secuencia de actividades.

Durante el proceso de diseño arquitectónico, los arquitectos del sistema deben tomar algunas decisiones estructurales que afectarán profundamente el sistema y su proceso de desarrollo. Con base en su conocimiento y experiencia, deben considerar las siguientes preguntas fundamentales sobre el sistema:

1. ¿Existe alguna arquitectura de aplicación genérica que actúe como plantilla para el sistema que se está diseñando?
2. ¿Cómo se distribuirá el sistema a través de algunos núcleos o procesadores?
3. ¿Qué patrones o estilos arquitectónicos pueden usarse?
4. ¿Cuál será el enfoque fundamental usado para estructurar el sistema?
5. ¿Cómo los componentes estructurales en el sistema se separarán en subcomponentes?
6. ¿Qué estrategia se usará para controlar la operación de los componentes en el sistema?
7. ¿Cuál organización arquitectónica es mejor para entregar los requerimientos no funcionales del sistema?
8. ¿Cómo se evaluará el diseño arquitectónico?
9. ¿Cómo se documentará la arquitectura del sistema?

Aunque cada sistema de software es único, los sistemas en el mismo dominio de aplicación tienen normalmente arquitecturas similares que reflejan los conceptos fundamentales del dominio. Por ejemplo, las líneas de producto de aplicación son aplicaciones que se construyen en torno a una arquitectura central con variantes que cubren requerimientos específicos del cliente. Cuando se diseña una arquitectura de sistema, debe decidirse qué tienen en común el sistema y las clases de aplicación más amplias, con la finalidad de determinar cuánto conocimiento se puede reutilizar de dichas arquitecturas de aplicación. En la sección 6.4 se estudian las arquitecturas de aplicación genéricas y en el capítulo 16 las líneas de producto de aplicación.

Para sistemas embebidos y sistemas diseñados para computadoras personales, por lo general, hay un solo procesador y no tendrá que diseñar una arquitectura distribuida para el sistema. Sin embargo, los sistemas más grandes ahora son sistemas distribuidos donde el software de sistema se distribuye a través de muchas y diferentes computadoras. La elección de arquitectura de distribución es una decisión clave que afecta el rendimiento y

la fiabilidad del sistema. Éste es un tema importante por derecho propio y se trata por separado en el capítulo 18.

La arquitectura de un sistema de software puede basarse en un patrón o un estilo arquitectónico particular. Un patrón arquitectónico es una descripción de una organización del sistema (Garlan y Shaw, 1993), tal como una organización cliente-servidor o una arquitectura por capas. Los patrones arquitectónicos captan la esencia de una arquitectura que se usó en diferentes sistemas de software. Usted tiene que conocer tanto los patrones comunes, en que éstos se usen, como sus fortalezas y debilidades cuando se tomen decisiones sobre la arquitectura de un sistema. En la sección 6.3 se analizan algunos patrones de uso frecuente.

La noción de un estilo arquitectónico de Garlan y Shaw (estilo y patrón llegaron a significar lo mismo) cubre las preguntas 4 a 6 de la lista anterior. Es necesario elegir la estructura más adecuada, como cliente-servidor o estructura en capas, que le permita satisfacer los requerimientos del sistema. Para descomponer las unidades del sistema estructural, usted opta por la estrategia de separar los componentes en subcomponentes. Los enfoques que pueden usarse permiten la implementación de diferentes tipos de arquitectura. Finalmente, en el proceso de modelado de control, se toman decisiones sobre cómo se controla la ejecución de componentes. Se desarrolla un modelo general de las relaciones de control entre las diferentes partes del sistema.

Debido a la estrecha relación entre los requerimientos no funcionales y la arquitectura de software, el estilo y la estructura arquitectónicos particulares que se elijan para un sistema dependerán de los requerimientos de sistema no funcionales:

1. *Rendimiento* Si el rendimiento es un requerimiento crítico, la arquitectura debe diseñarse para localizar operaciones críticas dentro de un pequeño número de componentes, con todos estos componentes desplegados en la misma computadora en vez de distribuirlos por la red. Esto significaría usar algunos componentes relativamente grandes, en vez de pequeños componentes de grano fino, lo cual reduce el número de comunicaciones entre componentes. También puede considerar organizaciones del sistema en tiempo de operación que permitan a éste ser replicable y ejecutable en diferentes procesadores.

2. *Seguridad* Si la seguridad es un requerimiento crítico, será necesario usar una estructura en capas para la arquitectura, con los activos más críticos protegidos en las capas más internas, y con un alto nivel de validación de seguridad aplicado a dichas capas.

3. *Protección* Si la protección es un requerimiento crítico, la arquitectura debe diseñarse de modo que las operaciones relacionadas con la protección se ubiquen en algún componente individual o en un pequeño número de componentes. Esto reduce los costos y problemas de validación de la protección, y hace posible ofrecer sistemas de protección relacionados que, en caso de falla, desactiven con seguridad el sistema.

4. *Disponibilidad* Si la disponibilidad es un requerimiento crítico, la arquitectura tiene que diseñarse para incluir componentes redundantes de manera que sea posible sustituir y actualizar componentes sin detener el sistema. En el capítulo 13 se describen dos arquitecturas de sistema tolerantes a fallas en sistemas de alta disponibilidad.

5. *Mantenibilidad* Si la mantenibilidad es un requerimiento crítico, la arquitectura del sistema debe diseñarse usando componentes autocontenidos de grano fino que

puedan cambiarse con facilidad. Los productores de datos tienen que separarse de los consumidores y hay que evitar compartir las estructuras de datos.

Evidentemente, hay un conflicto potencial entre algunas de estas arquitecturas. Por ejemplo, usar componentes grandes mejora el rendimiento, y utilizar componentes pequeños de grano fino aumenta la mantenibilidad. Si tanto el rendimiento como la mantenibilidad son requerimientos importantes del sistema, entonces debe encontrarse algún compromiso. Esto en ocasiones se logra usando diferentes patrones o estilos arquitectónicos para distintas partes del sistema.

Evaluar un diseño arquitectónico es difícil porque la verdadera prueba de una arquitectura es qué tan bien el sistema cubre sus requerimientos funcionales y no funcionales cuando está en uso. Sin embargo, es posible hacer cierta evaluación al comparar el diseño contra arquitecturas de referencia o patrones arquitectónicos genéricos. Para ayudar con la evaluación arquitectónica, también puede usarse la descripción de Bosch (2000) de las características no funcionales de los patrones arquitectónicos.

6.2 Vistas arquitectónicas

En la introducción a este capítulo se explicó que los modelos arquitectónicos de un sistema de software sirven para enfocar la discusión sobre los requerimientos o el diseño del software. De manera alternativa, pueden emplearse para documentar un diseño, de modo que se usen como base en el diseño y la implementación más detallados, así como en la evolución futura del sistema. En esta sección se estudian dos temas que son relevantes para ambos:

1. ¿Qué vistas o perspectivas son útiles al diseñar y documentar una arquitectura del sistema?
2. ¿Qué notaciones deben usarse para describir modelos arquitectónicos?

Es imposible representar toda la información relevante sobre la arquitectura de un sistema en un solo modelo arquitectónico, ya que cada uno presenta únicamente una vista o perspectiva del sistema. Ésta puede mostrar cómo un sistema se descompone en módulos, cómo interactúan los procesos de tiempo de operación o las diferentes formas en que los componentes del sistema se distribuyen a través de una red. Todo ello es útil en diferentes momentos de manera que, para el diseño y la documentación, por lo general se necesita presentar múltiples vistas de la arquitectura de software.

Existen diferentes opiniones relativas a qué vistas se requieren. Krutchen (1995), en su bien conocido modelo de vista 4+1 de la arquitectura de software, sugiere que deben existir cuatro vistas arquitectónicas fundamentales, que se relacionan usando casos de uso o escenarios. Las vistas que él sugiere son:

1. Una vista lógica, que indique las abstracciones clave en el sistema como objetos o clases de objeto. En este tipo de vista se tienen que relacionar los requerimientos del sistema con entidades.

