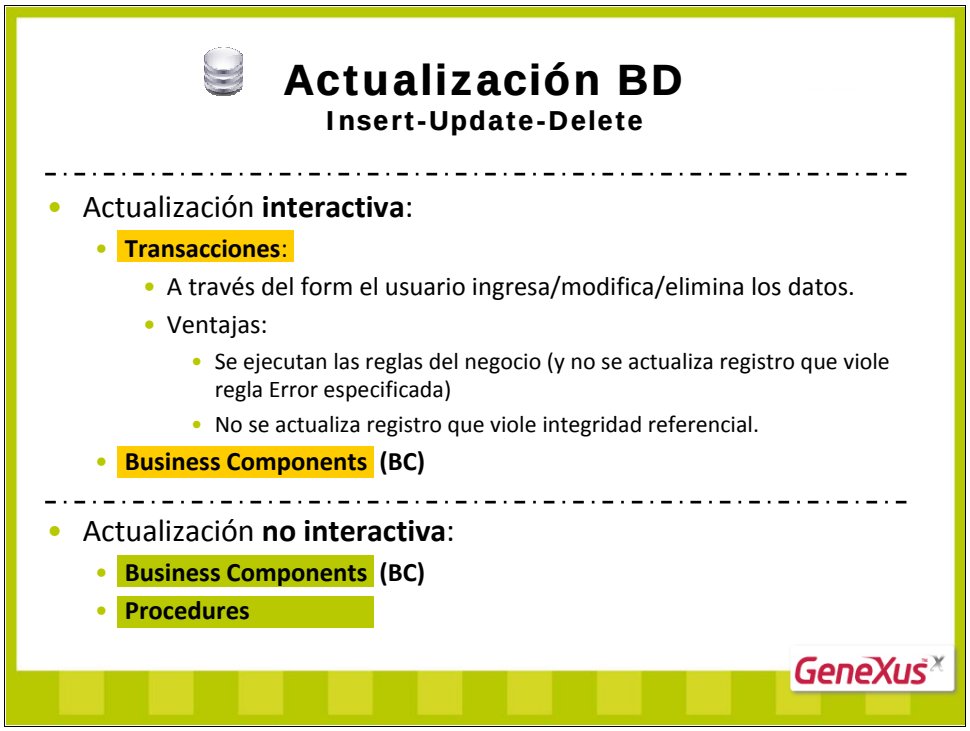
Actualización de la Base de Datos



Hasta aquí sólo conocemos una forma de actualizar la base de datos de nuestra aplicación: las transacciones.

Como sabemos, una transacción determina la o las tablas requeridas para almacenar su información. A su vez, al ser generada, se convertirá en un programa que implementa la lógica de inserción, eliminación y modificación de datos en esas tablas, en forma totalmente transparente para el programador. También la transacción permite definir todas las reglas de negocio que deberán cumplir los datos asociados. El programa generado se encarga de ejecutarlas. Así, si en una transacción tenemos una o varias reglas Error, que se disparan al satisfacerse determinadas condiciones, esas reglas estarán incluidas en el código generado, y no se permitirá actualizar la base de datos hasta tanto las condiciones que disparan esas reglas dejen de satisfacerse para los datos que se están manejando en esa oportunidad.

Asimismo, como vimos, las transacciones aseguran el control de integridad referencial, muy importante para asegurar la consistencia de la base de datos.

Pero las transacciones no cubren todas las necesidades en cuanto a la actualización de datos. También se necesitará una forma no interactiva, batch, de realizar actualizaciones sobre tablas.

Para ello existen dos alternativas: actualización utilizando lo que se conoce como Business Component, absolutamente relacionado con las transacciones, o utilizar comandos específicos para tal fin dentro de objetos de tipo Procedure.

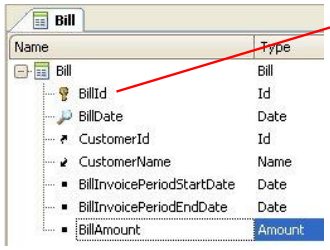
Luego veremos que los Business Components permiten gran flexibilidad, dado que permiten actualizar la base de datos de cualquier forma: tanto interactiva como no interactivamente.

En lo que sigue introduciremos los Business Components y luego los comandos de actualización directa dentro de Procedimientos.

Business Components

Escenario

- Necesitamos registrar los recibos que efectuamos a nuestros clientes.



autonumber

Rules:
Default(BillDate, &today);
Error('The invoice period considered must be past') if BillInvoicePeriodEndDate > BillDate;
Error('Wrong invoice period') if BillInvoicePeriodStartDate > BillInvoicePeriodEndDate;

Pero este es un caso en el que no nos sirve la transacción, dado que la generación de recibos es una tarea batch: se lanza una vez al mes un proceso que calcula y registra todos los recibos correspondientes a las facturas dentro de determinado período.

GeneXus

Business Components
Escenario

- Ya teníamos declarado el Data Provider 'GetBills' que realizaba el proceso de cálculo de recibos deseado, pero no grababa:

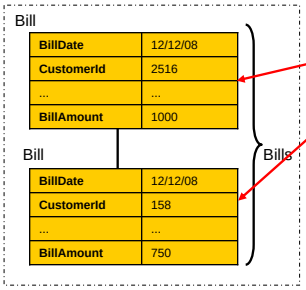
```
Bill
  {
    BillDate = &today
    CustomerId
    BillInvoicePeriodStartDate = &start
    BillInvoicePeriodEndDate = &end
    BillAmount = sum( InvoiceAmount, InvoiceDate >= &start and InvoiceDate <= &end and
                                                    InvoicePendingFlag )
  }
```

```
parm( in: &start, in: &end);
Output: Bill
Collection: True
Collection Name: Bills
```

Y nos devolvía una colección de tipos de datos estructurados (SDT).
Si observamos, la estructura jerárquica de este DP es casi idéntica a la estructura de la transacción Bill...

Business Components
Escenario

- ...¿y si pudiéramos guardar en una estructura los datos y luego grabarlos en la base de datos?




```
For &bill in GetBills( &start, &end )
    &bill.Save()
endfor
```

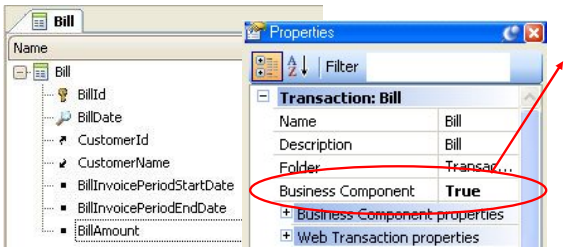
¿y si además esta grabación se realizara ejecutando las mismas reglas que se ejecutan cuando se graba mediante la transacción Bill, y controlando además la integridad referencial?

¡&bill no será una variable de tipo SDT, sino Business Component!

GeneXus

Business Components
BC

- A partir de una transacción, se puede crear su **Business Component (BC)**.



Permite encapsular la lógica de la transacción (reglas, eventos, controles de integridad referencial, estructura de los datos) de manera de poder realizar inserciones,modificaciones y eliminaciones, sin necesidad del form...

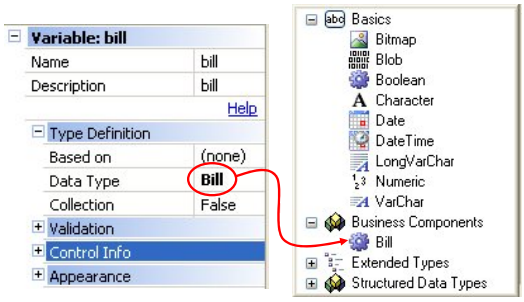
Se creará automáticamente en la KB un tipo de datos Business Component, que podrá asociarse a variables, y cuyo nombre será el de la transacción.

GeneXus

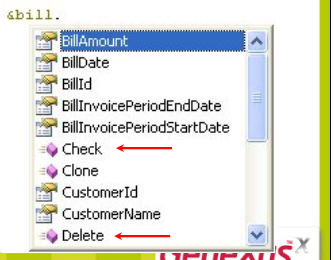
Por defecto la propiedad Business Component estará apagada. Eso significará que la lógica de negocio dada por la transacción, solo se utilizará dentro de la propia transacción, y no se creará un tipo de datos de igual nombre que la transacción, Bill, que permita encapsular la lógica de la transacción para utilizarla desde otros objetos.

