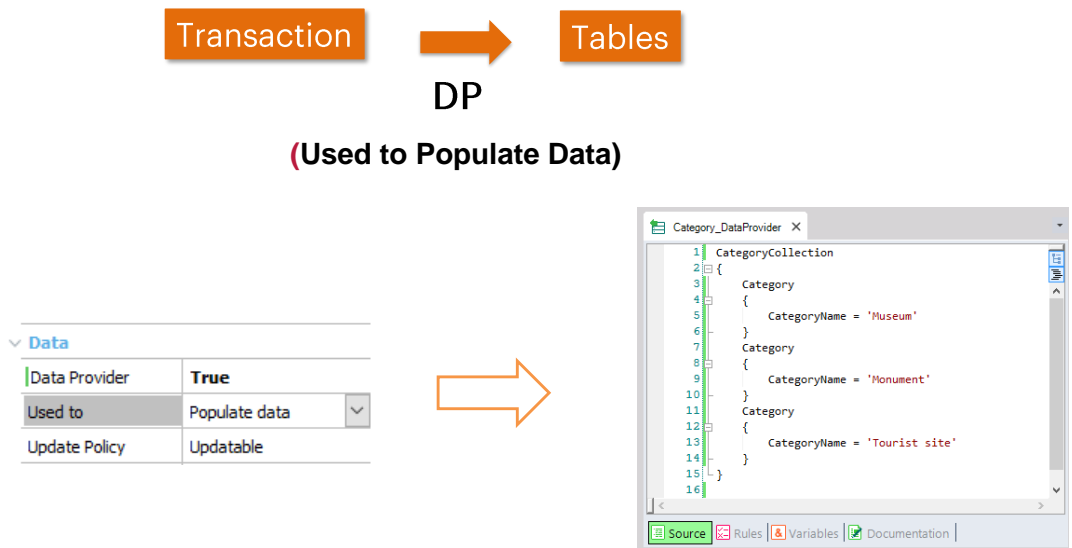# Dynamic Transactions

Transactions as "Views"

GeneXus™

So far, we've seen that in every transaction object, a table is created for each level to store its data and retrieve it later.

Transaction with Data Provider to initialize data

Transaction ➡ Tables

DP

(Used to Populate Data)

Data

Data Provider	**True**
Used to	Populate data
Update Policy	Updatable

```
Category_DataProvider  ×
 1  CategoryCollection
 2  {
 3      Category
 4      {
 5          CategoryName = 'Museum'
 6      }
 7      Category
 8      {
 9          CategoryName = 'Monument'
10      }
11      Category
12      {
13          CategoryName = 'Tourist site'
14      }
15  }
16
```

Source | Rules | Variables | Documentation

We have already seen that we can associate a Data Provider with a transaction in order to populate its tables with data.

Transaction with Data Provider to initialize data

Transaction → Tables

DP

(Used to Populate Data)

