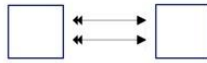# More Use Cases of Subtypes

GeneXus™

Direct
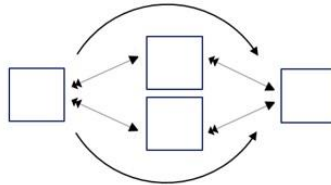
In previous videos we have studied the case of multiple references from one table to another directly related to it
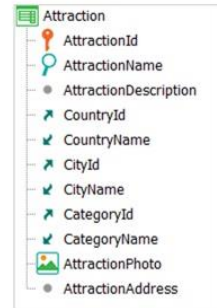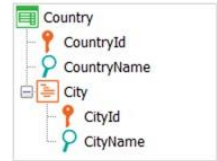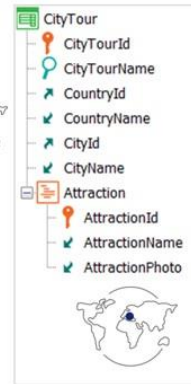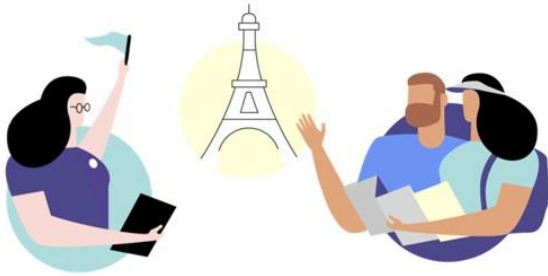
Direct

Indirect

...and also the case in which these references are indirectly related, since from one table we have two paths to get to another, so it is often necessary to perform disambiguation using subtypes.

In this video, we will study another case of indirect multiple reference, its problems, and possible solutions.

Suppose we need to register the tours that are offered to the travel agent's customers for visiting the different tourist attractions of a given city.

To do so, we will create the CityTour transaction, where in the first level, in addition to registering the name of the tour, we will specify its country and city. The second level will indicate the tourist attractions visited during the tour.

Note that each tourist attraction has a country and city defined, so if we do nothing, the user may enter for a city tour an attraction that is not in the same country or city of the city tour.
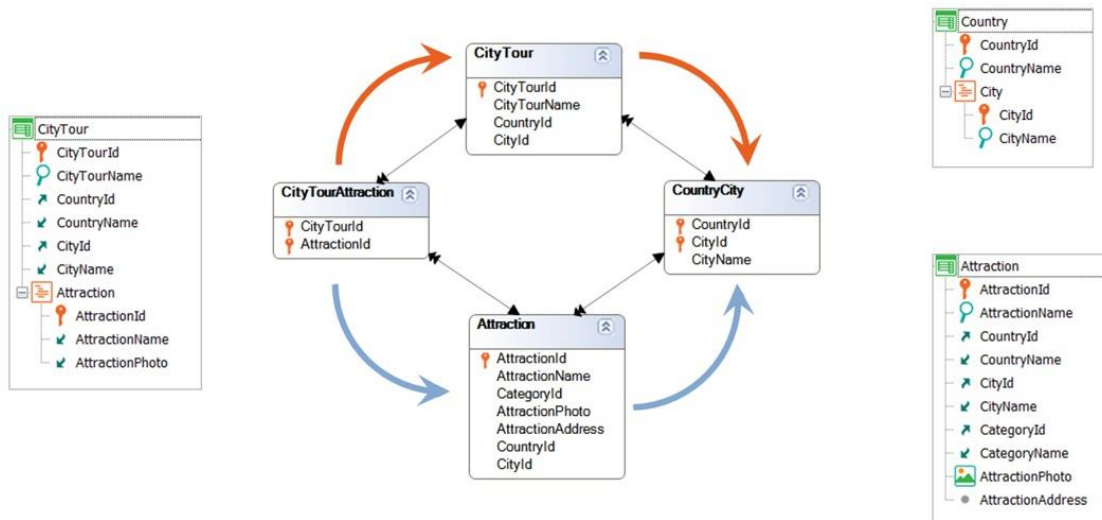
If we look at the table diagram, we can clearly see that we have two different ways to get from the second level of CityTour to the cities table. That is, in the extended table of CityTourAttraction there is the cities table, CountryCity, but we get to it by two different ways, and starting from a CityTourAttraction record, we are not sure if the city of the city tour matches the city of the attraction.