Business Components

En cualquier objeto GeneXus (ej: un procedimiento):



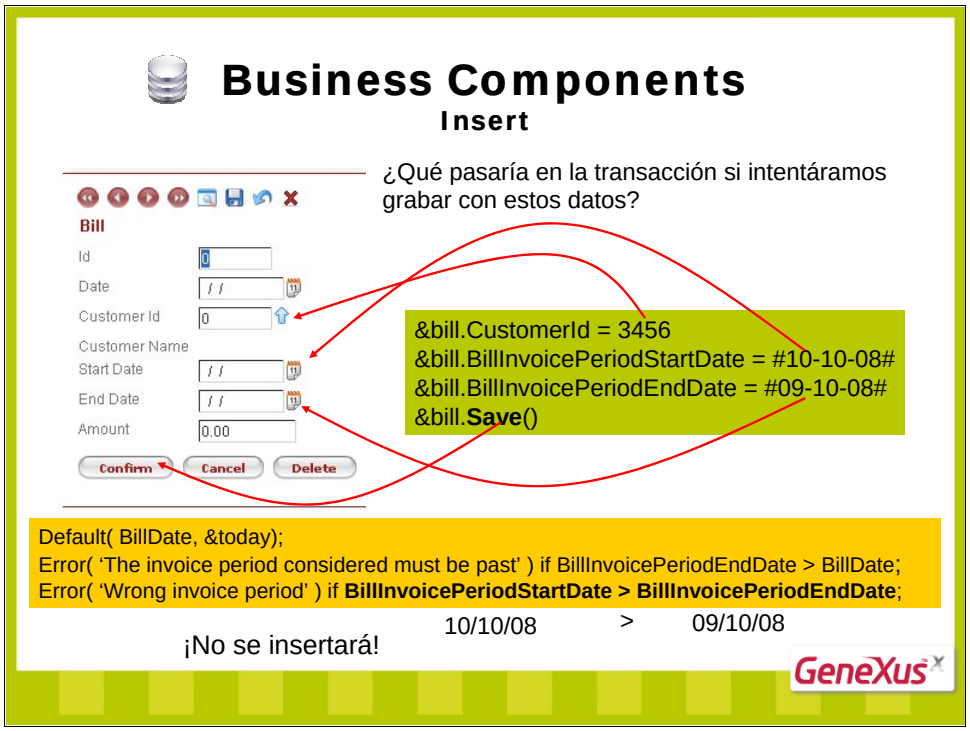
La variable &bill tendrá la misma estructura que el SDT obtenido a partir de la transacción, y se trabaja de forma análoga, pero además ofrecerá **métodos** para **actualizar** la base de datos en función de esos valores.




```
&bill.BillInvoicePeriodStartDate = #10-10-08#
&bill.BillInvoicePeriodEndDate = #09-10-08#
&bill.Save()
```

¿Se insertará registro en la tabla BILL?

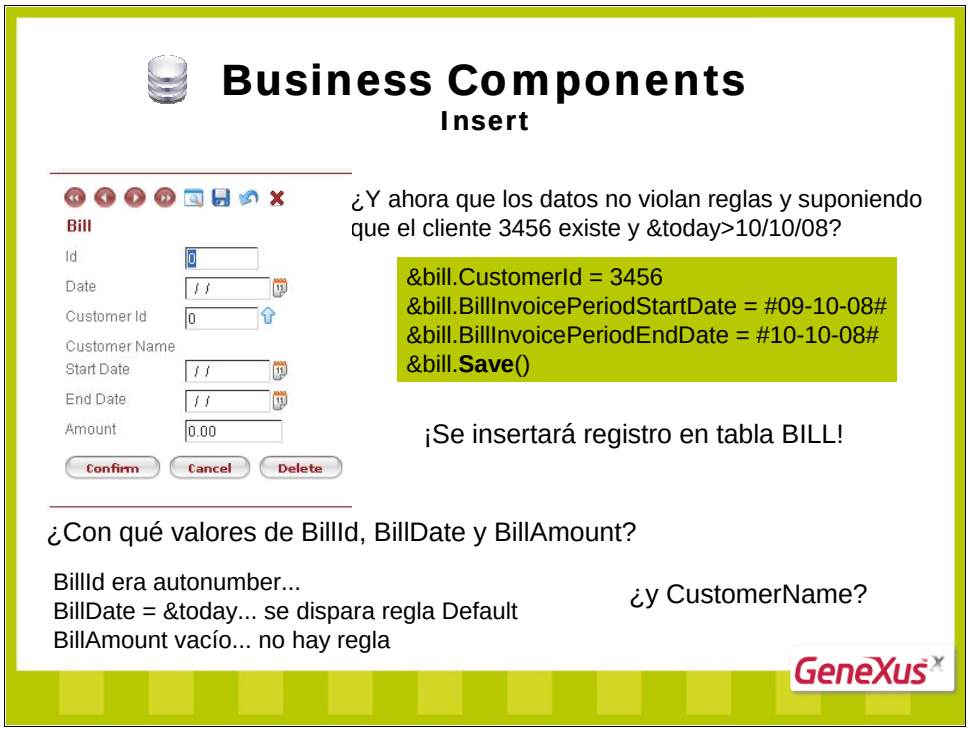
Obsérvese que en este caso la fecha inicial corresponde al día 10 de octubre, mientras la fecha final corresponde al día 9 de octubre... es decir, la fecha inicial será posterior a la fecha final...



Obsérvese que se llenan algunos miembros de la estructura de la variable &bill, Business Component, y luego se ejecuta el método Save (inexistente en SDTs).

Esto es equivalente a lo que en forma interactiva ocurriría si el usuario final ingresa esos mismos valores en los atributos correspondientes del form de la transacción, y luego presionara el botón 'Confirm'.

Por tanto, al igual que como ocurre con la transacción, deberán dispararse las reglas y realizarse los controles de integridad referencial. ¿Qué pasará con ese juego de datos si se intenta grabar mediante la transacción? Se impedirá la grabación, debido a que se está cumpliendo la condición de la segunda regla de error (también podría cumplirse la de la primera si la fecha de hoy fuera anterior al 10 de setiembre de 2008). Por otro lado, si no existiera en la tabla CUSTOMER un cliente con identificador 3456, fallará la integridad referencial y así tampoco se dejaría ingresar el registro en la tabla BILL de base de datos.



Antes de ejecutar el método Save del Business Component, el valor de BillId en la estructura de &bill será vacío. Sin embargo, una vez que se ejecute el Save, como el atributo BillId era autonumber en la tabla BILL, se ejecutará la propiedad a nivel de la tabla y el registro se insertará con el número siguiente al último dado.

En el caso de BillDate, análogamente, como no se le asigna valor en la estructura de &bill, antes del Save estará vacío, pero cuando éste se ejecute, se disparará la lógica de la transacción, y en particular la regla Default(BillDate, &today) declarada. Por tanto en el registro insertado en la base de datos, el valor del atributo BillDate corresponderá a la fecha de hoy.

Para BillAmount podríamos hacer el mismo análisis. Pero a diferencia del caso anterior, no hay regla que le asigne valor, por lo cuál se insertará el registro con valor vacío en este atributo.

De asignarse valor para &bill.CustomerName no será considerado a la hora de la inserción, dado que CustomerName es un atributo inferido, y no tiene regla Update que permita modificarlo.

Lo mismo sucedería para atributos definidos a nivel de la estructura de la transacción pero que fueran fórmulas. Es decir, son atributos virtuales en la transacción, por lo que también lo son a nivel del Business Component. Se disparará su valor.

Business Components

Update-Delete

¿Y si lo que deseo es actualizar un registro, no insertar uno nuevo?

```
&bill.Load( 100 )
&bill.BillDate = #06-01-08#
&bill.CustomerId = 1258
&bill.Save()
```

¡Se actualizará registro 100 de tabla BILL!

