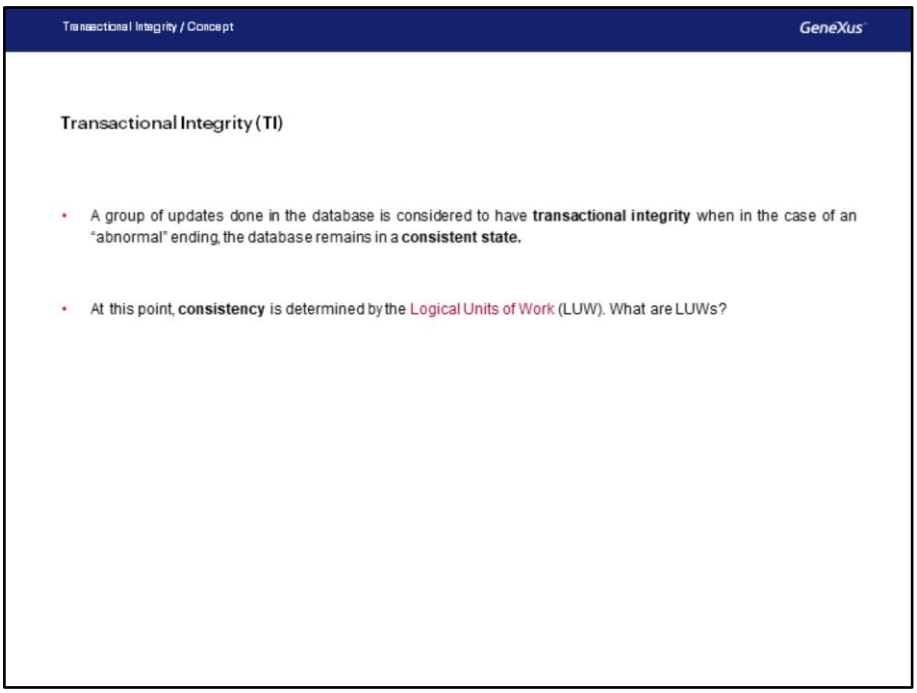
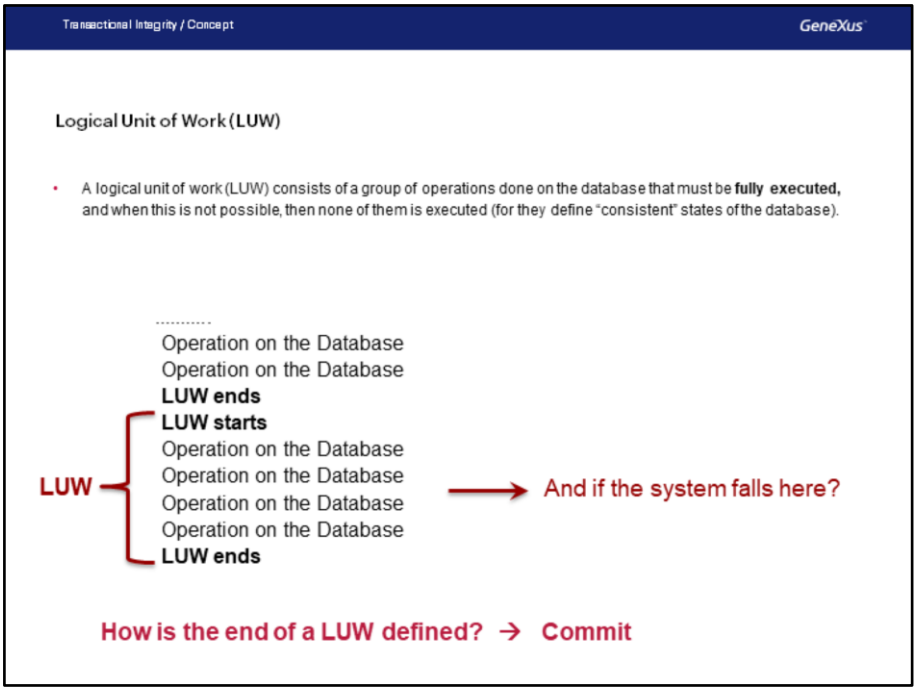
Transactional Integrity
Consistent database for failures

GeneXus 16

Concept



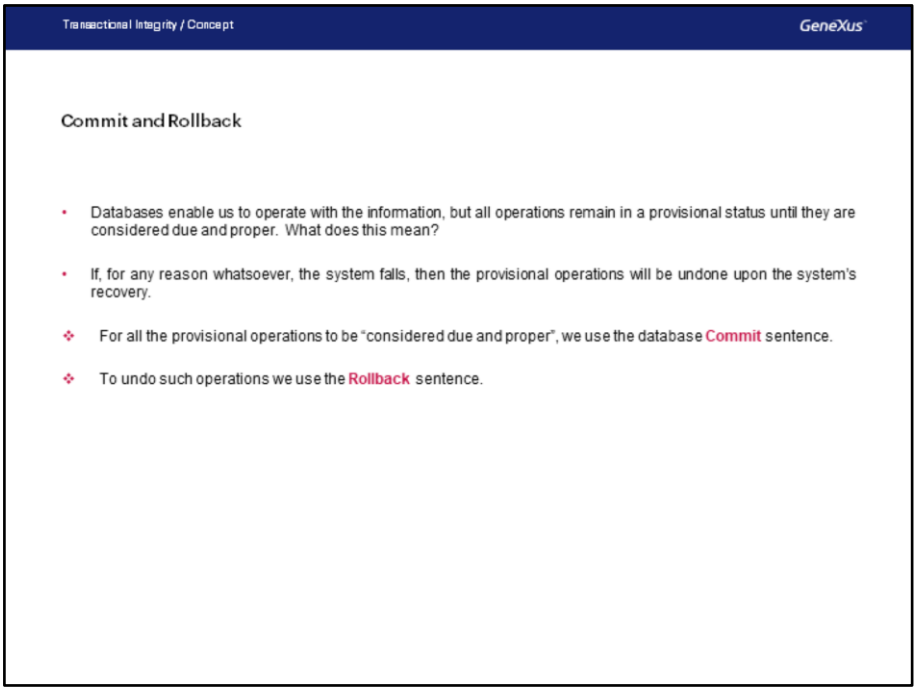
Many database managers (DBMSs) include recovery systems for the event of failures. These systems enable the database to remain in a consistent state for the case of contingencies such as power failures or falls of the system.



