**UNIVERSIDAD DE ORIENTE**

**NUCLEÓ ANZOÁTEGUI**

**ESCUELA DE INGENIERÍA Y CIENCIAS APLICADAS**

**DEPARTAMENTO DE COMPUTACION Y SISTEMAS**

**DESARROLLO DE SOFTWARE**



**UML 2.0 PARTE II**

**Estudiantes:**

Correa, Luis. C.I: 19.840.230 Ingeniería En Computación

Dun, Manuel. C.I: 19.257.821 Ingeniería En Computación

Rondón, Frank. C.I: 19.738.854 Ingeniería En Computación

**PROFESOR:**

Ing. Víctor Mujica.

**BARCELONA, JUNIO DE 2015**

**INDICE**

**Pág.**

**1.** **Diagrama de Clases** 2

**2.** **DIAGRAMA DE OBJETOS** 6

**3.** **Características de diagrama de objetos** 6

**4.** **Usos comunes** 7

**5.** **Técnicas comunes de modelado** 7

**6.** **Diagrama de Componentes** 8

**7.** **Estructura interna** 11

**8.** **Partes dentro de un Componente** 12

**9.** **Partes del Mismo Tipo** 13

**10.** **Técnicas comunes de Modelado** 15

**11.** **Clase Estructurada** 16

**12.** **Sugerencias y consejos** 17

## **Diagrama de Clases**

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia y de uso. Muestra los bloques de construcción de cualquier sistema orientado a objetos. Los diagramas de Clase son más útiles para ilustrar relaciones entre clases e interfaces.

Un diagrama de clases está compuesto por los siguientes elementos:

* [Clase](http://users.dcc.uchile.cl/~psalinas/uml/modelo.html#clase): atributos, métodos y visibilidad.
* [Relaciones](http://users.dcc.uchile.cl/~psalinas/uml/modelo.html#relacion): Herencia, Composición, Agregación, Asociación y Uso.
  1. Elementos
* **Clase**: Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, etc.).

En UML, una clase es representada por un rectángulo que posee tres divisiones:



En donde:

* **Superior**: Contiene el nombre de la Clase
* **Intermedio**: Contiene los atributos (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public).
* **Inferior**: Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: private, protected o public).

**Atributos:**

* + **Atributos:**

Los atributos o características de una Clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son:

* + - **public** (+): Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
    - **private** (-): Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden accesar).
    - **protected** (#): Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de las subclases que se deriven (ver herencia).
  + **Métodos:**

Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características:

* + - **public** (+): Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
    - **private** (-): Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden accesar).
    - **protected** (#): Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de métodos de las subclases que se deriven (ver herencia).

**Relaciones entre Clases:**

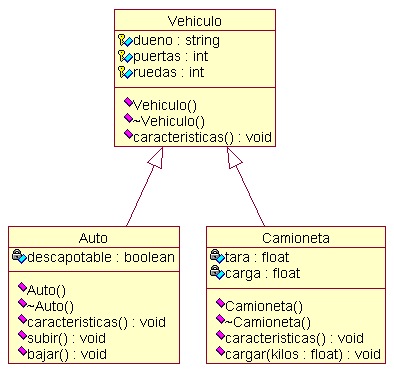
Ahora ya definido el concepto de Clase, es necesario explicar cómo se pueden interrelacionar dos o más clases (cada uno con características y objetivos diferentes).

En UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

* + **uno o muchos**: 1..\* (1..n)
  + **0 o muchos**: 0..\* (0..n)
  + **número fijo**: m (m denota el número).

