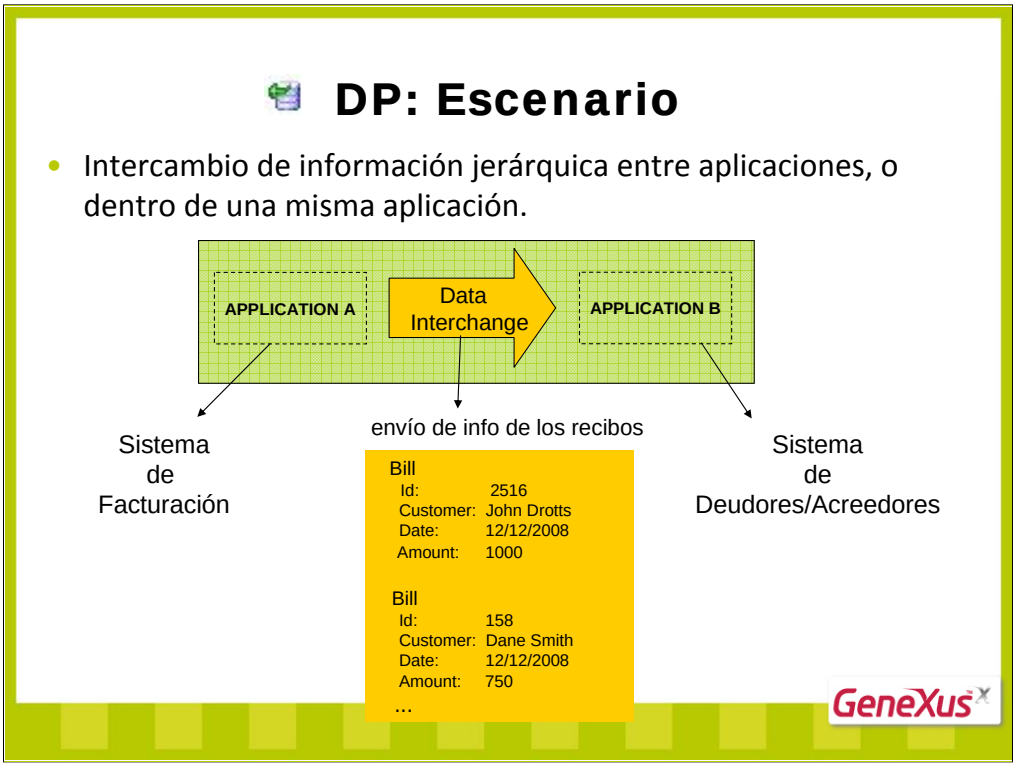
Objeto Data Provider

DP 

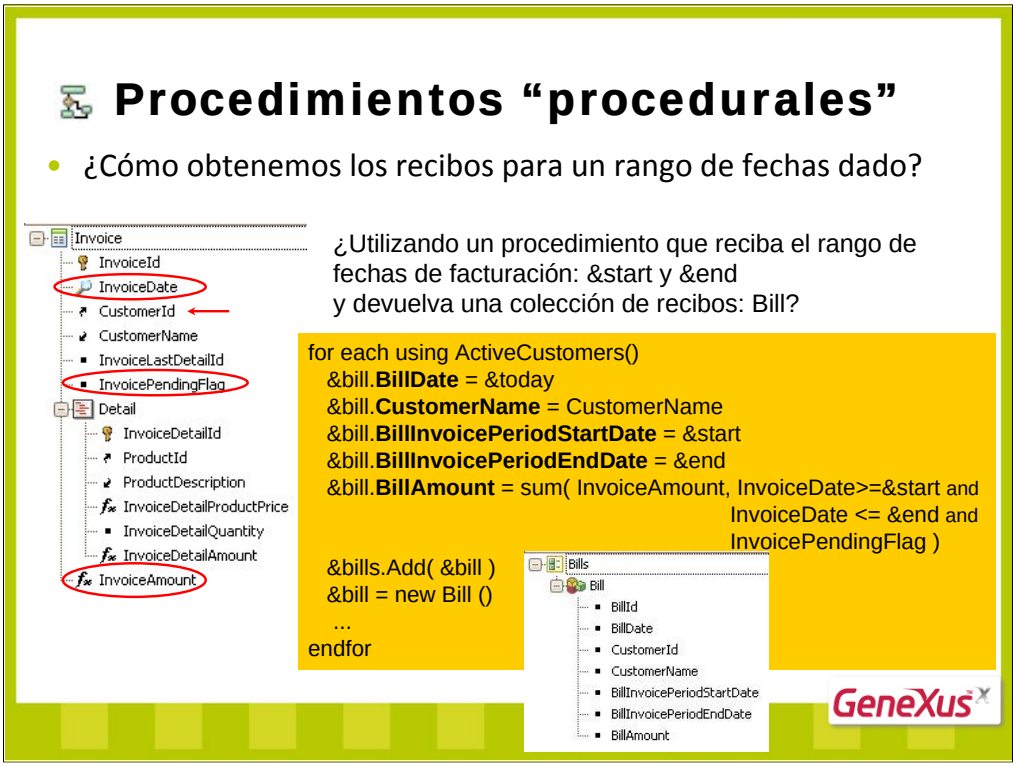
GeneXus



Supongamos que desde nuestro Billing System, queremos enviar a un sistema de deudores y acreedores, un listado de los recibos correspondientes a cierto período de facturación (es decir, para un período determinado se desea sumarizar para cada cliente el importe total que se le ha facturado y generarle un recibo).

Se trata de información jerárquica (enviaremos info de recibos, cada uno de los cuáles tiene determinados datos).

El formato más usual de intercambio de información jerárquica suele ser Xml, aunque no sabemos qué deparará el futuro en ese aspecto (por ejemplo se está volviendo común el formato Json).