Transaction uses:
 Insert, Update, Delete data
 Navigate (retrieve) data

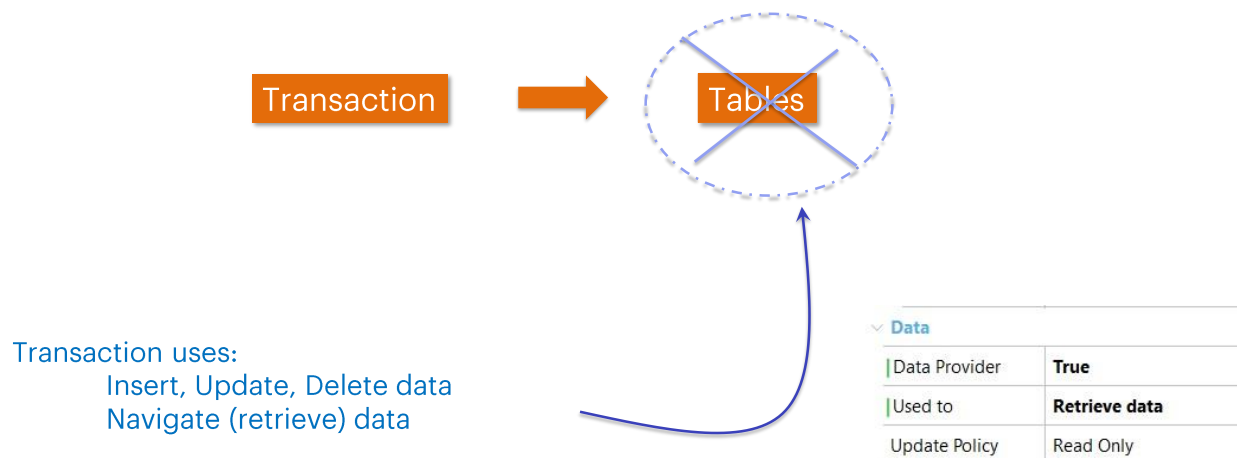| Data	
Data Provider	**True**
Used to	Populate data
Update Policy	Updatable ˅
Data warehousing	Updatable
DW transaction	Read Only

In that scenario, the Data Provider is used only for initialization. Then the transaction will behave normally; that is to say, it will access its tables as usual to retrieve the data, and will allow inserting, updating and deleting records as usual.

Note that the **Update Policy** property takes the Updatable value to enable these updates and this behavior.

But we can also change this transaction's behavior and prevent the data from being updated. That is, once the tables with data have been initialized, they cannot be modified and no new data can be added.
To do this you must change the update policy, which by default takes the Updatable value, and set it to Read only.
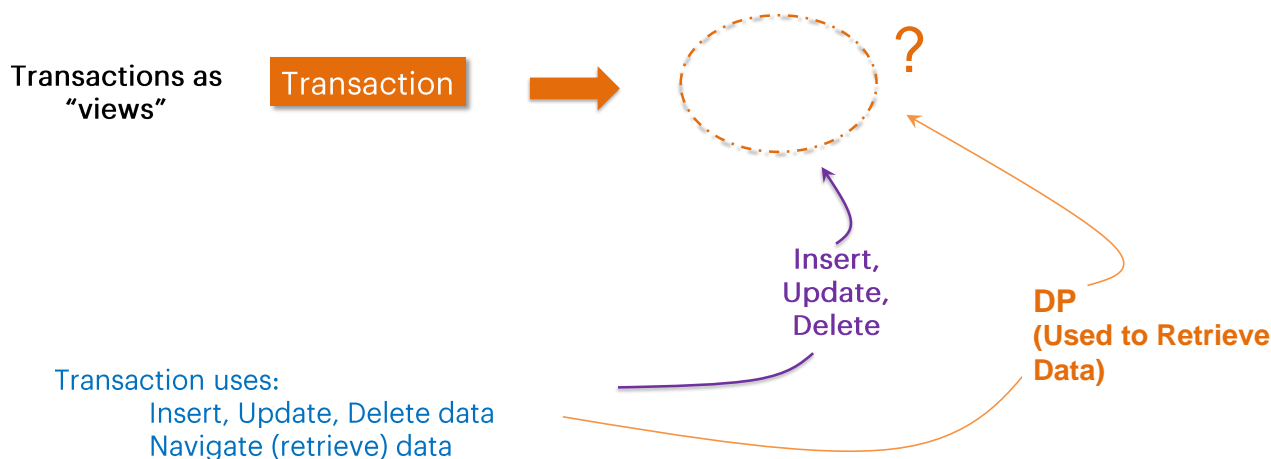
Transaction with Data Provider to retrieve data

Transactions as "views"

Transaction ➡️ Tables

Transaction uses:
    Insert, Update, Delete data
    Navigate (retrieve) data

Data	
Data Provider	**True**
Used to	**Retrieve data**
Update Policy	Read Only

We will now see that it is possible to continue using the transaction as usual but without storing the data in the associated tables. In short, we will have a case of transactions from which no tables are created in the application's database.

