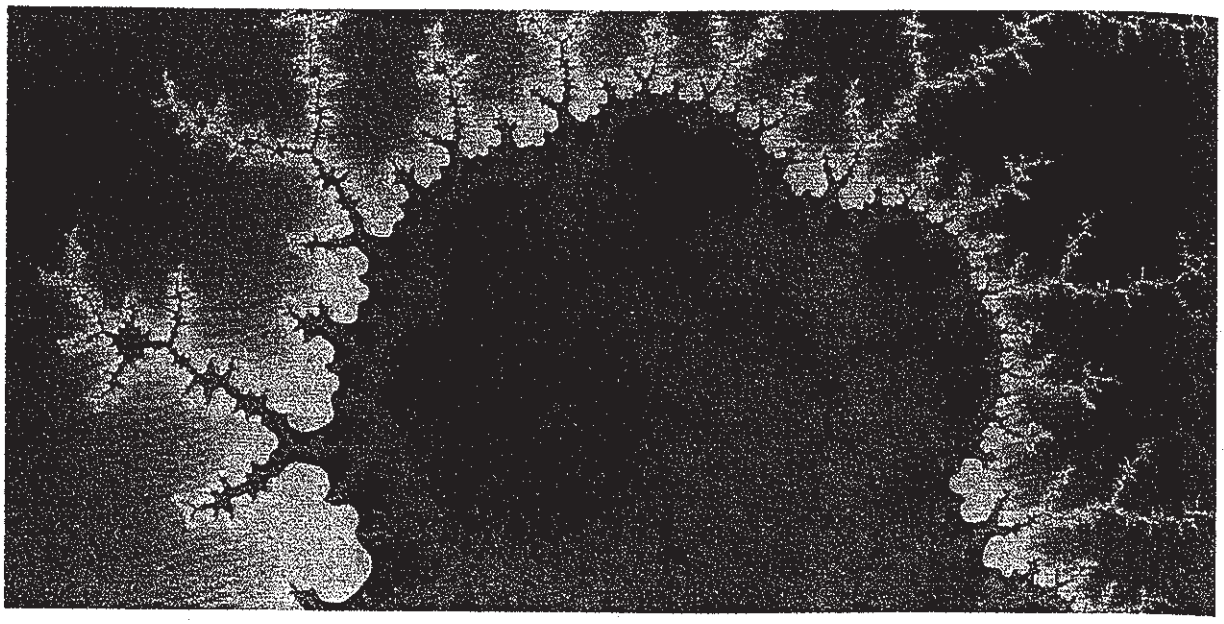

19
Interfaces y componentes

19.1. Presentación del capítulo

Este capítulo trata dos temas principales, interfaces y componentes. Tratamos estos dos temas juntos porque, como verá, están íntimamente relacionados. También verá cómo utilizar interfaces combinados con un tipo especial de componente denominado un subsistema, permite la creación de arquitecturas flexibles de sistemas.

19.2. Actividad UP: Diseñar un subsistema

Este capítulo trata principalmente sobre la actividad UP `Diseñar un subsistema`. Un subsistema es solamente un tipo especial de componente, por lo que en este capítulo tratamos componentes y desarrollo basado en componentes junto con subsistemas.

Los temas en este capítulo también tienen un impacto en las otras actividades de diseño de UP, como verá.

La actividad de UP `Diseñar un subsistema` se muestra en la figura 19.2. Hemos modificado la figura original para ponerla en línea con UML 2 donde los subsistemas son tipos de componentes en lugar de paquetes estereotipados. Los artefactos cambiados se muestran en gris en la figura.

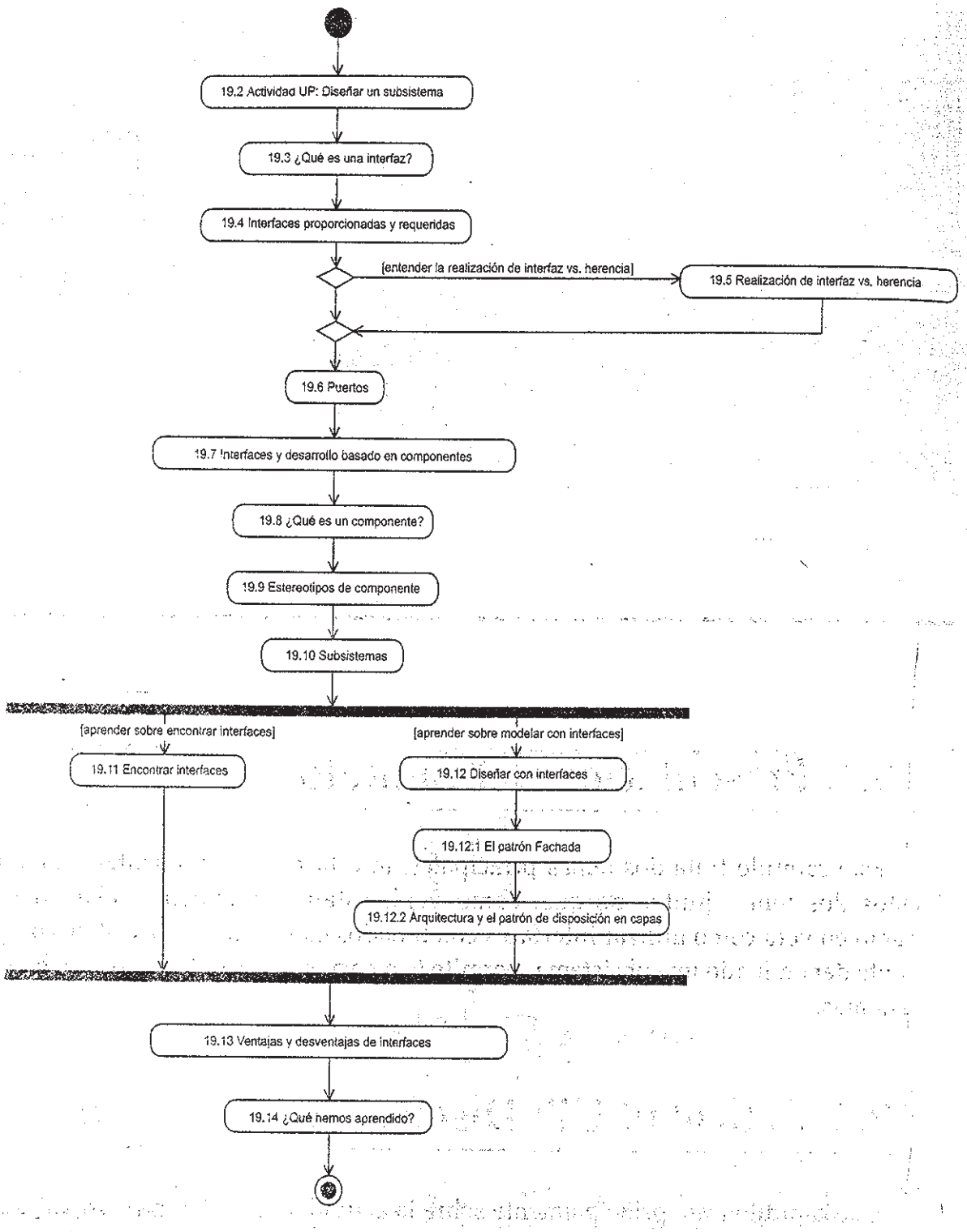

Figura 19.1.

La actividad `Diseñar un subsistema` trata sobre desglosar el sistema en partes que son tan independientes como sea posible. Estas partes se denominan subsistemas.

Las interacciones entre los subsistemas están mediadas por interfaces. Los objetivos del diseño de subsistema son minimizar el acoplamiento en el sistema al diseñar interfaces apropiadas y asegurarse de que cada subsistema realiza correctamente el comportamiento especificado por sus interfaces.

