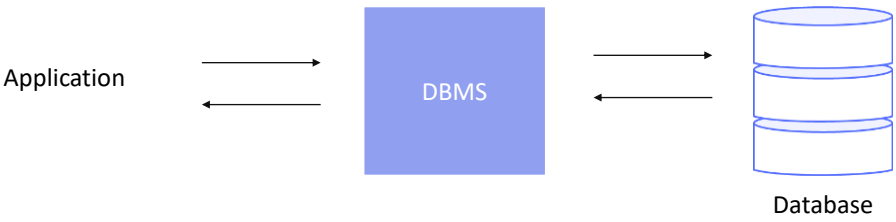
Transactional Integrity

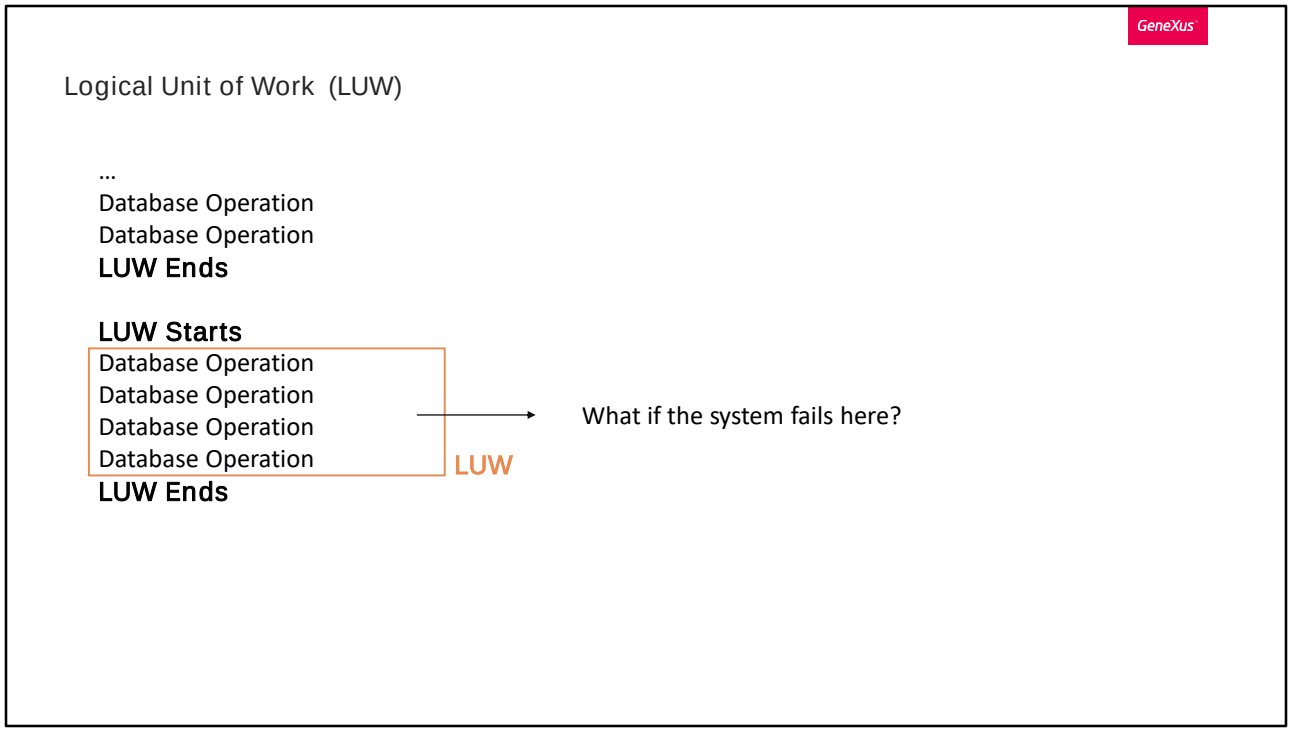
GeneXus™

GeneXus

Database Management System



Muchos manejadores de bases de datos (DBMSs) cuentan con sistemas de recuperación ante fallos, que permiten dejar la base de datos en estado consistente cuando ocurren imprevistos tales como apagones o caídas del sistema.