**Herencia (Especialización/Generalización)**: http://users.dcc.uchile.cl/~psalinas/uml/img/modelo/herencia1.jpg

Indica que una subclase hereda los métodos y atributos especificados por una Super Clase, por ende la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Super Clase (public y protected), ejemplo:

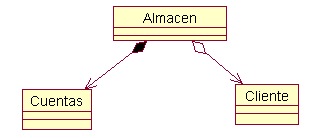


**Agregación**: http://users.dcc.uchile.cl/~psalinas/uml/img/modelo/agregacion1.jpg

Para modelar objetos complejos, n bastan los tipos de datos básicos que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación, tenemos dos posibilidades:

* + - **Por Valor**: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada **Composición** (el Objeto base se construye a partir del objeto incluido, es decir, es "parte/todo").
    - **Por Referencia**: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada **Agregación** (el objeto base utiliza al incluido para su funcionamiento).

Un Ejemplo es el siguiente:



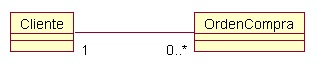
En donde se destaca que:

* Un Almacén posee Clientes y Cuentas (los rombos van en el objeto que posee las referencias).
* Cuando se destruye el Objeto Almacén también son destruidos los objetos Cuenta asociados, en cambio no son afectados los objetos Cliente asociados.
* La composición (por Valor) se destaca por un rombo relleno.
* La agregación (por Referencia) se destaca por un rombo transparente.

La flecha en este tipo de relación indica la navegabilidad del objeto referenciado. Cuando no existe este tipo de particularidad la flecha se elimina.

**Asociación**: http://users.dcc.uchile.cl/~psalinas/uml/img/modelo/asociacion1.jpgLa relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre si. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

Ejemplo:



Un cliente puede tener asociadas muchas Órdenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.

**Dependencia o Instanciación (uso)**: http://users.dcc.uchile.cl/~psalinas/uml/img/modelo/dependencia1.jpg

Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada.

El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra, como por ejemplo una aplicación grafica que instancia una ventana (la creación del Objeto Ventana esta condicionado a la instanciación proveniente desde el objeto Aplicacion):

http://users.dcc.uchile.cl/~psalinas/uml/img/modelo/edependencia.jpg

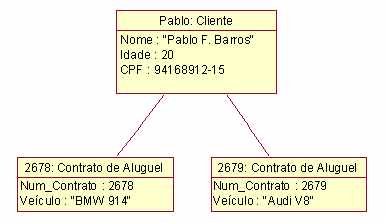
Cabe destacar que el objeto creado (en este caso la Ventana gráfica) no se almacena dentro del objeto que lo crea (en este caso la Aplicación).

## **DIAGRAMA DE OBJETOS**

Es una representación gráfica de la vista de diseño estática o de la vista de interacción estática de un sistema.

Los diagramas de objetos generalmente contienen:

* **Objetos**
* **Enlaces**

****

## **Características de diagrama de objetos**

* Los objetos se representan etiquetando el nombre de la instancia seguido de dos puntos (:) frente al nombre de la clase El icono de un objeto es un rectángulo dividido en secciones.
* Se centra en comunicar un aspecto de la vista de diseño estática o la vista de procesos estática de un sistema.
* Representa una escena de la historia representada por un diagrama de interacción.
* Contiene solo aquellos elementos esenciales para comprender ese aspecto.
* No es tan minimalista, no deja de informar al lector sobre la semántica importante.
* Al igual que los demás diagramas, los diagramas de objetos `pueden contener notas y restricciones.
* A veces se colocarán clases en los diagramas de objetos, especialmente cuando se quiera mostrar la clase que hay detrás de cada instancia.

## **Usos comunes**

Los diagramas de objetos se utilizan para modelar la vista de diseño estática o de la vista procesos estáticas de un sistema al igual que se hace con los diagramas de clases, pero desde la perspectiva de instancias reales o prototípicas.

Esta vista sustenta principalmente los requisitos funcionales de un sistema (o sea, los servicios que debe proporcionar el sistema a sus usuarios finales). Los diagramas de objetos permiten modelar estructuras de datos estáticas.

Al modelar la vista de diseño estática o la vista de interacción estática de un sistema, normalmente los diagramas de objetos se utilizan para modelar estructuras de objetos.