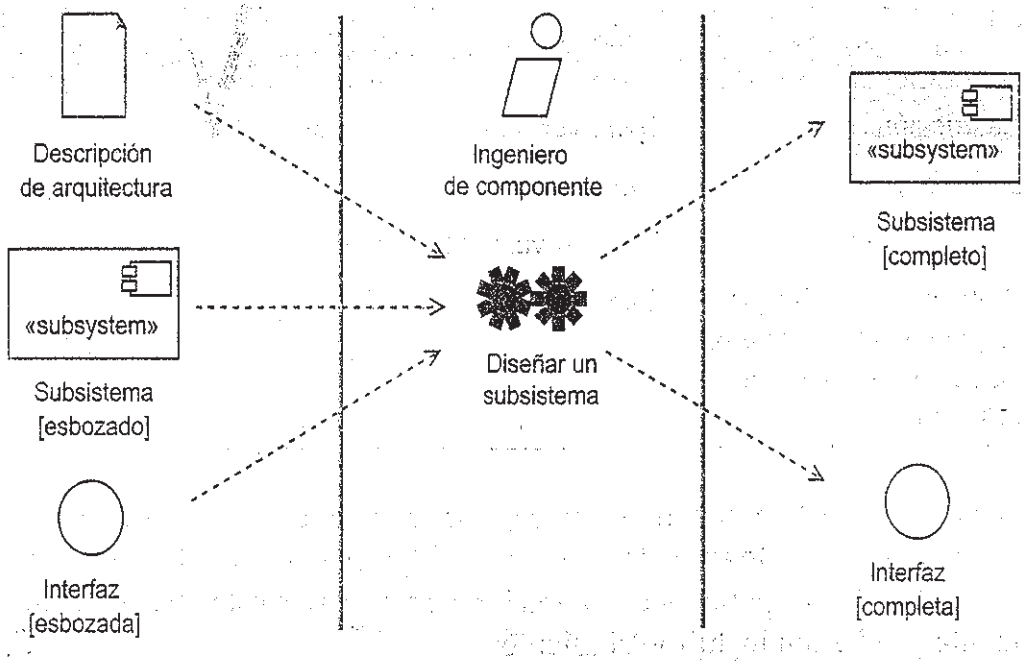

Figura 19.2. Adaptada de la figura 9.44 [Jacobson 1] con permiso de Addison-Wesley.

19.3. ¿Qué es una interfaz?

Una interfaz especifica un conjunto de características públicas.

La idea clave detrás de las interfaces es separar la especificación de funcionalidad (la interfaz) de su implementación por un clasificador como una clase o subsistema. Una interfaz no se puede instanciar; simplemente declara un contrato que se puede realizar por cero o más clasificadores. Cualquier cosa que realice una interfaz acepta y está de acuerdo en ajustarse a lo que el contrato de esa interfaz define.

Las interfaces pueden especificar las características listadas en la tabla 19.1. Esta tabla también muestra las responsabilidades de dichos clasificadores con respecto a la interfaz.

Tabla 19.1.

Interfaz especifica	Clasificador de realización
Operación.	Debe tener una operación con la misma firma y semántica.
Atributo.	Debe tener operaciones públicas para establecer y obtener el valor del atributo: el clasificador que lo realiza no es necesario para tener el atributo especificado por la interfaz, pero se debe comportar como si lo tuviera.
Asociación.	Debe tener una asociación con el clasificador destino; si una interfaz especifica una asociación con otra interfaz, los clasificadores que implementan estas interfaces deben tener una asociación entre ellos.

Interfaz específica	Clasificador de realización
Restricción.	Debe soportar la restricción.
Estereotipo.	Tiene el estereotipo.
Valor etiquetado.	Tiene el valor etiquetado.
Protocolo (por ejemplo según se define por una máquina de estado de protocolo).	Debe realizar el protocolo.

Las interfaces también necesitan especificaciones de la semántica de sus características (normalmente en texto o pseudocódigo) para guiar a los implementadores.

Los atributos y operaciones en una interfaz deberían estar especificados completamente y deberían incluir lo siguiente:

- La firma completa de la operación (nombre, tipos de todos los parámetros y tipo de retorno).
- La semántica de la operación, esto se puede grabar como texto o pseudocódigo.
- El nombre y tipo de los atributos.
- Cualquier operación o estereotipo de atributo, restricción y valor etiquetado.

Es importante recordar que una interfaz solamente define una especificación por sus características y nunca implica ninguna implementación en particular.

Las interfaces pueden tener un efecto profundo sobre la forma que diseña. Hasta ahora, hemos estado diseñando al conectar clases específicas, a lo que se podría llamar "diseñar para una implementación". Sin embargo, es más flexible "diseñar para un contrato" donde conecta a una interfaz y esta interfaz se puede realizar por un número de clases y otros clasificadores. Las librerías estándar de Java dan testimonio de la flexibilidad y potencial de este enfoque. A menudo decimos que una interfaz define un servicio ofrecido por una clase, subsistema o componente. Las arquitecturas modernas de software están a menudo basadas en servicio.

Java es el primer lenguaje comúnmente utilizado para introducir un constructor específico de lenguaje para interfaces. Sin embargo, es bastante posible programar utilizando interfaces en lenguajes que no tienen constructor para interfaces. Por ejemplo, en C++ simplemente definirá una clase abstracta, ésta es una clase abstracta cuyas operaciones son todas abstractas.

19.4. Interfaces proporcionadas y requeridas

El conjunto de interfaces realizadas por un clasificador se conoce como sus interfaces proporcionadas. Cuando un clasificador requiere una o más interfaces

para su operación, éstas se conocen como interfaces requeridas. La sintaxis UML para interfaces proporcionadas se muestra en la figura 19.3. Existen dos variantes, la notación al estilo de "clase" (en la que tiene que mostrar los atributos y operaciones), completa, y la notación abreviada (en la que puede no mostrar los atributos y operaciones).

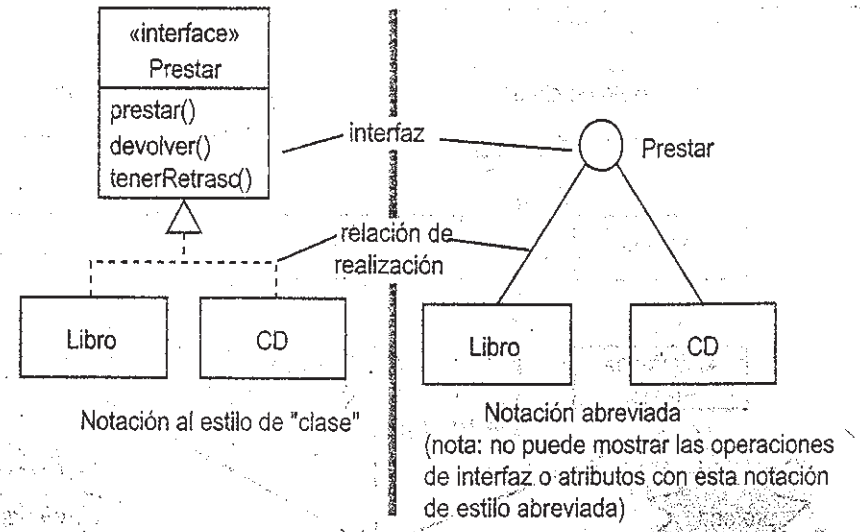

Figura 19.3.

El ejemplo en la figura 19.3 está tomado de un sencillo sistema de gestión de biblioteca donde la clase `Biblioteca` mantiene una colección de `Libros` y una colección de `CD`. Cada `Libro` y `CD` realiza la interfaz `Prestar` que especifica el protocolo para un elemento que se puede prestar. Aunque `Libros` y `CD` son semánticamente bastante diferentes, la interfaz `Prestar` permite que la `Biblioteca` los trate de forma uniforme, al menos en lo que se refiere a su protocolo de préstamo.

Las interfaces se nombran como clases, poniendo en mayúscula la primera letra de cada palabra con la primera letra de todas en mayúsculas. Sin embargo, en Visual Basic y C# existe un estándar común para prefijar cada nombre de interfaz con la mayúscula I, por ejemplo `IPrestar`. Otra forma de nombrarlos es añadir "able" al final del nombre, por ejemplo, `Prestable` o `IPrestable`.

La relación de realización es la relación entre una especificación (en este caso, una interfaz) y lo que realiza la especificación (en este caso, las clases `Libro` y `CD`). Veremos más adelante que no solamente son las clases las que pueden realizar una interfaz, otros clasificadores como paquetes y componentes pueden hacerlo también.

Observe que en la notación de estilo de clase, la relación de realización se dibuja como una línea de puntos con una punta de flecha no rellena, mientras que en la notación abreviada se convierte en una línea gruesa sin punta de flecha. La idea aquí es que la notación abreviada se mantenga lo más concisa posible. Utilice la notación de clase cuando quiera mostrar las características de la interfaz y la notación abreviada cuando no quiera. Sin embargo, ambas significan exactamente lo mismo.

La figura 19.4 muestra la clase `Biblioteca` que requiere la interfaz `Prestar`. Puede mostrar una interfaz requerida como una dependencia a una interfaz (en la notación completa o abreviada) o como un elemento en el que encaja la interfaz requerida. De cualquier forma, lo que esta notación nos dice es que `Biblioteca` entiende y necesita el protocolo especifico definido por la interfaz `Prestar`. Cualquier cosa que realice esta interfaz se puede conectar a `Biblioteca` y la `Biblioteca` entenderá qué se puede prestar y devolver.