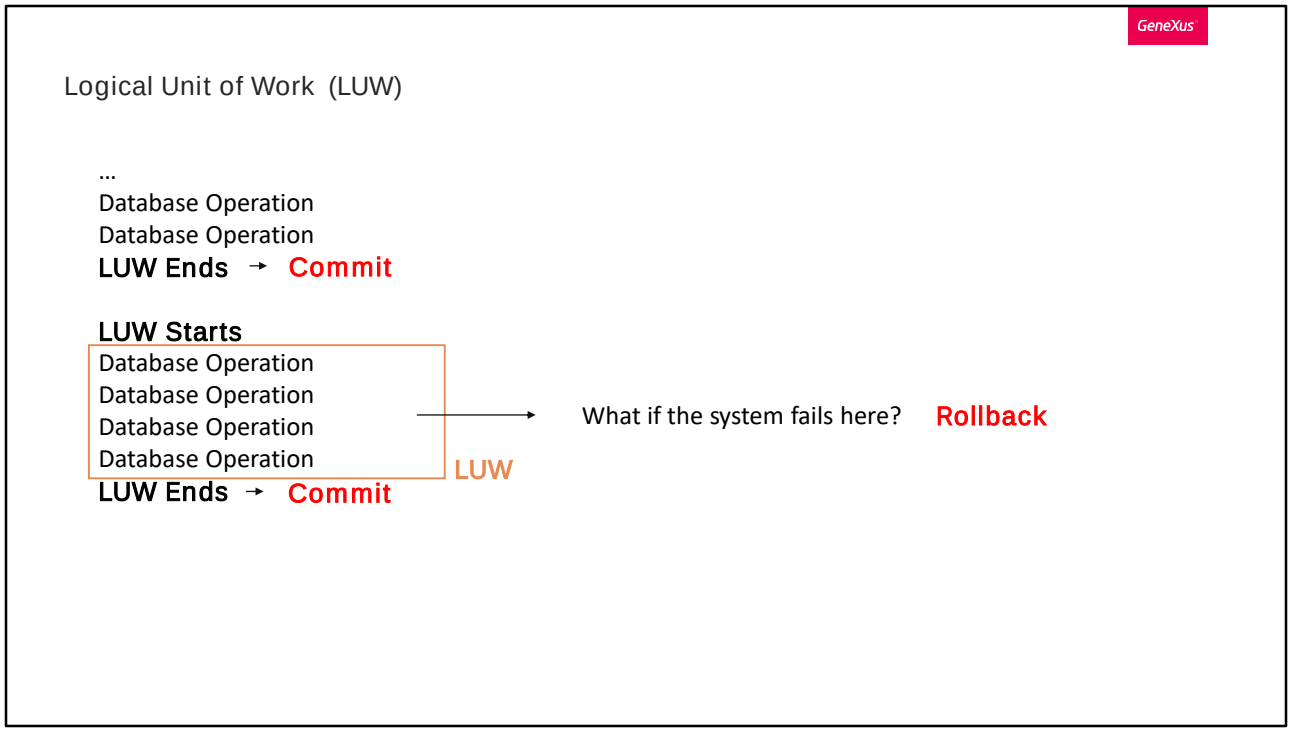


Los manejadores de bases de datos (DBMSs) que ofrecen integridad transaccional permiten establecer unidades de trabajo lógicas (UTL), en Inglés
Logical Unit of Work, que corresponden ni más ni menos que al concepto de “transacciones” de base de datos.

Básicamente, una UTL es un conjunto de operaciones a la base de datos, las cuales deberán ejecutarse todas o ninguna. O sea, no es posible, que dentro de una UTL se ejecuten algunas operaciones, y otras no. Esto nos asegura la integridad de la base de datos.

En el ejemplo mostramos una UTL que consiste de cuatro operaciones sobre la base de datos, podrían ser inserciones, actualizaciones o eliminaciones. Supongamos que las dos primeras fueron realizadas exitosamente, pero antes de ejecutarse la tercera se cae el sistema. Como la UTL no se completó deberán deshacerse las dos operaciones que sí habían llegado a efectuarse. De no hacerlo, como las UTLs son las que definen a nivel lógico los estados consistentes de la base de datos, ésta quedaría inconsistente.

En este caso el DBMS realiza lo que se llama un Rollback para recuperarse, conservando el último estado consistente de la base de datos.



Es el comando Commit el que determina el fin de una UTL.
Por lo que una UTL queda definida por las operaciones que se realizan entre dos commits.

Si el sistema se cae donde se indica, las dos operaciones realizadas después del último Commit (que son las operaciones pendientes de Commit) se deshacen con el Rollback automático que realiza el DBMS al recuperarse del fallo.

Es decir, como no habían quedado efectuadas todas las operaciones que conforman la UTL, se deshacen o se revierten las operaciones parciales que se habían realizado.

GeneXus

LUW in GeneXus

Transactions: At the end of each instance, immediately before the AfterComplete rule.

Procedure: At the end of the Source

Business Component: GeneXus does not write Commit.

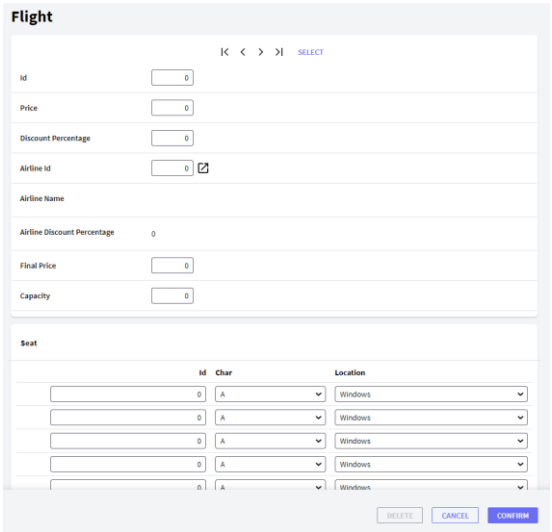
Las transacciones y los procedimientos son los objetos GeneXus creados para actualizar la información de la base de datos. Es por esta razón que GeneXus escribe el comando Commit cuando genera los programas en el lenguaje que se haya definido. ¿Dónde?

En el objeto transacción: al final de cada instancia, inmediatamente antes de las reglas con evento de disparo AfterComplete (es decir, después de haber manipulado el cabezal y las líneas).
En el objeto procedimiento: al final del Source.
Los Business Components, que se crean a partir de las transacciones, no incluyen Commit puesto que pueden ser utilizados en cualquier objeto, y será el desarrollador quien determine dónde "commitear".

Lo veremos en lo que sigue.