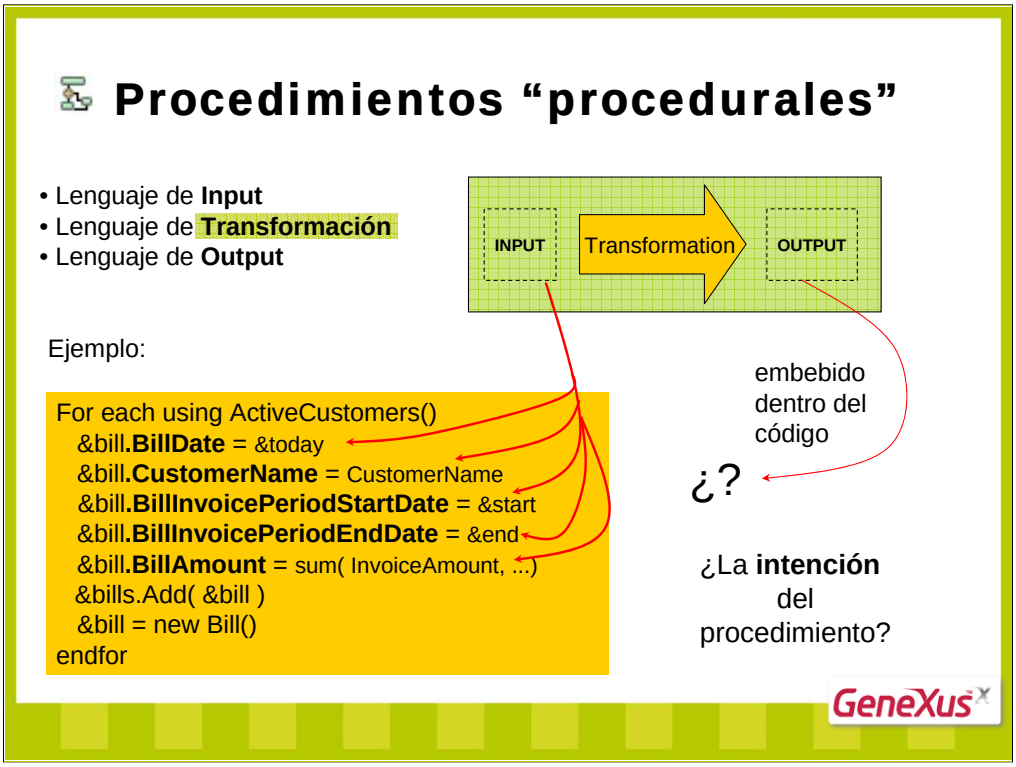
Teniendo el mismo SDT Bill que definimos cuando estudiamos los tipos de datos estructurados (allí le llamamos Bill_SDT, aquí le llamaremos Bill), estamos utilizando un procedimiento con los siguientes parámetros:

parm(in: &start, in: &end, out: &bills);

siendo &start y &end variables de tipo Date que reciben el rango de facturación, y &bills una variable collection del SDT Bill.

Obsérvese que estamos utilizando el Data Selector ActiveCustomers para filtrar por clientes activos y utilizando la fórmula inline sum, en la que únicamente sumamos los totales de aquellas facturas que pertenezcan al cliente de la actual iteración del For each y cuyas fechas se encuentren en el rango dado y tengan el valor True en el atributo Booleano InvoicePendingFlag.

Este atributo será cambiado a False una vez que la factura haya sido procesada dentro del proceso de generación de recibos que iniciamos aquí pero que completaremos más adelante, cuando estudiemos las formas de actualizar la información de la base de datos (aquí solamente estamos obteniendo la información de los recibos, pero aún no los registraremos en la base de datos...).



Todo procedimiento toma un Input, y mediante alguna transformación, obtiene un Output.

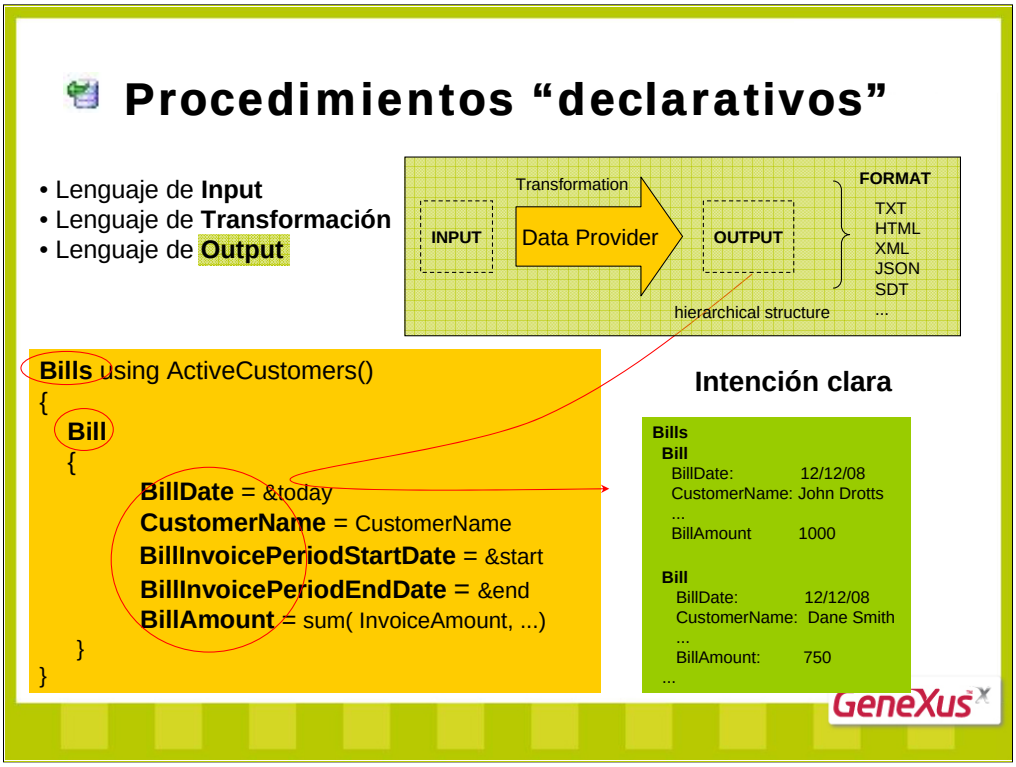
Para obtener el Input se tiene cierto lenguaje (en GeneXus si se trata de la base de datos, nombrando los atributos alcanza, como se puede ver en el ejemplo, a la derecha de las asignaciones; así como también se obtiene de los parámetros).

Para realizar la Transformación se tiene otro lenguaje (el código que podemos ver en el ejemplo) y luego para expresar el Output otro más.

Como puede verse, en el caso de los procedimientos GeneXus, tanto el Input como el Output se encuentran embebidos dentro del propio código de Transformación. Puede decirse entonces que el **foco** está puesto en la **transformación**.

De esta forma la **intención** del procedimiento, su **salida**, queda oscurecida dentro de ese código, que mezcla:

- elementos de salida,
- con elementos de transformación (en nuestro caso, el comando for each, el método add y el operador new, que implementan el armado de la colección de recibos)
- y con elementos de entrada.