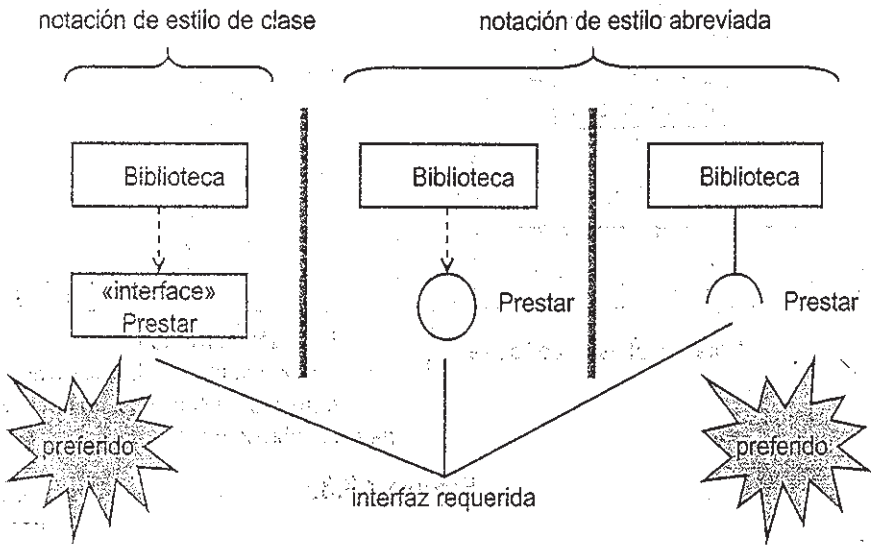

Figura 19.4.

La figura 19.5 muestra el sistema `Biblioteca` ensamblado. El conector de ensamblaje indica que `Libro` y `CD` proporcionan el conjunto de servicios (definidos por `Prestar`) que requiere la `Biblioteca`.

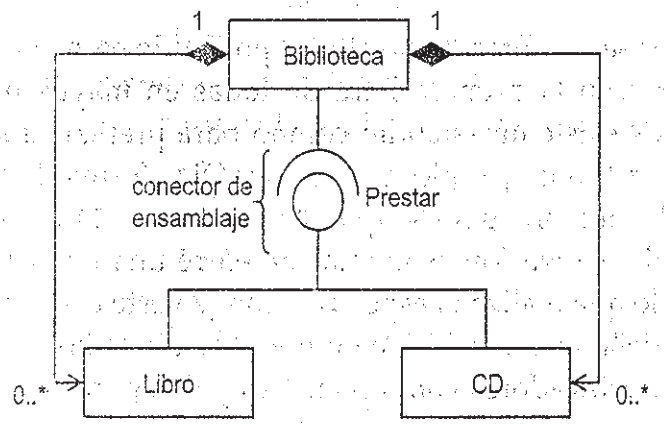

Figura 19.5.

Otro buen ejemplo de interfaces con múltiples implementaciones se puede encontrar en las clases de colección en las bibliotecas estándar de Java (véase la figura 19.6). Aunque se definen ocho interfaces, Java proporciona muchas implementaciones diferentes que tienen características diferentes. Al diseñar para una interfaz, el diseñador de Java puede dejar la realización de la interfaz para el momento de

implementación y dejar que el programador Java elija la implementación con las características más apropiadas.

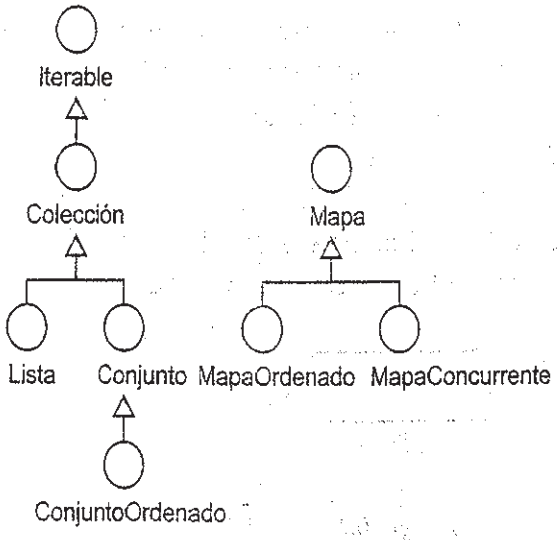

Figura 19.6.

19.5. Realización de interfaz vs. herencia

En este punto, merece la pena considerar la diferencia entre realización de interfaz y herencia. La semántica de realización de interfaz es "realiza el contrato especificado por" mientras que la semántica de herencia es "es un". El principio de sustitución se aplica tanto a herencia como a la realización de interfaz, por lo que ambos tipos de relación pueden generar polimorfismo. Para ilustrar las diferencias entre realización de interfaz y herencia, hemos creado una solución alternativa para el problema de biblioteca basándose en herencia (véase la figura 19.7). Esto parece ser bastante posible y en algunos casos una solución más sencilla para el sistema de biblioteca.

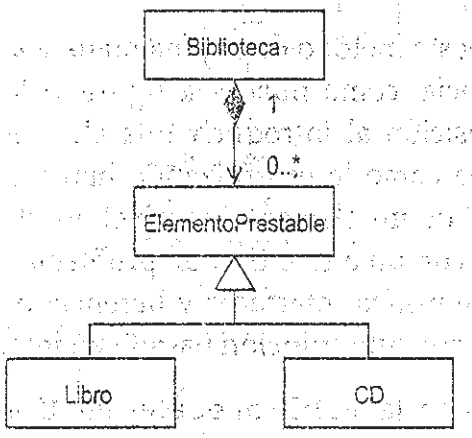

Figura 19.7.

Lo que es más importante es que este modelo basado en herencia no parece demasiado correcto porque estamos realizando una declaración bastante concreta

de que Libros y CD son del tipo ElementoPrestable. Pero, ¿es ésta posibilidad de poder tomar prestados Libros y CD suficiente para especificar su tipo? Quizá esto se podría considerar simplemente un aspecto determinado de su comportamiento que da la casualidad que tienen en común en el contexto del sistema de la biblioteca. Semánticamente es más correcto ver ElementoPrestable como un rol particular que tanto Libros como CD desempeñan con respecto a la Biblioteca, en lugar de como un supertipo común.

Para ilustrar un problema práctico con el modelo en la figura 19.7, considere añadir Revistas a la Biblioteca. Revistas como Nature no se prestan. Esto da lugar al modelo basado en herencia en la figura 19.8.

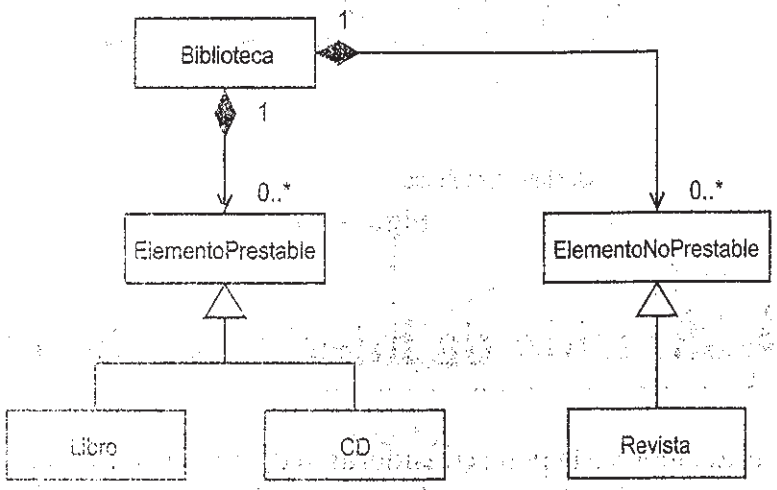

Figura 19.8.

Observe que la Biblioteca ahora tiene que mantener dos listas de objetos, los que se prestan y los que no. Esta solución funcionaría pero no es demasiado elegante ya que mezcla dos aspectos diferentes de la Biblioteca:

- Almacenar objetos.
- Prestar objetos.

Podemos mejorar este método de alguna manera al añadir un nivel adicional en la jerarquía de herencia, como ilustra la figura 19.9. Esto se libra de una de las relaciones de composición al introducir una clase ElementoBiblioteca. Esta solución es tan buena como lo es utilizando herencia sencilla. Hemos resuelto el protocolo "prestable" de un elemento en nivel aparte en la jerarquía de herencia. Esto es una solución común a este tipo de problema. Como se ilustra en la figura 19.10, un modelo que utiliza interfaces y herencia proporciona una solución más elegante. Las ventajas de esta solución basada en interfaz son las siguientes:

- Todo elemento en la Biblioteca es un ElementoBiblioteca: esto lo hace intuitivo.
- Hemos resuelto la noción de "prestabilidad" en una interfaz aparte, Prestable, que podemos aplicar a ElementosBiblioteca según sea necesario.