GeneXus

Transaction and Automatic Commit



Header

BeforeValidate

VALIDATION

AfterValidate / BeforeInsert – BeforeUpdate - BeforeDelete

RECORDING

AfterInsert - AfterUpdate - AfterDelete

For each line

VALIDATION

AfterValidate / BeforeInsert – BeforeUpdate - BeforeDelete

RECORDING

AfterInsert - AfterUpdate - AfterDelete

END ITERATION LEVEL 2

AfterLevel Level attLevel2 / BeforeComplete

COMMIT

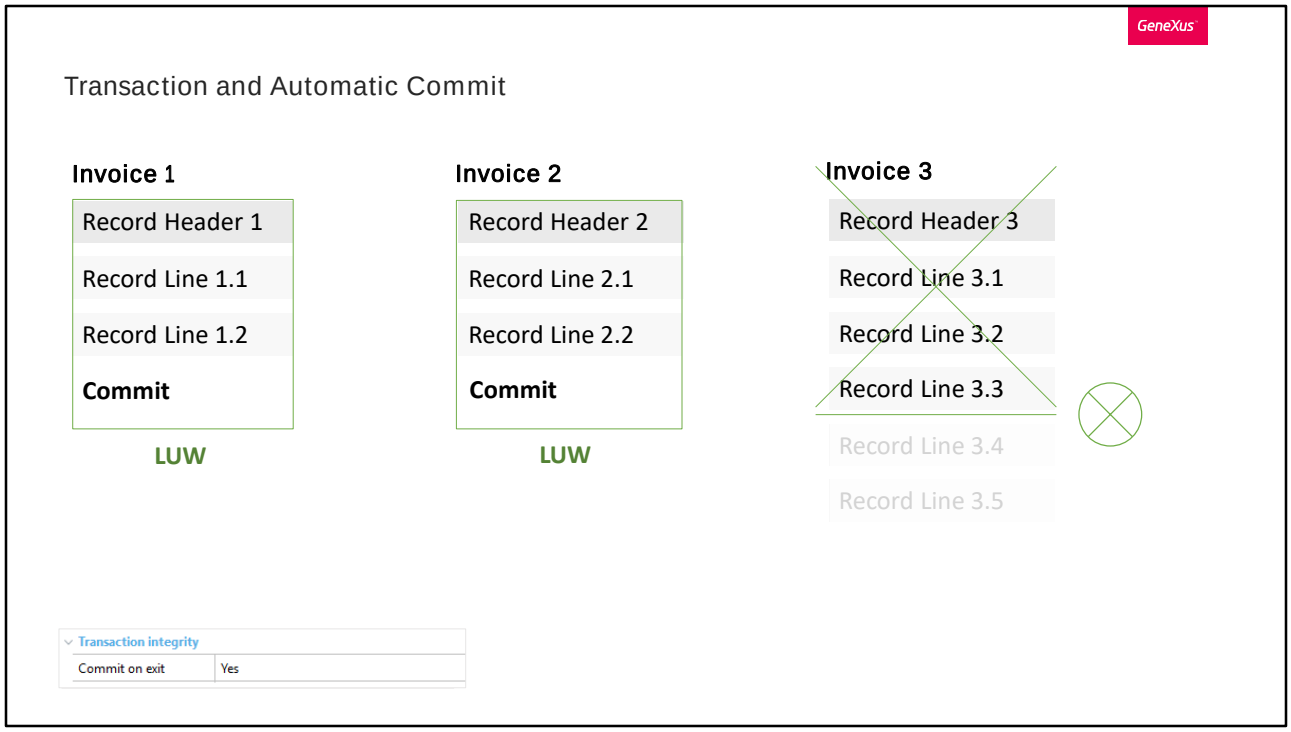
AfterComplete

El usuario manipula el cabezal y las líneas y presiona “Confirm”. En el servidor se ejecutan las reglas y fórmulas según el árbol de evaluación para el primer nivel y luego se disparan las reglas condicionadas al evento BeforeValidate. Luego de que la información del cabezal se da por válida, se disparan las reglas condicionadas a los eventos AfterValidate y, dependiendo del modo, las condicionadas a BeforeInsert, BeforeUpdate o BeforeDelete. Después de eso se graba el cabezal y se disparan las reglas que estuvieran condicionadas a AfterInsert, AfterUpdate o AfterDelete, según el modo.

Luego para cada línea:

Se ejecutan las reglas según el árbol de evaluación
Se ejecutan las reglas que tuvieran evento de disparo BeforeValidate a nivel de las líneas.
Se realiza la validación de la línea (se la da por buena).
Se ejecutan las reglas que tengan evento de disparo AfterValidate o, dependiendo del modo, BeforeInsert, BeforeUpdate o BeforeDelete.
Se inserta/modifica/elimina el registro correspondiente a la línea en la base de datos
Se ejecutan las reglas que tengan evento de disparo AfterInsert, AfterUpdate o AfterDelete, dependiendo del modo de la línea.

Al terminar con la última línea se ejecutan las reglas que tengan como evento de disparo After Level de un atributo del segundo nivel.
Si hay otro nivel paralelo, se repite lo mismo para ese otro nivel.
Cuando se termina con el último nivel se disparan las reglas que estén condicionadas al evento BeforeComplete. Luego de esto es que GeneXus coloca el comando Commit en forma automática. Por tanto, se ejecuta el Commit. Es decir, queda commiteada la información de cabezal y líneas.
A continuación se disparan las reglas que estuvieran condicionadas al evento AfterComplete.



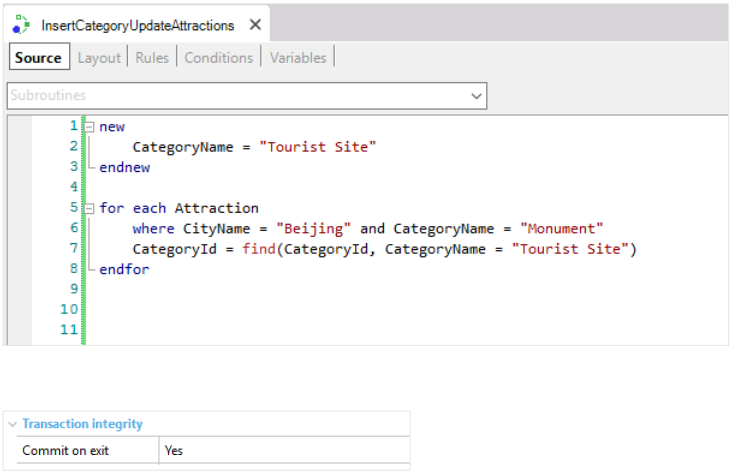
Supongamos que el usuario desea ingresar 3 facturas al sistema. Ingresa la primera, ingresa la segunda, y cuando Confirma la tercera, imaginemos que cuando el programa está procesando la tercera línea, después de haberla grabado, falla el sistema y la base de datos debe ser levantada nuevamente. ¿En qué estado quedará la base de datos?