El modelado de estructuras de objetos implica tomar una instantánea de los objetos de un sistema en un momento dado. Un diagrama de objetos representa una escena estática dentro de la historia representada por un diagrama de interacción.

Los diagramas de objetos se emplean para visualizar, especificar, construir y documentar la existencia de ciertas instancias en el sistema, junto a las relaciones entre ellas. El comportamiento dinámico y la ejecución se pueden representar como una secuencia de escenas.

## **Técnicas comunes de modelado**

* **Modelado de estructuras de objetos**
* **Ingeniería invertida**

**Modelado de estructuras de objetos:** Si se congela un sistema en ejecución o uno se imagina un instante concreto en un sistema modelado, aparecerá un conjunto de objetos, cada uno de ellos en un estado específico, y con unas relaciones particulares con otros objetos. Los diagramas de objetos se pueden usar para visualizar, especificar, construir y documentar la estructura de esas instantáneas. Los diagramas de objetos son especialmente útiles para modelar estructuras complejas de datos.

Con los diagramas de objetos no se pueden especificar completamente la estructura de objetos del sistema. Puede existir una multitud de posibles instancias de una clase particular, y para un conjunto de clases con relaciones entre ellas, pueden existir muchas más configuraciones posibles de esos objetos. Por lo tanto, al utilizar diagramas de objetos solo se pueden mostrar significativamente conjuntos interesantes de objetos concretos o prototípicos. Esto es lo que significa modelar una estructura de objetos.

**Para modelar una estructura de objetos:**

* Hay que identificar el mecanismo que se desea modelar. Un mecanismo representa alguna función o comportamiento del parte del sistema que se está modelando, que resulta de la interacción de una sociedad de clases, interfaces y otros elementos**.**
* Hay que crear una colaboración para describir un mecanismo.
* Para cada mecanismo, hay que identificar las clases, interfaces y otros elementos que participan en esa colaboración, también hay que identificar las relaciones entre estos elementos.
* Hay que considerar un escenario en el que intervenga este mecanismo. También hay que congelar ese escenario en un momento concreto, y representar cada objeto que participe en el mecanismo.
* Hay que mostrar el estado y los valores de los atributos de cada uno de esos objetos, si son necesarios para comprender el escenario,
* Análogamente, hay que mostrar los enlaces entre esos objeto, que representarán instancias de asociaciones entre ellos.

**Ingeniería Inversa:** Hacer ingeniería inversa (creación de un modelo a partir del código) con un diagrama de objetos puede ser muy útil. De hecho, mientras se está depurando el sistema, esto es algo que el programador o las herramientas están haciendo continuamente. Por ejemplo, si se persigue un enlace perdido, uno debe dibujar mental o literalmente un diagrama con los objetos afectados para saber dónde se invalida,

**Para hacer ingeniería inversa:**

* Hay que elegir el objetivo al que se desea aplicar la ingeniería inversa. Normalmente, se establecerá el contexto dentro de una operación o en relación con una instancia de una determinada clase.
* Hay que detener la ejecución en un determinado instante, utilizando una herramienta o simplemente recorriendo un escenario.
* Hay que identificar el conjunto de objetos interesantes que colaboran en el contexto, y representarlos en un diagrama de objetos.

Si es necesario para comprender la semántica, hay que mostrar el estado de estos objetos.

* Si es necesario para entender la semántica, hay que identificar los enlaces existentes entre estos objetos.
* Si el diagrama se complica en exceso, se puede cortar, eliminando aquellos objetos que no sean pertinentes para las cuestiones sobre el escenario que necesitan respuestas. Si el diagrama es demasiado simple, hay que expandir los vecinos de los objetos interesantes para mostrar con mayor profundidad el estado de cada objeto.
* Normalmente, habrá que añadir manualmente estructura que no sea explícita en el código. La información ausente proporciona el objetivo de diseño que solo es implícito en el código final.

## **Diagrama de Componentes**

**Un componente: es** una parte reemplazable de un sistema que conforma y proporciona la implementación de un conjunto de interfaces.