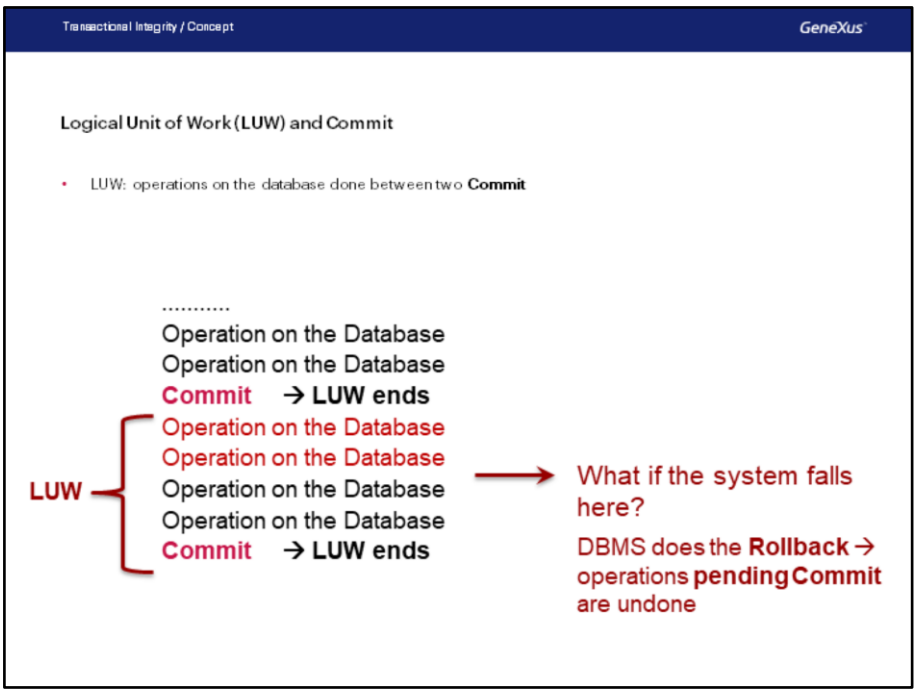
Database managers (DBMSs) that provide transactional integrity enable us to establish logical units of work (LUW), which correspond, no less and no more, to the concept of database "transactions".

Our example shows a LUW consisting of four operations on the database. If we suppose that the first two were accomplished successfully and the systems falls before the third operation is executed, since the LUW was not completed, then the two operations completed will be undone. If this were not so, and because the LUWs are the ones defining the consistent states of the database at the logical level, then the database would become inconsistent.

And how is a LUW defined?



Following recovery of the system after it has fallen, the DBMS will do a Rollback for recovering, while maintaining the last consistent status of the database.



The Commit command is the one that determines the end of a LUW. Therefore, a LUW is defined by the operations performed between two Commits.

When the system fall where shown, then the two operations done following the last Commit (which are the operations pending Commit) will be undone with the automatic Rollback performed by the DBMS upon recovery from the failure.

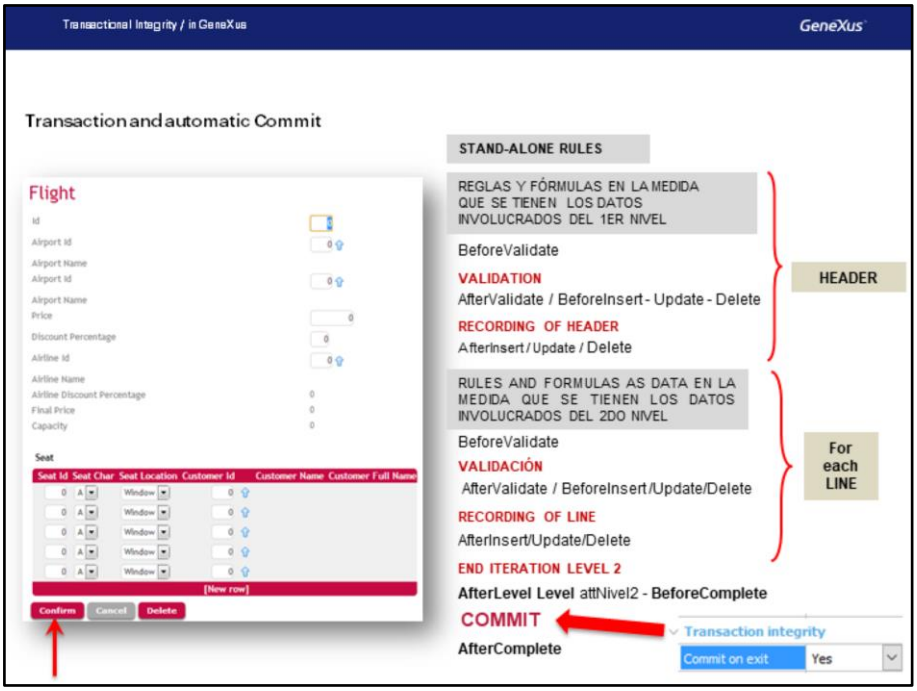
Transactional Integrity in GeneXus

Transactions and procedures are the GeneXus objects that are **created to update** the database information. This is why GeneXus writes the Commit command when it generates the programs in the language defined.

Where?

- In the Transaction object: at the end of each instance, immediately prior to the rules with the AfterComplete trigger event (that is: after handling the header and lines).
- In the Procedure object: at the end of the Source.

The Business Components created from transactions do not include Commit because they may be used for any object, and the developer will be the one deciding where to "Commit".
We will see this further ahead.