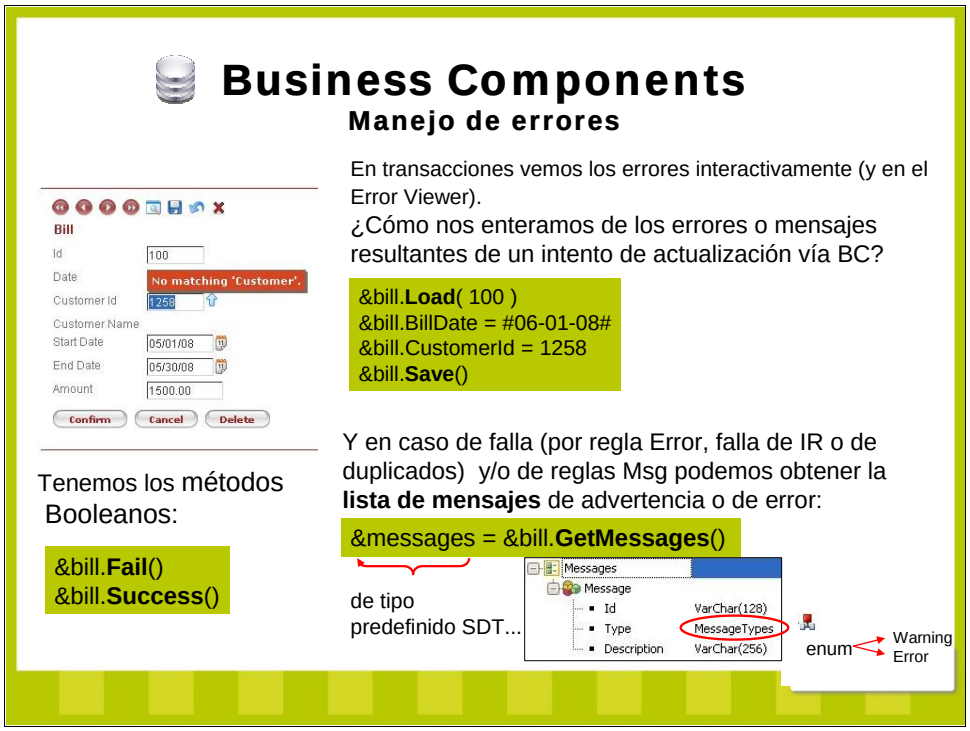
(siempre que exista cliente 1258 y 06/01/08>=BillInvoicePeriodEndDate)

¿Y para eliminar el recibo 100?

```
&bill.Load( 100 )
&bill.Delete()
```

¡Se realizará siempre y cuando no exista ninguna tabla que referencie al Bill 100!

GeneXus



Para manejar los errores habrá que definir una variable de tipo de datos SDT predefinido (viene con la KB) Messages (collection).

Cuando se ejecutan los métodos: **Save, Check, Load, Delete** se disparan y cargan los mensajes generados automáticamente por GeneXus así como las reglas Msg y Error definidos en la transacción. Se recomienda que siempre se recupere la lista de estos mensajes y se haga un “manejo de errores”.

Los mensajes más comunes generados automáticamente por GeneXus son:

Id	Type	Description
PrimaryKeyNotFound	MessageTypes.Warning	Data with the specified key could not be found
DuplicatePrimaryKey	MessageTypes.Warning	Record already exist
ForeingKeyNotFound	MessageTypes.Error	No matching 'TableName'
CandidateKeyNotFound	MessageTypes.Error	You can't update a record without reading it first
CannotDeleteReferencedRecord	MessageTypes.Error	Invalid delete, related information in 'TableName'
RecordIsLocked	MessageTypes.Error	Record is in use by another
OutOfRange	MessageTypes.Error	'AttributeName' is out of range
RecordWasChanged	MessageTypes.Error	'TableName' was changed

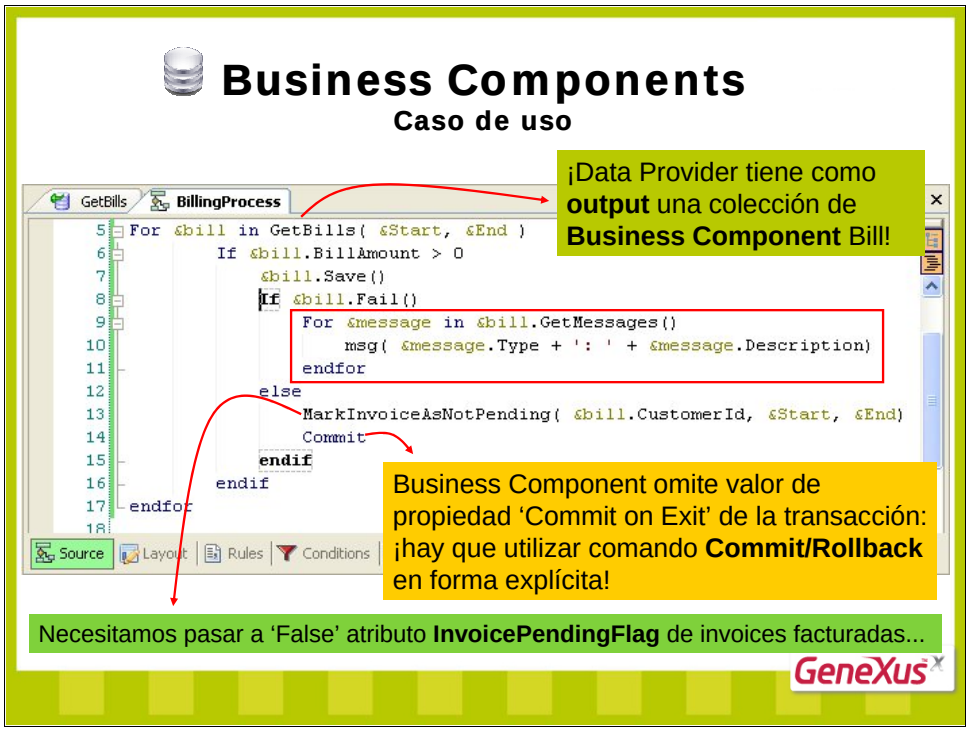
Las reglas Msg y Error aceptan en su definición además del parámetro con el mensaje a desplegar, un segundo parámetro que define el Identificador del mensaje. El objetivo es que cuando se ejecute la transacción como Bussiness Component, y se obtenga la lista de mensajes ocurridos luego de ejecutar una acción sobre la base de datos, se tenga de cada mensaje, además del mensaje en sí, su identificador, siendo posible así evaluar el identificador del mensaje para codificar el comportamiento en consecuencia:

Msg|Error(<mensaje>, <Id del mensaje>)

Ejemplos de <Id del mensaje>: "1", "2", "Error1", "Error2" o una descripción como ser "CustomerNameCannotBeEmpty", etc.

De no especificar un identificador para el mensaje, habrá que preguntar por el texto del mensaje.

Nota: Para los mensajes generados automáticamente por GeneXus, el **Id** es siempre en Inglés (independientemente del idioma seleccionado en el modelo).



Aquí completaríamos el proceso de generación de recibos.

Observemos primeramente un hecho importante: un data provider no solo puede devolver un SDT simple o colección, sino también un BC simple o colección.