**Una interfaz: es** una colección de operaciones que especifican un servicio proporcionado o solicitado por una clase o componente.

**Diagrama de componentes**

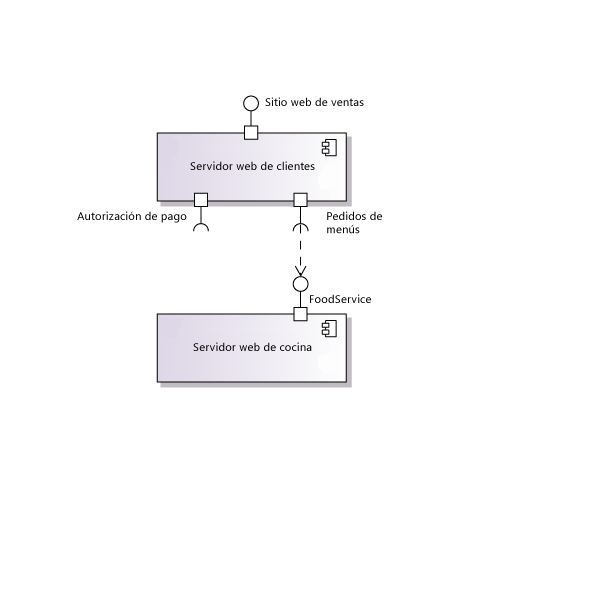
Permite visualizar con más facilidad la estructura general del sistema y el comportamiento del servicio que estos componentes proporcionan y utilizan a través de las interfaces.

Para construir un sistema basado en componentes, se descompone el sistema especificando interfaces que representan las principales líneas de separación del sistema.

Luego se proporcionan componentes que realizan las interfaces, junto con otros componentes que acceden a los servicios a través de sus interfaces

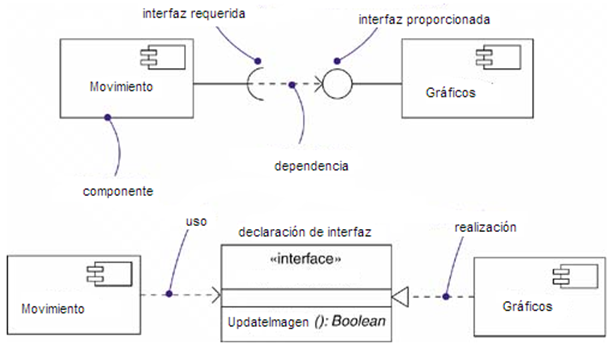
Una interfaz que es realizada por un componente se denomina una interfaz proporcionada, lo que significa que el componente proporciona la interfaz a otros componentes.

La interfaz que utiliza un componente se denomina interfaz requerida, lo que significa una interfaz con la que conforma el componente cuando solicita servicios de otros componentes. Un componente puede tanto proporcionar cómo requerir interfaces



Como se indica en la Figura, un componente se representa como un rectángulo con un pequeño icono con dos pestañas en su esquina superior derecha. El nombre del componente aparece en el rectángulo. Un componente puede tener atributos y operaciones, pero éstos a menudo se ocultan en los diagramas.

La relación entre un componente y sus interfaces se puede mostrar de dos formas diferentes. La primera (y más frecuente) consiste en representar la interfaz en su formato icónico, abreviado. Una interfaz proporcionada se representa como un círculo unido al componente por una línea (una “piruleta”). Una interfaz requerida se representa como un semicírculo unido al componente por una línea (un "enchufe”). En ambos casos, el nombre de la interfaz se coloca jumo al símbolo. La segunda manera representa la interfaz en su formato expandido, mostrando quizás sus operaciones. El componente que realiza la interfaz se conecta a ella con una relación de realización. El componente que accede a los servicios del otro componente a través de la interfaz se conecta a ella mediante una relación de dependencia.



