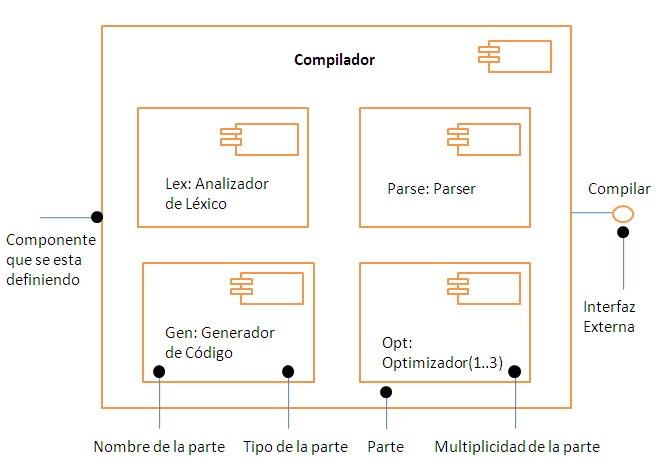
**Estructura interna**

Un componente puede implementarse como una única pieza de código, pero en los sistemas más grandes lo deseable es poder construir componentes más grandes utilizando componentes menores como bloques de construcción.

La estructura interna del componente junto con las conexiones entre ellos. En muchos casos, las partes internas pueden ser instancias de componentes menores que se enlazan estáticamente a través de sus puertos para proporcionar el comportamiento necesario sin necesidad de que el modelador especifique una lógica adicional.

Una parte es una unidad de implementación de un componente. Una parte tiene nombre y un tipo. En una instancia de un componente, hay una o más instancias que se corresponden con cada parte que tiene el tipo especificado. Una parte tiene una multiplicidad dentro del componente. Si la multiplicidad de la parte es mayor que uno, puede haber más de una instancia de la parte en una instancia particular del componente. Si la multiplicidad es distinta de un simple número entero, el número de instancias de la parte puede variar de una instancia del componente a otra. Una instancia de un componente se crea con el mínimo número de partes; las partes adicionales se añaden posteriormente. Un atributo de una clase es una especie de parte: tiene un tipo y una multiplicidad, y cada instancia de la clase tiene una o más instancias del tipo dado.

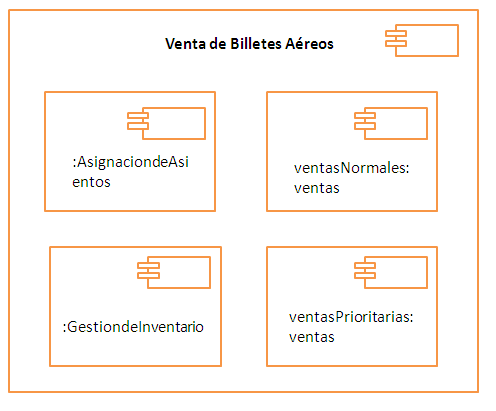


**Partes dentro de un Componente**

El la figura anterior se muestra un componente de un compilador construido a partir de cuatro tipos de partes. Hay un analizador léxico. Un analizador sintáctico, un generador de código y uno de tres optimizadores. Algunas versiones más completas del compilador pueden configurarse con distintos niveles de optimización; dentro de una versión particular, puede seleccionarse el optimizador apropiado en tiempo de ejecución.

Hay que tener en cuenta que una parte no es lo mismo que una clase. Cada parte es potencialmente distinguible por su nombre, como cada atributo de una clase. Puede haber más de una parte del mismo tipo, pero pueden diferenciarse por los nombres y, presumiblemente cumplen funciones distintas dentro del componente.

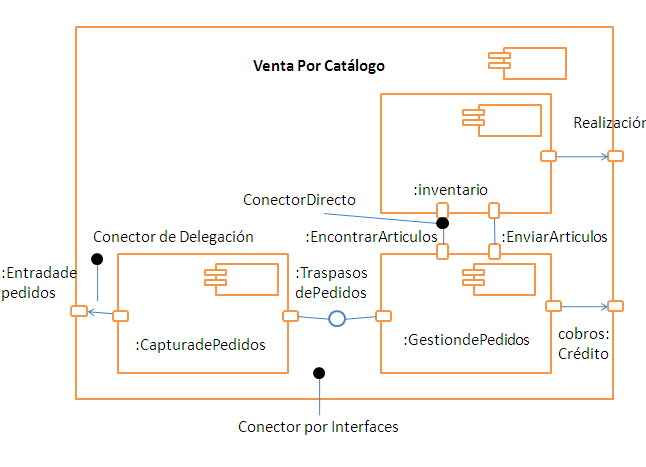
En la Siguiente figura se muestra un ejemplo del mismo, donde un componente Ventas de Billetes puede tener diferentes partes Ventas, para los pasajeros frecuentes y para los pasajeros normales; ambos trabajan igual pero la parte de pasajeros frecuentes solo está disponible para ciertos clientes especiales y conlleva una menor probabilidad de tiempo en la fila además de incluir algunos privilegios adicionales. Como ambos componentes tienen el mismo tipo, han de tener nombres distintos para poder distinguirse. Los otros dos componentes de tipo AsignacióndeAsientos y GestiondeInventario no requieren nombres ya que solo hay uno de cada tipo dentro del componente Venta de Billetes Aéreos.