2. Una vista de proceso, que muestre cómo, en el tiempo de operación, el sistema está compuesto de procesos en interacción. Esta vista es útil para hacer juicios acerca de las características no funcionales del sistema, como el rendimiento y la disponibilidad.
3. Una vista de desarrollo, que muestre cómo el software está descompuesto para su desarrollo, esto es, indica la descomposición del software en elementos que se implementen mediante un solo desarrollador o equipo de desarrollo. Esta vista es útil para administradores y programadores de software.
4. Una vista física, que exponga el hardware del sistema y cómo los componentes de software se distribuyen a través de los procesadores en el sistema. Esta vista es útil para los ingenieros de sistemas que planean una implementación de sistema.

Hofmeister y sus colaboradores (2000) sugieren el uso de vistas similares, pero a éstas agregan la noción de vista conceptual. Esta última es una vista abstracta del sistema que puede ser la base para descomponer los requerimientos de alto nivel en especificaciones más detalladas, ayudar a los ingenieros a tomar decisiones sobre componentes que puedan reutilizarse, y representar una línea de producto (que se estudia en el capítulo 16) en vez de un solo sistema. La figura 6.1, que describe la arquitectura de un robot de empacado, es un ejemplo de una vista conceptual del sistema.

En la práctica, las vistas conceptuales casi siempre se desarrollan durante el proceso de diseño y se usan para apoyar la toma de decisiones arquitectónicas. Son una forma de comunicar a diferentes participantes la esencia de un sistema. Durante el proceso de diseño, también pueden desarrollarse algunas de las otras vistas, al tiempo que se discuten diferentes aspectos del sistema, aunque no haya necesidad de una descripción completa desde todas las perspectivas. Además se podrían asociar patrones arquitectónicos, estudiados en la siguiente sección, con las diferentes vistas de un sistema.

Hay diferentes opiniones respecto de si los arquitectos de software deben o no usar el UML para una descripción arquitectónica (Clements *et al*., 2002). Un estudio en 2006 (Lange *et al*., 2006) demostró que, cuando se usa el UML, se aplica principalmente en una forma holgada e informal. Los autores de dicho ensayo argumentan que esto era incorrecto. El autor no está de acuerdo con esta visión. El UML se diseñó para describir sistemas orientados a objetos y, en la etapa de diseño arquitectónico, uno quiere describir con frecuencia sistemas en un nivel superior de abstracción. Las clases de objetos están muy cerca de la implementación, como para ser útiles en la descripción arquitectónica.

Para el autor, el UML no es útil durante el proceso de diseño en sí y prefiere notaciones informales que sean más rápidas de escribir y puedan dibujarse fácilmente en un pizarrón. El UML es de más valor cuando se documenta una arquitectura a detalle o se usa un desarrollo dirigido por modelo, como se estudió en el capítulo 5.

Algunos investigadores proponen el uso de lenguajes de descripción arquitectónica (ADL, por las siglas de *Architectural Description Languages*) más especializados (Bass *et al*., 2003) para describir arquitecturas del sistema. Los elementos básicos de los ADL son componentes y conectores, e incluyen reglas y lineamientos para arquitecturas bien formadas. Sin embargo, debido a su naturaleza especializada, los expertos de dominio y aplicación tienen dificultad para entender y usar los ADL. Esto dificulta la valoración de su utilidad para la ingeniería práctica del software. Los ADL diseñados para un dominio particular (por ejemplo, sistemas automotores) pueden usarse como una base

Nombre	MVC (modelo de vista del controlador)
Descripción	Separa presentación e interacción de los datos del sistema. El sistema se estructura en tres componentes lógicos que interactúan entre sí. El componente Modelo maneja los datos del sistema y las operaciones asociadas a esos datos. El componente Vista define y gestiona cómo se presentan los datos al usuario. El componente Controlador dirige la interacción del usuario (por ejemplo, teclas oprimidas, clics del mouse, etcétera) y pasa estas interacciones a Vista y Modelo. Véase la figura 6.3.
Ejemplo	La figura 6.4 muestra la arquitectura de un sistema de aplicación basado en la Web, que se organiza con el uso del patrón MVC.
Cuándo se usa	Se usa cuando existen múltiples formas de ver e interactuar con los datos. También se utiliza al desconocerse los requerimientos futuros para la interacción y presentación.
Ventajas	Permite que los datos cambien de manera independiente de su representación y viceversa. Soporta en diferentes formas la presentación de los mismos datos, y los cambios en una representación se muestran en todos ellos.
Desventajas	Puede implicar código adicional y complejidad de código cuando el modelo de datos y las interacciones son simples.

Figura 6.2 Patrón modelo de vista del controlador (MVC)

para el desarrollo dirigido por modelo. Sin embargo, se considera que los modelos y las notaciones informales, como el UML, seguirán siendo las formas de uso más común para documentar las arquitecturas del sistema.

Los usuarios de métodos ágiles afirman que, por lo general, no se utiliza la documentación detallada del diseño. Por lo tanto, desarrollarla es un desperdicio de tiempo y dinero. El autor está en gran medida de acuerdo con esta visión y considera que, para la mayoría de los sistemas, no vale la pena desarrollar una descripción arquitectónica detallada desde estas cuatro perspectivas. Uno debe desarrollar las vistas que sean útiles para la comunicación sin preocuparse si la documentación arquitectónica está completa o no. Sin embargo, una excepción es al desarrollar sistemas críticos, cuando es necesario realizar un análisis de confiabilidad detallado del sistema. Tal vez se deba convencer a reguladores externos de que el sistema se hizo conforme a sus regulaciones y, en consecuencia, puede requerirse una documentación arquitectónica completa.

6.3 Patrones arquitectónicos

La idea de los patrones como una forma de presentar, compartir y reutilizar el conocimiento sobre los sistemas de software se usa ahora ampliamente. El origen de esto fue la publicación de un libro acerca de patrones de diseño orientados a objetos (Gamma *et al*., 1995), que incitó el desarrollo de otros tipos de patrón, como los patrones para el diseño organizacional (Coplien y Harrison, 2004), patrones de usabilidad (Usability Group, 1998), interacción (Martin y Sommerville, 2004), administración de la configuración

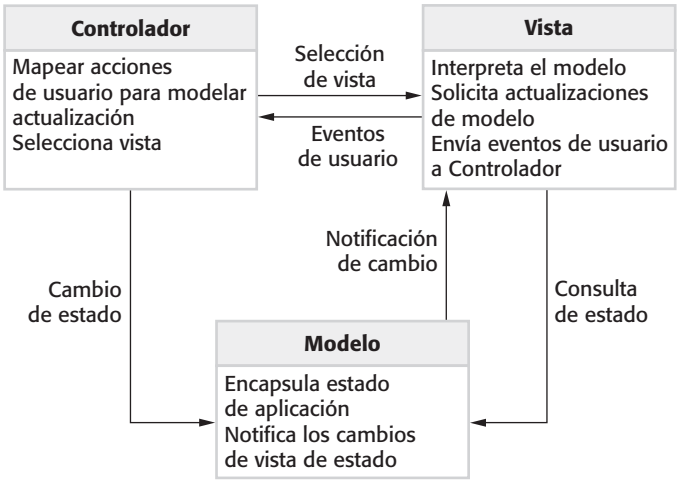

Figura 6.3 La organización del MVC

(Berczuk y Appleton, 2002), etcétera. Los patrones arquitectónicos se propusieron en la década de 1990, con el nombre de "estilos arquitectónicos" (Shaw y Garlan, 1996), en una serie de cinco volúmenes de manuales sobre arquitectura de software orientada a patrones, publicados entre 1996 y 2007 (Buschmann *et al*., 1996; Buschmann *et al*., 2007a; Buschmann *et al*., 2007b; Kircher y Jain, 2004; Schmidt *et al*., 2000).