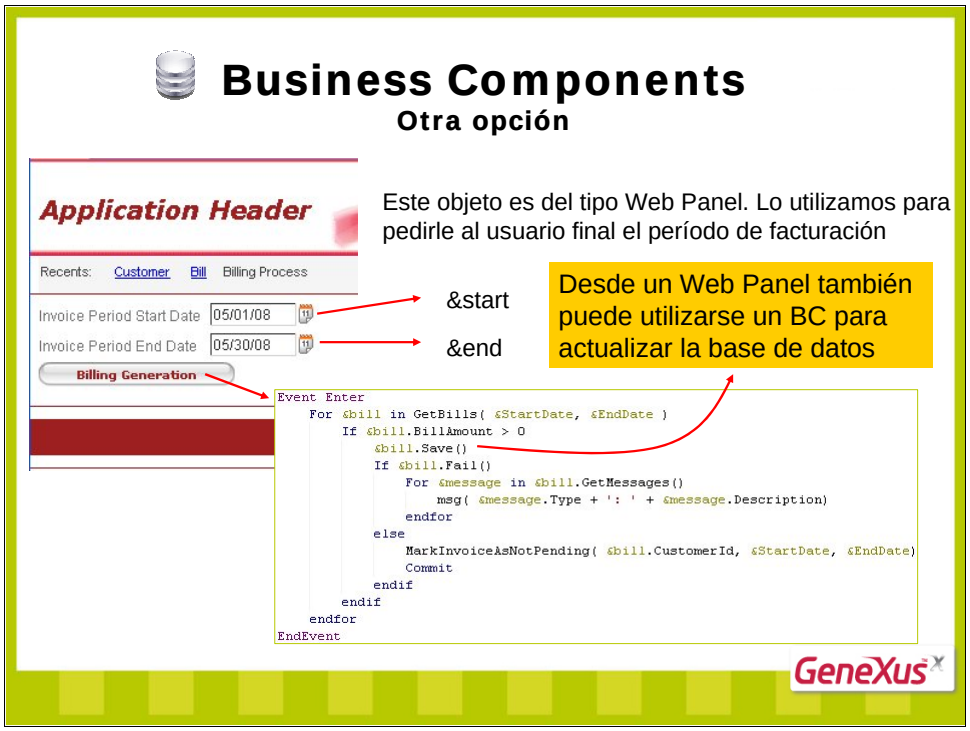
Importante: el BC devuelto solo podrá insertarse en la base de datos. Es decir, un Data Provider solo permite llenar su estructura, para luego hacer una operación de INSERT sobre la base de datos. No será posible hacer una actualización o eliminación con un BC cargado vía Data Provider.

Obsérvese por otro lado cómo se ha definido la variable &message de tipo de datos Messages.Message, correspondiente a los ítems del SDT predefinido Messages devuelto por el método **GetMessages** del BC.

Una vez que hemos obtenido e ingresado en la tabla BILL todos los recibos correspondientes a la facturación, deberíamos recorrer las facturas procesadas, y modificar el valor de su atributo InvoicePendingFlag, pasándolo a False. Recordemos que el valor de este atributo se utiliza para no procesar dos veces una misma factura. Recordemos que este valor se utiliza en el Data Provider en el cálculo del elemento BillAmount:

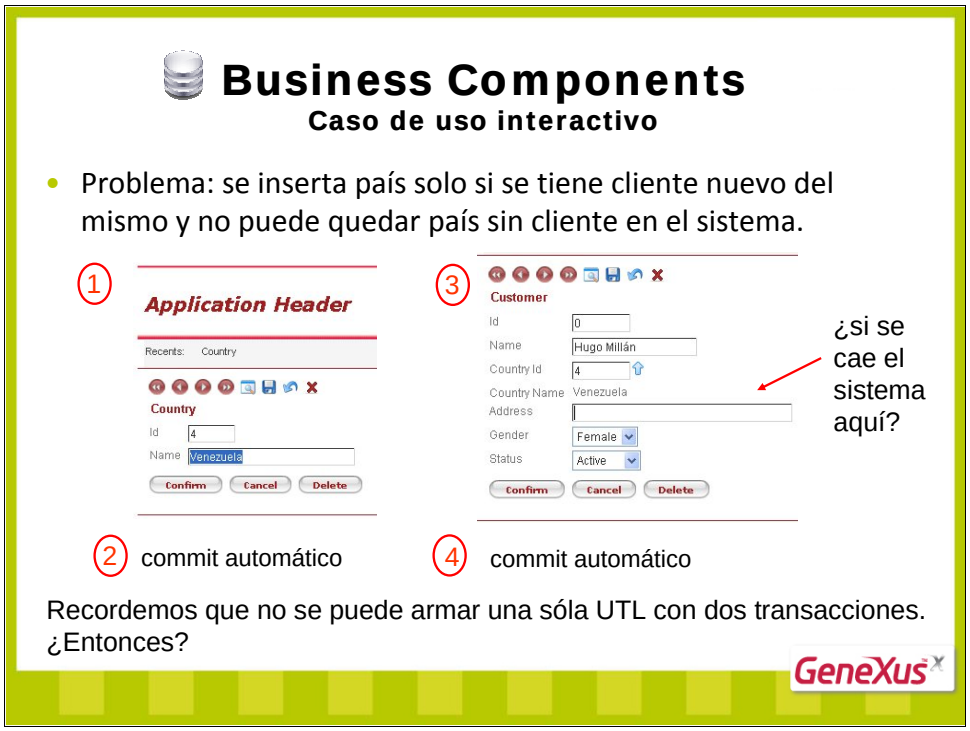
sum(InvoiceAmount, InvoiceDate >= &start and InvoiceDate <= &end and InvoicePendingFlag)

Dejaremos pendiente esta última parte para cuando estudiemos, unas páginas más adelante, las formas de actualización directa (dentro de procedimientos exclusivamente), de los registros de las tablas.



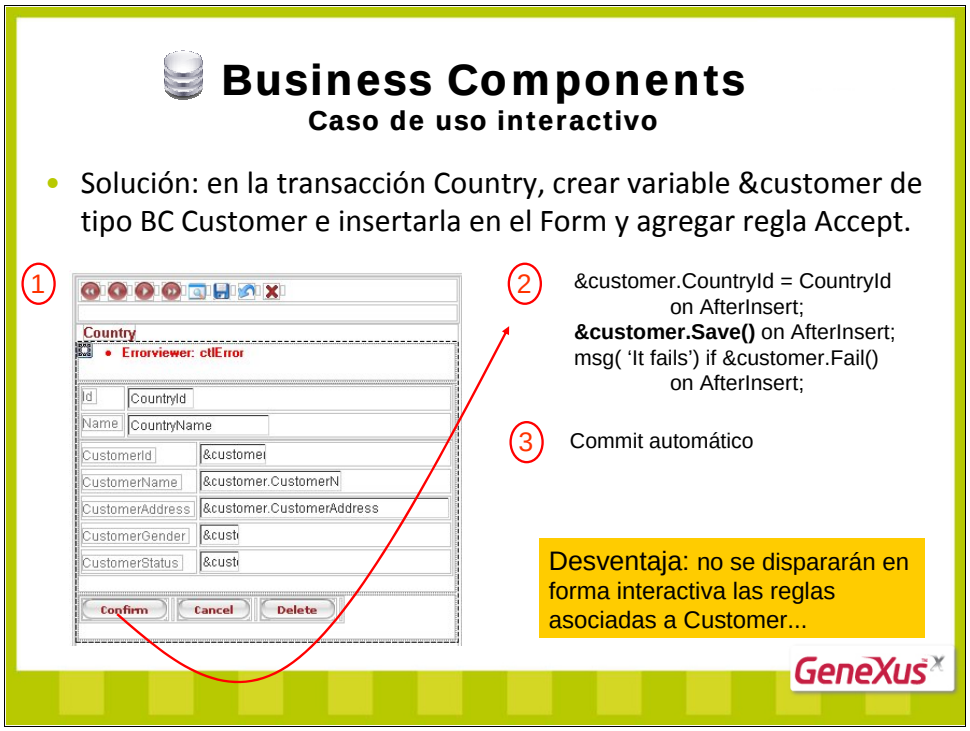
Adelantándonos aquí a presentar el otro tipo de objeto interactivo que estudiaremos un poco más adelante, el Web Panel, mostramos un objeto de este tipo en ejecución. Su función será pedir al usuario final un par de valores, que almacenará en las variables correspondientes de tipo Date (&start y &end) y luego, cuando el usuario presione el botón asociado al Evento Enter del Web Panel, se ejecutará su código.

Obsérvese que el código es idéntico al del procedimiento visto antes. Con este ejemplo pretendemos mostrar que mediante un Business Component puede actualizarse la base de datos desde cualquier objeto GeneXus.



Supongamos que cada vez que se inserta un nuevo país en el sistema, es necesariamente debido a que se tiene un nuevo cliente de ese país. Supongamos que como requerimiento, además, no debe quedar bajo ninguna circunstancia ingresado un país sin por lo menos un cliente asociado.