Como el DBMS hará un rollback deshará todas las operaciones que no se hayan "commiteado". En nuestro caso, se eliminarán los registros correspondientes al cabezal y las tres líneas de la factura 3.

Obsérvese que si se hubiese deshabilitado el commit automático de la transacción Invoice (propiedad "Commit on exit" = "No") ninguno de los registros insertados (ni los de la invoice 1 ni los de la 2, y por supuesto, no los de la 3) quedará en la base de datos. En este caso todas las operaciones conformarán una UTL, mientras que si se deja el valor por defecto a la propiedad Commit on Exit, "Yes", cada cabezal con sus líneas conformará una UTL distinta.

Para transacciones, se recomienda configurar esta propiedad en Yes.

Procedure and Automatic Commit



En todo procedimiento que acceda a la base de datos GeneXus agregará automáticamente (a menos que se indique lo contrario a través de la propiedad Commit on exit) un Commit.

Como los procedimientos se usan para otras cosas aparte de actualizar la base de datos (por ejemplo, para listar información o realizar cálculos o simplemente consultar la base de datos) GeneXus insertará automáticamente el comando Commit en el programa generado si entiende que ese procedimiento intenta actualizar la base de datos. De lo contrario no lo coloca, independientemente del valor de la propiedad Commit on Exit.

En este caso en el que se están usando los comandos new para insertar una categoría y el for each para actualizar el atributo CategoryId de la tabla asociada a la transacción Attraction, dado que la propiedad Commit on exit está en Yes, GeneXus escribirá automáticamente el Commit en el programa generado, al final del código. Esto hace que si luego de haber insertado la nueva categoría en la tabla CATEGORY, y al estar modificando el tercer monumento de Beijing cambiándole la categoría por la nueva, se cae el sistema, ninguno de los cambios anteriores (ni la categoría nueva, ni los cambios en los dos monumentos anteriores) quedará en la base de datos. Todas las operaciones de este procedimiento serán parte de una

misma UTL. ¿Dónde inicia esa UTL?

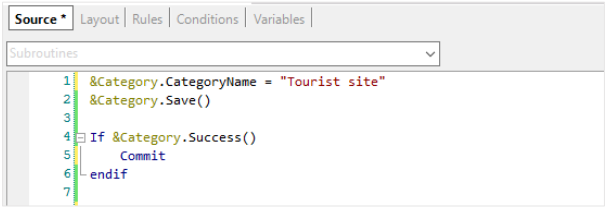
Dependerá de cuándo se realizó el último Commit. Si tras la última operación sobre la base de datos realizada antes de invocar a este procedimiento se realizó un Commit, entonces la UTL inicia con el new de este procedimiento. De lo contrario, todo este código será parte de una UTL que comenzó antes. ¿Dónde? Donde se encuentre el anterior Commit, inmediatamente después.

¿Dónde termina la UTL? Si la propiedad Commit on Exit está en "Yes", termina al final del procedimiento. Si no, termina donde se encuentre el siguiente Commit (habrá que ver en el que llamó a este procedimiento, qué es lo que sigue a su invocación).

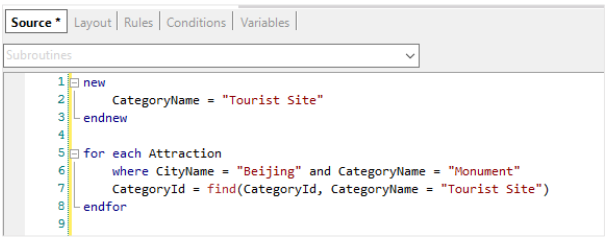
GeneXus

Procedure and Explicit Commit

Explicit Commit



Automatic Commit



New

For each that updates

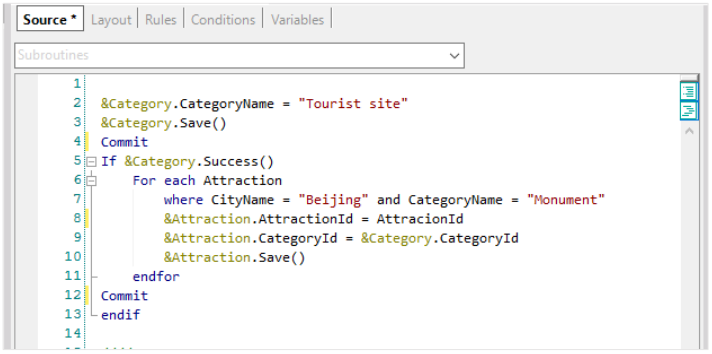
Delete

GeneXus reconoce que debe realizar un acceso a la base de datos dentro de un procedimiento cuando se utiliza new, for each para actualizar o el comando delete. En esos casos coloca el Commit implícito. En caso contrario, no entiende que hay acceso a la base de datos, y es por eso que cuando en un procedimientos se hace lo siguiente. O incluso si en lugar de .Save() hacemos .Insert(), .Update() o .InsertOrUpdate(). No agrega el Commit. GeneXus no se da cuenta de que estamos queriendo hacer un Insert en la base de datos, incluso utilizando el método insert, por lo que hay que escribir el comando Commit explícitamente.

Como acabamos de decir, si dentro del código del procedimiento hay un new, for each que actualiza o Delete, en cualquiera de esos casos no habría que escribir explícitamente el Commit. Si sólo tenemos en nuestro procedimiento este código, allí sí tendremos que agregalo en forma explícita.