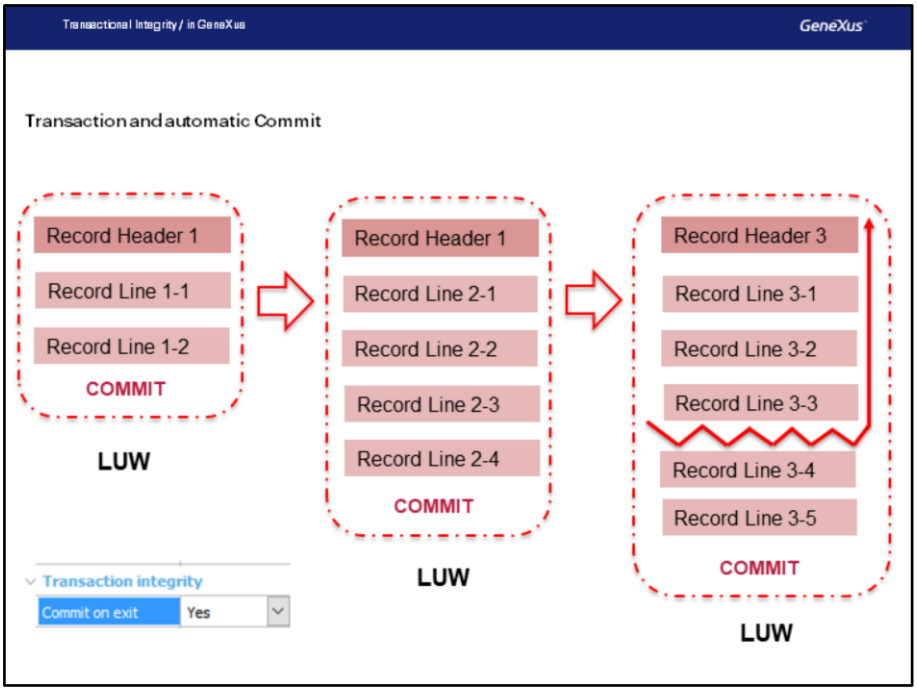


The user handles the header and lines and presses “Confirm”. Rules and formulas are executed on the server according to the assessment tree for the first level and then the rules conditioned to the BeforeValidate event are triggered. After the header information has been deemed valid, the rules conditioned to the AfterValidate events are triggered, and, depending on the mode, those conditioned to BeforeInsert, BeforeUpdate or BeforeDelete, as well. Then the header is recorded and the rules conditioned to AfterInsert, AfterUpdate or AfterDelete –depending on the mode- will be triggered.

Then for each line:

- The rules are executed according to the assessment tree.
- The rules with BeforeValidate triggering event at the line level are executed.
- The line is validated (it is deemed valid).
- The rules with AfterValidate -or, depending on the mode, BeforeInsert, BeforeUpdate or BeforeDelete- triggering event, are executed.
- The record corresponding to the line in the database is inserted/modified/deleted.
- The rules with AfterInsert, AfterUpdate or AfterDelete triggering event –depending on the line mode- are executed.

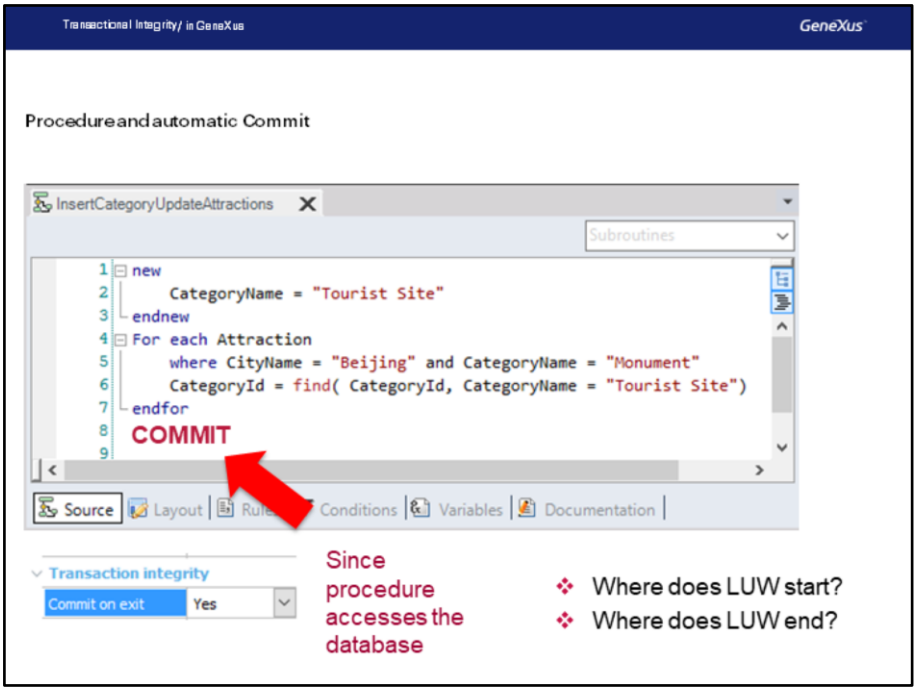
After the last line has been concluded, the rules with triggering event After Level of an attribute of the second level are executed.
If there is another parallel level, the same takes place for that other level.
After the last level has been concluded, the rules conditioned to the BeforeComplete event are triggered. It is after this that GeneXus inserts the Commit command automatically. Therefore, the Commit is executed, meaning that **the information of the header and lines is committed.**
After that, the rules any rules conditioned to the AfterComplete event will be triggered.



Suppose that the customer is willing to enter three invoices in the system. The first invoice is entered, then the second one, and when the third invoice is Confirmed, imagine that -during the processing of the third line by the program, after it is recorded- the system fails and the database needs to be picked up again. What would be the state of the database then?

Due to the rollback that the DBMS will carry out, all operations that have not been “committed” will be then undone. In our case, the records corresponding to the header and the three lines of Invoice 3 will be deleted.

Note that, had the automatic Commit of the Invoice Transaction (“Commit on exit” property = “No”) been disabled, then none of the records entered (neither those of Invoices 1 and 2, nor –obviously– those in Invoice 3) would remain in the database. In such case all these operations would comprise a LUW, whereas by leaving the default value for the Commit on Exit property (“Yes”), then each header with its lines would comprise a different LUW.