To make sure that both paths match when inserting or modifying city tours, the use of subtypes is not mandatory.
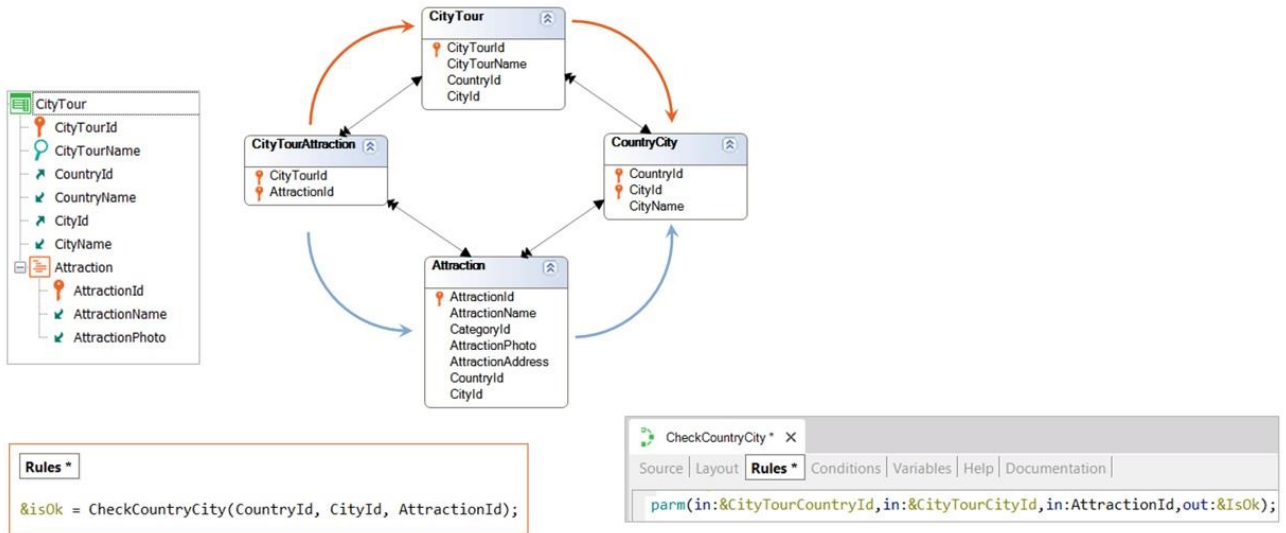
We could, for example, invoke a procedure in the rules of the CityTour transaction to which we send in a parameter the attributes CountryId, CityId and AttractionId, and what it will do is to check that the pair CountryId, CityId, which is clearly that of CityTour, matches the pair that is found when accessing the Attraction record according to the attraction that we want to add to the city tour.

Here we see that the procedure receives in variables the ID of the country and city of the CityTour, and in an attribute the AttractionId of the line that we want to check. And it will return a Boolean value that indicates whether country and city match or not.
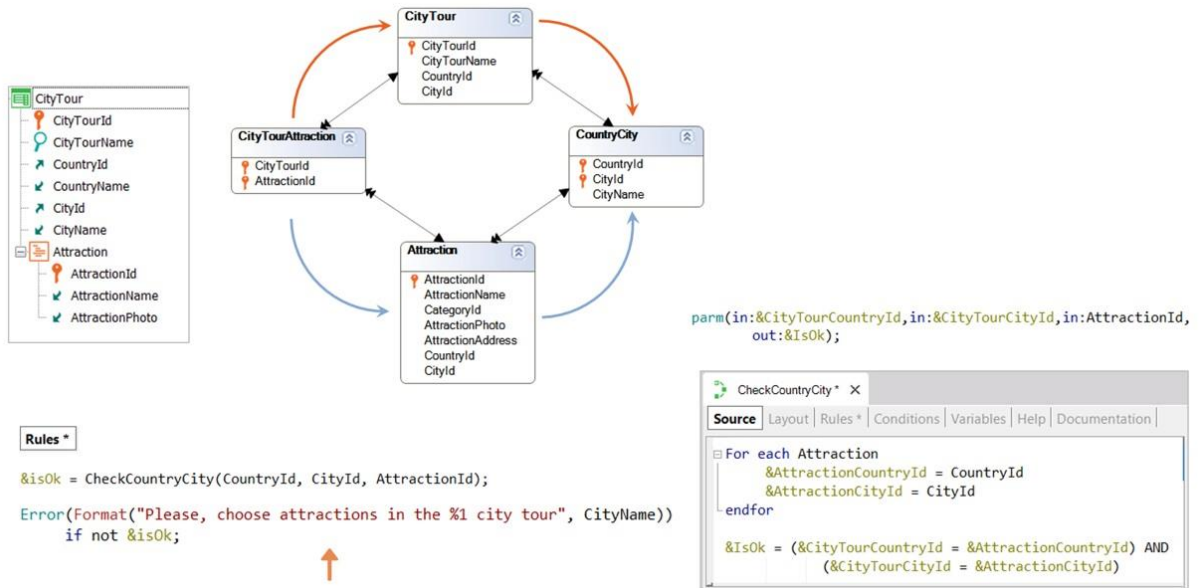
Then in the Source we access the table of the Attraction transaction and in two new variables we load the country and city of the attraction (through the automatic filter).