We can achieve this by indicating that the Data Provider associated with the transaction will be used to Retrieve data.

Transaction with Data Provider to retrieve data

Transactions as "views"

Transaction → ⭕ **?**

Insert, Update, Delete

DP (Used to Retrieve Data)

Transaction uses:
Insert, Update, Delete data
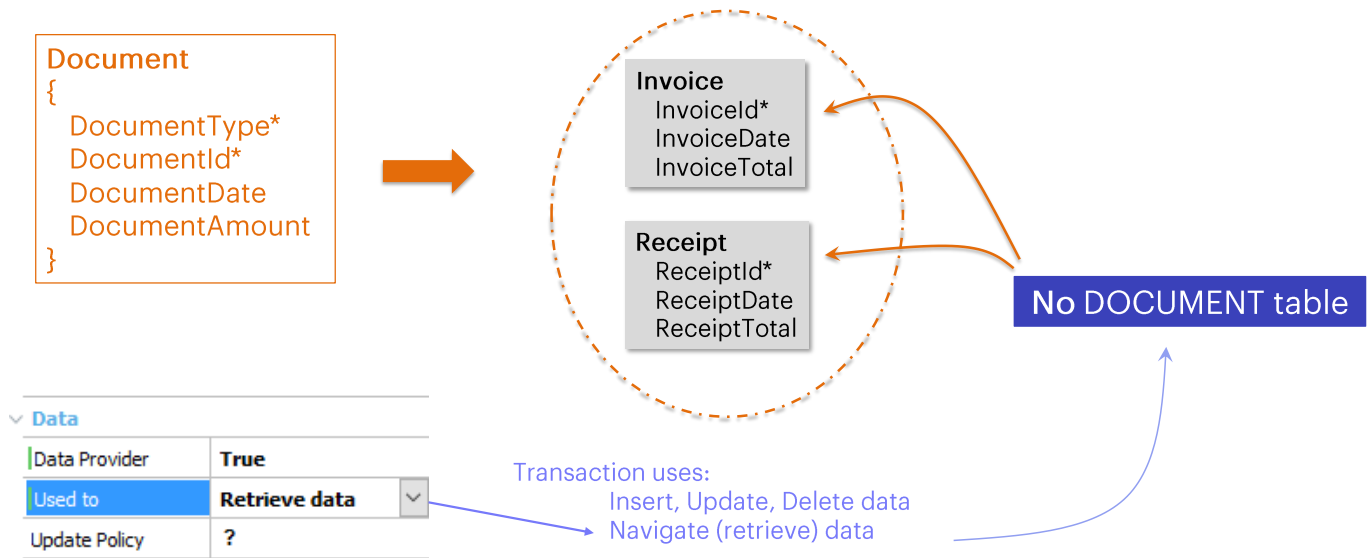Navigate (retrieve) data

But if no tables are created, somehow we will have to specify where the information will be retrieved from every time the user wants to browse his or her data. In addition, we will have to indicate what to do when the user enters data on the screen and wants to insert, update or delete it from the corresponding tables.

To Insert, Update or Delete we will have to explicitly program three events with these names.

To retrieve the transaction's data, we will have to program the Data Provider associated with it.
Just like data population, the Update Policy property will allow us to indicate if the transaction can be used only to retrieve information, or also to update it.

Scenario: integrity relationship of "or" type

```
Document
{
    DocumentType*
    DocumentId*
    DocumentDate
    DocumentAmount
}
```

Invoice
InvoiceId*
InvoiceDate
InvoiceTotal

Receipt
ReceiptId*
ReceiptDate
ReceiptTotal

No DOCUMENT table

Data	
Data Provider	True
Used to	Retrieve data
Update Policy	?

Transaction uses:
Insert, Update, Delete data
Navigate (retrieve) data

Let's see an example.