GeneXus

Customizing LUWs

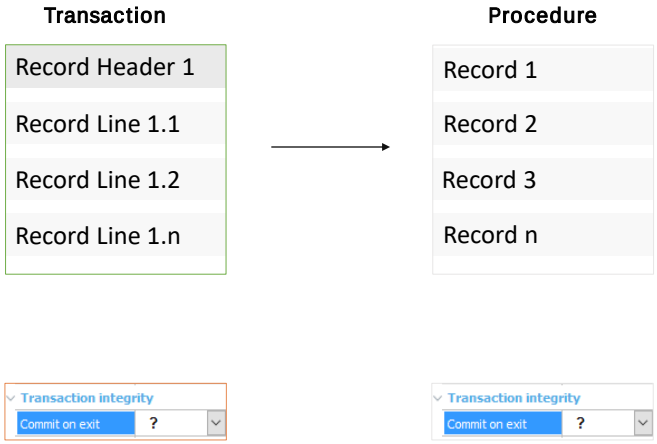


En el ejemplo que habíamos visto Genexus colocaba un commit al final automáticamente. Pero si quisiéramos que la grabación de la categoría sea parte de una UTL y las grabaciones de las atracciones sean parte de otra, de modo que si se cae el sistema antes de terminar de modificar las atracciones la categoría quede ingresada pero las atracciones no, ¿cómo hacemos?

Será a través del uso del Commit. Tendremos que escribir un Commit luego de haber insertado la categoría, y otro luego de haber insertado todas las atracciones. Este segundo podemos obviarlo si tenemos prendida la propiedad Commit on exit. De todos modos parece una buena práctica escribirlo explícitamente, por si más adelante se agregan más operaciones al Source del procedimiento, que deban quedar en otra UTL.

GeneXus

Customizing LUWs

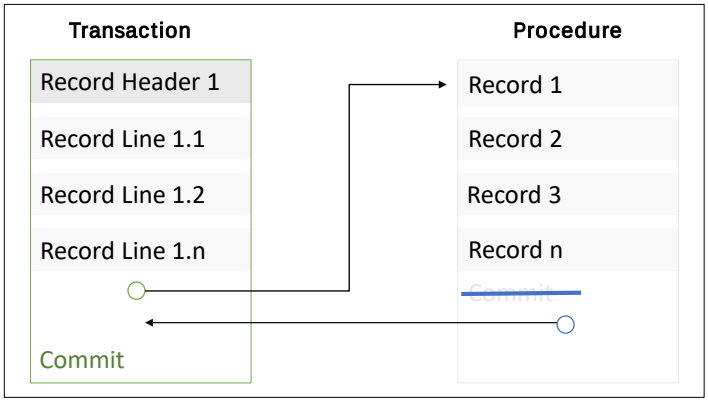


Si necesitamos invocar desde una transacción “A” a un procedimiento “B” que realiza operaciones sobre la base de datos, de modo tal que las actualizaciones del registro del cabezal de la transacción y todas las líneas, así como de todos los registros del procedimiento conformen una única UTL (y por tanto si se cae el sistema antes de completar todo se deshagan todos los cambios), ¿qué debemos programar?

Tenemos varias posibilidades para lograrlo.

GeneXus

Customizing LUWs



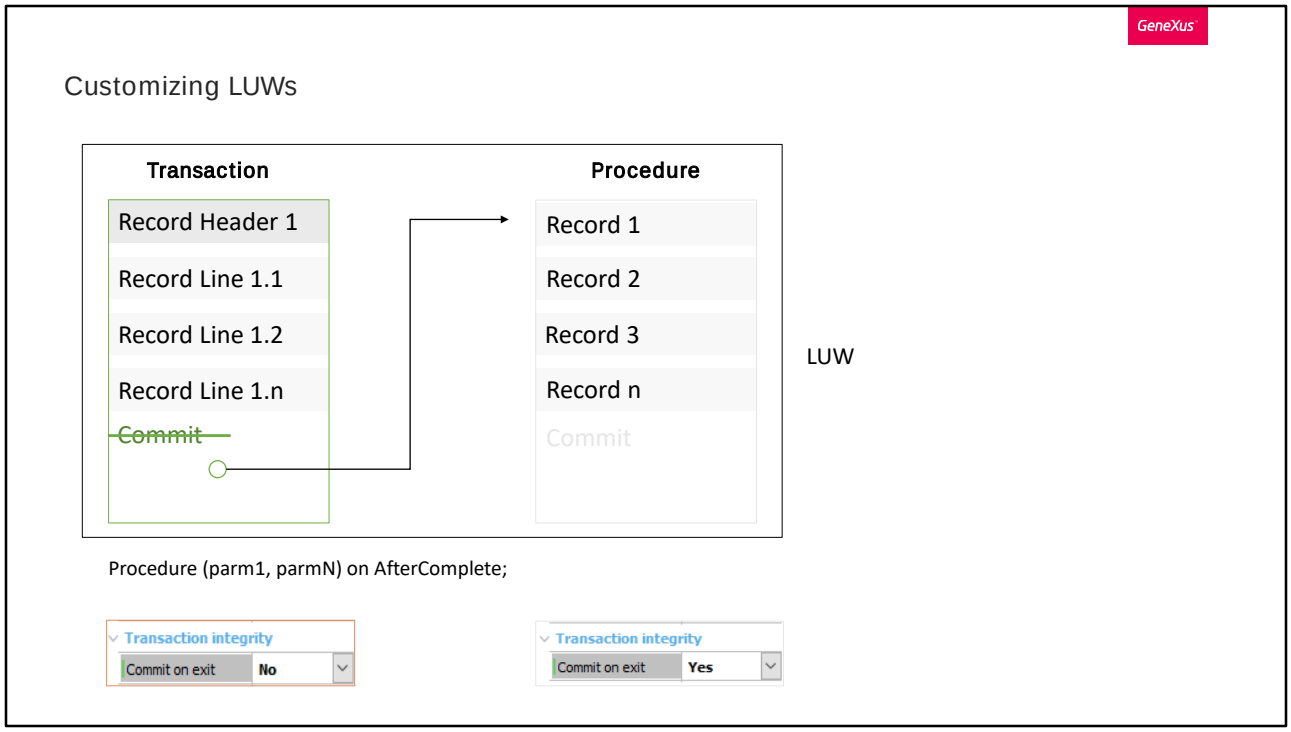
LUW

Procedure (parm1, parmN) on BeforeComplete;

Una alternativa es realizar los siguientes pasos:

1. Invocar al procedimiento en algún momento ANTERIOR al Commit automático. Por ejemplo "on BeforeComplete"
2. Deshabilitar el Commit automático del procedimiento.