En esta sección se introducen los patrones arquitectónicos y se describe brevemente una selección de patrones arquitectónicos de uso común en diferentes tipos de sistemas. Para más información de patrones y su uso, debe remitirse a los manuales de patrones publicados.

Un patrón arquitectónico se puede considerar como una descripción abstracta estilizada de buena práctica, que se ensayó y puso a prueba en diferentes sistemas y entornos. De este modo, un patrón arquitectónico debe describir una organización de sistema que ha tenido éxito en sistemas previos. Debe incluir información sobre cuándo es y cuándo no es adecuado usar dicho patrón, así como sobre las fortalezas y debilidades del patrón.

Por ejemplo, la figura 6.2 describe el muy conocido patrón Modelo-Vista-Controlador. Este patrón es el soporte del manejo de la interacción en muchos sistemas basados en la Web. La descripción del patrón estilizado incluye el nombre del patrón, una breve descripción (con un modelo gráfico asociado) y un ejemplo del tipo de sistema donde se usa el patrón (de nuevo, quizá con un modelo gráfico). También debe incluir información sobre cuándo hay que usar el patrón, así como sobre sus ventajas y desventajas. En las figuras 6.3 y 6.4 se presentan los modelos gráficos de la arquitectura asociada con el patrón MVC. En ellas se muestra la arquitectura desde diferentes vistas: la figura 6.3 es una vista conceptual; en tanto que la figura 6.4 ilustra una posible arquitectura en tiempo de operación, cuando este patrón se usa para el manejo de la interacción en un sistema basado en la Web.

En una breve sección de un capítulo general es imposible describir todos los patrones genéricos que se usan en el desarrollo de software. En cambio, se presentan algunos ejemplos seleccionados de patrones que se utilizan ampliamente y captan buenos principios de diseño arquitectónico. En las páginas Web del libro se incluyen más ejemplos acerca de patrones arquitectónicos genéricos.

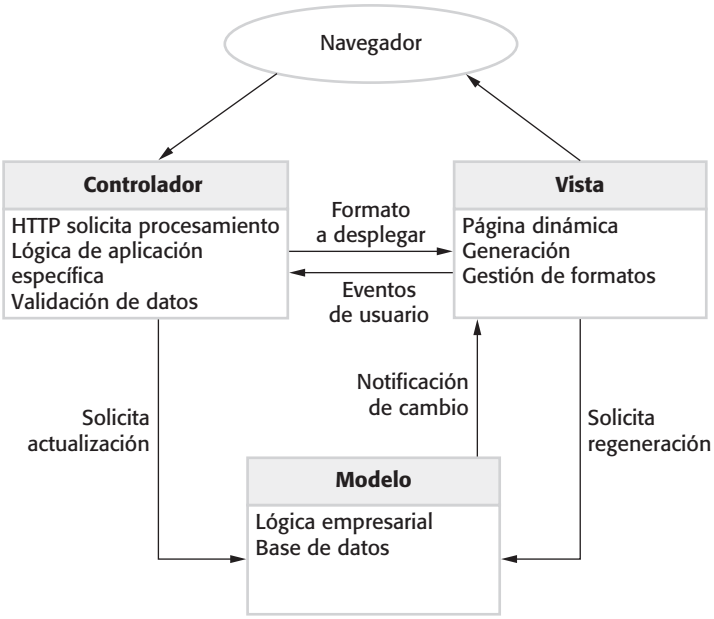

Figura 6.4 Arquitectura de aplicación Web con el patrón MVC

6.3.1 Arquitectura en capas

Las nociones de separación e independencia son fundamentales para el diseño arquitectónico porque permiten localizar cambios. El patrón MVC, que se muestra en la figura 6.2, separa elementos de un sistema, permitiéndoles cambiar de forma independiente. Por ejemplo, agregar una nueva vista o cambiar una vista existente puede hacerse sin modificación alguna a los datos subyacentes en el modelo. El patrón de arquitectura en capas es otra forma de lograr separación e independencia. Este patrón se ilustra en la figura 6.5. Aquí, la funcionalidad del sistema está organizada en capas separadas, y cada una se apoya sólo en las facilidades y los servicios ofrecidos por la capa inmediatamente debajo de ella.

Este enfoque en capas soporta el desarrollo incremental de sistemas. Conforme se desarrolla una capa, algunos de los servicios proporcionados por esta capa deben quedar a disposición de los usuarios. La arquitectura también es cambiable y portátil. En tanto su interfaz no varíe, una capa puede sustituirse por otra equivalente. Más aún, cuando las interfaces de capa cambian o se agregan nuevas facilidades a una capa, sólo resulta afectada la capa adyacente. A medida que los sistemas en capas localizan dependencias de máquina en capas más internas, se facilita el ofrecimiento de implementaciones multiplataforma de un sistema de aplicación. Sólo las capas más internas dependientes de la máquina deben reimplantarse para considerar las facilidades de un sistema operativo o base de datos diferentes.

La figura 6.6 es un ejemplo de una arquitectura en capas con cuatro capas. La capa inferior incluye software de soporte al sistema, por lo general soporte de base de datos y sistema operativo. La siguiente capa es la de aplicación, que comprende los componentes relacionados con la funcionalidad de la aplicación, así como los componentes de utilidad que usan otros componentes de aplicación. La tercera capa se relaciona con la gestión de interfaz del usuario y con brindar autenticación y autorización al usuario, mientras que la

Nombre	Arquitectura en capas
Descripción	Organiza el sistema en capas con funcionalidad relacionada con cada capa. Una capa da servicios a la capa de encima, de modo que las capas de nivel inferior representan servicios núcleo que es probable se utilicen a lo largo de todo el sistema. Véase la figura 6.6.
Ejemplo	Un modelo en capas de un sistema para compartir documentos con derechos de autor se tiene en diferentes bibliotecas, como se ilustra en la figura 6.7.
Cuándo se usa	Se usa al construirse nuevas facilidades encima de los sistemas existentes; cuando el desarrollo se dispersa a través de varios equipos de trabajo, y cada uno es responsable de una capa de funcionalidad; cuando exista un requerimiento para seguridad multinivel.
Ventajas	Permite la sustitución de capas completas en tanto se conserve la interfaz. Para aumentar la confiabilidad del sistema, en cada capa pueden incluirse facilidades redundantes (por ejemplo, autenticación).
Desventajas	En la práctica, suele ser difícil ofrecer una separación limpia entre capas, y es posible que una capa de nivel superior deba interactuar directamente con capas de nivel inferior, en vez de que sea a través de la capa inmediatamente abajo de ella. El rendimiento suele ser un problema, debido a múltiples niveles de interpretación de una solicitud de servicio mientras se procesa en cada capa.

Figura 6.5 Patrón de arquitectura en capas

capa superior proporciona facilidades de interfaz de usuario. Desde luego, es arbitrario el número de capas. Cualquiera de las capas en la figura 6.6 podría dividirse en dos o más capas.

La figura 6.7 es un ejemplo de cómo puede aplicarse este patrón de arquitectura en capas a un sistema de biblioteca llamado LIBSYS, que permite el acceso electrónico controlado a material con derechos de autor de un conjunto de bibliotecas universitarias. Tiene una arquitectura de cinco capas y, en la capa inferior, están las bases de datos individuales en cada biblioteca.

En la figura 6.17 (que se encuentra en la sección 6.4) se observa otro ejemplo de patrón de arquitectura en capas. Muestra la organización del sistema para atención a la salud mental (MHC-PMS) que se estudió en capítulos anteriores.