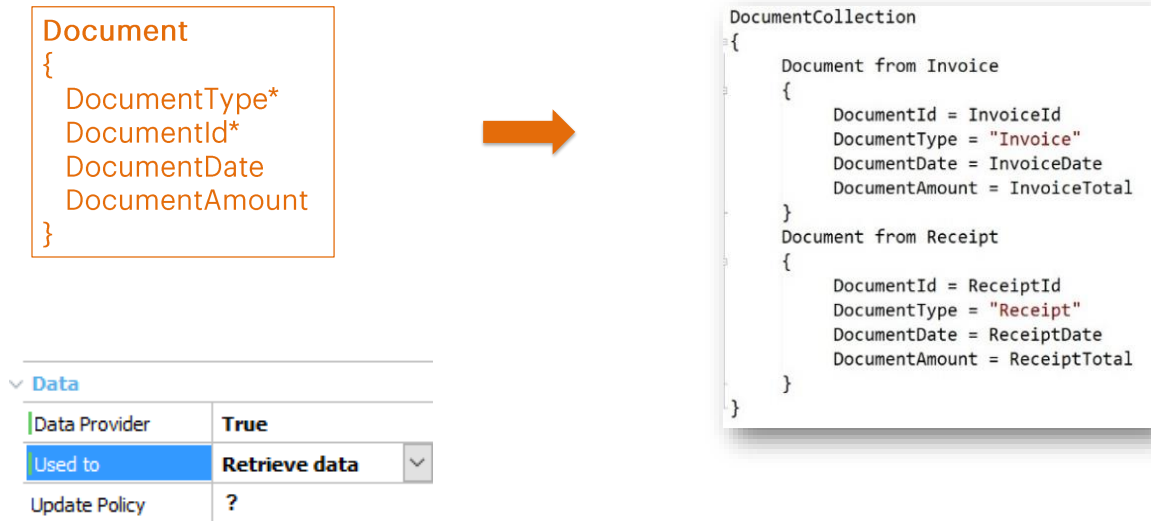Let's suppose that we have two standard transactions:
- **Invoice**, to represent the invoices issued by the travel agency to its clients for purchasing tickets, trips, and so on. These invoices are identified with a sequential number.
- **Receipts**, to represent the receipts issued by the travel agency to its clients for their purchases. Receipts these also identified with sequential numbers.

The accounting system of the Travel Agency needs to handle these invoices and receipts as Documents in general, to then be able to handle these Documents in Accounting Movements.

First, we create the Document transaction, with an identifier made up by DocumentType and DocumentId. Why do we need a compound identifier? In order to determine, for example, whether we are dealing with invoice 1 or receipt 1.

This transaction will be like a "view" that will unify the information contained in the Invoice and Receipt tables. That is to say, it will not create a table to contain the data; instead, it will take it from the tables corresponding to Invoice and Receipt.

Scenario: integrity relationship of "or" type

```
Document
{
   DocumentType*
   DocumentId*
   DocumentDate
   DocumentAmount
}
```



```
DocumentCollection
{
    Document from Invoice
    {
        DocumentId = InvoiceId
        DocumentType = "Invoice"
        DocumentDate = InvoiceDate
        DocumentAmount = InvoiceTotal
    }
    Document from Receipt
    {
        DocumentId = ReceiptId
        DocumentType = "Receipt"
        DocumentDate = ReceiptDate
        DocumentAmount = ReceiptTotal
    }
}
```

Data	
Data Provider	True
Used to	Retrieve data
Update Policy	?

To this end, we set its Data Provider property to True, and indicate that we will use this Data Provider to receive information.

After that, GeneXus will automatically understand that the table associated with the transaction must not be created because this Data Provider will be used to indicate from where to obtain the data. In this case, it will be from the INVOICE and RECEIPT tables associated with the corresponding transactions.

Let's look at the source of the Data Provider associated with this transaction

We have a Document group to retrieve all the documents that are invoices, and another group to retrieve all the documents that are receipts.

From now on, every time the transaction is run to navigate its data, this Data Provider will be run to load the corresponding information on the screen, in a completely transparent way for both the developer and the user. Nobody will notice that the transaction doesn't have a table.