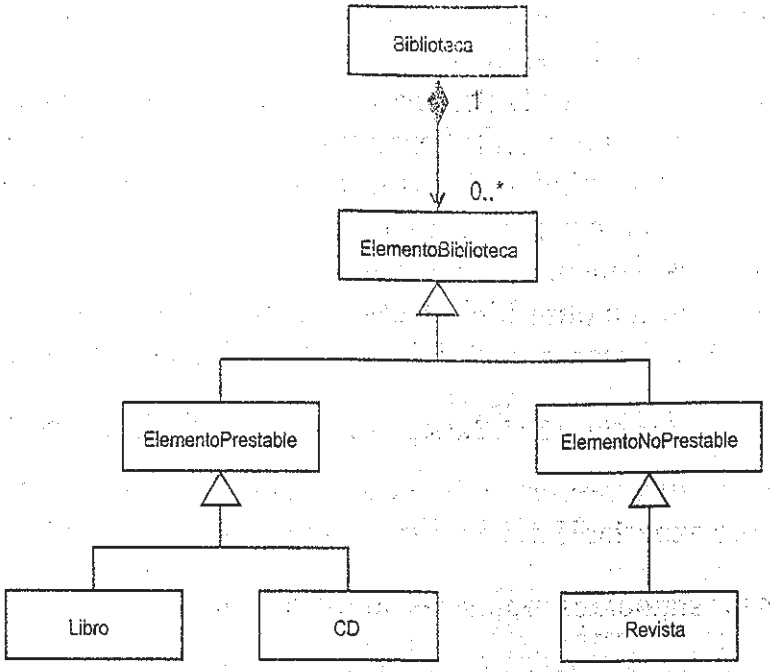

Figura 19.9.

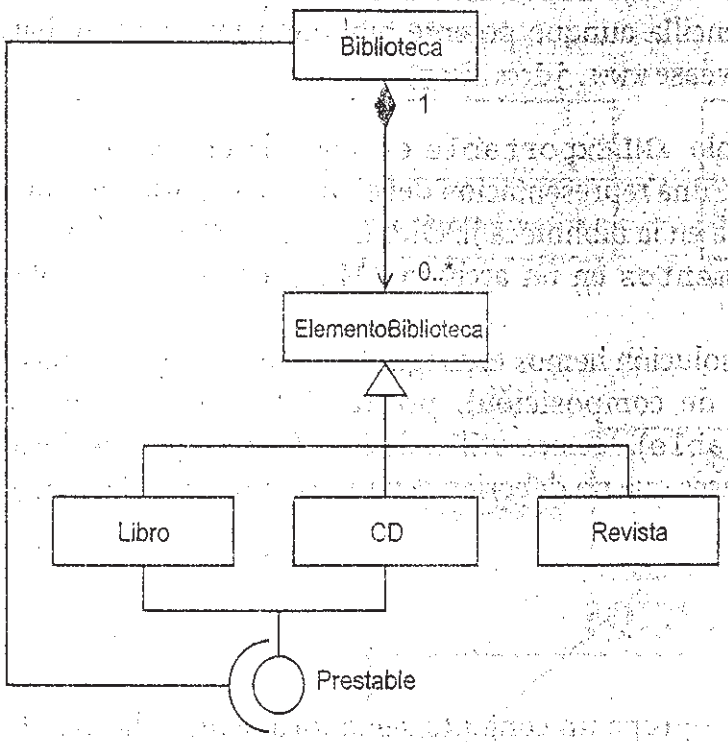

Figura 19.10.

- Tenemos algunas menos clases, cinco clases y una interfaz, frente a siete clases para la otra solución.
- Poseemos algunas menos relaciones de composición; una frente a dos en la otra solución.
- Hay una jerarquía de herencia más sencilla con solamente dos niveles.
- Tenemos menos relaciones de herencia; tres frente a cinco en la otra solución.

De esta forma, la solución basada en la interfaz es más sencilla y tiene una mejor semántica. Situamos elementos como `númeroCatálogo` que tienen todos los `ElementosBiblioteca`, en la clase base `ElementoBiblioteca` para que se puedan heredar, y definimos el protocolo "prestable" aparte en la interfaz `Prestable`.

Para ilustrar la flexibilidad de las interfaces, desarrollemos más este ejemplo. Suponga que necesita exportar detalles de `Libros` y `Revistas` (pero no `CD`) a archivos XML. Los elementos de negocio para esto tienen que permitir el intercambio de información con otras bibliotecas y permitir mostrar el catálogo de material impreso en la Web. Diseñamos la solución de la siguiente manera:

- Introducir una clase `XMLExportador` que realiza la exportación XML.
- Introducir una interfaz, `XMLExportable`, que define el protocolo que cada elemento exportable debe tener para funcionar con `XMLExportador`.

Tenemos los siguientes requisitos no funcionales:

- El lenguaje de implementación debería ser Java.
- La biblioteca JDOM se debería utilizar para el procesamiento XML (JDOM es una sencilla aunque potente biblioteca Java para trabajar con documentos XML; véase `www.jdom.org`).

El protocolo `XMLExportable` es una sola operación, `obtenerElemento()`, que devuelve una representación del elemento exportable como un `Elemento`, una clase definida en la biblioteca JDOM. La clase `XMLExportador` utiliza JDOM para escribir `Elementos` en un archivo XML. La solución completa se muestra en la figura 19.11.

Con esta solución hemos conseguido separar los problemas de almacenamiento (la relación de composición), prestabilidad (`Prestable`) y exportabilidad (`XMLExportable`). Hemos utilizado interfaces para especificar los protocolos comunes de clases que no deberían estar normalmente relacionadas por herencia.

19.6. Puertos

Un puerto agrupa un conjunto semánticamente cohesivo de interfaces proporcionadas y requeridas. Indica un punto específico de la interacción entre un clasificador y su entorno.

El ejemplo en la figura 19.12 ilustra la notación de puerto. Muestra una clase `Libro` que tiene un puerto denominado `presentación`. Este puerto consta de una interfaz requerida, `MostrarMedio`, y una interfaz proporcionada, `Mostrar`. El nombre del puerto es opcional. La figura muestra dos variantes de la notación del puerto. La parte izquierda de la figura muestra el caso normal y la parte derecha muestra una alternativa más concisa. Sin embargo, esta alternativa es solamente aplicable si el puerto tiene un solo tipo de interfaz proporcionada (puede tener cero

o más interfaces requeridas). Muestre el nombre del tipo detrás del nombre del puerto como se muestra en la figura.

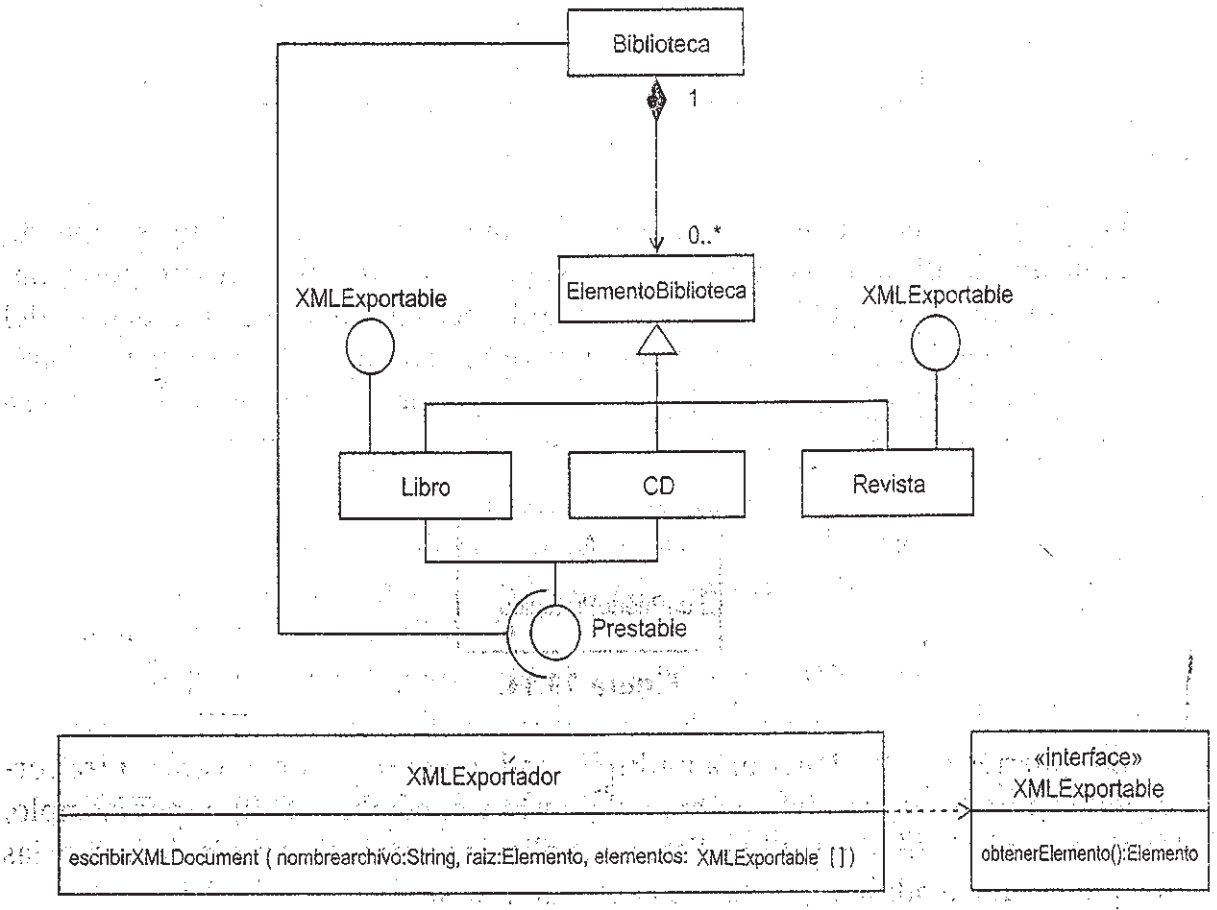

Figura 19.11.