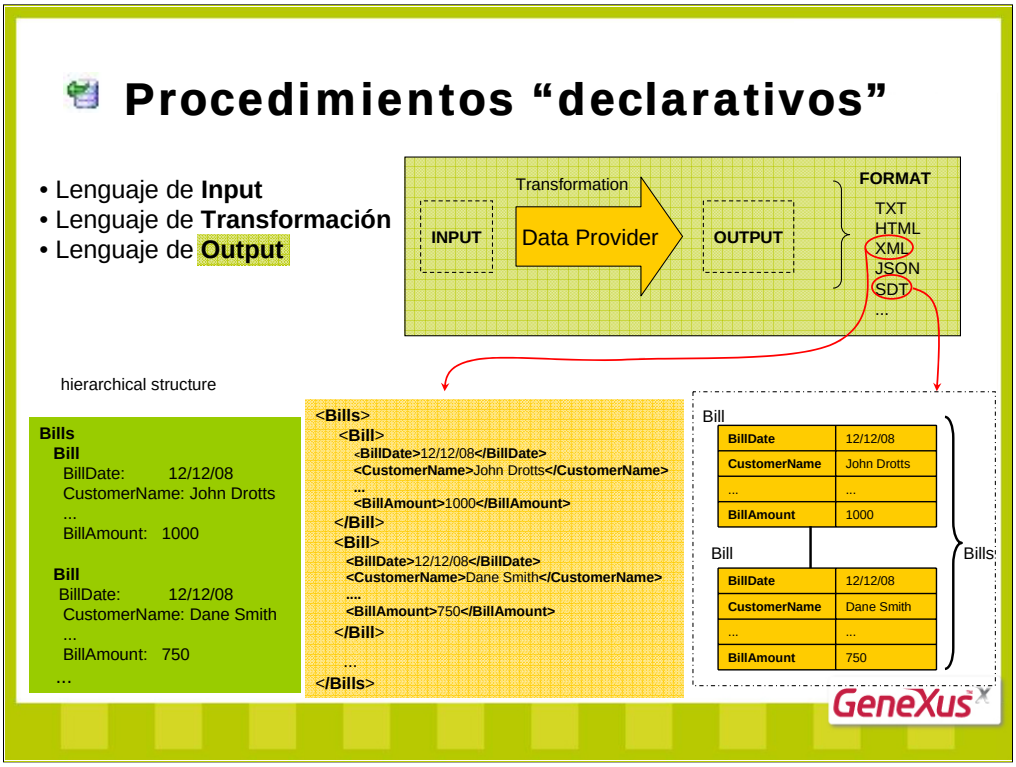


Con un Data Provider, el **foco** está ubicado en el lenguaje de **salida**: obsérvese que se indica en una estructura jerárquica de qué se compone ese Output.

Luego, para cada elemento de la estructura jerárquica, habrá que indicar en el Source del Data Provider, cómo se calcula.

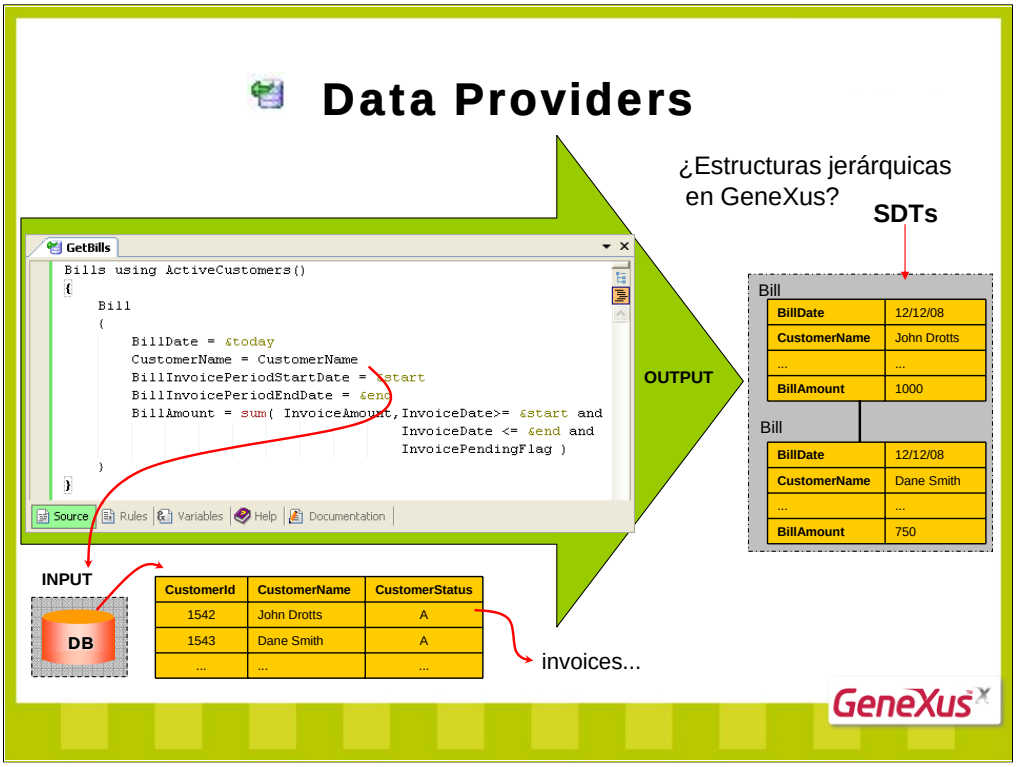
Por tanto, alcanza con observar el lado izquierdo para deducir cuál será la estructura resultante.

Luego, alcanzará con indicar el formato deseado para esa estructura jerárquica resultante...



Luego, una misma información estructurada, podrá representarse utilizando los diferentes formatos existentes.

Esa es la idea del Data Provider. Si en el futuro aparece un nuevo formato de representación de información estructurada, el Data Provider continuará invariable... GeneXus implementará el método de transformación a ese formato, y solo habrá que utilizarlo.