**Sustitución**

Ei objetivo básico de cualquier recurso del sistema operativo basado en componentes es permitir el ensamblado de sistemas a partir de artefactos reemplazables. Esto significa que se puede diseñar un sistema utilizando componentes y luego implementar esos componentes utilizando artefactos. Posteriormente, el sistema puede evolucionar añadiendo nuevos componentes y sustituyendo los antiguos, sin reconstruir el sistema.

Las interfaces son la clave para conseguir esto. En el sistema ejecutable, podemos usar cualquier artefacto que imple- mente a un componente que conforma o proporciona una interfaz. Podernos extender el sistema haciendo que los componentes proporcionen nuevos servicios a través de otras interfaces, las cuales pueden ser descubiertas y utilizadas a su vez por otros componentes.

Un componente es una parte de un sistema. Un componente raramente se encuentra aislado. En vez de ello, un componente dado colabora con otros componentes y al hacer esto existe en el contexto arquitectónico o tecnológico para el que se ha planificado. Por tanto, un componente representa un bloque de construcción fun­damental, que puede utilizarse para diseñar y componer sistemas.

**Puertos**

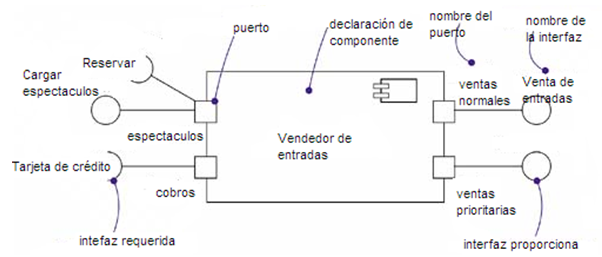
Las interfaces son útiles al declarar el comportamiento global de un componente pero no tienen identidad individual la implementación del componente debe asegurar simplemente que todas las operaciones de todas las interfaces proporcionadas son implementadas. Para tener más control sobre la implementación pueden utilizarse puertos.

Un puerto es una ventana explícita dentro de un componente encapsulado. En un componente encapsulado, todas las interacciones dentro y fuera del componente pasan a través de los puertos. Otro componente puede comunicarse con un componente dado a través de un puerto específico. En la implementación, las partes internas del componente pueden interactuar a través de un puerto externo específico, de forma que cada parte pue­de ser independiente de los requisitos de las otras partes.

Un puerto se representa como un pequeño cuadrado insertado en el borde de un componente). Tanto las interfaces requeridas como las proporcionadas pueden enlazarse al puerto.

Cada puerto tiene un nombre, de forma que pueda ser identificado de manera única, dados el componente y el nombre del puerto. El nombre del puerto puede ser utilizado por algunas partes internas del componente para identificar el puerto al cual enviar y del cual recibir los mensajes.

Los puertos también pueden tener multiplicidad: ésta indica el número posible de instancias de un puerto particular dentro de una instancia del componente. Cada puerto en una instancia de un componente tiene un vector de instancias del puerto. Aunque todas las instancias de un puerto que están en un vector satisfacen la misma interfaz y aceptan el mismo tipo de solicitudes, pueden tener diferentes estados y valores para los datos. Por ejemplo, cada instancia en el vector puede tener un nivel de prioridad diferente, de forma que las instancias de puerto mis grandes sean servidas antes.

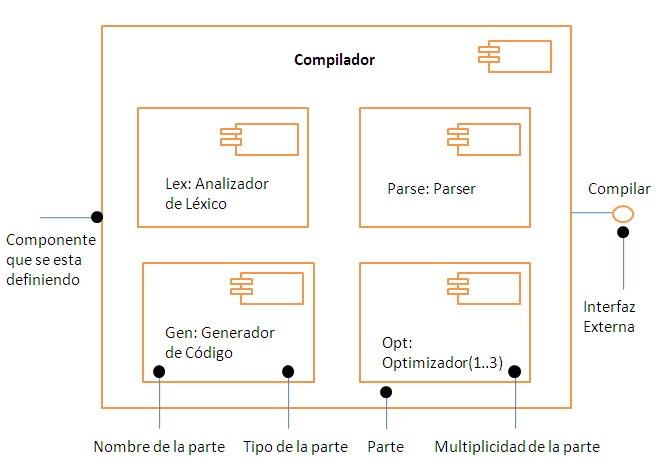


## **Estructura interna**

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.