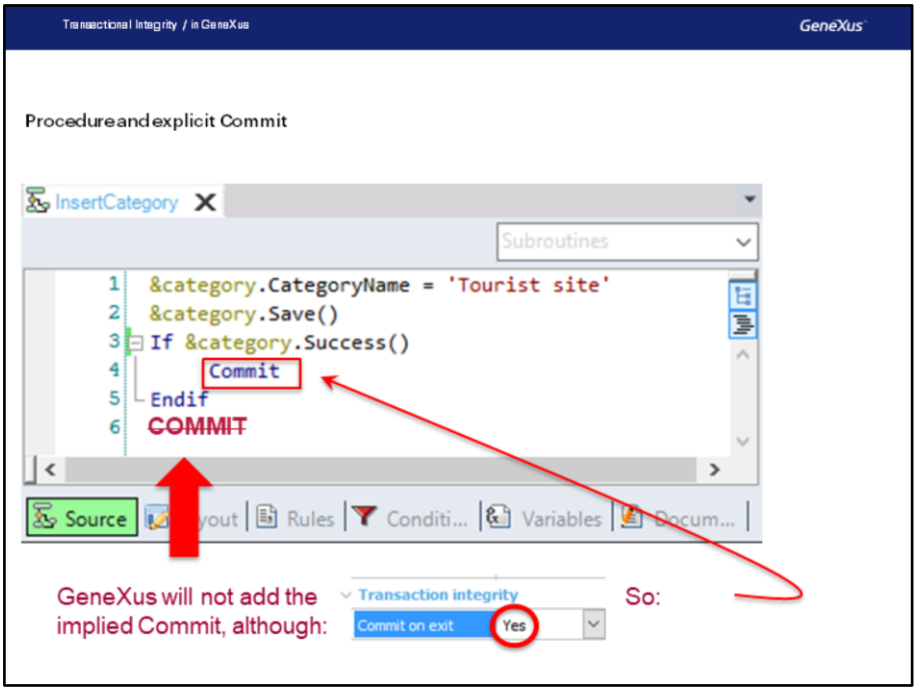
For every procedure accessing the database, GeneXus will automatically add a Commit (unless the Commit on exit property indicates otherwise).

Since procedures are used for other things besides updating the database (such as listing information, making calculations, or just to query the database), GeneXus will automatically insert the Commit command in the program generated when it understands that the procedure is trying to update the database. Otherwise, it will not insert it, regardless of the value of the Commit on Exit property.

In this case, where the new commands are being used to insert a category and the For each to update the CategoryId attribute of the table associated with the Attraction Transaction, since the Commit on exit property is set to Yes, then GeneXus will automatically write the Commit in the program generated, at the end of the code. This means that, upon modifying the third monument in Beijing –after inserting the new category in the CATEGORY table– to change its category for the new one, if the system happens to fall, then none of the previous changes (nor the new category or the changes to the two previous monuments) will remain in the database. All the operations in this procedure will be part of the same LUW. And where does this LUW start?

That will depend on when the last Commit was done. If following the last operation on the database that was done prior to invoking this procedure a Commit was performed, then the LUW will start with the new of this procedure. Otherwise, this whole code will be part of an LUW that started previously. Where? At the point immediately following the previous Commit.

And where does the LUW end? If the Commit on Exit property is set to "Yes" it will end at the end of the procedure. If not, it will end at the point of the following Commit (we will have to look into the one that called this procedure, to see what follows the invocation).

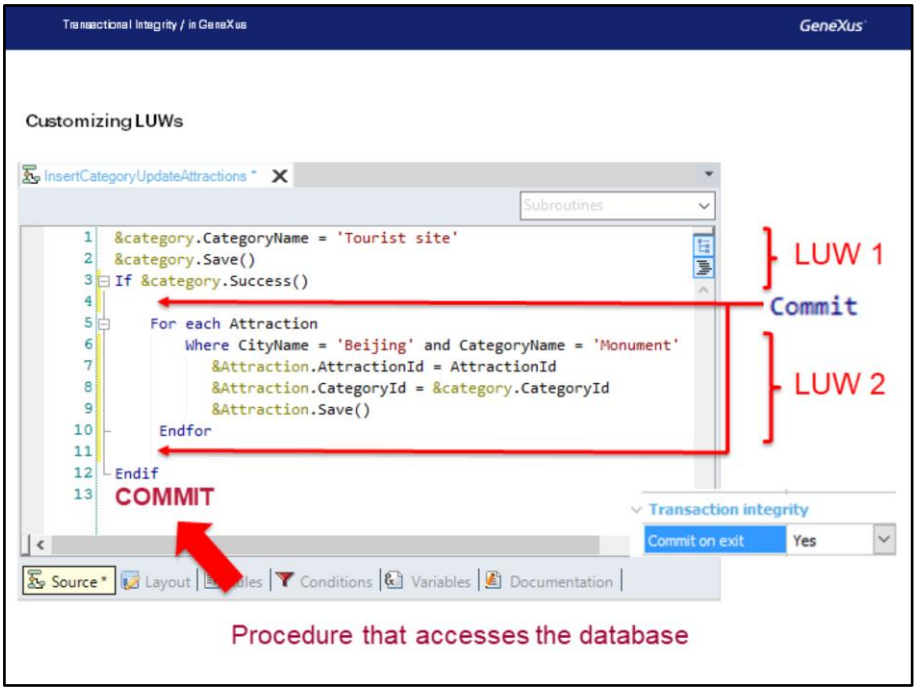


GeneXus acknowledges that it must access the database inside a procedure for cases where new, for each to update, delete, the Load() of BC method, or Delete() are used. It will then include the implied Commit. Otherwise, it will not understand that the database is accessed, so in procedures where only:

&BC.element1 = …
&BC.element2 = …
&BC.Save()