De esta manera se estará invocando al procedimiento luego de haberse grabado todos los registros (el correspondiente al cabezal y los correspondientes a las líneas), habiéndose ya disparado todas las reglas condicionadas a eventos AfterLevel. Es decir, se estará llamando al procedimiento un instante antes del Commit. El procedimiento realizará todas sus actualizaciones de registros de la base de datos, y como hemos deshabilitado su Commit, si el desarrollador no incorporó este comando explícitamente dentro de su código, la UTL no se cerrará. Una vez finalizada la ejecución del código del procedimiento, se retorna al llamador, a la sentencia que siga a la invocación. Aquí es donde se encontrará el Commit.



Otra alternativa es realizar los siguientes pasos:

1. No importa cuándo se invoque al procedimiento, siempre que sea luego de haber grabado todos los registros (cabezal y líneas) de la transacción, incluso después de donde iría el Commit: Por ejemplo on AfterComplete.

2. Siempre y cuando se deshabilite ese Commit automático de la transacción y se deje el commit automático del procedimiento.

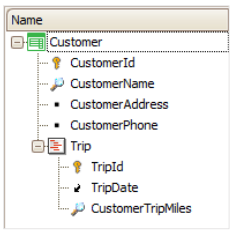
Invocando al procedimiento en este momento, estamos seguros de que todos los registros de cabezal y líneas se habrán grabado. Luego el procedimiento realizará sus actualizaciones de registros a la base de datos, y hará su commit, commiteando, por tanto, todos los registros (los suyos y los de la transacción).

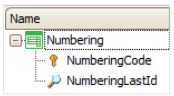
Estas son sólo dos de las múltiples alternativas. La elegida dependerá de la lógica que está queriendo implementar (normalmente no le será indiferente el momento de invocación al procedimiento).

GeneXus

Customizing LUWs

Transactions






```
CustomerId = GetNextNumber( "Customer" )
    on BeforeInsert;
```

Procedure


```
parm( in: &who, out: &nextNumber );
```

Dada la transacción Customer que registra los clientes y sus excursiones contratadas.

Supongamos que no queremos usar la estrategia de autonumeración de la base de datos, sino que queremos tener una tabla interna propia en la base de datos que registre el último número dado a cada entidad para numerar su identificador.

Para ello creamos una transacción Numbering cuyo atributo identificador NumberingCode registra el nombre de la entidad de la que se trate (por ejemplo “Customer”, “Trip”, “Invoice”, “Category”, “Attraction”, etc.) y su atributo NumberingLastId, el último número dado para esa entidad.

Luego, programaremos un procedimiento, GetNextNumber, que será el responsable de obtener el siguiente número a ser asignado al identificador de la entidad que lo está llamando.

Analizaremos el código en un momento.

Por ejemplo, desde la transacción Customer, cuando el usuario quiera ingresar un cliente nuevo, dejará vacío el campo CustomerId en la pantalla y cuando abandone el campo, pasando al siguiente campo,

CustomerName, la transacción sabrá que se está queriendo ingresar un registro nuevo (quedará en modo "INS", insert).
Como no tenemos el atributo CustomerId autonumerado, tendremos que invocar al procedimiento GetNextNumber, para obtener el número a asignarle a CustomerId.
Debemos pasarle por parámetro al procedimiento quién somos, es decir, en este caso, "Customer" para que el procedimiento vaya a buscar a la tabla Numbering el último número que se le había dado a un "customer".

Veamos en detalle lo que hace el procedimiento creado.
Primero que nada, en las reglas declaramos dos parámetros, uno de entrada y otro de salida. El parámetro de entrada será el que utilizaremos para saber qué queremos enumerar, un cliente, un viaje, una factura, etc. Esta variable será del tipo character.
Y de salida declaramos la variable NextNumber, del tipo numérico, la cual devolverá el valor que nos interesa.

Volviendo al source, se recorrerá la tabla de la transacción Numbering. Con el where indicamos que queremos recuperar el registro en el que NumberingCode tenga igual valor al de la variable que pasamos por parámetro, en este caso Customers.
Si lo encuentra, al atributo NumberingLastId se le sumará uno, y ese valor se le asignará a la variable nextNumber. Luego, al atributo NumberingLastId le asignamos el valor de nextNumber.

En caso que no se encuentre un registro en el que NumberingCode tenga el valor de la variable que le pasamos, crearemos un registro en la base de datos mediante el comando new. En el que NumberingCode tendrá el valor de la variable pasada por parámetro, por ejemplo Invoice. Luego le asignamos a la variable nextNumber el valor uno, Y al atributo NumberingLastId le asignamos el valor de nextNumber, ya que al ser un nuevo registro en esta tabla iniciaremos con este valor.

Si invocáramos a este procedimiento mediante la regla de asignación sin evento de disparo:
CustomerId = GetNextNumber("Customer") if Insert; el procedimiento se disparará:

1. Una vez ni bien el usuario en la pantalla deje vacío el campo CustomerId y abandone el campo.
2. Una segunda vez cuando el usuario confirme, y las reglas vuelvan a dispararse en orden, en el servidor.