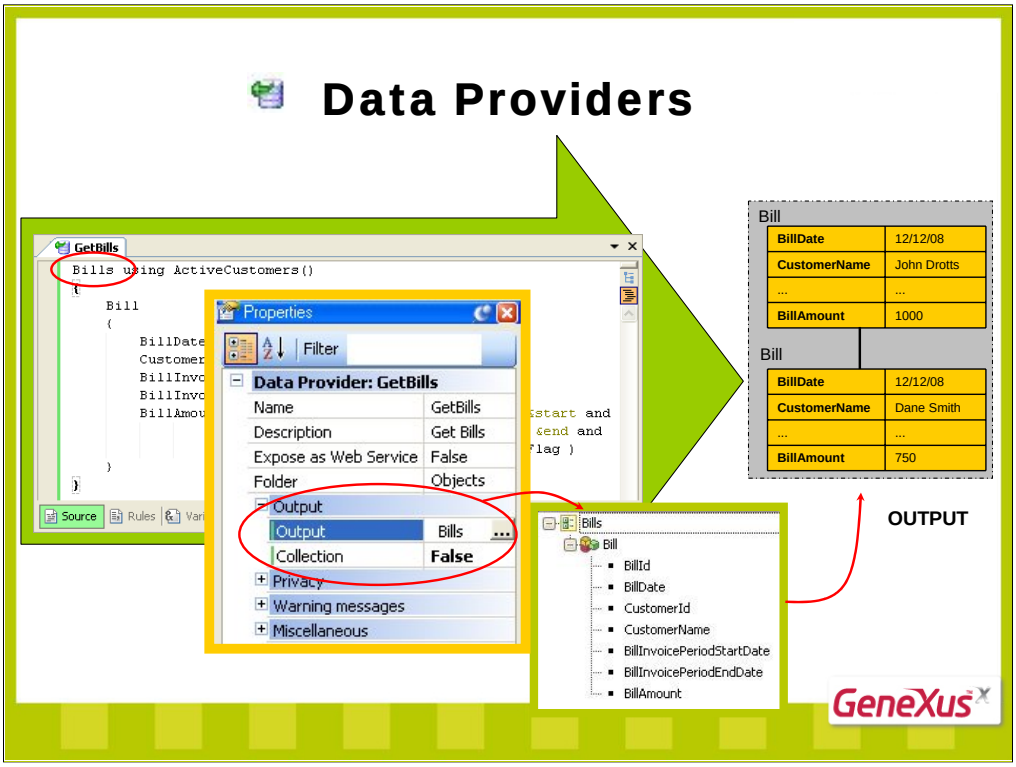


Hasta aquí hicimos un análisis descriptivo. Surge inmediatamente la siguiente pregunta: ¿cómo se representan en GeneXus las estructuras jerárquicas de datos?

Por tanto, la salida de un Data Provider será un SDT (o una colección de SDTs).

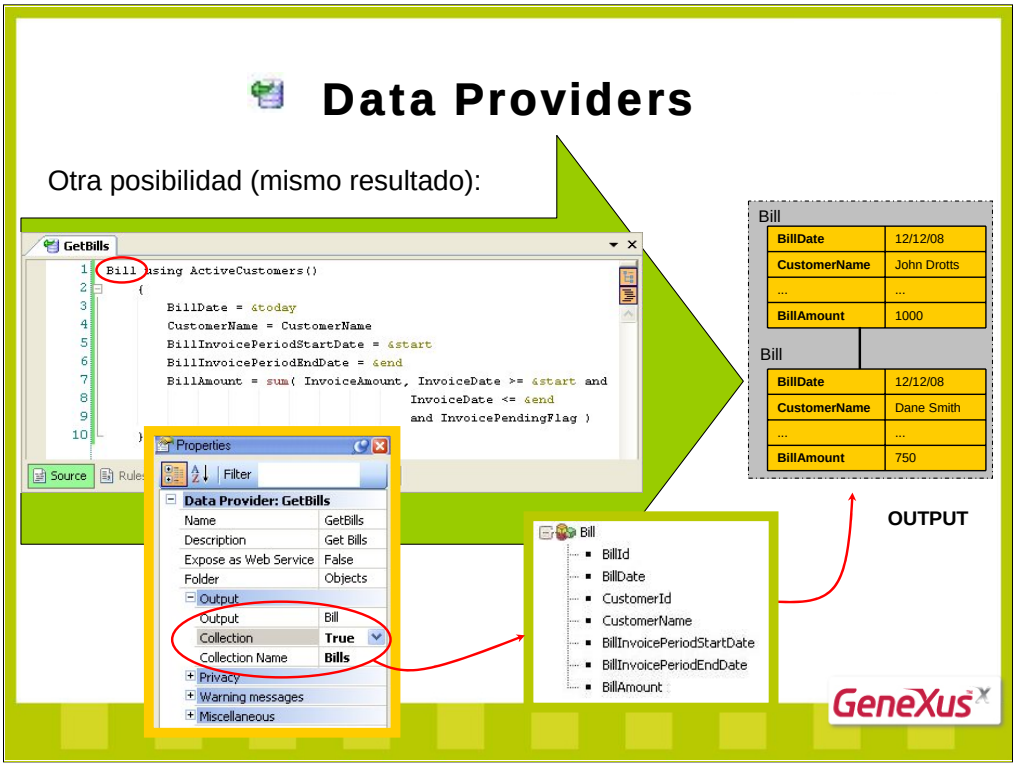
Luego, con esa salida puede realizarse lo que se desee, en particular, convertirla a formato XML, con el método toXml de los SDTs.

Veamos cómo declaramos la salida... no será de la forma convencional (como parámetro de out de la regla parm)...



Aquí vemos para el ejemplo que venimos estudiando, que teniendo el SDT definido Bills, collection, que coincide con la estructura que se infiere del Source del Data Provider 'GetBills', habrá que declarar ese SDT en la propiedad Output del Data Provider.

Pero no es la única posibilidad... veremos otra en la página siguiente, donde cobrará sentido la propiedad Collection.