Tal como tenemos implementada la aplicación, esto no se está controlando. ¿Cómo se ingresa un país y un cliente para el mismo? El usuario debe ejecutar la transacción Country, ingresar el país. Luego abrir la transacción Customer, e ingresar los datos del cliente y confirmar. ¿Pero qué sucederá si se cae el sistema cuando se está ingresando el cliente? Quedará el país ingresado (recordar que cada transacción por defecto hace commit al final).



Necesitamos que la inserción del país en la tabla COUNTRY y la inserción del cliente en la tabla CUSTOMER se realicen dentro de una misma UTL. Como hemos visto, las operaciones efectuadas por dos transacciones no pueden incluirse en una única UTL. Por esta razón, necesitaremos hacer la inserción dentro de la misma transacción.

¿Cómo? Dentro de la transacción Country, definimos una variable &customer, de tipo business component Customer (para ello, tendremos previamente que haber prendido la propiedad Business Component de la transacción Customer, de manera tal que se cree este tipo de datos en la KB). Una vez efectuado esto, simplemente insertando en el form la variable (al igual que sucedía con un SDT), GeneXus colocará controles para cada uno de los miembros de la estructura: &customer.CustomerId, &customer.CustomerName, &customer.CustomerAddress, etc.

Las variables por defecto en las transacciones son de salida, por lo que habrá que especificar regla Accept para que puedan ser de escritura en ejecución y el usuario pueda ingresar valores para las mismas.

El Save del customer deberá realizarse necesariamente luego de insertarse el país en su tabla (de lo contrario fallará la integridad referencial). Por tanto en las reglas:

&customer.Save() on AfterInsert;

Sabemos que en este momento ya se habrá grabado el registro correspondiente al país, y luego vendrá este Save... a continuación, el commit automático.

Business Components
Resumen

- Objetivo:
  - Reutilizar la lógica del negocio definida en las transacciones. Usar el poder de las transacciones (sin sus forms) desde otros objetos GeneXus:
    - Procedimientos, Web Panels...
    - desde otra Transacción (al igual que una variable podía llenarse interactivamente insertándola en el Form, también un BC)

- Beneficios:
  - Actualización a la BD garantizando integridad de los datos y ejecución de las reglas del negocio.
  - Reutilización de código (no necesidad de duplicar reglas de negocio)
  - Varios objetos GeneXus pueden actualizar la BD.

Si existe una regla que invoca a un objeto que tiene interfaz, es decir, form, esa regla no se incluye en el BC. Existe una forma de especificar que una regla declarada en la transacción no aplique cuando se ejecuta la transacción, sino solo cuando se ejecuta el Business Component asociado: es calificando la regla con [BC].

Ejemplo:

```
[BC] Default( BillDate, &today);
```

Si se quiere calificar de una sola vez un conjunto de reglas:

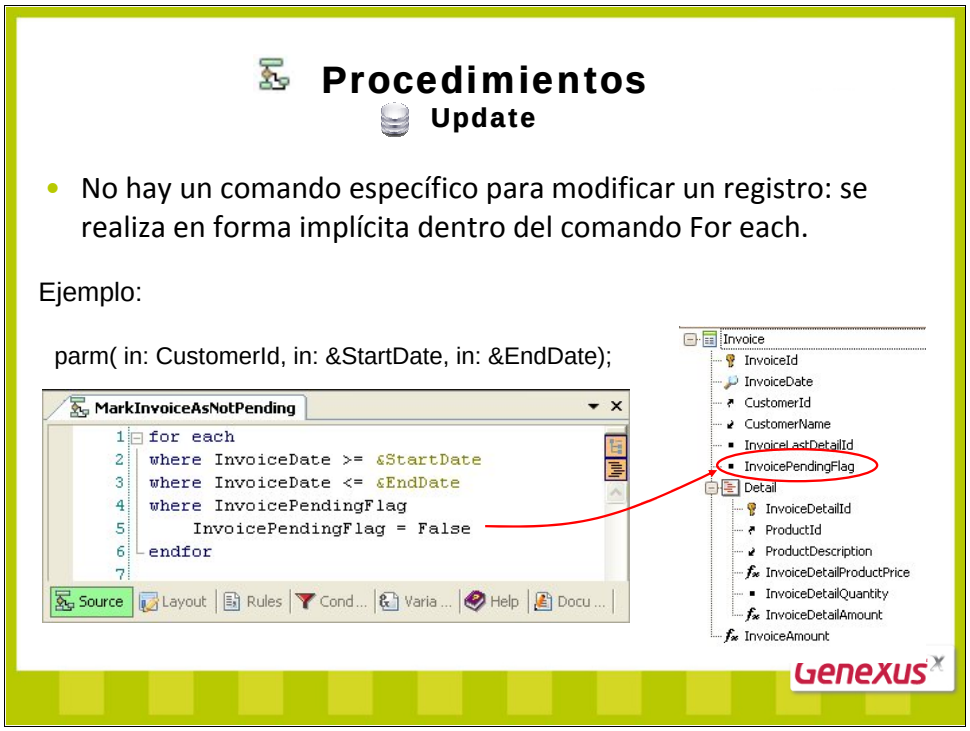
```
[BC]
{
   regla1;
   regla2;
   ...
   reglan;
}
```

Lo mismo vale para eventos.

Análogamente, existen calificadores para indicar que una regla solo se ejecute si se está corriendo la transacción con su form web: [WEB].

Actualización no interactiva directa

Procedimientos



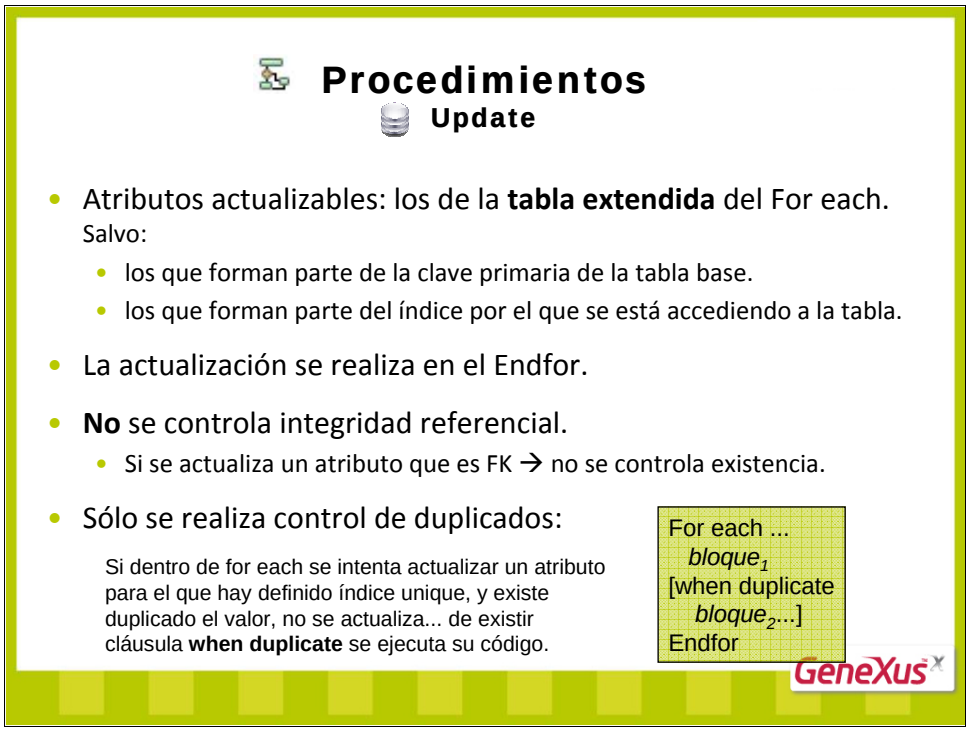
La modificación de datos de la base de datos se realiza en forma implícita: no hay un comando específico de actualización.

Para actualizar uno o varios atributos de una tabla se utiliza el comando For each, y dentro del mismo el comando de asignación.