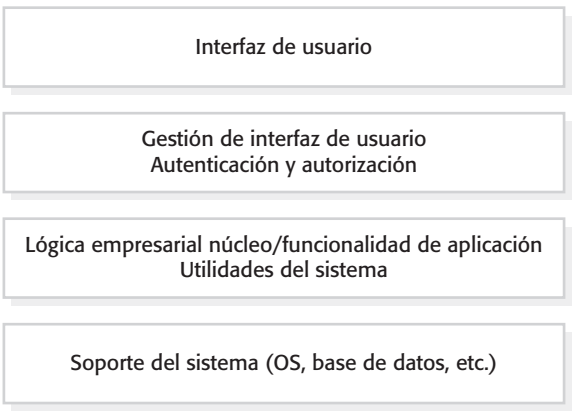

Figura 6.6 Arquitectura genérica en capas

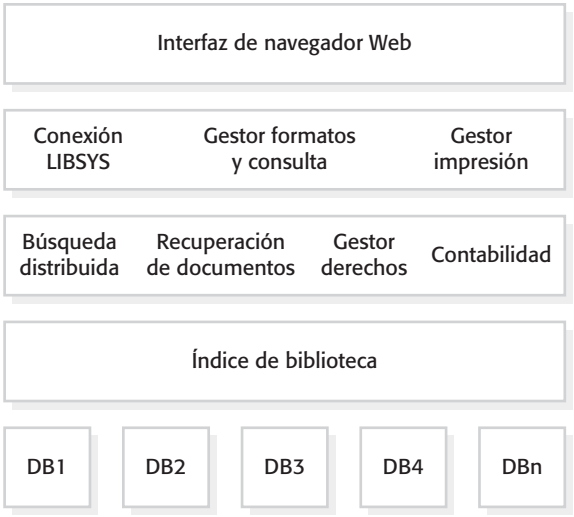

Figura 6.7 Arquitectura del sistema LIBSYS

6.3.2 Arquitectura de repositorio

Los patrones de arquitectura en capas y MVC son ejemplos de patrones en que la vista presentada es la organización conceptual de un sistema. El siguiente ejemplo, el patrón de repositorio (figura 6.8), describe cómo comparte datos un conjunto de componentes en interacción.

La mayoría de los sistemas que usan grandes cantidades de datos se organizan sobre una base de datos o un repositorio compartido. Por lo tanto, este modelo es adecuado

Figura 6.8 El patrón de repositorio

Nombre	Repositorio
Descripción	Todos los datos en un sistema se gestionan en un repositorio central, accesible a todos los componentes del sistema. Los componentes no interactúan directamente, sino tan sólo a través del repositorio.
Ejemplo	La figura 6.9 es un ejemplo de un IDE donde los componentes usan un repositorio de información de diseño de sistema. Cada herramienta de software genera información que, en ese momento, está disponible para uso de otras herramientas.
Cuándo se usa	Este patrón se usa cuando se tiene un sistema donde los grandes volúmenes de información generados deban almacenarse durante mucho tiempo. También puede usarse en sistemas dirigidos por datos, en los que la inclusión de datos en el repositorio active una acción o herramienta.
Ventajas	Los componentes pueden ser independientes, no necesitan conocer la existencia de otros componentes. Los cambios hechos por un componente se pueden propagar hacia todos los componentes. La totalidad de datos se puede gestionar de manera consistente (por ejemplo, respaldos realizados al mismo tiempo), pues todos están en un lugar.
Desventajas	El repositorio es un punto de falla único, de modo que los problemas en el repositorio afectan a todo el sistema. Es posible que haya ineficiencias al organizar toda la comunicación a través del repositorio. Quizá sea difícil distribuir el repositorio por medio de varias computadoras.

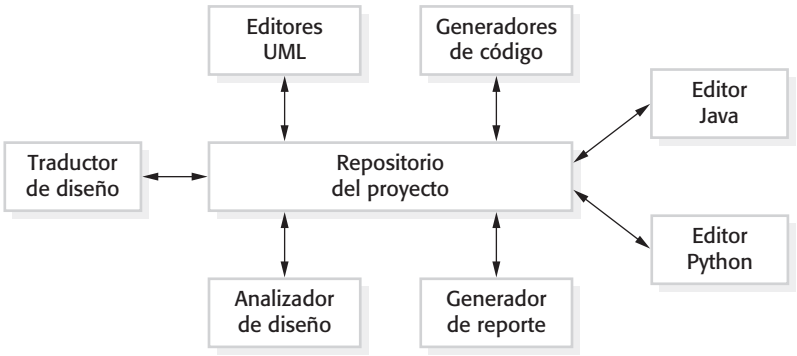

Figura 6.9 Arquitectura de repositorio para un IDE

para aplicaciones en las que un componente genere datos y otro los use. Los ejemplos de este tipo de sistema incluyen sistemas de comando y control, sistemas de información administrativa, sistemas CAD y entornos de desarrollo interactivo para software.

La figura 6.9 ilustra una situación en la que puede usarse un repositorio. Este diagrama muestra un IDE que incluye diferentes herramientas para soportar desarrollo dirigido por modelo. En este caso, el repositorio puede ser un entorno controlado por versión (como se estudia en el capítulo 25) que hace un seguimiento de los cambios al software y permite regresar (*rollback*) a versiones anteriores.

Organizar herramientas alrededor de un repositorio es una forma eficiente de compartir grandes cantidades de datos. No hay necesidad de transmitir explícitamente datos de un componente a otro. Sin embargo, los componentes deben operar en torno a un modelo de repositorio de datos acordado. Inevitablemente, éste es un compromiso entre las necesidades específicas de cada herramienta y sería difícil o imposible integrar nuevos componentes, si sus modelos de datos no se ajustan al esquema acordado. En la práctica, llega a ser complicado distribuir el repositorio sobre un número de máquinas. Aunque es posible distribuir un repositorio lógicamente centralizado, puede haber problemas con la redundancia e inconsistencia de los datos.

En el ejemplo que se muestra en la figura 6.9, el repositorio es pasivo, y el control es responsabilidad de los componentes que usan el repositorio. Un enfoque alternativo, que se derivó para sistemas IA, utiliza un modelo “blackboard” (pizarrón) que activa componentes cuando los datos particulares se tornan disponibles. Esto es adecuado cuando la forma de los datos del repositorio está menos estructurada. Las decisiones sobre cuál herramienta activar puede hacerse sólo cuando se hayan analizado los datos. Este modelo lo introdujo Nii (1986). Bosch (2000) incluye un buen análisis de cómo este estilo se relaciona con los atributos de calidad del sistema.

6.3.3 Arquitectura cliente-servidor

El patrón de repositorio se interesa por la estructura estática de un sistema sin mostrar su organización en tiempo de operación. El siguiente ejemplo ilustra una organización en tiempo de operación, de uso muy común para sistemas distribuidos. En la figura 6.10 se describe el patrón cliente-servidor.

Nombre	Cliente-servidor
Descripción	En una arquitectura cliente-servidor, la funcionalidad del sistema se organiza en servicios, y cada servicio lo entrega un servidor independiente. Los clientes son usuarios de dichos servicios y para utilizarlos ingresan a los servidores.
Ejemplo	La figura 6.11 es un ejemplo de una filmoteca y videoteca (videos/DVD) organizada como un sistema cliente-servidor.
Cuándo se usa	Se usa cuando, desde varias ubicaciones, se tiene que ingresar a los datos en una base de datos compartida. Como los servidores se pueden replicar, también se usan cuando la carga de un sistema es variable.
Ventajas	La principal ventaja de este modelo es que los servidores se pueden distribuir a través de una red. La funcionalidad general (por ejemplo, un servicio de impresión) estaría disponible a todos los clientes, así que no necesita implementarse en todos los servicios.
Desventajas	Cada servicio es un solo punto de falla, de modo que es susceptible a ataques de rechazo de servicio o a fallas del servidor. El rendimiento resultará impredecible porque depende de la red, así como del sistema. Quizás haya problemas administrativos cuando los servidores sean propiedad de diferentes organizaciones.

Figura 6.10 Patrón cliente-servidor

Un sistema que sigue el patrón cliente-servidor se organiza como un conjunto de servicios y servidores asociados, y de clientes que acceden y usan los servicios. Los principales componentes de este modelo son:

1. Un conjunto de servidores que ofrecen servicios a otros componentes. Ejemplos de éstos incluyen servidores de impresión; servidores de archivo que brindan servicios de administración de archivos, y un servidor compilador, que proporciona servicios de compilación de lenguaje de programación.
2. Un conjunto de clientes que solicitan los servicios que ofrecen los servidores. Habrá usualmente varias instancias de un programa cliente que se ejecuten de manera concurrente en diferentes computadoras.
3. Una red que permite a los clientes acceder a dichos servicios. La mayoría de los sistemas cliente-servidor se implementan como sistemas distribuidos, conectados mediante protocolos de Internet.