Scenario: integrity relationship of "or" type

Dynamic transaction as a base transaction: Documents list

```
Document
{
  DocumentType*
  DocumentId*
  DocumentDate
  DocumentAmount
}
```

```
For each Document order (DocumentDate)
      print Documents
Endfor
```

After this, the dynamic transaction is used as any other transaction. For example, to print all the documents ordered by date in descending order, we can create a procedure with a For Each command similar to this:

It is very interesting to note that in this For Each command Document is declared as a base transaction. And from this, since the attributes mentioned in the printblock belong to Document, GeneXus determines that the base table of this For Each command is Document.

But Document does not exist as a table in our database but is a view through the Data Provider.

Scenario: integrity relationship of "or" type

Database update: Insert, Update and Delete

```
Document
{
   DocumentType*
   DocumentId*
   DocumentDate
   DocumentAmount
}
```

→ Events →

```
Event Insert
    If DocumentType = Type.Invoice
        &Invoice = New()
        &Invoice.InvoiceId = DocumentId
        &Invoice.InvoiceDate = DocumentDate
        &Invoice.InvoiceTotal = DocumentAmount
        &Invoice.Insert()
    else
        &Receipt = New()
        &Receipt.ReceiptId = DocumentId
        &Receipt.ReceiptDate = DocumentDate
        &Receipt.ReceiptTotal = DocumentAmount
        &Receipt.Insert()
    endif
Endevent
```

Transactions are not only used to retrieve their data but also to update it. How do we go about it, since we don't have a table associated with this transaction?

Let's look at the Update Policy property. If this property is set to "Updatable," the events Insert, Update and Delete will be offered in the transaction to program how to insert, update and delete the data entered by the user on the screen. Only the developer will know what to do in each case with this information.
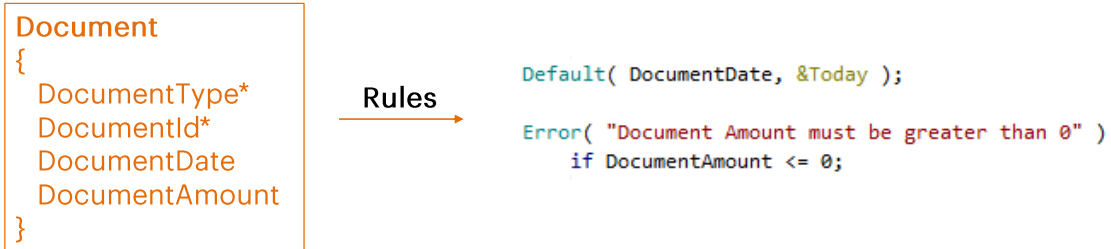
These actions will be enabled depending on the reality.

Look at the Form of the transaction. When the user has finished completing the fields in this screen to insert a new document and has pressed the Confirm button, we will have to insert a new record in the Invoice or Receipt table as appropriate, depending on the value entered in the DocumentType attribute.

So let's look at the Events sector of the transaction. We have programmed the Insert event, using the Invoice and Receipt variables based on the Invoice and Receipt Business Component data types, respectively.