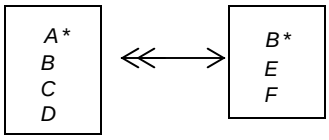
Se pueden actualizar varios atributos dentro del mismo For each, pudiendo éstos pertenecer tanto a la propia **tabla base** como a la **tabla extendida**.

El ejemplo que presentamos completa el que habíamos iniciado respecto al proceso de facturación.

Obsérvese que aquí se está recorriendo la tabla INVOICE, filtrando por InvoiceDate, por InvoicePendingFlag, así como por CustomerId (en el listado de navegación se observará que si bien no se ha especificado order, al recibir en el atributo CustomerId, que es foreign key, se ordena por este atributo para optimizar). Dentro del for each, para cada factura que cumple las condiciones, se actualiza el valor del atributo InvoicePendingFlag. Aquí podrían actualizarse no solo ese atributo, sino cualquiera de la propia tabla INVOICE, o de su extendida, con las excepciones que se indican a continuación.



Supongamos que tenemos el siguiente diagrama de Bachman genérico:



Y en el Source de un procedimiento hacemos:

```
For each
    C = &C
    E = &E
    D = &D
Endfor
```

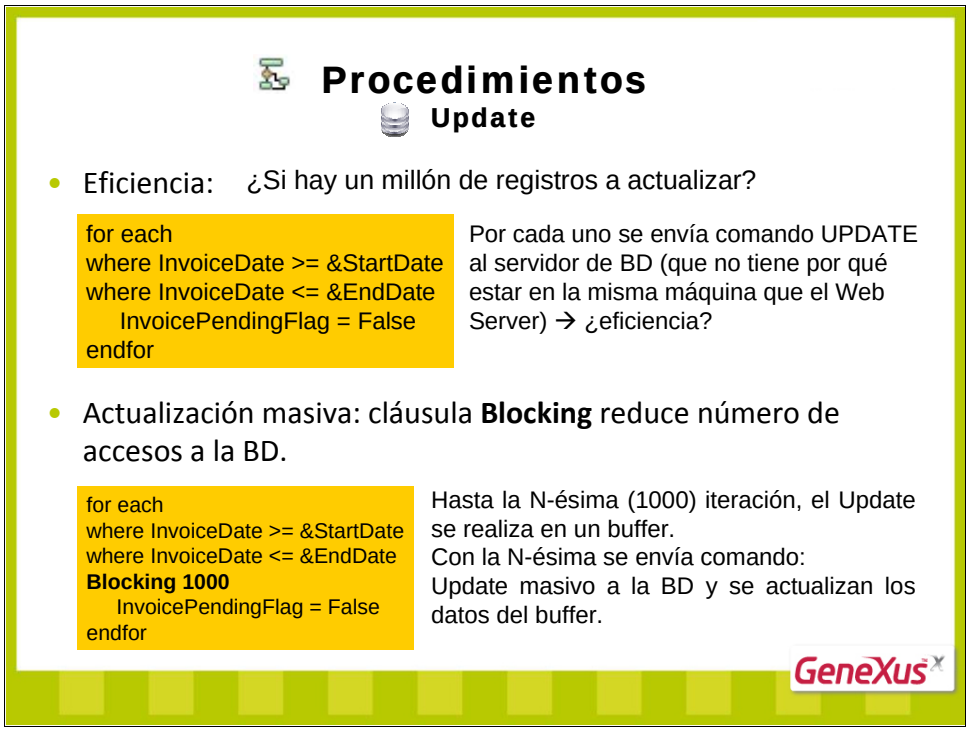
Aquí la tabla base del For each será claramente la de clave primaria *A* y dentro del For each estamos actualizando tanto atributos de la propia tabla base como de la extendida. ¿En qué momento ocurre efectivamente la actualización de los registros involucrados?

La actualización no ocurre ni bien se encuentra un comando de asignación dentro del For each, sino luego de que se encuentran todos, para cada instancia de la tabla base, es decir, cuando se llega al **Endfor** para cada iteración.

Como vimos antes, no podemos actualizar dentro del comando For each atributos de la clave primaria. Sin embargo podríamos querer actualizar un atributo que sin ser clave primaria, está definido como clave candidata (mediante un índice unique).

Si el atributo es clave candidata, debe controlarse que no se dupliquen sus valores, por lo que de encontrarse duplicado el registro en este sentido, no se permitirá hacer la actualización.

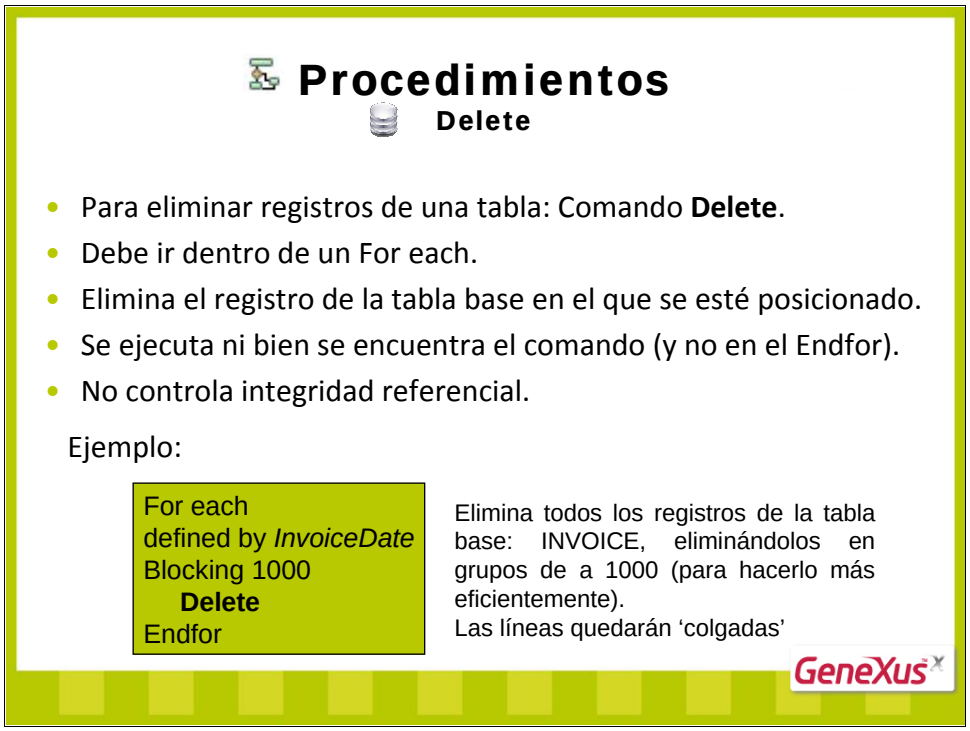
Si se desea tomar una acción en caso de que esto ocurra, el comando For each agrega la **cláusula when duplicate**. Solo tiene sentido si existe alguna clave candidata para ese For each.



Realizar un “blocking” a las operaciones de actualización de la BD significa almacenarlas en memoria y enviarlas en grupo al DBMS. En lugar de interactuar con el DBMS en cada operación de actualización, la interacción tiene lugar solamente cada N operaciones de actualización, donde N es el número que se establece en la cláusula Blocking.

No será el caso de nuestro ejemplo, pero ¿qué sucedería si se está haciendo una actualización masiva que incluye algún atributo clave candidata de la tabla, y se encuentran duplicados para algunos de los registros del grupo de 1000 que se está procesando?

En ese caso, una vez llenado el buffer con las 1000 actualizaciones, al enviar a la BD el comando UPDATE del grupo, saltará el error de duplicados y se iterará sobre el grupo, realizando un comando UPDATE individual de BD, uno por uno.