**Partes del Mismo Tipo**

Por otra parte si las partes son componentes con puertos, pueden enlazarse a través de éstos. La regla es bien sencilla: dos componentes pueden conectarse si uno proporciona una interfaz que el otro requiere. Conectar los puertos significa que el puerto que requiere invocará el puerto que proporciona para obtener servicios. La ventaja de los puertos e interfaces es que no hace falta saber nada más; si las interfaces son compatibles los puertos pueden conectarse. Una herramienta podría generar automáticamente un código de invocación desde un componente hacia otro. También puede reconectar el componente a otros que proporcionen la misma interfaz, si estos están disponibles. Una línea entre dos puertos es un *conector*. En la instancia del componente global, el conector representa un enlace o un enlace transitorio. Un enlace transitorio representa una relación de uso entre dos componentes. En vez de corresponder a una asociación común, el enlace transitorio puede ser proporcionado por un parámetro de procedimiento o una variable local que sirve como destinataria de una operación. La ventaja de los puertos e interfaces e que los dos componentes no tienen que conocer nada acerca del otro en tiempo de diseño, siempre y cuando sus interfaces sean compatibles.

Los conectores pueden conectarse de dos formas, como lo muestra la siguiente figura. Si dos componentes se enlazan de manera explícita, ya sea directamente o a través de sus puertos, simplemente se dibuja una línea entre ellos o los puertos. Por otro lado si dos componentes se conectan porque tienen interfaces compatibles se puede usar una notación de una junta circular para mostrar que no hay relación inherente entre los componentes.

También se pueden conectar puertos internos a puertos externos del componente global. Esto se denomina un colector de delegación ya que los mensajes sobre el puerto externo son delegados al puerto interno. Esto se represente con una flecha del puerto interno al puerto externo. Se puede ver de dos maneras según se prefiera. Por un lado, el puerto interno es el mismo que el puerto externo; ha sido movido a la frontera y se le ha permitido asomarse al exterior. En el segundo enfoque, cualquier mensaje hacia el puerto externo es transmitido inmediatamente hacia el puerto interno, y viceversa. En realidad no importa, el comportamiento es el mismo.

La figura muestra un ejemplo con puertos internos y diferentes tipos de conectores. Las solicitudes externas sobre el puerto EntradaDePedidos se delegan hacia el puerto interno del subcomponente CapturaDePedidos. Este componente a su vez envía su salida al puerto TrasopasoDePedidos, que está conectado con una junta circular al subcomponente GestiondePedidos. Este tipo de conexión implica que no existe un conocimiento especial entre ambos componentes; la salida podría estar conectada a cualquier componente que cumpliese la interfaz TraspasoDePedidos. El componente GestionDePedidos se comunica con el componente Inventario para encontrar los artículos que quedan en el almacén. Esto representa un conector directo; como no se muestra ninguna interfaz, se está sugiriendo que la conexión está más fuertemente acoplada. Una vez encontrados los artículos en el almacén, el componente GestionDePedidos accede a un servicio Externo Crédito; estos se muestra por el conector de delegación hacia el puerto externo denominado Cobros.

Una vez el servicio externo Crédito responde, el componente GestiosDePedidos se comunica con un puerto diferente EnvíodeArticulos del componente inventario para preparar el envío del pedido. El componente Inventario accede a un servicio externo Realización para ejecutar finalmente el envío.

Debe notarse que el diagrama de componentes muestra la estructura y los caminos potenciales de los mensajes del componente. El diagrama de componentes por si mismo no revela la secuencia de mensajes a través del componente. La secuenciación y otros tipos de formación dinámica pueden representarse a través de diagramas.

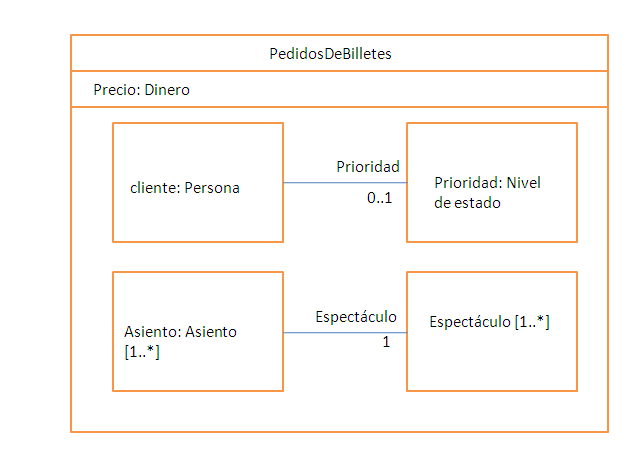
**Técnicas comunes de Modelado**

* **Modelado de Instancias discretas**

Una clase estructurada puede utilizarse para modelar estructuras de datos en las que las partes tienen conexiones contextuales que solo se aplican dentro de ella. Los atributos o las asociaciones normales pueden definir partes compuestas de una clase, pero puede que las partes no estén relacionadas en un diagrama de clases plano. Una clase cuya estructura interna se muestra con partes y conectores evita este problema.