Imaginemos que el último id de cliente es el 5.

El procedimiento se ejecutará actualizando a 6 el registro de la tabla Numbering correspondiente y mostrándole al usuario inmediatamente en la pantalla el número 6. Pero ¿qué pasa si el usuario se arrepiente y nunca presiona Confirm, sino que cancela? Se habrá perdido el número 6.

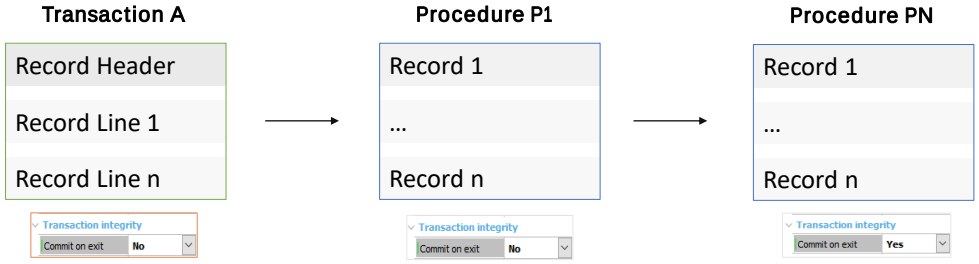
Incluso si el usuario sí confirma, se volverá a ejecutar el procedimiento, y el número que se le asignará al cliente será el 7, por lo que el número 6 también se perderá.

¿Cómo resolvemos esta situación? Condicionando la invocación al procedimiento a un evento de disparo (de esta manera la regla de asignación no se disparará en el cliente, mientras el usuario trabaja en la pantalla). ¿Cuál es el momento apropiado? El último momento posible es BeforeInsert del cabezal. ¿Por qué? Porque después de ese momento, el valor que asignemos a cualquiera de sus atributos no tendrá efecto, dado que el registro ya se habrá grabado.

Ahora bien, el procedimiento GetNextNumber modifica un registro de la tabla Numbering, o inserta uno si éste no existía. Entonces, por defecto, colocará un Commit automático al final. De este modo, si enseguida de retornar a la transacción supongamos que sí se inserta el cliente en su tabla, y cuando se está insertando la tercera excursión se cae el sistema, al reestablecerse no quedarán registrados los registros asociados al cliente, pero sí quedará perdido ese número, puesto que ya se había commiteado. Para que todas las operaciones realizadas por transacción y procedimiento queden dentro de la misma UTL, alanzará en este caso con no hacer Commit on Exit en el procedimiento, y que el Commit de todo lo haga la transacción.

GeneXus

Customizing LUWs

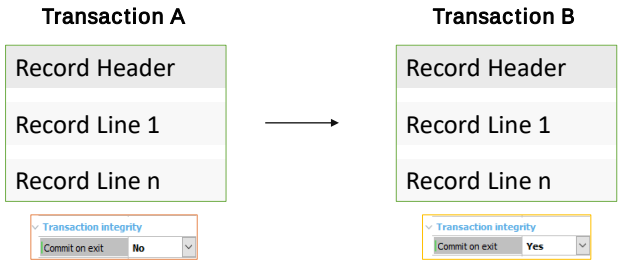


No puede conformarse una única UTL entre transacciones.

Si desde una transacción se invoca a un procedimiento que invoca a otro procedimiento, todos con el Commit deshabilitado salvo el último (o también puede tener esté su commit deshabilitado y quien hace Commit es la transacción cuando se le retorna el control) su commit commiteará los registros de la transacción y de todos los procs.

GeneXus

Customizing LUWs

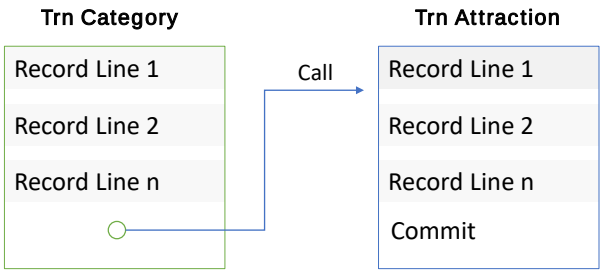


Sin embargo, si la transacción (“A”) llama a otra transacción (“B”), el commit de la segunda (“B”) NO commiteará los registros manipulados por la primera (“A”). Son Commits independientes. Por lo que si deshabilitamos el commit de “A” y llamamos a “B”, que hace Commit, los registros de “A” ¡no serán commiteados por nadie!

¿Cómo hacemos, entonces, para lograr que se ingresen el cabezal y líneas de una transacción “A”, se llame a una “B” para ingresar información en cierta forma relacionada, y que todas estas operaciones conformen una única UTL?

GeneXus

Customizing LUWs



Desde el trabajar con Categories queremos que al presionar Insert se le ofrezca al usuario la posibilidad de ingresar una nueva categoría e inmediatamente una atracción de esa categoría. Porque no queremos dejar insertadas categorías sin atracciones relacionadas.

Para ello desde la transacción Category alcanzará con invocar a la transacción Attraction on AfterInsert (para que el CategoryId ya tenga el valor autonumerado correcto, dado por la base de datos), y con declarar en Attraction que recibirá un parámetro.

De esta manera, cuando ingresemos una Categoría, nos llevará a la transacción Attraction para ingresar una atracción. Ya quedando seleccionada la categoría que acabamos de pasar por parámetro.

Pero...¿qué pasa si se cae el sistema inmediatamente después de que Attraction haya hecho su commit? Ese Commit, ¿habrá commiteado también el registro de Category que ya se había insertado? La respuesta es No, por lo que veíamos antes.