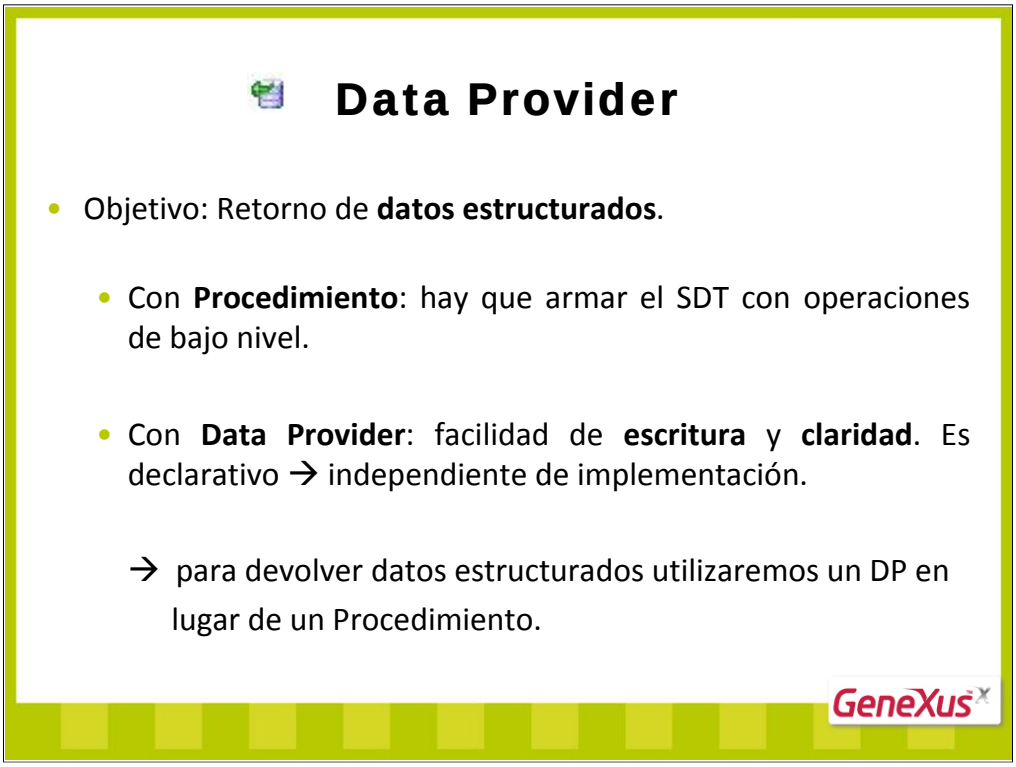
Si en lugar de tener el SDT collection Bills, tuviéramos el que aparece arriba, Bill, entonces podríamos programar el Source del Data Provider como vemos y luego configurar la propiedad Collection en 'True', con lo cuál se abrirá una nueva propiedad, Collection Name, que permitirá dar nombre a la colección.

Este Data Provider es equivalente al anterior.

Obsérvese que en este caso no es necesario poner la raíz de la jerarquía, Bills, puesto que se infiere de la propiedad 'Collection Name'. De todas formas, si bien no es requerido, podría programarse el Source exactamente igual al anterior, es decir, con el grupo Bills encabezando la jerarquía.

```
Bills
{
   Bill  using ActiveCustomers()
   {
      ...
   }
}
```

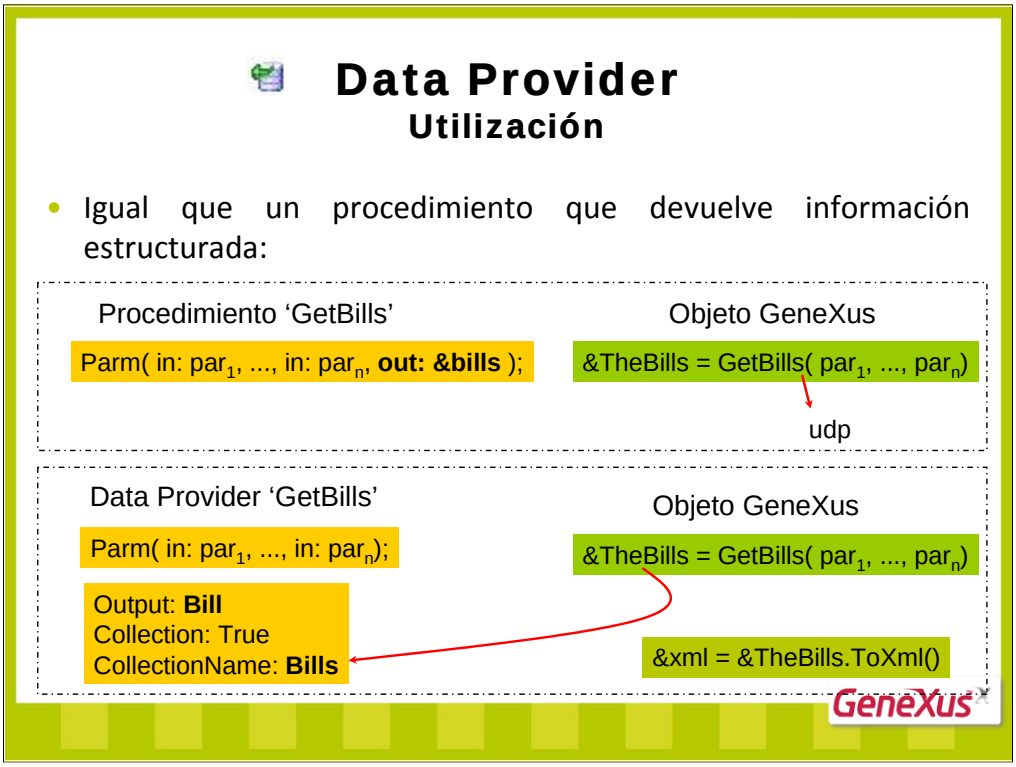
Nota: la cláusula using, así como las where que veremos luego, order, etc., podrán especificarse tanto a nivel del grupo Bills, como del grupo Bill, en casos como este. Volveremos sobre este tema más adelante.



Los Data Providers atacan eficientemente un tipo de problema: aquel que consiste en retornar datos estructurados. Para ese tipo de problemas contábamos con los procedimientos, pero su desventaja evidente era que necesitábamos implementar la forma de armar esas estructuras con operaciones de bajo nivel, como agregar un ítem a una colección con Add, y pedir memoria (new). Asimismo, si hubiera que hacer cálculos complejos para dar valor a cada miembro de la estructura, quedarían en el código embebidos los elementos de la salida, sin que fuera sencillo visualizar en un golpe de vista la salida resultante del Procedimiento.

Todos estos inconvenientes se evitan con un Data Provider. Aquí el foco está puesto en la salida, razón por la cuál con un solo vistazo quedará evidenciada la estructura del Output. Por otro lado, siendo absolutamente declarativo, nos independizamos de la forma en que se implementa realmente la carga del SDT. De eso se encarga GeneXus.

Cuanto más declarativa sea una herramienta, más fácil de programar, más adaptable al cambio, más independiente de una implementación particular. GeneXus tiende a permitir declarar lo más posible y cada vez programar 'proceduralmente' menos. Lo declarativo traslada el problema de la implementación a la herramienta, y se la quita al programador. Cuanto más inteligente una herramienta, menos necesidad tendrá el programador de resolver el problema: le alcanzará con enunciarlo.