Las arquitecturas cliente-servidor se consideran a menudo como arquitecturas de sistemas distribuidos; sin embargo, el modelo lógico de servicios independientes que opera en servidores separados puede implementarse en una sola computadora. De nuevo, un beneficio importante es la separación e independencia. Los servicios y servidores pueden cambiar sin afectar otras partes del sistema.

Es posible que los clientes deban conocer los nombres de los servidores disponibles, así como los servicios que proporcionan. Sin embargo, los servidores no necesitan conocer la identidad de los clientes o cuántos clientes acceden a sus servicios. Los clientes acceden a los servicios que proporciona un servidor, a través de llamadas a procedimiento remoto usando un protocolo solicitud-respuesta, como el protocolo http utilizado

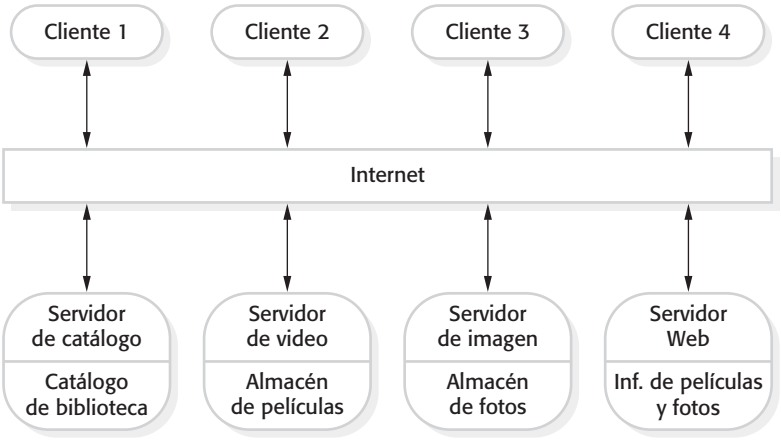

Figura 6.11 Arquitectura cliente-servidor para una filmoteca

en la WWW. En esencia, un cliente realiza una petición a un servidor y espera hasta que recibe una respuesta.

La figura 6.11 es un ejemplo de un sistema que se basa en el modelo cliente-servidor. Se trata de un sistema multiusuario basado en la Web, para ofrecer un repertorio de películas y fotografías. En este sistema, varios servidores manejan y despliegan los diferentes tipos de medios. Los cuadros de video necesitan transmitirse rápidamente y en sincronía, aunque a una resolución relativamente baja. Tal vez estén comprimidos en un almacén, de manera que el servidor de video puede manipular en diferentes formatos la compresión y descompresión del video. Sin embargo, las imágenes fijas deben conservarse en una resolución alta, por lo que es adecuado mantenerlas en un servidor independiente.

El catálogo debe manejar una variedad de consultas y ofrecer vínculos hacia el sistema de información Web, que incluye datos acerca de las películas y los videos, así

Figura 6.12 Patrón de tubería y filtro (*pipe and filter*)

Nombre	Tubería y filtro (*pipe and filter*)
Descripción	El procesamiento de datos en un sistema se organiza de forma que cada componente de procesamiento (filtro) sea discreto y realice un tipo de transformación de datos. Los datos fluyen (como en una tubería) de un componente a otro para su procesamiento.
Ejemplo	La figura 6.13 es un ejemplo de un sistema de tubería y filtro usado para el procesamiento de facturas.
Cuándo se usa	Se suele utilizar en aplicaciones de procesamiento de datos (tanto basadas en lotes [*batch*] como en transacciones), donde las entradas se procesan en etapas separadas para generar salidas relacionadas.
Ventajas	Fácil de entender y soporta reutilización de transformación. El estilo del flujo de trabajo coincide con la estructura de muchos procesos empresariales. La evolución al agregar transformaciones es directa. Puede implementarse como un sistema secuencial o como uno concurrente.
Desventajas	El formato para la transferencia de datos debe acordarse entre las transformaciones que se comunican. Cada transformación debe analizar sus entradas y sintetizar sus salidas al formato acordado. Esto aumenta la carga del sistema, y puede significar que sea imposible reutilizar transformaciones funcionales que usen estructuras de datos incompatibles.

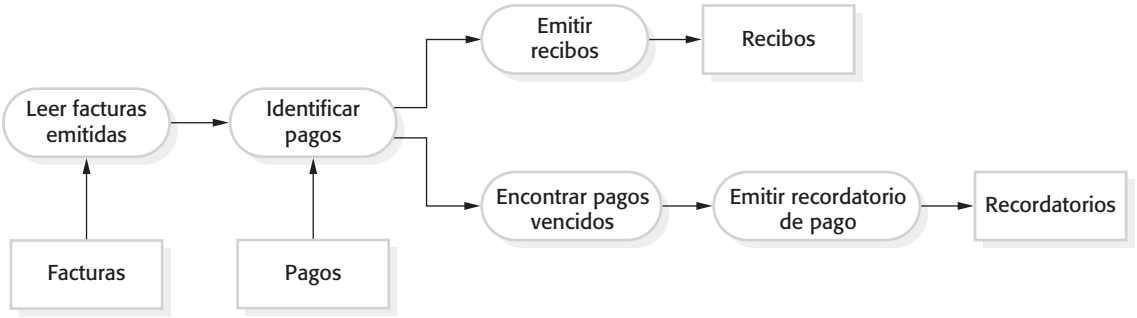

Figura 6.13 Ejemplo de arquitectura de tubería y filtro

como un sistema de comercio electrónico que soporte la venta de fotografías, películas y videos. El programa cliente es simplemente una interfaz integrada de usuario, construida mediante un navegador Web, para acceder a dichos servicios.

La ventaja más importante del modelo cliente-servidor consiste en que es una arquitectura distribuida. Éste puede usarse de manera efectiva en sistemas en red con distintos procesadores distribuidos. Es fácil agregar un nuevo servidor e integrarlo al resto del sistema, o bien, actualizar de manera clara servidores sin afectar otras partes del sistema. En el capítulo 18 se estudian las arquitecturas distribuidas, incluidas las arquitecturas cliente-servidor y las arquitecturas de objeto distribuidas.

6.3.4 Arquitectura de tubería y filtro

El ejemplo final de un patrón arquitectónico es el patrón tubería y filtro (*pipe and filter*). Éste es un modelo de la organización en tiempo de operación de un sistema, donde las transformaciones funcionales procesan sus entradas y producen salidas. Los datos fluyen de uno a otro y se transforman conforme se desplazan a través de la secuencia. Cada paso de procesamiento se implementa como un transformador. Los datos de entrada fluyen por medio de dichos transformadores hasta que se convierten en salida. Las transformaciones pueden ejecutarse secuencialmente o en forma paralela. Es posible que los datos se procesen por cada transformador ítem por ítem o en un solo lote.

El nombre "tubería y filtro" proviene del sistema Unix original, donde era posible vincular procesos empleando "tuberías". Por ellas pasaba una secuencia de texto de un proceso a otro. Los sistemas conformados con este modelo pueden implementarse al combinar comandos Unix, usando las tuberías y las instalaciones de control del intérprete de comandos Unix. Se usa el término "filtro" porque una transformación "filtra" los datos que puede procesar de su secuencia de datos de entrada.

Se han utilizado variantes de este patrón desde que se usaron por primera vez computadoras para el procesamiento automático de datos. Cuando las transformaciones son secuenciales, con datos procesados en lotes, este modelo arquitectónico de tubería y filtro se convierte en un modelo secuencial en lote, una arquitectura común para sistemas de procesamiento de datos (por ejemplo, un sistema de facturación). La arquitectura de un sistema embebido puede organizarse también como un proceso por entubamiento, donde cada proceso se ejecuta de manera concurrente. En el capítulo 20 se estudia el uso de este patrón en sistemas embebidos.