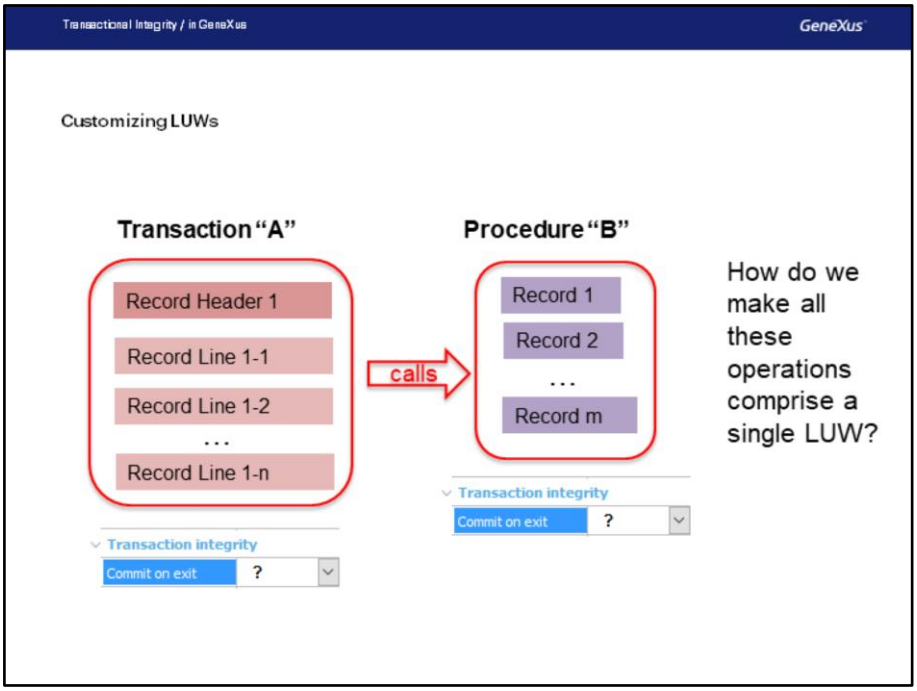
are done, it will not add the Commit. That would be the case of our example, had we not programmed the Load of the attraction seeking only to insert the category, as shown in the Source above. GeneXus does not realize that what we want to do is an Insert (it considers the BC as if it were any SDT), so we have to explicitly write the Commit command.

If inside the code of the procedure we also have a new, an updating For each, or even a Load() (as in the case of the previous page) or a Delete, for any of such cases we will not have to explicitly write the Commit. If our procedure only includes the above code, then we will have to add it explicitly.



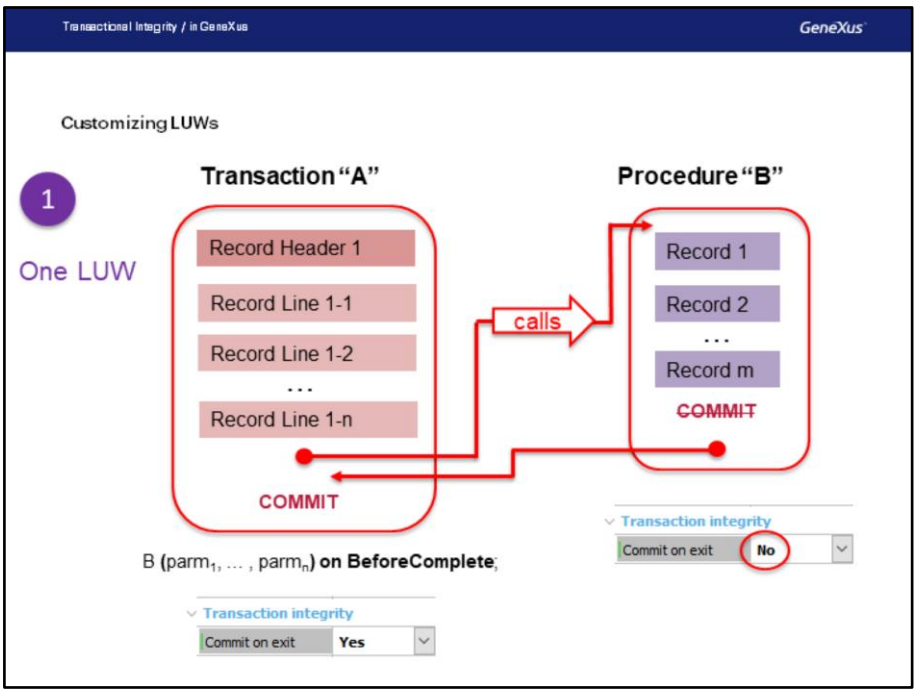
In the example that we saw, GeneXus automatically added a Commit at the end (the Load causes it to open connection to the database and add Commit at the end).
But if we want the recording of the category to be part of an LUW and the recordings of attractions to be part of another LUW, so that upon a fall of the system prior to finishing the modifications of attractions the category remains entered while the attractions don't, then what do we do?

We will have to use the Commit. We write one Commit after inserting the category and another one after inserting all the attractions. We can avoid the second one when we have the Commit on exit property set up. However, it is always good practice to write it explicitly, just in case more operations are added later to the Source of the procedure, which must remain in the other LUW.



When we need to invoke, from a transaction ("A"), a procedure ("B") that performs operations on the database, so that the record updates of the record of the transaction header and all lines, as well as all the records of the procedure, comprise a single LUW (so, if the system falls prior to completing all this then all changes will become undone), what should we program?

There are different ways of achieving this.

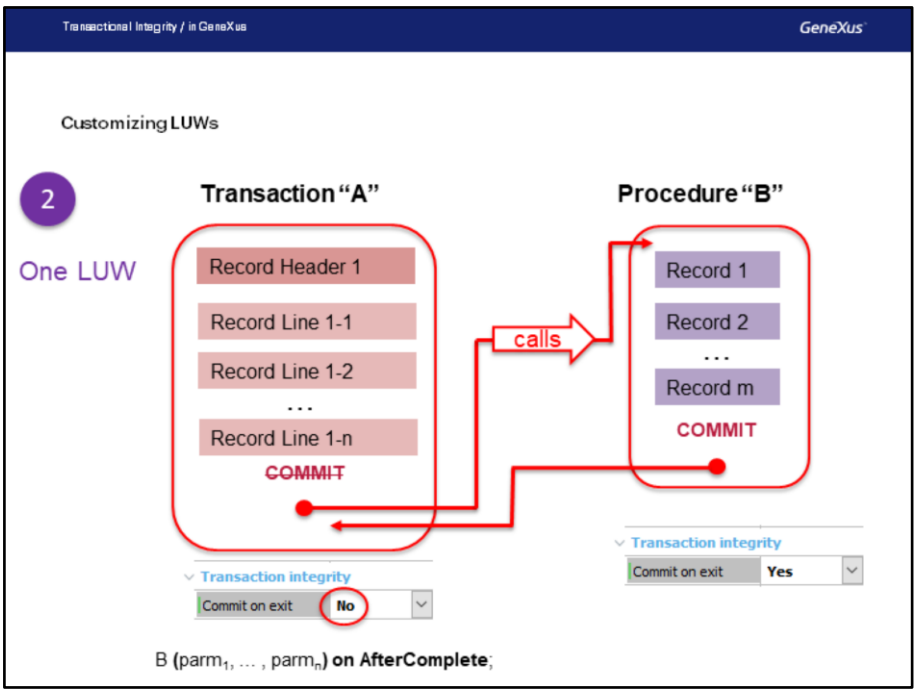


One alternative is to follow these steps:

1. Invoke the procedure at some point PRIOR to the automatic Commit. For example:
   B($parm_1$, … , $parm_n$**) on BeforeComplete**;

2. Disable the procedure's automatic Commit.

By doing this, we will be invoking the procedure after all the records (the one corresponding to the header and those corresponding to the lines) have been saved, and after all the rules conditioned to AfterLevel events have been triggered. This means that we will be calling the procedure an instant before the Commit. The procedure will do all its updates or records in the database, and since we disabled its Commit, if the developer did not explicitly include this command in its code, then the LUW will not be closed. Once the execution of the procedure code is finalized, then it will return to the caller, and to the sentence that follows the invocation. Here is where the Commit will be located.



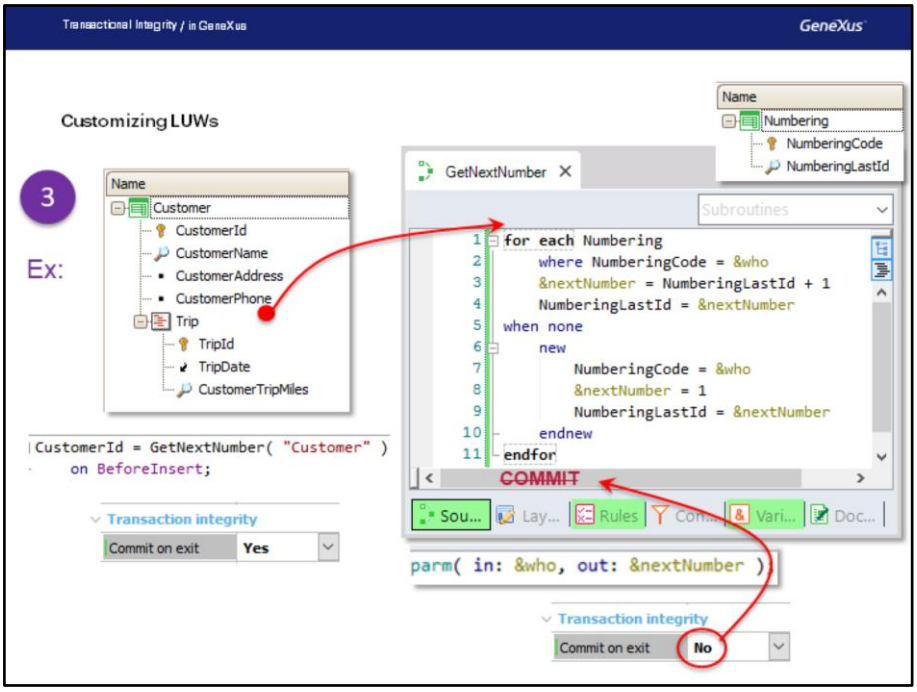
Another option is to follow these steps:

1. Regardless of the time when the procedure is invoked, to the extent that it is invoked after all the transaction's records have been saved (header and lines), and even after the location where the Commit would be:
   B($parm_1$, … , $parm_n$**) on AfterComplete**;

2. Provided that the automatic Commit of the transaction is disabled, leaving the procedure's automatic Commit.

By invoking the procedure between the AfterLevel Level 2nd-level-attibute/Before Complete event and the AfterComplete, we will be sure that all header and lines records will be saved. Then the procedure will do its updates of records to the database, and it will also do its Commit, thus committing all the records (its own and those of the transaction).

These are just two of the multiple alternatives possible. The one we choose will depend on the logic that we want to implement (usually, the time at which the procedure is invoked will not be indifferent).



Upon the Customer transaction that records customers and their hired trips, suppose that we do not want to use the autonumbering strategy of the database, but rather have an internal table of our own in the database to record the last number given to each entity for numbering its identifier.

To do this we create a Numbering transaction whose identifier attribute NumberingCode records the name of the entity involved (for example, “Customer”, “Trip”, “Invoice”, “Category”, “Attraction”, etc.) and its NumberingLastId attribute –the last number given for that entity.

Then we program a procedure, GetNextNumber, that will be responsible for obtaining the following number that is to be assigned to the identifier of the entity calling it.

For example, from the Customer transaction, when the user wishes to enter a new customer, the CustomerId field will be left blank on screen, and upon exiting the field to go to the next field that is CustomerName, the transaction will know that the idea is to enter a new record (it will remain in insert mode -“INS”).
Since we don’t have the CustomerId attribute autonumbered, we will have to invoke the GetNextNumber procedure to obtain the number that is to be assigned to CustomerId.
We must pass who we are, by parameter to the procedure, in this case: “Customer”, so that the procedure fetches from the Numbering table the last number used for a “customer”, then add one, and update that value in the Numbering table to return the new number.
If we invoked this procedure through the assignment rule with no triggering event:

CustomerId = GetNextNumber(“Customer”) if Insert;

then the procedure will be triggered:

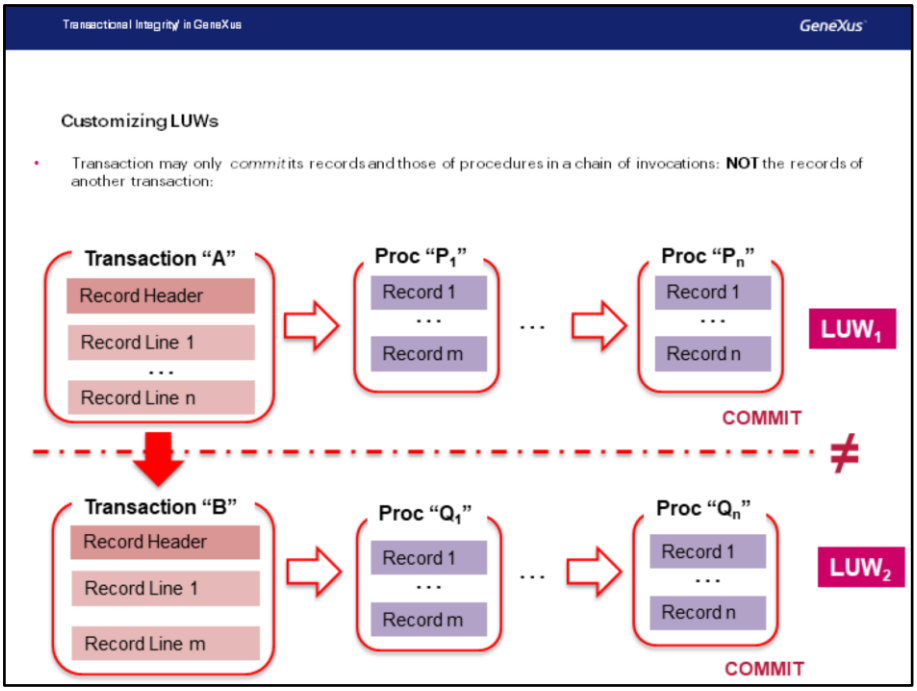
1. Once immediately after the user on screen leaves the CustomerId field blank and abandons the field.
2. For the second time when the user confirms and the rules are

triggered again, in order, on the server.

Imagine that the last Customer ID is 5. Considering 1., the procedure se will be executed updating to 6 the record of the corresponding Numbering table, and immediately showing the user the number 6 on screen. But, what if the user decides to not press Confirm and to cancel instead? Then the number 6 will be lost.
Moreover, due to 2., if the user does Confirm, then the procedure will be executed again, and that will be assigned to the customer will be number 7, so number 6 will be lost either way.

How can we solve this situation? By conditioning the invocation to the procedure with a triggering event (this will prevent the assignment rule from being triggered in the customer while the user works on screen). And when is the exact time for this? The last moment possible is the header's BeforeInsert. Why? Because from that point on, the value that we assign to any of the attributes will have no effects because the record will be already saved.

Thus, the GetNextNumber procedure modifies a record from the Numbering table or inserts one when it did not exist. Therefore, it will add an automatic Commit at the end, by default. So if immediately after returning to the Transaction, suppose that the customer is indeed inserted in its table and the system falls while the third trip is being inserted, upon recovery the records associated with the customer will not be recorded, though that number will be actually lost. This is because it had been committed already. For the all operations made by transaction and procedure to be inside the same LUW in this case we would have to avoid doing the Commit on Exit in the procedure, and the Commit of everything should be done by the transaction.

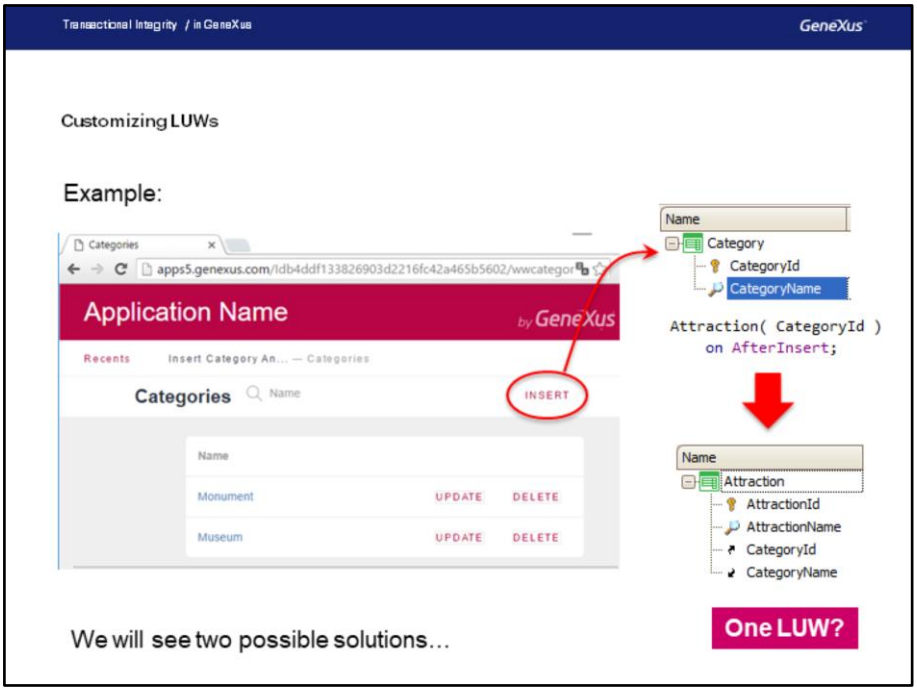


It is not possible to comprise a single LUW between transactions.

When from a transaction we invoke a procedure that invokes another procedure, which in turn invokes another procedure, all of them with the Commit disabled except for the last one (or when the last one also has its Commit disabled and the one doing the Commit is the transaction upon the return to it of the control) then its Commit will commit the records of the transaction and of all the procedures.

However, if Transaction ("A") calls another Transaction ("B"), then the Commit of the second transaction ("B") **WILL NOT commit** the records handled by the first transaction ("A"), because they are independent Commits. So, if we disable the commit of "A" and call "B", which does commit, then the records of "A" will not be committed at all!

What do we do then for the header and lines of Transaction "A" to be recorded, the transaction "B" be called to enter information that is in some way related, and to make all such operations to comprise a single LUW?