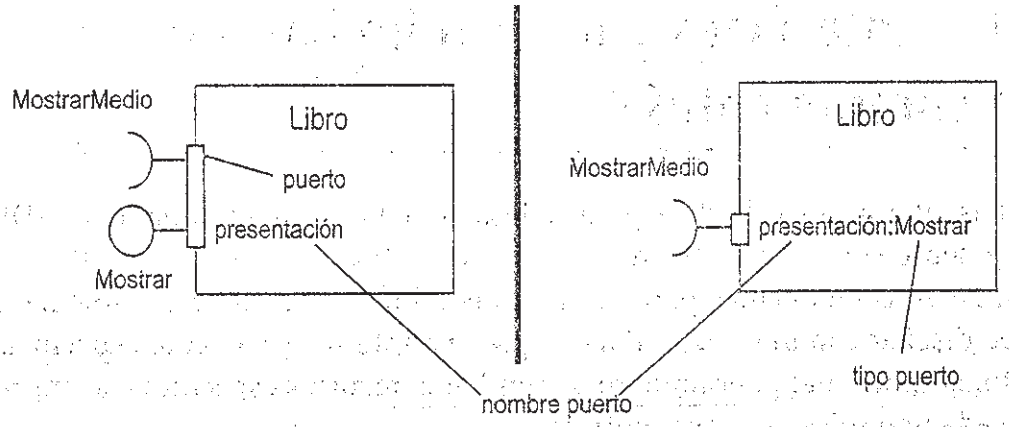

Figura 19.12.

Los puertos son una forma muy útil de estructurar las interfaces proporcionadas o requeridas de un clasificador. También se pueden utilizar para simplificar un diagrama. Por ejemplo, la figura 19.13 muestra una clase `Espectador` que conecta con el puerto `presentación` de la clase `Libro`. Para conectar los puertos, sus interfaces proporcionadas y requeridas deben coincidir. Utilizar puertos es mucho más conciso que mostrar todas las interfaces proporcionadas y requeridas pero puede ser más difícil de leer.

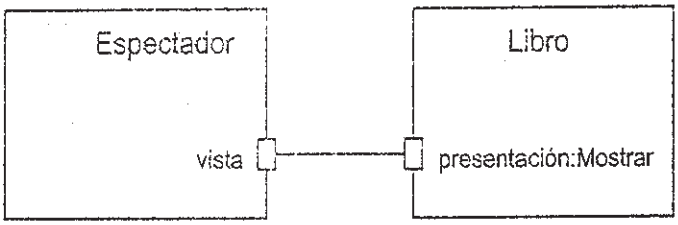

Figura 19.13.

Los puertos pueden tener una visibilidad. Cuando un puerto se dibuja solapando el límite del clasificador, es público, y esto significa que las interfaces proporcionadas y requeridas son públicas. Si el rectángulo del puerto se muestra dentro del límite del clasificador (como ilustra la figura 19.14), entonces el puerto tiene visibilidad protegida (predeterminado) o visibilidad privada. La visibilidad real no se muestra gráficamente pero se graba en la especificación del puerto.

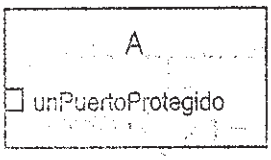

Figura 19.14.

Los puertos pueden tener una multiplicidad y esto se puede mostrar entre corchetes cuadrados detrás del nombre del puerto y el nombre del tipo (por ejemplo, `presentación:Mostrar[1]`). Esta multiplicidad indica el número de instancias del puerto que tendrán las instancias del clasificador.

19.7. Interfaces y desarrollo basado en componentes

Las interfaces son la clave para el desarrollo basado en componentes (DBC). Esto trata sobre construir software a partir de partes. Si desea crear software flexible basado en componentes para lo que puede incorporar nuevas implementaciones, debe diseñar con interfaces. Puesto que una interfaz solamente especifica un contrato, permite cualquier número de implementaciones específicas, siempre y cuando cada uno se ajuste a ese contrato.

En los siguientes apartados examinamos los componentes y luego exploramos cómo los componentes y las interfaces se pueden combinar en el DBC.

19.8. ¿Qué es un componente?

La especificación UML 2.0 [UML2S] indica "un componente representa una parte modular de un sistema que encapsula sus contenidos y cuya manifestación se

reemplaza dentro de su entorno". Un componente actúa como una caja negra cuyo comportamiento externo está completamente definido por sus interfaces proporcionadas y requeridas. Debido a esto, un componente se puede reemplazar por otro que soporte el mismo protocolo.

Los componentes pueden tener atributos y operaciones y pueden participar en relaciones de asociación y generalización. Los componentes son clasificadores estructurados y pueden tener una estructura interna que constan de partes y conectores. Hemos presentado los clasificadores estructurados en el capítulo 18 y si todavía no lo ha leído, le aconsejamos que lo lea ahora antes de continuar con este apartado.

Los componentes pueden representar algo que se puede instanciar en tiempo de ejecución, como un EJB (Enterprise JavaBean) o pueden representar un constructor lógico, como un subsistema, que solamente se instancia indirectamente por virtud de sus partes siendo instanciadas.

Un componente se puede mostrar por uno o más artefactos. Un artefacto representa algo en el mundo físico como un archivo fuente. Por ejemplo, un componente EJB podría mostrarse por un archivo JAR (Java Archive). Tratamos los artefactos con más detalle en el capítulo 24.

El diagrama de componente puede mostrar componentes, dependencias entre componentes y la forma en la que los clasificadores están asignados a los componentes. Como se ha ilustrado en la figura 19.15, un componente se dibuja como un cuadro con el estereotipo `<<component>>` y/o un icono de componente en su esquina superior derecha. Los componentes pueden tener interfaces proporcionadas y requeridas y puertos.

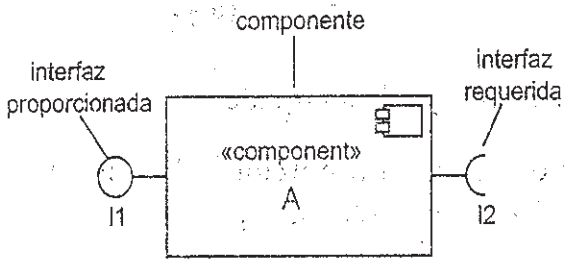

Figura 19.15.

Un componente puede tener una estructura interna. Puede mostrar las partes anidadas dentro del componente (véase la figura 19.16) o externamente, conectadas a éste por relaciones de dependencia. Ambas formas son sintácticamente equivalentes, aunque creemos que la notación anidada es más clara.

Cuando un componente tiene estructura interna, normalmente delegará las responsabilidades definidas por sus interfaces a uno o más de sus partes internas. En la figura 19.16, el componente A proporciona la interfaz `I1` y requiere la interfaz `I2`. Encapsula dos partes de tipo `b` y `c`. Delega el comportamiento especificado por sus interfaces proporcionadas y requeridas a `b` y `c` respectivamente.

Los componentes pueden depender de otros componentes. Para desacoplar los componentes, siempre media la dependencia con una interfaz. Cuando un compo-

nente requiere una interfaz, puede mostrar esto como una dependencia entre el componente y la interfaz o puede utilizar un conector de ensamblaje según se ilustra en la figura 19.17.

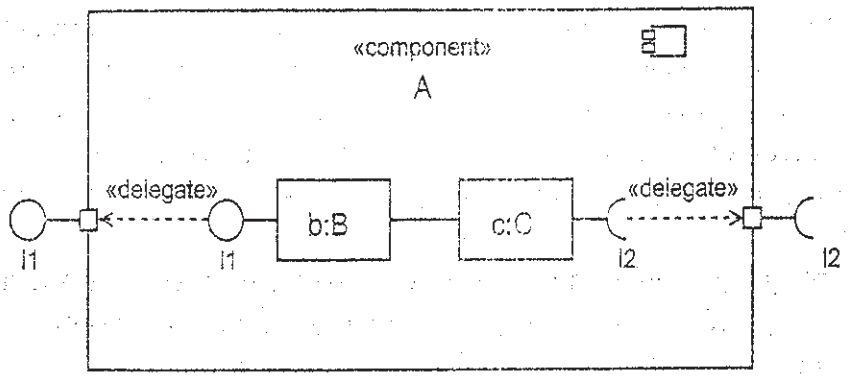

Figura 19.16.