Para modelar clases estructurada:

* Hay que identificar partes internas de la clase y sus tipos.
* Hay que dar a cada parte un nombre que indique su propósito dentro de la clase estructurada, no su tipo genérico.
* Hay que dibujar conectores entre partes que se comunican o tienen relaciones contextuales.
* Hay que sentirse libre para usar otras clases estructuradas como tipos de las partes, pero recordando que no se pueden hacer conexiones directas a las partes dentro de otra clase; hay que conectar a través de sus puertos externos.

La siguiente figura muestra el diseño de la clase estructurada *PedidosDeBilletes*. Esta clase tiene cuatro partes y un atributo normal, *Precio*. El cliente es un objeto *Persona.* El cliente puede tener o no tener una estatus de prioridad, así que la parte prioridad se representa con una multiplicidad de 0…1; el conector desde *cliente* a prioridad también tiene la misma multiplicidad. Hay uno o más asientos reservados; asiento tiene un valor de multiplicidad. No es necesario mostrar un conector desde cliente a asiento porque de todos modos están en la misma clase estructurada. Obsérvese que espectáculo se ha dibujado con un borde discontinuo. Esto significa que la parte es una referencia a un objeto que no está contenido dentro de la clase estructurada. La referencia se crea y se destruye con una instancia de la clase PedidosDeBilletes, pero las instancias de Espectáculo son independientes de la clase PedidosDeBilletes. La parte asiento está conectada a la referencia espectáculo porque el pedido puede incluir asientos para más de un espectáculo, y cada reserva de asiento debe estar conectada a un espectáculo específico. De la multiplicidad del conecto podemos deducir que cada reserva *Asientos* estáconectada exactamente a un objeto E*spectáculo*.

**Clase Estructurada**

* **Modelado de una API**

Los desarrolladores que ensamblan un sistema a partir de partes componentes a menudo desean ver las interfaces de programación de aplicaciones (API) que se utilizan para enlazar las partes. Las API representan las líneas de separación programáticas de un sistema y se pueden modelar mediante interfaces y componentes.

Una API es sencillamente una interfaz realizada por uno o más componentes. Los desarrolladores realmente solo se preocupan de la propia interfaz. El componente que realice las operaciones de interfaz no es relevante siempre y cuando haya *algún* componente que las realice. Sin embargo, desde la perspectiva de la gestión de la configuración de un sistema, estas realizaciones son importantes porque hay que asegurar que cuando se publica una API hay alguna realización disponible que cumple las obligaciones de la API. Afortunadamente, con UML podemos modelas ambas perspectivas.

Para modelar una API:

* Hay que identificar las líneas de separación del sistema y modelar cada una como una interfaz, recogiendo los atributos y operaciones que forman su frontera.
* Hay que exponer solo aquellas propiedades de la interfaz que son importantes para comprender dentro del contexto dado; en otro caso, hay que esconder sus propiedades, manteniéndolas en la especificación de la interfaz por referencia, si son necesarias.
* Hay que modelar la realización de cada API solo en tanto que sea importante para mostrar la configuración de una implementación específica.

En la siguiente figura se muestra la API de un componente de animación. Podemos ver cuatro interfaces que forman la API: implicación, IModelos, IVisualizacion y IGuiones. Otro componente pueden ser una o más interfaces.

**Sugerencias y consejos**

Los componentes permiten encapsular las partes de un sistema para reducir las dependencias, hacerlas explícitas y mejorar la flexibilidad y la posibilidad de sustitución cuando el sistema debe modificarse a futuro. Un buen componente:

* Encapsula un servicio que tiene una interfaz y una frontera.
* Tiene suficiente estructura interna para que merezca la pena describirla.
* No combina funcionalidades que no estén relacionadas a su única pieza.
* Organiza su comportamiento externo utilizando unas cuantas interfaces y puertos.
* Interactúa solo a través de los puertos que ha declarado.

Su se requiere mostrar la implementación de un componente utilizando subcomponentes animados:

* Hay que utilizar un número pequeño de subcomponentes. Si hay demasiados para que quepan cómodamente en una página, hay que utilizar niveles adicionales de descomposición en algunos de los componentes.
* Hay que asegurarse que los componentes solo interactúan entre los puertos y conectores definidos.
* Hay que determinar que componentes interactúan directamente con el modulo exterior y deben modelarse estos con conectores de delegación.

Cuando se dibuje un componente en UML:

* Hay que darle un nombre que indique claramente su propósito.
* Hay que dar nombres a los subcomponentes y a los puertos si su significado no está claro a partir de sus tipos si hay varias partes del mismo tipo.
* Hay que ocultar los detalles innecesarios. No hay que mostrar todos los detalles de la implementación sobre el diagrama de componentes.
* Hay que mostrar la dinámica de un componente mediante diagramas de interacción.