Un Data Provider también puede recibir parámetros vía regla parm, pero a diferencia de un procedimiento donde todos los parámetros pueden ser de entrada/salida, aquí solo podrán ser de entrada.

Por otro lado, un procedimiento que devuelve información estructurada lo hace mediante una variable que se carga en el código, y que debe declararse como de salida, en el último parámetro de la regla parm.

En un Data Provider, la declaración del tipo de datos de salida se hace mediante las propiedades Output, **no** en la regla parm.

La invocación desde cualquier objeto GeneXus de un Data Provider es idéntica a la invocación de un procedimiento, es decir, con udp.

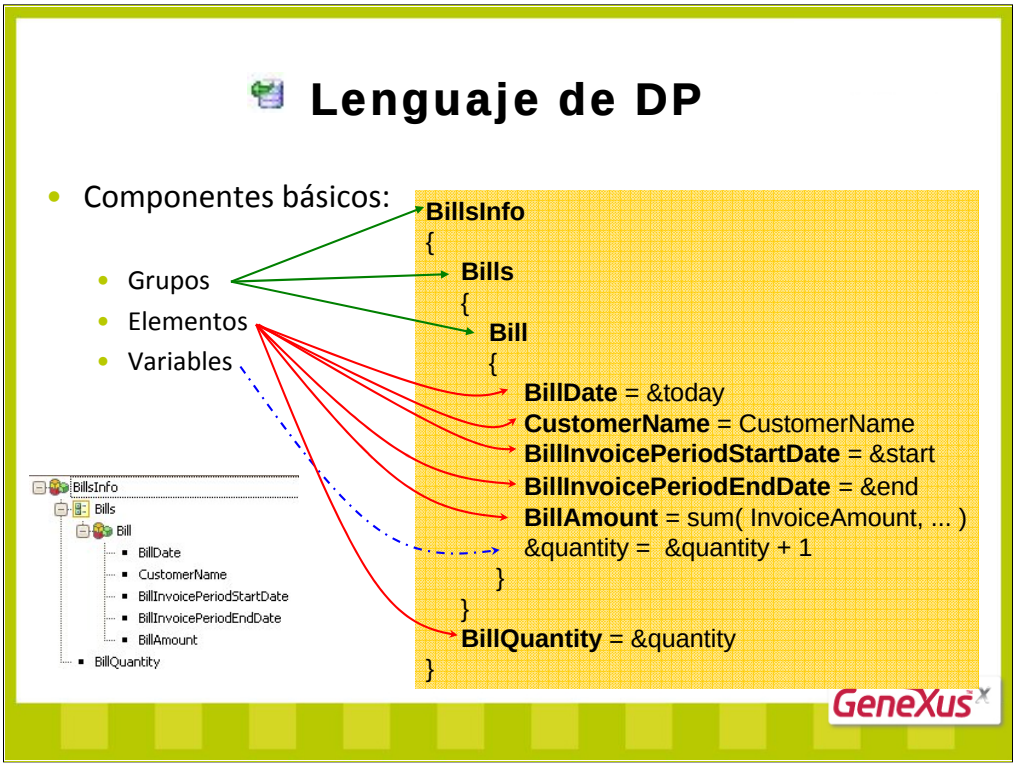
Recuerde que al invocar a un objeto y no especificar el método de invocación, se asume udp.

La variable &TheBills deberá estar declarada en el objeto GeneXus en el que se realiza la invocación. Es la variable en la que se devolverá el resultado. Recuerde que si el SDT definido en la KB es el correspondiente a los ítems individuales: Bill, (y no a la colección Bills), entonces la variable &TheBills se definirá como Collection, de tipo de datos Bill.

Luego, con la estructura jerárquica devuelta se podrá, por ejemplo, convertir al formato deseado, como XML.

Importante:

Como veremos, un Data Provider no solo puede retornar un SDT o colección de SDT, sino también otro tipo, el **Business Component**, que también representa información estructurada. Vea ese tema para completar el conocimiento de Data Providers.



El ejemplo que podemos ver aquí es parecido al que estuvimos trabajando. Hemos agregado el elemento BillQuantity. Tómese unos instantes para pensar cómo deberá ser el SDT BillsInfo, para que matchee con este Source. A la izquierda lo hemos presentado.

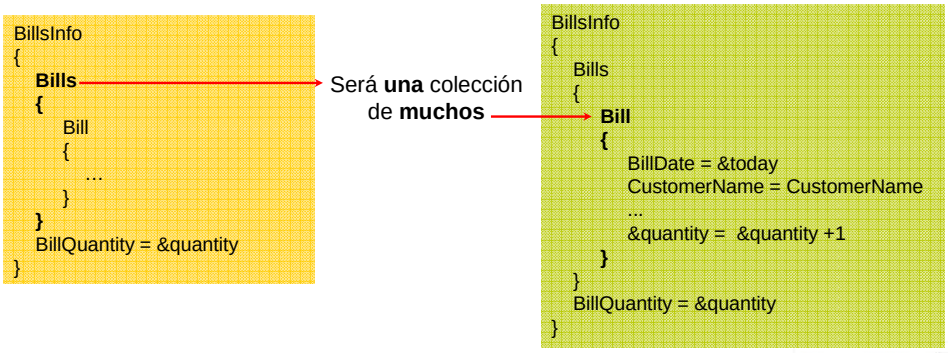
De todos modos, el Source no tiene por qué escribirse tal cual la estructura del SDT. Cuando en el SDT tenemos un miembro collection de un determinado ítem, podemos omitir del Source el Item como grupo. Ejemplo: con el SDT que tenemos arriba, podríamos haber escrito el Data Provider:

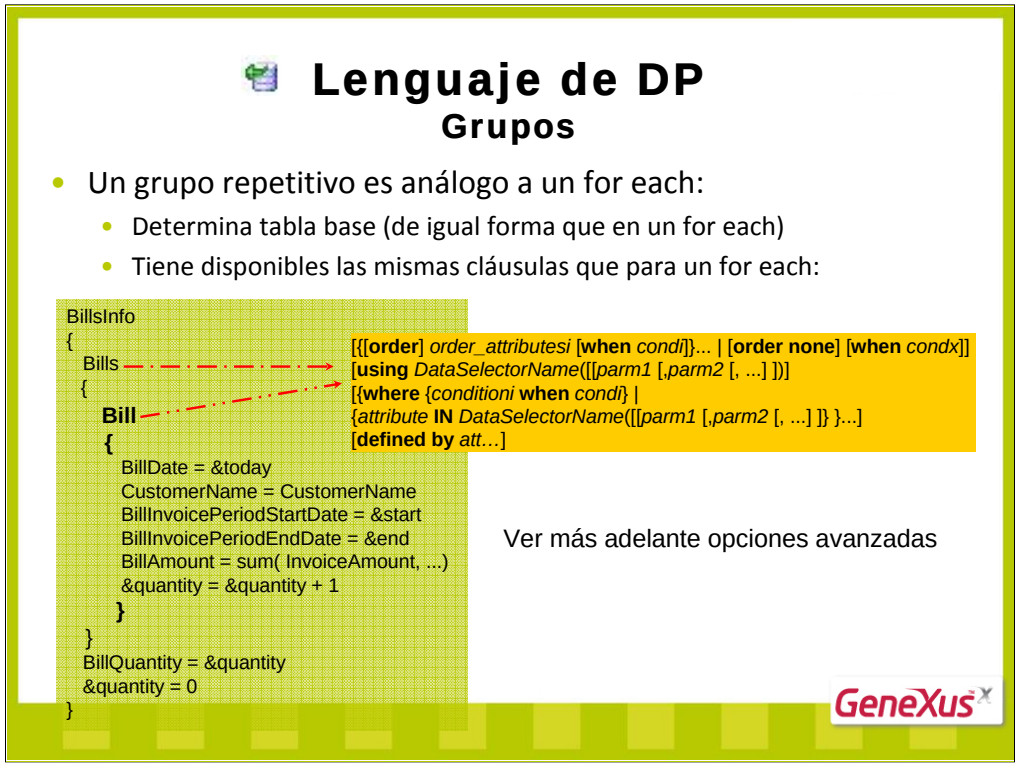
```
Billsnfo
{
   Bills
   {
      BillDate = &today
      CustomerName = CustomerName
      BillInvoicePeriodStartDate = &start
      BillInvoicePeriodEndDate = &end
      BillAmount = sum (InvoiceAmount, ... )
      &quantity = &quantity + 1
    }
   BillQuantity = &quantity
}
```

con el mismo resultado. GeneXus tiene la inteligencia suficiente como para matchear con el SDT.

Lenguaje de DP
Grupos

- Agrupan tanto **elementos**, **grupos** como **variables** en un mismo nivel de jerarquía.
- Puede ser repetitivo como no serlo (inteligencia para determinarlo).





En el ejemplo, el grupo de nombre Bill será repetitivo. ¿Por qué? Para contestar la pregunta, hagamos otra: ¿y si fuera un for each, donde los elementos de la izquierda de las asignaciones corresponden a los distintos elementos de una variable SDT? En este caso la presencia de *CustomerName* (a la derecha de la segunda asignación) permite afirmar que hay tabla base: CUSTOMER.

Por tanto el grupo será repetitivo, iterando sobre la tabla CUSTOMER.

En el ejemplo que veníamos trabajando, podíamos tener la cláusula:

Bill using ActiveCustomers()

que es como tener:

For each using ActiveCustomers()

por lo que así no estuviera el atributo *CustomerName* en la segunda asignación, de todos modos sería con tabla base, por la presencia del atributo *CustomerStatus* en el Data Selector 'ActiveCustomers'.

Obsérvese que el grupo de nombre BillsInfo, en cambio, no será repetitivo, no tiene cláusulas asociadas, y los elementos que contiene están definidos a base de variables y no de atributos:

BillQuantity = &quantity
&quantity = 0

¿Y qué pasa con el grupo Bills? Obsérvese que en este caso, es un grupo que sólo contiene otro grupo. El grupo contenido será repetitivo, por lo que Bills será una colección de Bill. Por este motivo, el subgrupo Bill podría omitirse (solamente dejar Bills) y que quede implícito. De este modo, las cláusulas del grupo que permiten definir order, filtros, defined by, se pueden asociar a este grupo.

Lenguaje de DP
Grupos

- También pueden repetirse grupos:

```
Clients
{
   Client
   {
        Name =  'Lou Reed'
        Country = 'United States'
        City = 'New York'

   }
   Client where CountryName = 'Mexico'
   {
        Name =  CustomerName
        Country = CountryName
        City = CityName
   }
}
```

El resultado retornado será una colección de N+1 ítems: siendo N el número de clientes de México.

GeneXus

Si la condición se hubiese colocado en el grupo Clients, aplicaría a los dos subgrupos Client. Por eso es que se permite que las cláusulas operen a nivel de los grupos que se repiten (items), y no solo a nivel del grupo que es colección de ítems.

Lenguaje de DP
Elementos

- Un Elemento es un valor atómico en el Output.
- Cada Elemento debe ser asignado y su sintaxis es la de las fórmulas.

Ejemplo:

```
Clients
{
    Client
    {
        Name = CustomerName
        Active = True if CustomerStatus = Status.Active; False otherwise;
    }
}
```

GeneXus

Lenguaje de DP
Elementos

- Elementos y Atributos usualmente tienen el mismo nombre → notación más compacta:

Ejemplo:

```
BillsInfo
{
  Bills
  {
    Bill
    {
      BillDate = &today
      CustomerName = CustomerName
      BillInvoicePeriodStartDate = &start
      ...
    }
  }
}
```