Note that we could have used the Save method instead of the Insert method of the Business Component. We don't need to write the Commit command because we're in the Document transaction that still has the Commit on Exit property set to Yes by default. That is to say, it will implicitly run the Commit.
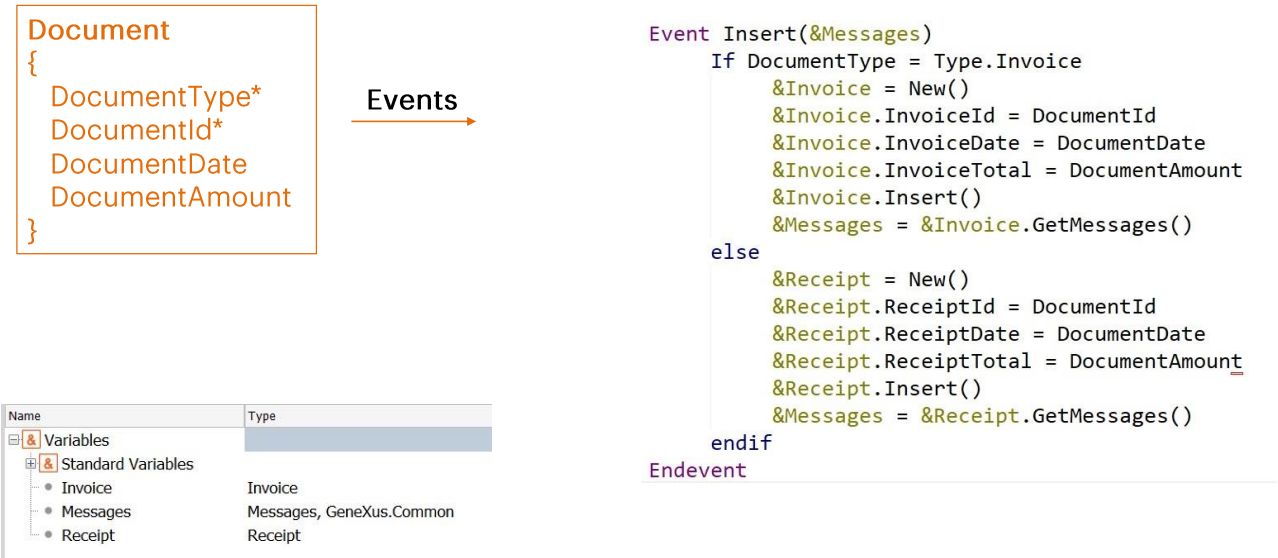
Rules? Triggering events?

```
Document
{
  DocumentType*
  DocumentId*
  DocumentDate
  DocumentAmount
}
```

Rules →

```
Default( DocumentDate, &Today );

Error( "Document Amount must be greater than 0" )
      if DocumentAmount <= 0;
```

❖ Evaluation tree and triggering moments are identical

❖ They are specified and triggered just like in a standard transaction

Let's see what happens if we have rules stated at the dynamic transaction level.

When are they triggered? What happens with the evaluation tree?

Both the evaluation tree and the rule triggering moments are the same as if we were dealing with a regular transaction.

Messages triggered in the Business Components

```
Document
{
   DocumentType*
   DocumentId*
   DocumentDate
   DocumentAmount
}
```

Events →

```
Event Insert(&Messages)
    If DocumentType = Type.Invoice
        &Invoice = New()
        &Invoice.InvoiceId = DocumentId
        &Invoice.InvoiceDate = DocumentDate
        &Invoice.InvoiceTotal = DocumentAmount
        &Invoice.Insert()
        &Messages = &Invoice.GetMessages()
    else
        &Receipt = New()
        &Receipt.ReceiptId = DocumentId
        &Receipt.ReceiptDate = DocumentDate
        &Receipt.ReceiptTotal = DocumentAmount
        &Receipt.Insert()
        &Messages = &Receipt.GetMessages()
    endif
Endevent
```