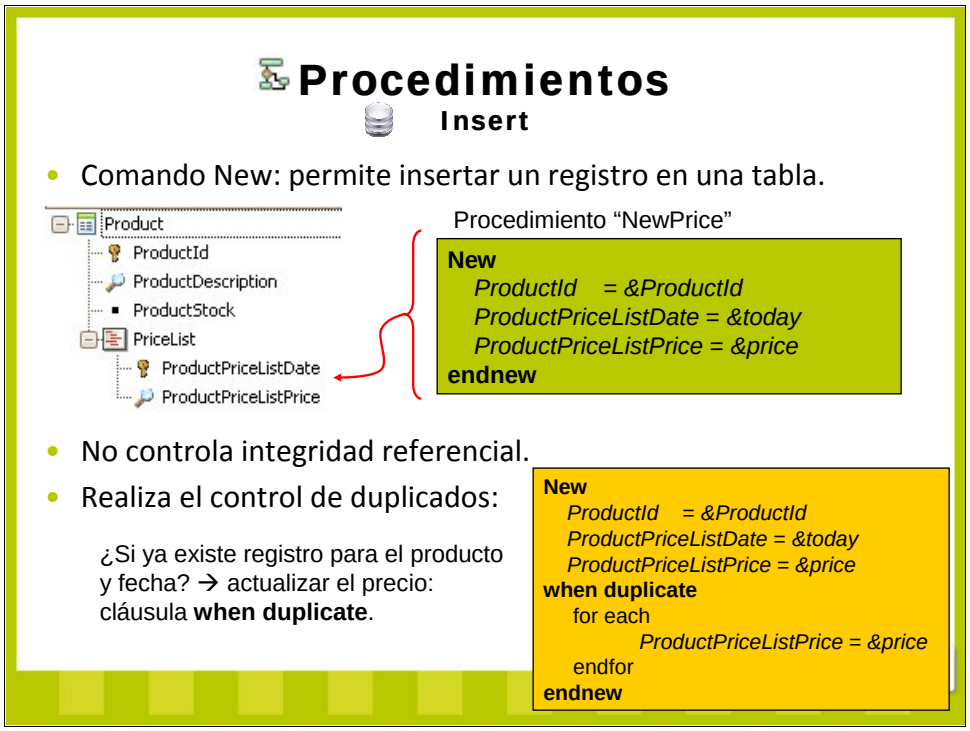
Para eliminar datos se utiliza el **comando Delete** dentro del comando For each.

El comando Delete elimina **el registro** en el que se está posicionado en un momento dado. Es por ello que no puede aparecer "suelto" dentro del Source. Debe colocarse dentro de un comando For each, cuya tabla base sea la tabla de la que se quieren eliminar registros. Solo se eliminan los registros de la tabla base, no de la extendida.

Si deseamos eliminar todas las facturas anteriores a una fecha dada, podemos programar un procedimiento:

```
For each
where InvoiceDate <=&date
      For each
     defined by InvoiceDetailQuantity
          DELETE      //se eliminan las líneas
     Endfor
     DELETE //luego de eliminar las líneas se elimina el cabezal
Endfor
```

Para mayor eficiencia en la eliminación, dependiendo del número de registros de la base de datos, convendrá agregar cláusula Blocking con un factor de bloqueo, N, adecuado. Por ejemplo, si se agrega "Blocking 1000" la eliminación física no se realizará en cada iteración, sino que cada 1000 veces que se llegue al Endfor, se eliminarán el grupo de 1000 registros en un único acceso a la Base de Datos (en lugar de 1000).



Supongamos que queremos implementar un procedimiento que haga lo siguiente: para el producto cuyo código es recibido por parámetro, dé de alta un nuevo precio (también recibido por parámetro) en su lista de precios, para la fecha correspondiente al día en que se ejecuta el procedimiento. Este debe crear un nuevo registro en la tabla PRODUCTPRICELIST, que está compuesta por los atributos *ProductId*, *ProductPriceListDate* y *ProductPriceListPrice*, siendo su clave primaria una compuesta, conformada por *ProductId* y *ProductPriceListDate*.

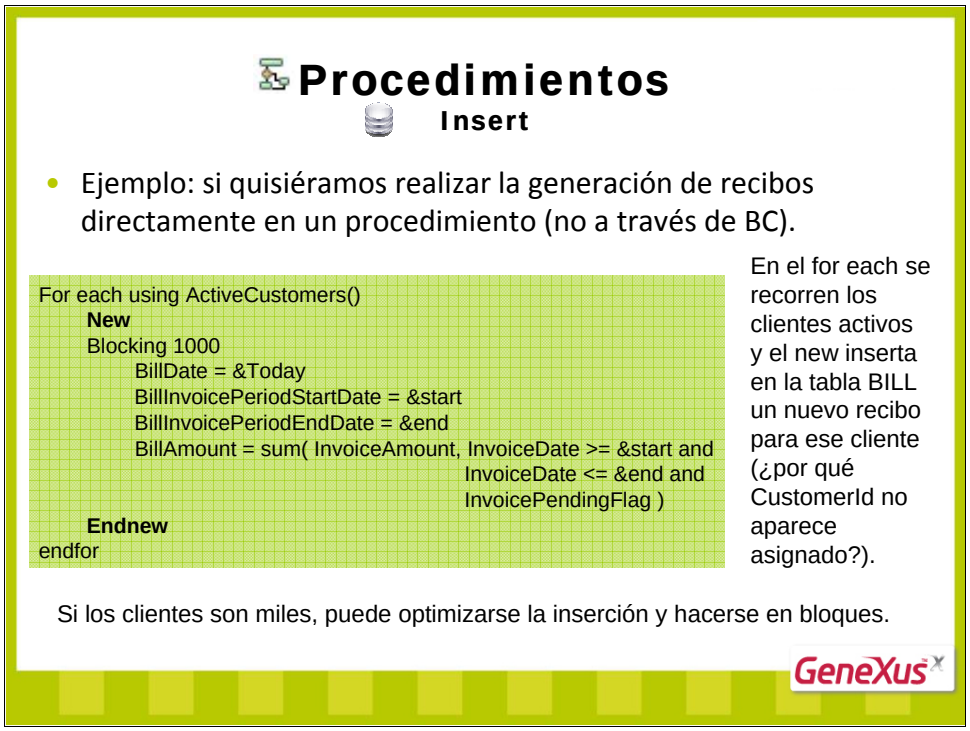
Para ello se utiliza el **comando new** que escribimos arriba. Observemos que dentro del mismo aparecen comandos de asignación, donde se le da valor a los atributos de la tabla en la que se quiere insertar el registro.

Cada vez que GeneXus encuentra un new, debe determinar la tabla en la que se realizará la inserción (**tabla base del new**). Es determinada a partir de los atributos que aparecen **dentro** del comando new, **del lado izquierdo en una asignación**.

El único control que realiza, es el de duplicados. En nuestro caso, si existiera un registro con los valores de *ProductId* y *ProductPriceListDate*, no se realizará la inserción (pero en cambio si lo que no existiera fuera un producto en la tabla PRODUCT con el *ProductId*, allí sí se insertará, pues no controla integridad referencial). Si se programa la cláusula **when duplicate**, se actualizará el registro encontrado.

Observemos que para realizar la actualización del atributo, la asignación debe estar dentro de un comando For each. Si no colocamos el For each no se realizará la actualización del precio para ese registro, es decir, es como si no se hubiera incluido cláusula when duplicate.

Si se quisiera insertar un registro para el cual existiera clave candidata duplicada, también se controlará y no se realizará la inserción. En caso de existir cláusula when duplicate, se ejecutará.