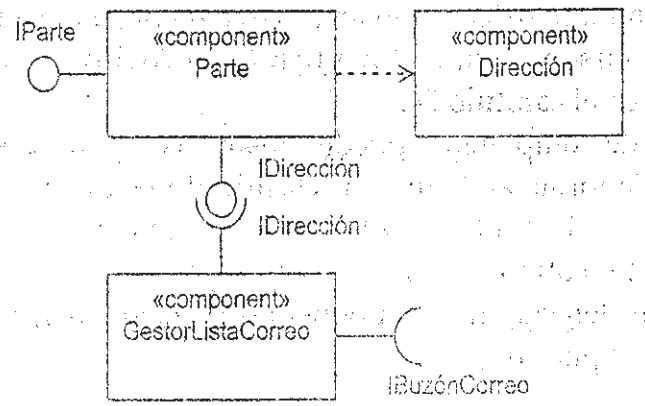

Figura 19.17.

En la figura 19.17 puede ver lo siguiente:

- El componente `Parte` proporciona dos interfaces del tipo `IParte` y `IDirección`. Estas interfaces se representan como círculos.
- El componente `GestorListaCorreo` requiere dos interfaces del tipo `IDirección` e `IBuzónCorreo`. Éstas se representan como sockets.
- Existe un conector de ensamblaje entre el componente `Parte` y el componente `GestorListaCorreo`. Esto muestra que `GestorListaCorreo` se está comunicando con el componente `Parte` por medio de la interfaz proporcionada `IDirección`.
- En este modelo, el componente `Parte` actúa como una fachada para desacoplar el componente `GestorListaCorreo` de los detalles del componente `Dirección`.

Los componentes a menudo se muestran simplemente como cuadros negros con sus interfaces proporcionadas y requeridas anexadas a éstas. Sin embargo, también puede tener una vista de caja blanca de un componente según se muestra en la

figura 19.18. Esta vista presenta los detalles internos del componente. Puede mostrar cualquier interfaz proporcionada, requerida, realizaciones o artefactos asociados.

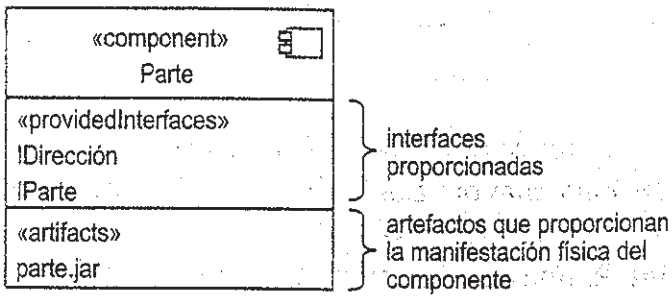

Figura 19.18.

19.9. Estereotipos de componente

Los componentes son posiblemente el elemento UML más estereotipado. Esto es porque un componente se puede utilizar para representar muchos tipos diferentes de elementos. UML 2 proporciona un pequeño conjunto de estereotipos estándar de componentes que se listan en la tabla 19.2. Examinamos uno de estos, `<<subsystem>>`, en más detalle en el siguiente apartado.

Tabla 19.2.

Estereotipo	Semántica
`<<buildComponent>>`	Un componente que define un conjunto de elementos para fines organizativos o de desarrollo a nivel de sistema.
`<<entity>>`	Un componente de información persistente que representa un concepto de negocio.
`<<implementation>>`	Una definición de componentes que no tiene especificación; es una implementación para una `<<specification>>` aparte con la que tiene una dependencia.
`<<specification>>`	Un clasificador que especifica un dominio de objetos sin definir la implementación física de esos objetos, por ejemplo, un componente estereotipado por `<<specification>>` solamente tiene interfaces proporcionadas y requeridas y no clasificadores de realización.
`<<process>>`	Un componente basado en transacción.
`<<service>>`	Un componente funcional sin estado que computa un valor.
`<<subsystem>>`	Una unidad de descomposición jerárquica para grandes sistemas.

Si utiliza perfiles UML en su modelado, estos probablemente definen sus propios estereotipos de componente.

19.10. Subsistemas

Un subsistema es un componente que actúa como una unidad de descomposición para un sistema mayor. Los subsistemas se dibujan como componentes estereotipados `<<subsystem>>`.

Un subsistema es una construcción lógica que se utiliza para descomponer un sistema grande en bloques manejables. Los subsistemas no se pueden instanciar en momento de ejecución, pero sí sus contenidos.

Desde la perspectiva de UP, los subsistemas son un concepto de estructuración clave. Desglosar un sistema en subsistemas resuelve un problema de desarrollo difícil en muchos subproblemas más pequeños y manejables. Ésta es la clave para el desarrollo de sistemas con éxito utilizando UP. Las interfaces van de la mano con los subsistemas como se muestra en la figura 19.19. En este ejemplo, el subsistema GUI solamente sabe de las interfaces `GestorCliente`, `GestorCuenta` y `GestorPedido`. No sabe nada del subsistema de `LógicaNegocio` que lo implementa. Esto significa que podría en principio reemplazar el subsistema `LógicaNegocio` con otro subsistema o incluso con varios subsistemas, siempre y cuando juntos proporcionen el mismo conjunto de interfaces. De forma similar, podría reemplazar el subsistema `GUI` con un subsistema GUI diferente siempre y cuando requiera el mismo conjunto de interfaces. Utilizar las interfaces de esta forma desacopla subsistemas y crea flexibilidad de arquitectura.

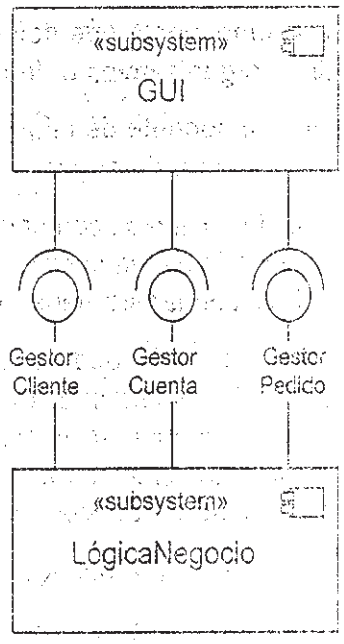

Figura 19.19.

Las interfaces conectan sus sistemas para crear una arquitectura de sistema.

19.11. Encontrar interfaces

A medida que diseña un sistema o parte de un sistema, merece la pena examinar el modelo de diseño para intentar encontrar algunas interfaces. Esto es bastante fácil de hacer; se procede como sigue:

- Cuestione cada asociación: Examine cada una de ellas y formule la pregunta. "¿Debería esta asociación ser a una clase particular de objetos o debería ser más flexible que esto?" Si decide que la asociación realmente necesita ser más flexible de lo que sería si estuviera unida a una clase particular, considere utilizar una interfaz.
- Cuestione cada envío de mensaje: Examine cada uno de ellos y formule la pregunta, "¿Debería el envío de este mensaje ser a objetos de solamente una clase o debería ser más flexible que esto?" Si debería ser más general (por ejemplo, puede pensar en casos en los que el mismo mensaje se podría enviar a objetos de otras clases), considere utilizar una interfaz.
- Resuelva grupos de operaciones que se podrían reutilizar en cualquier otra parte. Por ejemplo, si muchas clases en su sistema necesitan imprimirse en ciertos dispositivos de salida, piense en diseñar una interfaz `Imprimir`.
- Resuelva conjuntos de operaciones que se repiten en más de una clase.
- Resuelva conjuntos de atributos que se repiten en más de una clase.
- Busque clases que desempeñan el mismo rol en el sistema, el rol puede indicar una posible interfaz.
- Busque posibilidades para expansión futura. Algunas veces puede diseñar sistemas que se pueden expandir fácilmente en el futuro. La cuestión clave es, "en el futuro ¿se necesitarán añadir otras clases al sistema?" Si la respuesta es sí, trate de definir una o más interfaces que definirán el protocolo para añadir estas nuevas clases.
- Examine dependencias entre componentes; medie éstas por conectores de ensamblaje donde sea posible.

Como puede ver, puede haber muchas oportunidades para utilizar interfaces. Examinamos los detalles de diseñar con interfaces en el siguiente apartado.

19.12. Diseñar con interfaces

Cuando diseña un sistema, siempre es de utilidad si todo se comporta lo más uniformemente posible. Utilizando interfaces, puede diseñar protocolos comunes que se podrían realizar por muchas clases o componentes. Un buen ejemplo de esto

es un sistema en el que estuvimos trabajando para proporcionar una interfaz común a varios sistemas legacy. El problema era que cada sistema tenía un protocolo diferente de comunicación. Fuimos capaces de ocultar esta complejidad detrás de una sola interfaz que constaba de las operaciones `abrir(...)`, `leer(...)`, `escribir(...)` y `cerrar(...)`.