En la figura 6.13 se muestra un ejemplo de este tipo de arquitectura de sistema, que se usa en una aplicación de procesamiento en lote. Una organización emite facturas a los clientes. Una vez a la semana, los pagos efectuados se incorporan a las facturas. Para

Patrones arquitectónicos para control

Existen patrones arquitectónicos específicos que reflejan formas usadas comúnmente para organizar el control en un sistema. En ellos se incluyen el control centralizado, basado en un componente que llama otros componentes, y el control con base en un evento, donde el sistema reacciona a eventos externos.

http://www.SoftwareEngineering-9.com/Web/Architecture/ArchPatterns/

las facturas pagadas se emite un recibo. Para las facturas no saldadas dentro del plazo de pago se emite un recordatorio.

Los sistemas interactivos son difíciles de escribir con el modelo tubería y filtro, debido a la necesidad de procesar una secuencia de datos. Aunque las entradas y salidas textuales simples pueden modelarse de esta forma, las interfaces gráficas de usuario tienen formatos I/O más complejos, así como una estrategia de control que se basa en eventos como clics del mouse o selecciones del menú. Es difícil traducir esto en una forma compatible con el modelo *pipelining* (entubamiento).

6.4 Arquitecturas de aplicación

Los sistemas de aplicación tienen la intención de cubrir las necesidades de una empresa u organización. Todas las empresas tienen mucho en común: necesitan contratar personal, emitir facturas, llevar la contabilidad, etcétera. Las empresas que operan en el mismo sector usan aplicaciones comunes específicas para el sector. De esta forma, además de las funciones empresariales generales, todas las compañías telefónicas necesitan sistemas para conectar llamadas, administrar sus redes y emitir facturas a los clientes, entre otros. En consecuencia, también los sistemas de aplicación que utilizan dichas empresas tienen mucho en común.

Estos factores en común condujeron al desarrollo de arquitecturas de software que describen la estructura y la organización de tipos particulares de sistemas de software. Las arquitecturas de aplicación encapsulan las principales características de una clase de sistemas. Por ejemplo, en los sistemas de tiempo real puede haber modelos arquitectónicos genéricos de diferentes tipos de sistema, tales como sistemas de recolección de datos o sistemas de monitorización. Aunque las instancias de dichos sistemas difieren en detalle, la estructura arquitectónica común puede reutilizarse cuando se desarrollen nuevos sistemas del mismo tipo.

La arquitectura de aplicación puede reimplantarse cuando se desarrollen nuevos sistemas, pero, para diversos sistemas empresariales, la reutilización de aplicaciones es posible sin reimplementación. Esto se observa en el crecimiento de los sistemas de planeación de recursos empresariales (ERP, por las siglas de *Enterprise Resource Planning*) de compañías como SAP y Oracle, y paquetes de software vertical (COTS) para aplicaciones especializadas en diferentes áreas de negocios. En dichos sistemas, un sistema genérico se configura y adapta para crear una aplicación empresarial específica.

Arquitecturas de aplicación

En el sitio Web del libro existen muchos ejemplos de arquitecturas de aplicación. Se incluyen descripciones de sistemas de procesamiento de datos en lote, sistemas de asignación de recursos y sistemas de edición basados en eventos.

http://www.SoftwareEngineering-9.com/Web/Architecture/AppArch/

Por ejemplo, un sistema para suministrar la administración en cadena se adapta a diferentes tipos de proveedores, bienes y arreglos contractuales.

Como diseñador de software, usted puede usar modelos de arquitecturas de aplicación en varias formas:

1. *Como punto de partida para el proceso de diseño arquitectónico* Si no está familiarizado con el tipo de aplicación que desarrolla, podría basar su diseño inicial en una arquitectura de aplicación genérica. Desde luego, ésta tendrá que ser especializada para el sistema específico que se va a desarrollar, pero es un buen comienzo para el diseño.
2. *Como lista de verificación del diseño* Si usted desarrolló un diseño arquitectónico para un sistema de aplicación, puede comparar éste con la arquitectura de aplicación genérica y luego, verificar que su diseño sea consistente con la arquitectura genérica.
3. *Como una forma de organizar el trabajo del equipo de desarrollo* Las arquitecturas de aplicación identifican características estructurales estables de las arquitecturas del sistema y, en muchos casos, es posible desarrollar éstas en paralelo. Puede asignar trabajo a los miembros del grupo para implementar diferentes componentes dentro de la arquitectura.
4. *Como un medio para valorar los componentes a reutilizar* Si tiene componentes por reutilizar, compare éstos con las estructuras genéricas para saber si existen componentes similares en la arquitectura de aplicación.
5. *Como un vocabulario para hablar acerca de los tipos de aplicaciones* Si discute acerca de una aplicación específica o trata de comparar aplicaciones del mismo tipo, entonces puede usar los conceptos identificados en la arquitectura genérica para hablar sobre las aplicaciones.

Hay muchos tipos de sistema de aplicación y, en algunos casos, parecerían muy diferentes. Sin embargo, muchas de estas aplicaciones distintas superficialmente en realidad tienen mucho en común y, por ende, suelen representarse mediante una sola arquitectura de aplicación abstracta. Esto se ilustra aquí al describir las siguientes arquitecturas de dos tipos de aplicación:

1. *Aplicaciones de procesamiento de transacción* Este tipo de aplicaciones son aplicaciones centradas en bases de datos, que procesan los requerimientos del usuario mediante la información y actualizan ésta en una base de datos. Se trata del tipo más común de sistemas empresariales interactivos. Se organizan de tal forma que las acciones del usuario no pueden interferir unas con otras y se mantiene la integridad de la base

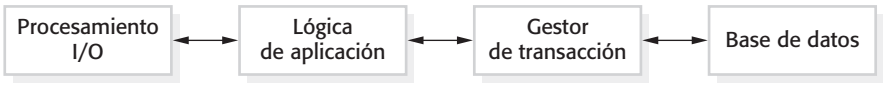

Figura 6.14 Estructura de las aplicaciones de procesamiento de transacción

de datos. Esta clase de sistema incluye los sistemas bancarios interactivos, sistemas de comercio electrónico, sistemas de información y sistemas de reservaciones.

2. *Sistemas de procesamiento de lenguaje* Son sistemas en los que las intenciones del usuario se expresan en un lenguaje formal (como Java). El sistema de procesamiento de lenguaje elabora este lenguaje en un formato interno y después interpreta dicha representación interna. Los sistemas de procesamiento de lenguaje mejor conocidos son los compiladores, que traducen los programas en lenguaje de alto nivel dentro de un código de máquina. Sin embargo, los sistemas de procesamiento de lenguaje se usan también en la interpretación de lenguajes de comandos para bases de datos y sistemas de información, así como de lenguajes de marcado como XML (Harold y Means, 2002; Hunter *et al*., 2007).

Se eligieron estos tipos particulares de sistemas porque una gran cantidad de sistemas empresariales basados en la Web son sistemas de procesamiento de transacciones, y todo el desarrollo del software se apoya en los sistemas de procesamiento de lenguaje.

6.4.1 Sistemas de procesamiento de transacciones

Los sistemas de procesamiento de transacciones (TP, por las siglas de *Transaction Processing*) están diseñados para procesar peticiones del usuario mediante la información de una base de datos, o los requerimientos para actualizar una base de datos (Lewis *et al*., 2003). Técnicamente, una transacción de base de datos es una secuencia de operaciones que se trata como una sola unidad (una unidad atómica). Todas las operaciones en una transacción tienen que completarse antes de que sean permanentes los cambios en la base de datos. Esto garantiza que la falla en las operaciones dentro de la transacción no conduzca a inconsistencias en la base de datos.