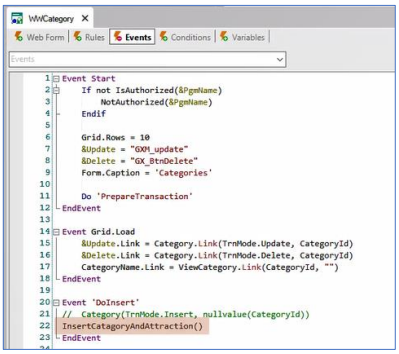
Necesitamos que la inserción de una categoría y acto seguido, de una atracción, conformen una misma UTL y el Commit final commitee ambos registros. ¿Cómo lo conseguimos?

Veremos dos opciones, de las múltiples que hay.

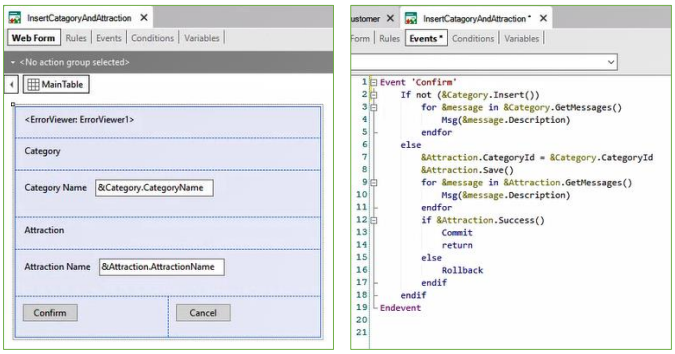
GeneXus

Customizing LUWs

WorkWithCategoy



InsertCategoryAndAttraction



Una solución será que el botón Insert del "trabajar con" invoque a un web panel, que es un objeto con interfaz donde los desarrolladores programamos libremente el comportamiento.

En él insertamos dos variables &category y &attraction en su form. Serán Buisiness Components de Category y Attraction respectivamente.
Sólo nos interesa que el usuario inserte el nombre de la categoría que está queriendo crear, y el nombre de la atracción de esa categoría. Al hacerlo y presionar Confirm, se disparará el evento asociado, que hemos programado como se ve en la pantalla de eventos.

Tengamos en cuenta para este ejemplo, que los atributos CountryId y AttractionId, de las transacciones Country y Attraction respectivamente, tienen la propiedad autonumber en true.

Primero intentamos hacer el Insert del BC de Category. Si falla (por ejemplo si el usuario dejó vacío el nombre de la categoría en la pantalla y la transacción tiene una regla error para prevenir ese caso) mostramos en el control ErrorViewer los mensajes producidos por el BC.
En ese ejemplo el mensaje de error se dispara dos veces, una del lado del cliente cuando dejamos el campo vacío. Y otra cuando confirmamos, del lado del servidor. Ambos casos arrojan el mismo resultado. A este tipo de

reglas las llamamos idempotentes.

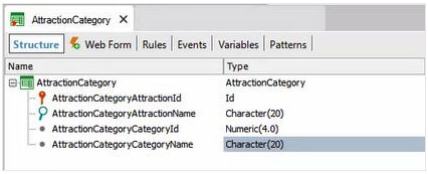
Si ingresamos todos los datos correctamente, entonces a la variable BC de Attraction le asignamos como categoría la recién insertada e intentamos hacer el Save. Mostramos los mensajes producidos y luego si la operación fue exitosa, hacemos Commit, tras lo que, ahora sí, quedarán commiteados ambos registros.

Si la operación falló, entonces observemos que contamos con el comando Rollback, para deshacer la inserción del registro de Category que había sido insertado exitosamente.

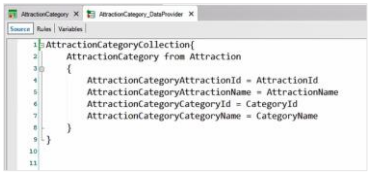
GeneXus

Customizing LUWs

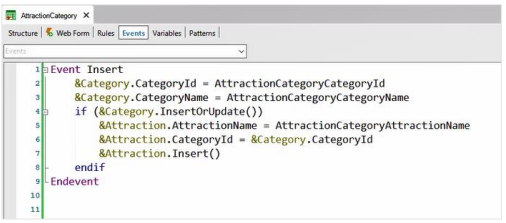
Trn AttractionCategory



Data Provider



Insert Event



Una segunda alternativa para resolver el mismo requerimiento es utilizar una transacción dinámica en vez del web panel.

Recordemos que las transacciones dinámicas son aquellas que no generan tabla física, se obtienen los datos consultados en tiempo de ejecución.
Son definidas configurando las siguientes propiedades de la transacción. Data Provider en true, y Used To en Retrieve Data.
Al poner el Data Provider en true, se creará automáticamente un objeto Data Provider. Y al configurar Used To en Retrieve Data, GeneXus entenderá que no deberá crear la tabla asociada a la transacción, ya que en ese Data Provider se declarará de dónde obtener los datos.

En el ejemplo creamos la transacción dinámica AttractionCategory, cuya información se tomará de la tabla de atracciones, sabiendo que cada atracción tiene una categoría. Será como una vista de atracciones.

Cuando el usuario quiera insertar en esta transacción una atracción, se le permitirá ingresar los datos de la atracción así como los de la categoría.
En el evento Insert utilizamos &category business component de Category, y &attraction business component de Attraction y copiamos a sus miembros los valores que el usuario especificó en los atributos de la

transacción dinámica, a través de su form.
Si la categoría no existe, se crea antes de insertar la atracción. Si existe, se le permite actualizar su nombre. Luego se inserta la atracción con la categoría.
Si dejamos la propiedad Commit on Exit de la transacción con su valor por defecto, Yes, luego de la ejecución del evento Insert, por tratarse de una transacción de un solo nivel, se realizará el Commit, que tendrá efecto sobre los dos registros insertados a través de los business components.

GeneXus™

training.genexus.com
wiki.genexus.com