Aquí tiene otro ejemplo. Si considera un sistema que modela una organización (un sistema de recursos humanos, por ejemplo), existen muchas clases que tienen un nombre y dirección, por ejemplo, `Persona`, `UnidadOrganizativa`, `Trabajo`. Todas estas clases pueden desempeñar un rol común. Tiene sentido que todas estas clases deberían tener la misma interfaz para gestionar los detalles del nombre y la dirección. Podría definir por lo tanto una interfaz `NombreYDirección` que todos podrían realizar. Otras soluciones a este problema pueden utilizar herencia, pero la solución de interfaz puede ser algunas veces más flexible.

Merece la pena recordar que las clases pueden tener asociaciones reflexivas (consigo mismas) y que puede haber roles que son internos a la clase. Éstos son también posibles candidatos para interfaces.

Un uso potente de las interfaces es proporcionar la posibilidad de conectar elementos a los sistemas. Una de las formas de hacer que los sistemas sean flexibles al cambio es diseñar el sistema de modo que las extensiones se puedan conectar fácilmente. Las interfaces son la clave para esto. Si puede diseñar sistemas en torno a interfaces, entonces las asociaciones y envíos de mensaje ya no están unidos a objetos de una clase determinada sino que están unidos a una interfaz determinada. Esto facilita el añadir nuevas clases a un sistema ya que las interfaces definen los protocolos que las nuevas clases deben soportar para conectarse sin problemas.

Los algoritmos de plug-in son un buen ejemplo de módulos de software que podría querer conectar. Hace tiempo estuvimos trabajando en un sistema que realizaba un cálculo complejo en un conjunto de datos. Los usuarios querían experimentar con el algoritmo de cálculo para tratar de localizar la estrategia óptima. Sin embargo, el sistema no se ha escrito teniendo esto en cuenta y cualquier pequeño cambio en el algoritmo llevó varios días ya que el código existente se tuvo que cambiar y se tuvo que reconstruir el sistema. Trabajamos con uno de los diseñadores para que el sistema utilizara una interfaz para los algoritmos que se pueden conectar. Después de este trabajo se pueden comprobar nuevos algoritmos en cuestión de horas. De hecho, pudimos incluso cambiar algoritmos mientras el sistema estaba todavía funcionando.

19.12.1. El patrón Fachada

Ocultar subsistemas complejos detrás de una interfaz sencilla y bien definida se conoce como el patrón Fachada. Esto está documentado en [Gamma 1]. Este libro es un tesoro de patrones de diseño reutilizables que se pueden utilizar en contextos muy diferentes en modelos de diseño. Gamma dice esto sobre el patrón Fachada: "Estructurar un sistema en subsistemas ayuda a reducir la complejidad. Un objetivo común de diseño es minimizar la comunicación y dependencias entre subsistemas.

Una forma de conseguir este objetivo es introducir un objeto fachada que proporciona una única interfaz simplificada a las facilidades más generales de un subsistema". El patrón Fachada permite la ocultación de información; puede ocultar detalles complejos del funcionamiento interno de un subsistema detrás de una interfaz sencilla. Esto reduce la complejidad del sistema y le permite controlar y gestionar el acoplamiento entre los subsistemas.

Las interfaces utilizadas como fachada se pueden utilizar para crear uniones en un sistema. Realice esto de la siguiente forma:

- Identificar partes cohesivas del sistema.
- Empaquetar éstas en un `<<subsystem>>`.
- Definir una interfaz para ese subsistema.

19.12.2. Arquitectura y el patrón de disposición en capas

La colección de subsistemas de diseño e interfaces constituye la arquitectura de alto nivel de un sistema. Sin embargo, para que esta arquitectura sea fácil de entender y mantener, necesita organizar la colección de subsistemas e interfaces de forma coherente. Puede realizar esto al aplicar un patrón de arquitectura conocido como disposición en capas. El patrón de disposición en capas organiza los subsistemas de diseño e interfaces en capas donde los subsistemas en cada capa son semánticamente cohesivos. La esencia de crear una arquitectura robusta en capas es para gestionar el acoplamiento entre subsistemas al:

- Introducir nuevas interfaces donde sea necesario.
- Reempaquetar clases en nuevos subsistemas en una forma que reduzca el acoplamiento entre subsistemas.

Las dependencias entre capas se deben de gestionar con cuidado ya que estas dependencias representan el acoplamiento entre capas. Idealmente quiere que las capas estén lo más desacopladas posibles, por lo tanto, trate de asegurarse de que:

- Las dependencias van en un sentido.
- Todas las dependencias están mediadas por interfaces.

Es decir, un subsistema en una capa debería requerir interfaces de la capa debajo de ésta y proporcionar interfaces a la capa encima de ésta allí donde sea posible.

Existen muchas formas de producir arquitecturas en capas y puede tener tantas capas como tenga sentido. Sin embargo, el patrón básico es una división entre las capas de presentación, la lógica de negocio y las capas de utilidad. Como muestra la figura 19.20 es bastante común subdividir aún más la capa de lógica de negocio. En este caso, tenemos dos capas, dominio y servicios. La capa de dominio contiene

subsistemas específicos a esta determinada aplicación, y la capa de servicios contiene subsistemas que pueden ser reutilizables en otras aplicaciones.

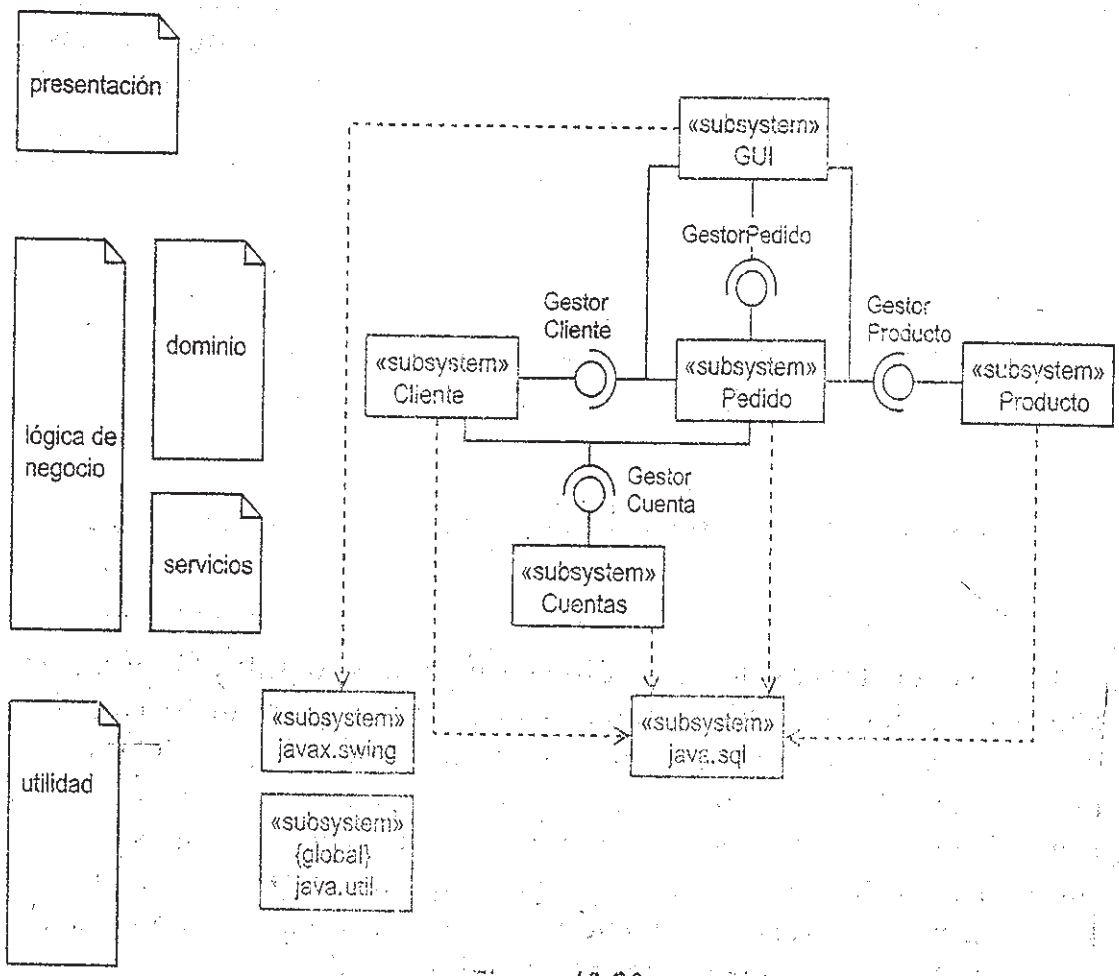

Figura 19.20.