Desde una perspectiva de usuario, una transacción es cualquier secuencia coherente de operaciones que satisfacen un objetivo, como "encontrar los horarios de vuelos de Londres a París". Si la transacción del usuario no requiere el cambio en la base de datos, entonces sería innecesario empaquetar esto como una transacción técnica de base de datos.

Un ejemplo de una transacción es una petición de cliente para retirar dinero de una cuenta bancaria mediante un cajero automático. Esto incluye obtener detalles de la cuenta del cliente, verificar y modificar el saldo por la cantidad retirada y enviar comandos al cajero automático para entregar el dinero. Hasta que todos estos pasos se completan, la transacción permanece inconclusa y no cambia la base de datos de la cuenta del cliente.

Por lo general, los sistemas de procesamiento de transacción son sistemas interactivos donde los usuarios hacen peticiones asíncronas de servicios. La figura 6.14 ilustra la estructura arquitectónica conceptual de las aplicaciones de TP. Primero, un usuario hace una petición al sistema a través de un componente de procesamiento I/O. La petición se procesa mediante alguna lógica específica de la aplicación. Se crea una transacción y pasa hacia un gestor de transacciones que, por lo general, está embebido en el sistema de manejo de la

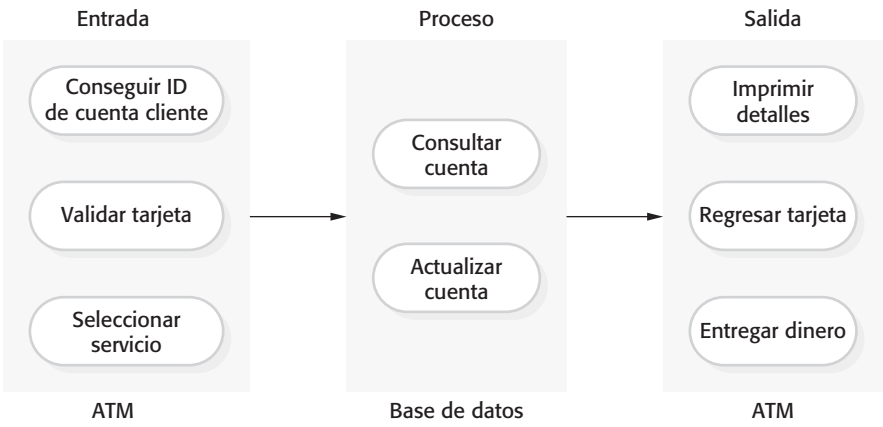

Figura 6.15 Arquitectura de software de un sistema ATM

base de datos. Después de que el gestor de transacciones asegura que la transacción se ha completado adecuadamente, señala a la aplicación que terminó el procesamiento.

Los sistemas de procesamiento de transacción pueden organizarse como una arquitectura "tubería y filtro" con componentes de sistema responsables de entradas, procesamiento y salida. Por ejemplo, considere un sistema bancario que permite a los clientes consultar sus cuentas y retirar dinero de un cajero automático. El sistema está constituido por dos componentes de software cooperadores: el software del cajero automático y el software de procesamiento de cuentas en el servidor de la base de datos del banco. Los componentes de entrada y salida se implementan como software en el cajero automático y el componente de procesamiento es parte del servidor de la base de datos del banco. La figura 6.15 muestra la arquitectura de este sistema e ilustra las funciones de los componentes de entrada, proceso y salida.

6.4.2 Sistemas de información

Todos los sistemas que incluyen interacción con una base de datos compartida se consideran sistemas de información basados en transacciones. Un sistema de información permite acceso controlado a una gran base de información, tales como un catálogo de biblioteca, un horario de vuelos o los registros de pacientes en un hospital. Cada vez más, los sistemas de información son sistemas basados en la Web, cuyo acceso es mediante un navegador Web.

La figura 6.16 presenta un modelo muy general de un sistema de información. El sistema se modela con un enfoque por capas (estudiado en la sección 6.3), donde la capa superior soporta la interfaz de usuario, y la capa inferior es la base de datos del sistema. La capa de comunicaciones con el usuario maneja todas las entradas y salidas de la interfaz de usuario, y la capa de recuperación de información incluye la lógica específica de aplicación para acceder y actualizar la base de datos. Como se verá más adelante, las capas en este modelo pueden trazarse directamente hacia servidores dentro de un sistema basado en Internet.

Como ejemplo de una instancia de este modelo en capas, la figura 6.17 muestra la arquitectura del MHC-PMS. Recuerde que este sistema mantiene y administra detalles de los pacientes que consultan médicos especialistas en problemas de salud mental. En el

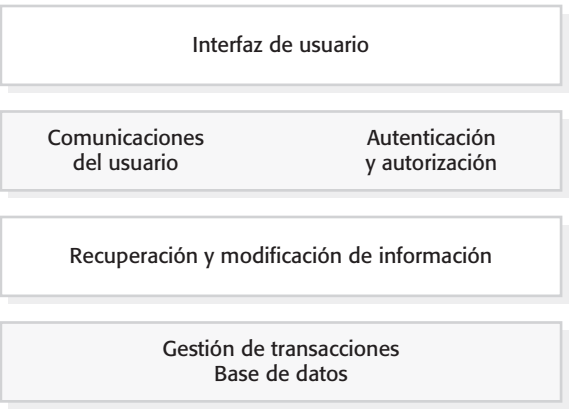

Figura 6.16 Arquitectura de sistema de información en capas

modelo se agregaron detalles a cada capa al identificar los componentes que soportan las comunicaciones del usuario, así como la recuperación y el acceso a la información:

1. La capa superior es responsable de implementar la interfaz de usuario. En este caso, la UI se implementó con el uso de un navegador Web.
2. La segunda capa proporciona la funcionalidad de interfaz de usuario que se entrega a través del navegador Web. Incluye componentes que permiten a los usuarios ingresar al sistema, y componentes de verificación para garantizar que las operaciones utilizadas estén permitidas de acuerdo con su rol. Esta capa incluye componentes de gestión de formato y menú que presentan información a los usuarios, así como componentes de validación de datos que comprueban la consistencia de la información.
3. La tercera capa implementa la funcionalidad del sistema y ofrece componentes que ponen en operación la seguridad del sistema, la creación y actualización de la información del paciente, la importación y exportación de datos del paciente desde otras bases de datos, y los generadores de reporte que elaboran informes administrativos.

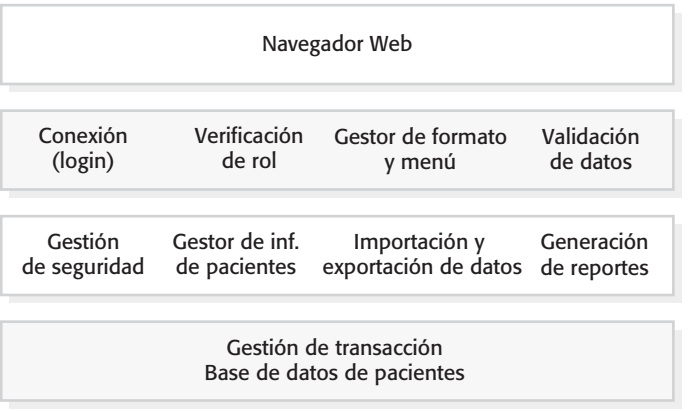

Figura 6.17 Arquitectura del MHC-PMS

4. Finalmente, la capa más baja, que se construye al usar un sistema comercial de gestión de base de datos, ofrece administración de transacciones y almacenamiento constante de datos.

Los sistemas de gestión de información y recursos, por lo general, son ahora sistemas basados en la Web donde las interfaces de usuario se implementan con el uso de un navegador Web. Por ejemplo, los sistemas de comercio electrónico son sistemas de gestión de recursos basados en Internet, que aceptan pedidos electrónicos por bienes o servicios y, luego, ordenan la entrega de dichos bienes o servicios al cliente. En un sistema de comercio electrónico, la capa específica de aplicación incluye funcionalidad adicional que soporta un "carrito de compras", donde los usuarios pueden colocar algunos objetos en transacciones separadas y, luego, pagarlos en una sola transacción.