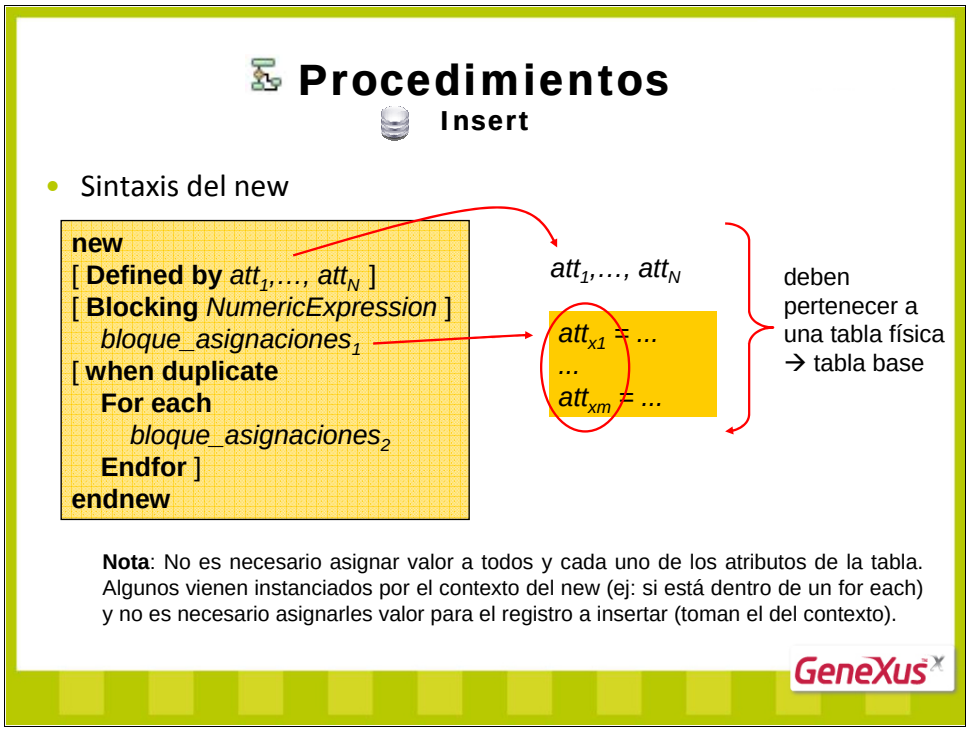
El caso más común será tener el comando new dentro de un for each, dado que en general se quieren insertar registros en base a cálculos efectuados en función de otros.

El mismo proceso de generación de recibos que habíamos resuelto anteriormente utilizando Data Provider y Business Component, podríamos realizarlo con un procedimiento que utilice el comando new de inserción. ¿Cuál alternativa elegiría usted? Recuerde que con los comandos de actualización dentro de procedimientos, el único control que se realiza es el de duplicados.

Si las inserciones son muchas, así como vimos para el caso de las actualizaciones, disponemos de la cláusula blocking para ir guardando en un buffer y luego hacer la inserción de todos esos registros a la vez.

En nuestro caso no fallará la inserción, debido a que la clave primaria es autonumber, y que no tenemos claves candidatas (vía índices unique) en la tabla de recibos (BILL).

Pero si no fuera el caso, como las inserciones se van realizando en el buffer hasta llegar a la 1000, no es posible hasta completar el buffer saber si alguna operación fallará. Cuando se ejecuta el INSERT masivo de base de datos, allí sí puede fallar alguna inserción. En ese caso, se itera en los elementos del buffer, ejecutando el INSERT simple, uno por uno. Si existe cláusula When duplicate en el New, entonces por los registros que fallen, se ejecutará la cláusula.



En la sintaxis presentada, *bloque_asignaciones*$_1$ es un bloque de código compuesto mayormente por sucesivos comandos de asignación (aquí se asigna valor a los atributos de la tabla en la que se insertará el registro, aunque también pueden asignarse valores a variables, así como anidarse otro comando new).

La **cláusula Defined By** opcional, se incorpora a los mismos efectos que lo hacía para el comando For each: ayudar a determinar la tabla base.

La tabla base del new se obtiene de los atributos del Defined By y los que aparezcan del lado izquierdo de asignaciones dentro del *bloque_asignaciones*$_1$. Aquí no se utiliza la tabla extendida.

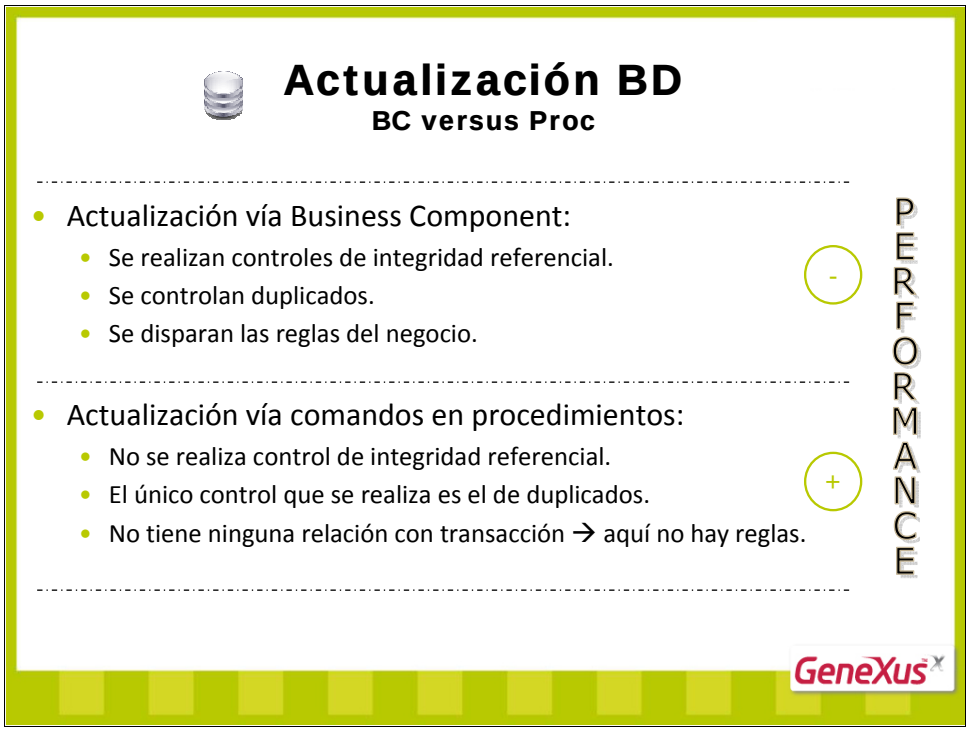
En el caso de que el **comando new** esté dentro de una cláusula repetitiva (ej. For each), es posible reducir el número de accesos a la base de datos usando la **cláusula blocking**.

El comando new realiza un **control de duplicados**, de manera tal que no se permitirá insertar un registro que ya exista en la tabla.

La **cláusula when duplicate** del comando permite programar la acción en caso de que el registro ya exista en la tabla base (tanto por clave primaria como por clave candidata).

Normalmente, de ocurrir lo anterior, se quiere actualizar algunos de los atributos de dicho registro. Para ello, en *bloque_asignaciones*$_2$ se realizan tales asignaciones, pero como lo que se hace es actualizar un registro (y no insertar uno nuevo), estas asignaciones aparecen rodeadas de “For each – Endfor”, pues como hemos visto, las actualizaciones solo pueden realizarse dentro de un For each.

De no especificarse cláusula when duplicate para un new, si el registro que quiere insertarse se encuentra duplicado no se realizará acción alguna y la ejecución continuará en el comando siguiente. Es decir, como no puede insertar el registro porque ya existe uno, no hace nada y sigue adelante, con el próximo comando.



En los procedimientos el único control de integridad que se realiza automáticamente es el control de duplicados. El control de integridad referencial queda a cargo del programador, lo que no ocurre en las transacciones (ergo, en un Business Component).

Queda claro de la enumeración mostrada arriba que los Business Components ofrecen todas las garantías y constituirán la forma de actualización privilegiada.

Cabe preguntarse entonces: ¿cuándo actualizar utilizando estos comandos? La respuesta es: cuando la performance sea un problema.

Un ejemplo discutible es el que utilizamos para cambiar el valor del atributo *InvoicePendingFlag* a False. No había ninguna regla que lo implicara, no involucraba integridad referencial, ni duplicados y podía involucrar millones de registros. ¿Cuál hubiese sido la alternativa? Prender la propiedad Business Component de la transacción Invoice, y en el procedimiento MarkInvoiceAsNotPending definir variable &invoice con ese tipo de datos y luego:

```
for each
where InvoiceDate >= &startDate
where InvoiceDate <= &endDate
where InvoicePendingFlag
        &invoice.Load ( InvoiceId )
        &invoice.InvoicePendingFlag = False
        &invoice.Save()
endfor
```

Pero esta solución será más ineficiente que la actualización directa utilizando “blocking factor”, sabiendo que se deberán actualizar millones de facturas.