Siempre que sea posible, es mejor diseñar para una interfaz. En la figura 19.20, puede ver que los subsistemas que hemos diseñado están todos conectados vía interfaces. Sin embargo, los paquetes Java están simplemente conectados con dependencias, aunque cada uno pone disponible varias interfaces. La razón para esto es que aunque mostrar sus interfaces es interesante y de utilidad, mostrar las interfaces que se encuentran disponibles por bibliotecas estándar Java no parece servir a ningún propósito de utilidad. Observe también que el paquete `java.util` que contiene componentes genéricos como `Strings` se utiliza en todas partes y por lo tanto está marcado con una etiqueta `{global}`. Esta etiqueta indica que todos los contenidos públicos del paquete son visibles en todas partes. Es una forma pragmática de indicar que ha elegido no mostrar las dependencias a este paquete porque no proporcionan ninguna información de utilidad.

19.13. Ventajas y desventajas de las interfaces

Cuando diseña con clases, está limitando el diseño a implementaciones específicas. Pero cuando diseña con interfaces, diseña para contratos que se pueden reali-

zar por muchas implementaciones diferentes. Diseñar para contratos libera el modelo (y, por tanto, el sistema) de dependencias de implementación y por lo tanto aumenta su flexibilidad y extensibilidad.

Diseñar con interfaces le permite reducir el número de dependencias entre clases, subsistemas y componentes y empieza a proporcionar control sobre la cantidad de acoplamiento en un modelo. En un sentido real, el acoplamiento es el peor enemigo del desarrollador de objetos, ya que los sistemas altamente acoplados son difíciles de entender, mantener y evolucionar. Un uso apropiado de las interfaces puede ayudar a reducir el acoplamiento y separar el modelo en subsistemas cohesivos.

Sin embargo, existen desventajas al utilizar interfaces. Por lo general, siempre que hace algo más flexible, lo hace más complejo. Por lo tanto, cuando diseña con interfaces, busca un equilibrio entre flexibilidad y complejidad. En principio, podría hacer que cada operación de cada clase fuera una interfaz; sencillamente no entendería dicho sistema. Existe también un coste de rendimiento a la flexibilidad, pero ésta es una consideración menor comparada con el aumento en complejidad.

Cuando diseña un sistema trata de capturar un conjunto muy definido de semántica de negocio en software. Parte de esta semántica es fluida y cambia bastante rápidamente, mientras que otra es relativamente estable. Necesita flexibilidad para tratar con aspectos fluidos, pero puede simplificar los sistemas al prescindir de cierta cantidad de flexibilidad para las partes más estables. En cierto sentido, éste es uno de los secretos de un buen análisis y diseño orientado a objetos, identificar las partes fluidas y estables de un sistema y modelar cada una de ellas en consecuencia.

Para ser franco, es más importante que un sistema esté correctamente modelado que esté modelado flexiblemente. Concéntrese siempre en modelar correctamente la semántica clave de negocio del sistema y luego piense en la flexibilidad. Mantenga las interfaces amigables y sencillas.

19.14. ¿Qué hemos aprendido?

Las interfaces permiten que el software se diseñe según un contrato en lugar de según una implementación específica. Ha aprendido lo siguiente:

- La actividad UP `Diseñar un subsistema` está relacionada con desglosar un sistema en subsistemas: partes que son tan independientes como sea posible.
  - Las interacciones entre subsistemas están mediadas por interfaces.
- Una interfaz especifica un conjunto de características públicas.
  - Las interfaces separan la especificación de funcionalidad de la implementación.
  - Las interfaces se pueden anexar a clases, subsistemas, componentes y cualquier otro clasificador y definen los servicios ofrecidos por estos.

- Si un clasificador dentro de un subsistema realiza una interfaz pública, el subsistema o componente también realiza la interfaz pública.
- Cualquier cosa que realice una interfaz está de acuerdo en ajustarse al contrato definido por el conjunto de operaciones especificadas en la interfaz.

- Semántica de la interfaz: El clasificador de realización tiene las siguientes responsabilidades para cada característica:
  - Operación: Debe tener una operación con la misma firma y semántica.
  - Atributo: Debe tener operaciones públicas para establecer y obtener el valor del atributo:
    - El clasificador de realización no es obligatorio para tener el atributo, pero se debe comportar como si lo tuviera.
  - Asociación: Debe tener una asociación con el clasificador destino:
    - Si una interfaz especifica una asociación con otra interfaz, los clasificadores de implementación de estas interfaces deben tener una asociación entre ellos.
  - Restricción: Debe soportar la restricción.
  - Estereotipo: Tiene el estereotipo.
  - Valor etiquetado: Tiene el valor etiquetado.
  - Protocolo: Debe realizar el protocolo.
- Diseñar para una implementación:
  - Clases específicas están conectadas.
  - Para mantener las cosas sencillas (aunque rígidas), diseñe para una implementación.
- Diseñar para un contrato:
  - Una clase está conectada a una interfaz que puede tener muchas posibles realizaciones.
  - Para hacer las cosas más flexibles (pero posiblemente más complejas), diseñe para un contrato.
- Interfaz proporcionada: Una interfaz proporcionada por un clasificador:
  - El clasificador realiza la interfaz.
  - Utilice la notación de estilo de "clase" cuando necesite mostrar las operaciones en el modelo.
  - Utilice la notación abreviada cuando quiera mostrar la interfaz sin operaciones.

- Interfaz requerida: Una interfaz requerida por un clasificador:
  - El clasificador requiere otro clasificador que realiza la interfaz.
  - Muestra una dependencia a una interfaz de estilo de clase, una interfaz abreviada o utiliza un conector de ensamblaje.
- Conector de ensamblaje: Une interfaces requeridas y proporcionadas.
- Realización de interfaz vs. herencia.
  - Realización de interfaz:"realiza un contrato especificado por".
  - Herencia: "es un".
  - La realización de interfaz y herencia generan polimorfismo.
  - Utilice interfaces para especificar los protocolos comunes de clases que no deberían estar normalmente relacionados por herencia.
- Puerto: Agrupa un conjunto semánticamente cohesivo de interfaces proporcionadas y requeridas:
  - Pueden tener un nombre, tipo y visibilidad.
- Componente: Una parte modular de un sistema que encapsula sus contenidos y cuya manifestación es reemplazable dentro de su entorno:
  - Puede tener atributos y operaciones.
  - Puede participar en relaciones.
  - Puede tener estructura interna.
  - Su comportamiento externo está completamente definido por sus interfaces proporcionadas y requeridas.
  - Los componentes manifiestan uno o más artefactos.
- El desarrollo basado en componentes (DBC) trata la construcción de software a partir de partes:
  - Utiliza interfaces para hacer que los componentes sean "conectables".
  - Al diseñar para una interfaz, permite la posibilidad de realizaciones diferentes por muchos componentes diferentes.
- Los componentes pueden representar:
  - Una entidad física (como un componente EJB).
  - Una entidad lógica (como un subsistema).
- Estereotipos estándar de componentes:
  - `<<buildComponent>>`: Un componente que define un conjunto de elementos para fines organizativos o de desarrollo a nivel de sistema.

- <<entity>>: Un componente persistente de información que representa un concepto de negocio.
- <<implementation>>: Un componente que no tiene especificación; es una implementación para una <<specification>> aparte con la que tiene una dependencia.
- <<specification>>: Un clasificador que especifica un dominio de objetos sin definir la implementación física de esos objetos. Por ejemplo, un componente estereotipado por <<specification>> solamente tiene interfaces proporcionadas y requeridas y ningún clasificador de realización.
- <<process>>: Un componente basado en transacciones.
- <<service>>: Un componente funcional sin estado que computa un valor.
- <<subsystem>>: Una unidad de descomposición jerárquica para sistemas grandes.

- Subsistema: Un componente que actúa como una unidad de descomposición para un sistema más amplio.
  - Un componente estereotipado <<subsystem>>.
  - Un constructor lógico que se utiliza para descomponer un sistema mayor en bloques manejables.
  - No se puede instanciar en tiempo de ejecución, pero sí sus contenidos.
  - Desglosar un sistema en subsistemas es clave para un desarrollo del sistema con éxito utilizando UP.
  - Los subsistemas se utilizan para:
    - Separar aspectos de diseño.
    - Representar componentes mayores.
    - Envolver sistemas legacy.
  - Utilice interfaces para ocultar los detalles de implementación de los subsistemas:
  - El patrón Fachada oculta una implementación compleja dentro de una sencilla interfaz.
  - El patrón por niveles organiza los subsistemas en niveles semánticamente cohesivos:
    - Las dependencias entre capas solamente deberían ir en un sentido.
    - Todas las dependencias entre capas deberían estar mediadas por una interfaz.

- Las capas de ejemplo incluyen las capas de presentación, lógica de negocio y utilidad.

- Encontrar interfaces:
  - Cuestionar asociaciones.
  - Cuestionar envíos de mensaje.
  - Resolver grupos de operaciones reutilizables.
  - Resolver grupos de operaciones que se repiten.
  - Resolver grupos de atributos que se repiten.
  - Buscar clases que desempeñan el mismo rol en el sistema.
  - Buscar posibilidades para expansión futura.
  - Buscar dependencias entre componentes.