La organización de servidores en dichos sistemas refleja usualmente el modelo genérico en cuatro capas presentadas en la figura 6.16. Dichos sistemas se suelen implementar como arquitecturas cliente-servidor de multinivel, como se estudia en el capítulo 18:

1. El servidor Web es responsable de todas las comunicaciones del usuario, y la interfaz de usuario se pone en función mediante un navegador Web;
2. El servidor de aplicación es responsable de implementar la lógica específica de la aplicación, así como del almacenamiento de la información y las peticiones de recuperación;
3. El servidor de la base de datos mueve la información hacia y desde la base de datos y, además, manipula la gestión de transacciones.

El uso de múltiples servidores permite un rendimiento elevado, al igual que posibilita la manipulación de cientos de transacciones por minuto. Conforme aumenta la demanda, pueden agregarse servidores en cada nivel, para lidiar con el procesamiento adicional implicado.

6.4.3 Sistemas de procesamiento de lenguaje

Los sistemas de procesamiento de lenguaje convierten un lenguaje natural o artificial en otra representación del lenguaje y, para lenguajes de programación, también pueden ejecutar el código resultante. En ingeniería de software, los compiladores traducen un lenguaje de programación artificial en código de máquina. Otros sistemas de procesamiento de lenguaje traducen una descripción de datos XML en comandos para consultar una base de datos o una representación XML alternativa. Los sistemas de procesamiento de lenguaje natural pueden transformar un lenguaje natural a otro, por ejemplo, francés a noruego.

En la figura 6.18 se ilustra una posible arquitectura para un sistema de procesamiento de lenguaje hacia un lenguaje de programación. Las instrucciones en el lenguaje fuente definen el programa a ejecutar, en tanto que un traductor las convierte en instrucciones para una máquina abstracta. Luego, dichas instrucciones se interpretan mediante otro componente que lee (*fetch*) las instrucciones para su ejecución y las ejecuta al usar (si es necesario) datos del entorno. La salida del proceso es el resultado de la interpretación de instrucciones en los datos de entrada.

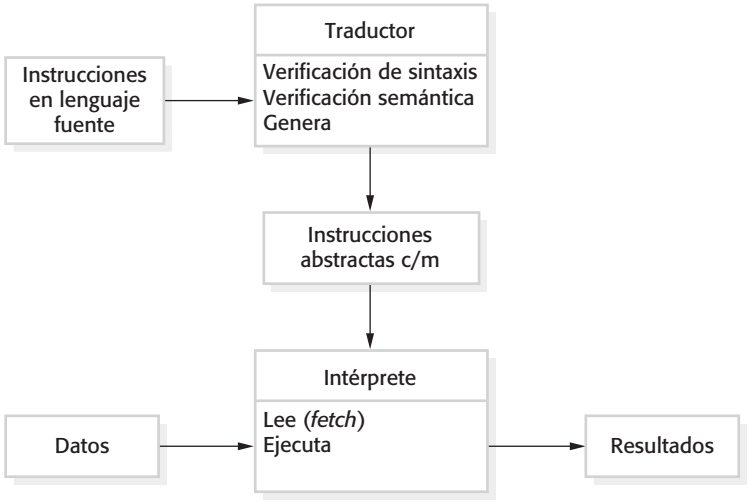

Figura 6.18 Arquitectura de un sistema de procesamiento de lenguaje

Desde luego, para muchos compiladores, el intérprete es una unidad de hardware que procesa instrucciones de la máquina, en tanto que la máquina abstracta es el procesador real. Sin embargo, para lenguajes escritos de manera dinámica, como Python, el intérprete puede ser un componente de software.

Los compiladores de lenguaje de programación que forman parte de un entorno de programación más general tienen una arquitectura genérica (figura 6.19) que incluye los siguientes componentes:

1. Un analizador léxico, que toma valores simbólicos (*tokens*) y los convierte en una forma interna.
2. Una tabla de símbolos, que contiene información de los nombres de las entidades (variables, de clase, de objeto, etcétera) usados en el texto que se traduce.
3. Un analizador de sintaxis, el cual verifica la sintaxis del lenguaje que se va a traducir. Emplea una gramática definida del lenguaje y construye un árbol de sintaxis.
4. Un árbol de sintaxis es una estructura interna que representa el programa a compilar.

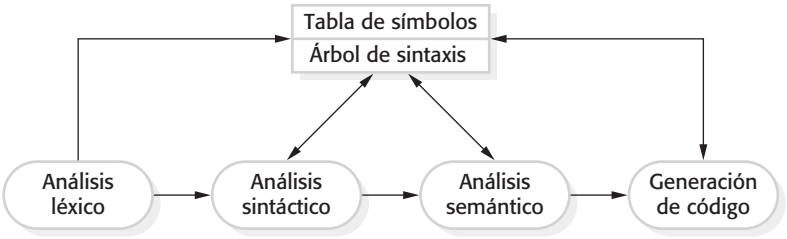

Figura 6.19 Una arquitectura de compilador de tubería y filtro

Arquitecturas de referencia

Las arquitecturas de referencia captan en un dominio características importantes de las arquitecturas del sistema. En esencia, incluyen todo lo que pueda estar en una arquitectura de aplicación aunque, en realidad, es muy improbable que alguna aplicación individual contenga todas las características mostradas en una arquitectura de referencia. El objetivo principal de las arquitecturas de referencia consiste en evaluar y comparar las propuestas de diseño y, en dicho dominio, educar a las personas sobre las características arquitectónicas.

http://www.SoftwareEngineering-9.com/Web/Architecture/RefArch.html

5. Un analizador semántico que usa información del árbol de sintaxis y la tabla de símbolos, para verificar la exactitud semántica del texto en lenguaje de entrada.
6. Un generador de código que "recorre" el árbol de sintaxis y genera un código de máquina abstracto.

También pueden incluirse otros componentes que analizan y transforman el árbol de sintaxis para mejorar la eficiencia y remover redundancia del código de máquina generado. En otros tipos de sistema de procesamiento de lenguaje, como un traductor de lenguaje natural, habrá componentes adicionales, por ejemplo, un diccionario, y el código generado en realidad es el texto de entrada traducido en otro lenguaje.

Existen patrones arquitectónicos alternativos que pueden usarse en un sistema de procesamiento de lenguaje (Garlan y Shaw, 1993). Pueden implementarse compiladores con una composición de un repositorio y un modelo de tubería y filtro. En una arquitectura de compilador, la tabla de símbolos es un repositorio para datos compartidos. Las fases de análisis léxico, sintáctico y semántico se organizan de manera secuencial, como se muestra en la figura 6.19, y se comunican a través de la tabla de símbolos compartida.

Este modelo de tubería y filtro de compilación de lenguaje es efectivo en entornos *batch*, donde los programas se compilan y ejecutan sin interacción del usuario; por ejemplo, en la traducción de un documento XML a otro. Es menos efectivo cuando un compilador se integra con otras herramientas de procesamiento de lenguaje, como un sistema de edición estructurado, un depurador interactivo o un programa de impresión estética (*prettyprinter*). En esta situación, los cambios de un componente deben reflejarse de inmediato en otros componentes. Por lo tanto, es mejor organizar el sistema en torno a un repositorio, como se muestra en la figura 6.20.

Esta figura ilustra cómo un sistema de procesamiento de lenguaje puede formar parte de un conjunto integrado de herramientas de soporte de programación. En este ejemplo, la tabla de símbolos y el árbol de sintaxis actúan como un almacén de información central. Las herramientas y los fragmentos de herramienta se comunican a través de él. Otra información que en ocasiones se incrusta en las herramientas, como la definición gramática y la definición del formato de salida para el programa, se toma de las herramientas y se coloca en el repositorio. En consecuencia, un editor enfocado en la sintaxis podría verificar que ésta sea correcta mientras se escribe un programa, y un *prettyprinter* puede crear listados del programa en un formato que sea fácil de leer.