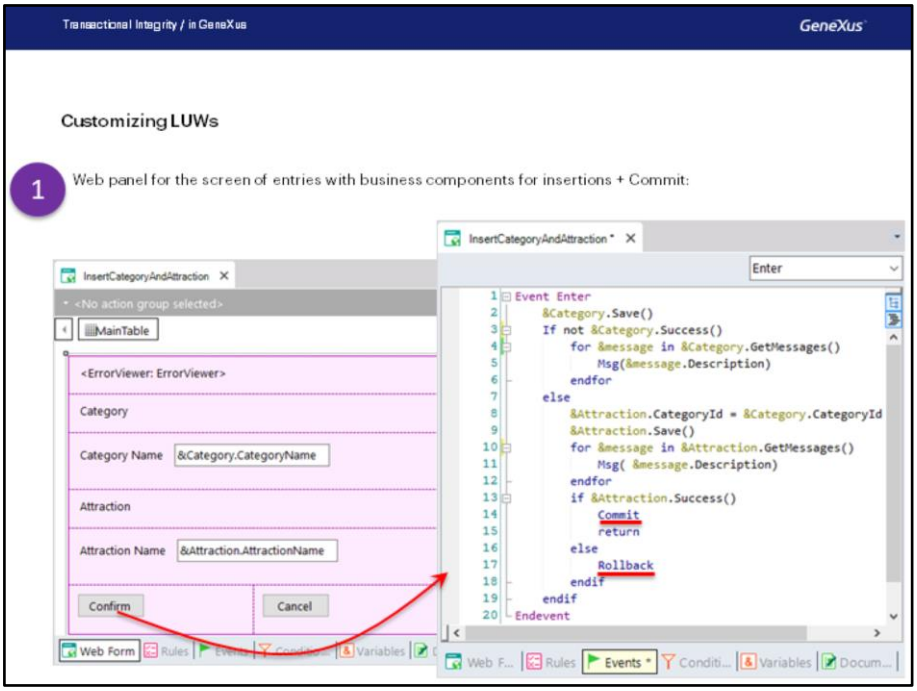
From the work with categories we want that, upon pressing Insert, the user be offered the possibility of entering a new category, and an attraction from that category immediately following. This is because we do not want to have categories inserted that do not have any related attractions.

To do this, from the Category transaction we will invoke the transaction Attraction on AfterInsert (so that the CategoryId already has the correct autonumbered value given by the database), and declare in Attraction that it will receive a parameter.

But…what happens if the system falls after Attraction has done its commit? Will that commit have committed also the record of Category that was already inserted? The answer is NO, for the reasons we just explained before.

We need that the insertion of a category and the act immediately following it of insertion of an attraction comprise a single LUW and for the final Commit to in fact commit both records. And how do we do this?

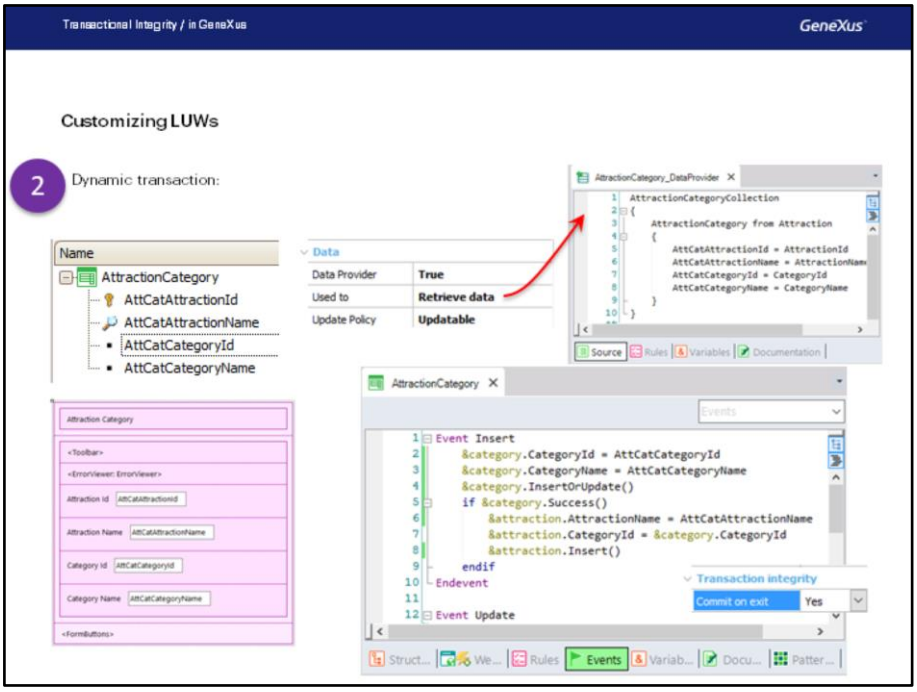
From the many options possible we will consider two alternatives.



One solution is for the Insert button of the “work with” to invoke a web panel, which is an object with interface where developers freely program behaviors.
There we insert two variables –namely: &category and &attraction- in its form. They will be Business Components of respectively Category and Attraction.
We just want the user to insert the name of the category that is to be created, and the name of the attraction in that category. When we do this and press Confirm, the associated event –which we programmed, as we can see in the events screen- will be triggered.
First we try to do the Save of the BC of Category. If it fails (for example, if the user left the name of the category empty on the screen and the transaction has an error rule to prevent such cases) then we show in the ErrorViewer control the messages produced by the BC.
If it does not fail, then we assign as category to the BC variable of Attraction the one just inserted, and we try to do the Save. We show the messages produced (whether error messages or success messages such as “Data was successfully added”) and then, if the operation was successful, we do **Commit**, following this this time both records will be indeed committed.
If the operation fails, then we should note that we have the **Rollback command** to undo the insertion of the record of Category that had been successfully inserted.

Customizing LUWs

- GeneXus provides the **Commit** and **Rollback** commands.
- They enable the customization of LUWs, together with the **automatic Commit** which we can turn on or off with the **Commit on Exit** property in transactions and procedures.
- They may be written in Procedures and Web Panels, in combination with Business Components, but **NOT** in Transactions.



Another alternative to solve the same requirement is to use a Dynamic Transaction instead of the web panel.

In the example we created the dynamic transaction AttractionCategory, whose information is taken from the attractions table in the understanding that each attraction has a category. It is like a view of attractions.

To insert an attraction in this transaction, the user will be allowed to enter the data of the attraction as well as the data of the category. In the Insert event we use &category business component of Category, and &attraction business component of Attraction and we copy the values of its members that the user specified in the attributes of the dynamic transaction, through its form.
If the category is inexistent, we create it prior to inserting the attraction. If it does exist, it is possible to update its name. Then we insert the attraction with the category.
If we leave the Commit on Exit property of the transaction with its default value (Yes) after executing the Insert event, since it is a single level transaction, the Commit will be performed, with effects on the two records inserted through the business components.

GeneXus™

The power of doing.

Videos	training.genexus.com
Documentation	wiki.genexus.com
Certifications	training.genexus.com/certifications