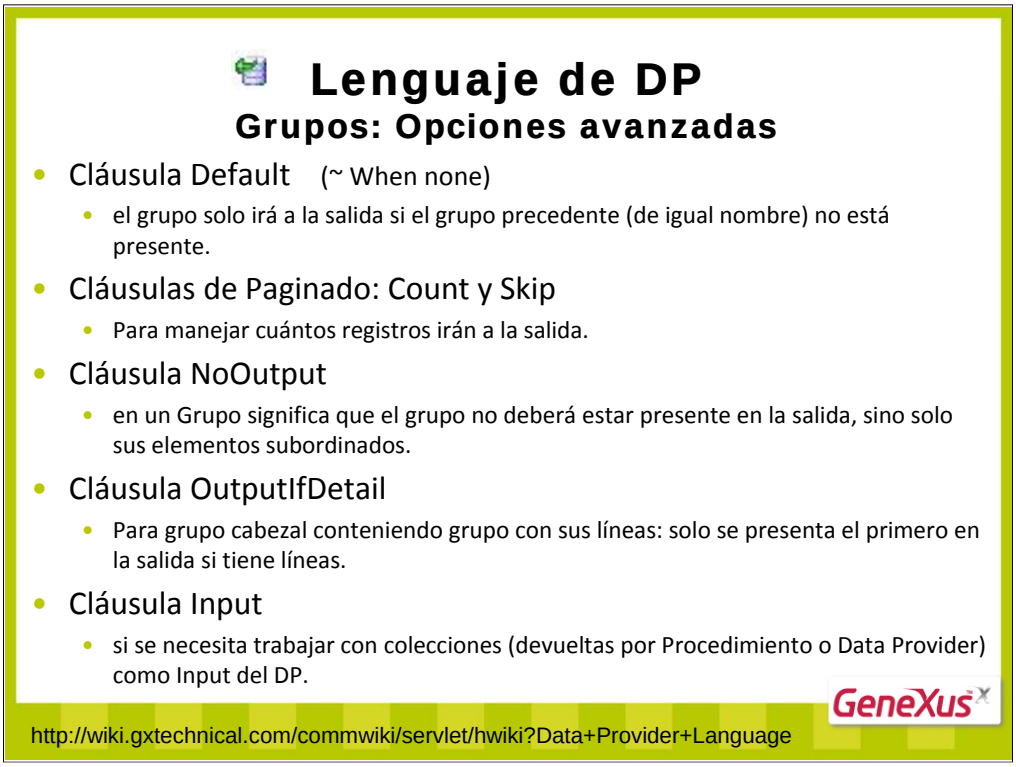
```
BillsInfo
{
  Bills
  {
    Bill
    {
      BillDate = &today
      CustomerName
      BillInvoicePeriodStartDate = &start
      ...
    }
  }
}
```

Lenguaje de DP
Variables

- Algunas veces es necesario realizar cálculos internos:

Ejemplo:

```
BillsInfo
{
   Bills
   {
      &quantity = 0
      Bill
      {
         BillDate = &today
         CustomerName
         BillInvoicePeriodStartDate = &start
         BillInvoicePeriodEndDate = &end
         BillAmount = sum( InvoiceAmount, ... )
         &quantity = &quantity + 1
      }
   }
   BillQuantity = &quantity
}
```



Podrá encontrar la información completa de este tema en inglés en nuestro community wiki, en la siguiente página:

http://wiki.gxtechnical.com/commwiki/servlet/hwiki?Data+Provider+Language

Lenguaje de DP
Group: cláusula Default

- Cláusula Default: el grupo solo irá a la salida si el grupo precedente (de igual nombre) no está presente.
  (~ When none en for each)

```
CurrentTaxes
   Where TaxInitialDate >= today()
   Where TaxFinalDate <= today()
   {
      VAT = TaxVAT
      Income = TaxIncome
   }
CurrentTaxes [Default]
   {
      VAT = 0.7
      Income = 0.3
   }
```

Tabla TAXES

TaxInitialDate*
TaxFinalDate*
TaxVAT
TaxIncome

Sólo se incluye en el Output si no se encontró registro en el día de hoy

Lenguaje de DP

Grupos: cláusulas de paginado

- Count y Skip.
- Para manejar cuántos registros irán a la salida.

```
Customers
{
   Customer [Count = 20] [Skip = 100]
   {
      Code = CustomerId
      Name = CustomerName
   }
}
```

Se saltean los primeros 100 customers y se incluyen en el Output los 20 siguientes.

Lenguaje de DP

Grupos: cláusula NoOutput

- En un Grupo significa que el grupo no deberá estar presente en la salida: sino solo sus elementos subordinados.

```
Employees
{
   Employee
   {
      Id = EmployeeId
      Name = EmployeeName
      EarningInfo [NoOutput]
      Where IsAutorized(&UserId)
      {
         Salary = EmployeeSalary
         Bonus = EmployeeBonus
      }
   }
}
```

```
<Employees>
   <Employee>
      <Id>123</Id>
      <Name>John Doe</Name>
      <Salary>30000</Salary>
      <Bonus>5000</Bonus>
   </Employee>
   ...
</Employees>
```

```
<Employees>
   <Employee>
      <Id>123</Id>
      <Name>John Doe</Name>
      <EarningInfo>
         <Salary>30000</Salary>
         <Bonus>5000</Bonus>
      </EarningInfo>
   </Employee>
   ...
</Employees>
```

…en lugar de…

No lo hemos mencionado, pero como intuitivamente podemos pensar, de tener un par de grupos “anidados” (un grupo que, entre otras cosas, contiene a otro), si cada uno debe acceder a la base de datos, entonces las tablas base, así como el criterio de navegación se determinan exactamente igual que en el caso de un par de for eachs anidados.

Por ese motivo, en el ejemplo, el grupo Country tiene tabla base COUNTRY; el grupo Customers tiene tabla base CUSTOMER, y se hará un join entre ambas tablas a la hora de recuperar la información para cargar la colección de países. Es decir, por cada registro de COUNTRY se cargará la información de sus atributos CountryId y CountryName en los elementos Id y Name del ítem Country del SDT colección “Countries” que se está cargando, y luego se recorrerá la tabla CUSTOMER filtrando por aquellos registros para los cuales CUSTOMER.CountryId = COUNTRY.CountryId, cargando para cada uno los elementos Id y Name.

La presencia de la cláusula OutputIfDetail hace que solamente se presenten en la salida aquellos países que cuenten con clientes asociados. Si un país de la base de datos no tiene cliente alguno, entonces no se presentará como item de la colección de salida, “Countries”.

Lenguaje de DP
Cláusula Input

- Hasta aquí asumimos que el Input venía de la BD; pero también se necesitan otro tipo de entradas. Por ejemplo, la salida de un Procedimiento o Data Provider. La forma obvia de trabajo es a través de variables, que una vez asignadas se manejan de forma usual:

```
&var = Proc.udp( parm1, ..., parmn)
```

```
&SdtVariable = DataProvider( parm1, ..., parmn )
```

- Pero si se necesita trabajar con una colección se necesita la cláusula Input…

Lenguaje de DP
Cláusula Input

- Para trabajar con colecciones.

```
VerySimple
{
   Month Input &i = 1 to 12
   {
      MonthNumber = &i
   }
}
```

Similar al 'For &i=1 to 12' del lenguaje procedural

Otro Input puede ser un SDT collection:

```
CustomersFromAnotherDataProvider
{
   &CustomersSDT = GetCustomers() // a DataProvider that Outputs Customers collection
   Customer Input &Customer in &CustomersSDT
   {
      Id = &Customer.Code
      Name = &Customer.Name
   }
}
```

Similar al 'For &var in Expression' del lenguaje procedural