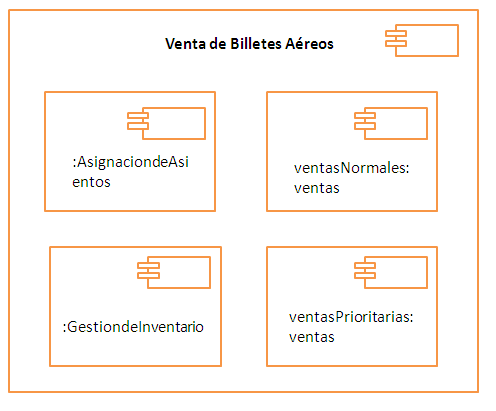


## **Partes dentro de un Componente**

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.

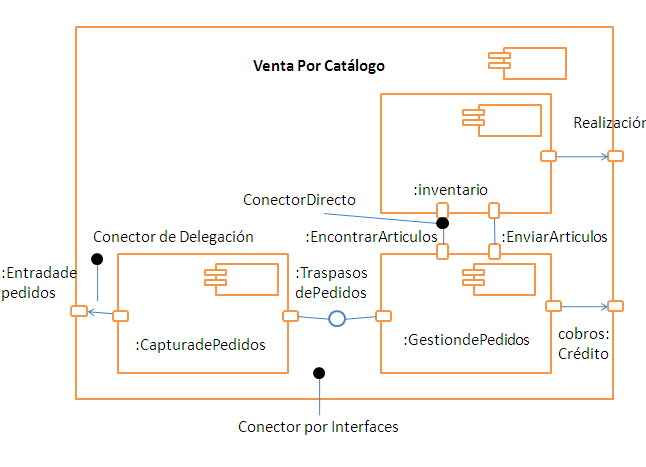
En la Siguiente figura se muestra un ejemplo del mismo, donde un componente Ventas de Billetes puede tener diferentes partes Ventas, para los pasajeros frecuentes y para los pasajeros normales; ambos trabajan igual pero la parte de pasajeros frecuentes solo está disponible para ciertos clientes especiales 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.

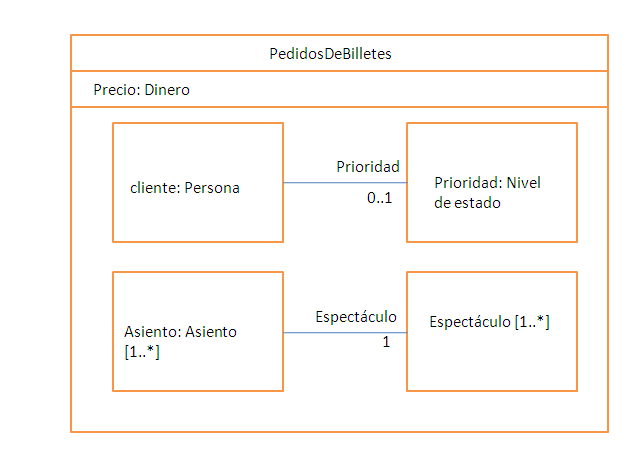
## **Técnicas comunes de Modelado**

* **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.

**Universidad de Oriente**

**Núcleo Anzoátegui**

**Escuela de Ingeniería y Ciencias Aplicadas**

**Departamento de Computación y Sistemas**

**Desarrollo de Software**

****

***Diagramas de: Clases,***

***Objetos y de Componentes***

**Integrantes:**

Tineo, Fanny

Rodríguez, Nelson

García, Wilmer

Moya, Drusmary C.I.: 19839122

**Prof.:** Mujica, Victor.

Puerto La Cruz, Mayo de 2015.