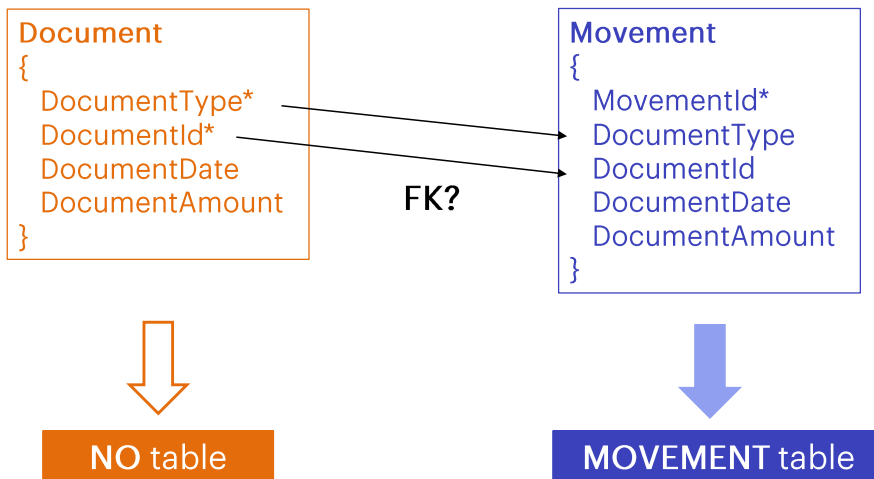| Name	Type
& Variables	
& Standard Variables	
Invoice	Invoice
Messages	Messages, GeneXus.Common
Receipt	Receipt

As for the success or failure messages triggered during the execution of the Business Components Invoice and Receipt, we can also retrieve them.

To this end, we set the Messages variable, based on the Messages data type, a collection, as a parameter in each of the Insert, Update and Delete events at the transaction level.

These messages are displayed in the form of the dynamic transaction in a completely transparent way.

Referential Integrity ?

Document
{
  DocumentType*
  DocumentId*
  DocumentDate
  DocumentAmount
}

FK?

Movement
{
  MovementId*
  DocumentType
  DocumentId
  DocumentDate
  DocumentAmount
}

NO table

MOVEMENT table

Moving forward with this example, remember that we have said that the Travel Agency's accounting system needs to handle all documents as accounting movements.

Since the DOCUMENT table associated to the Document transaction will not be created, we can assume that then in the MOVEMENT table associated with the standard Movement transaction, the pair of attributes made up of DocumentType and DocumentId cannot constitute the foreign key that they should.

So what's going to happen with the referential integrity check? Can GeneXus make it?

Since referential integrity must be ensured, GeneXus generates SQL triggers to do so. Therefore, it could be said that the pair DocumentType and DocumentId make up a "pseudo" foreign key in Movement.

In sum, in Document it will not be possible to delete invoices or receipts that have an associated movement. In addition, it will not be possible to add a Movement that doesn't exist as a Document.

Summary
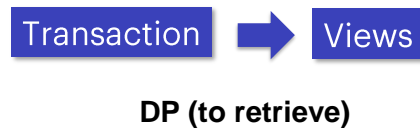
1. Data Provider: True

2. Used to: Populate Data

Transaction ➡ Tables

DP (to initialize)

3. Update Policy: Updatable

It allows (or not) making the usual updates (on the transaction tables)

1 Data Provider: True

2. Used to: Retrieve Data

Transaction ➡ Views

DP (to retrieve)

3. Update Policy: Updatable

It allows (or not) making updates, but they have to be programmed in special events: **Insert, Update, Delete**

Let's review the concepts we have learned:

When the Data Provider property at a transaction level takes the True value, GeneXus asks us what we are going to use it for.

If we indicate that we will use it to populate the table with data, then the transaction will generate its corresponding associated tables.
Through the Update policy property we will allow, or not, the updating of records in the usual way.

When the Data Provider property associated with a transaction is set to True, and we indicate that we are going to use it to receive data, GeneXus understands that the transaction will not have any associated tables and that it will make a view from that Data Provider.

Then, according to the value that we indicate in the Update policy itself, the corresponding events should be programmed to allow the insertion, modification or deletion of records in the corresponding tables.

More about Dynamic Transactions...

- Other use cases:

  Selection
  Data grouping
  Temporary relations

- More examples of use of dynamic transactions

  http://wiki.genexus.com/commwiki/servlet/wiki?28062,Dynamic%20Transactions.

Finally, it is worth mentioning that we have seen a single use case of dynamic transactions, but there are multiple cases, such as Selection, Data Grouping and Temporary Relationships that we will address in other courses.

You can access more information using the link on the screen.

training.genexus.com
wiki.genexus.com
training.genexus.com/certifications