A Boolean variable returns True if the country received in a parameter matches that of the attraction and also the city received in a parameter matches that of the attraction. Otherwise, False is returned.

This is how in the transaction we condition the error rule to be triggered if the procedure returned False.

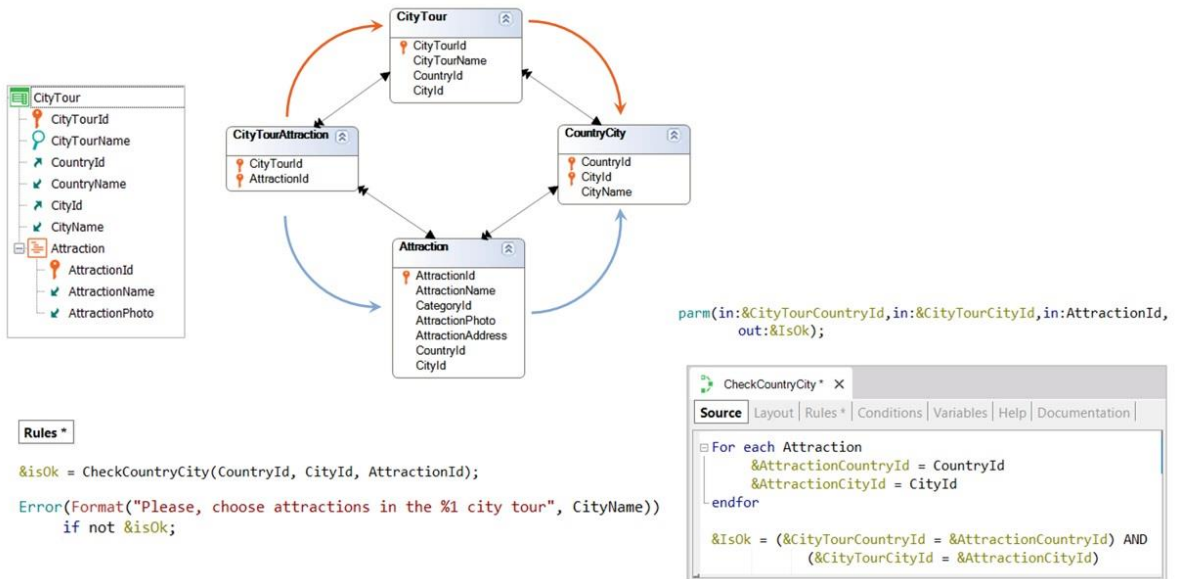Let's see this in GeneXus.

If we now run the transaction with a city tour that we had already loaded and that runs through Beijing, with the two attractions that we see in Beijing, and we want to add another one, which is not from Beijing, such as the Eiffel Tower, we see how the error is being thrown correctly.

And if we add one from Beijing, such as the forbidden city, we can save without any problem.

```
parm(in:&CityTourCountryId,in:&CityTourCityId,in:AttractionId,
     out:&IsOk);
```

CheckCountryCity * ×

| Source | Layout | Rules * | Conditions | Variables | Help | Documentation |

```
For each Attraction
      &AttractionCountryId = CountryId
      &AttractionCityId = CityId
endfor

&IsOk = (&CityTourCountryId = &AttractionCountryId) AND
        (&CityTourCityId = &AttractionCityId)
```

Rules *

```
&isOk = CheckCountryCity(CountryId, CityId, AttractionId);

Error(Format("Please, choose attractions in the %1 city tour", CityName))
      if not &isOk;
```

In this way, we did not have to use subtypes to ensure this check through the transaction. Note, however, that we had to call a procedure to be able to access the attraction's CountryId and CityId attributes without ambiguity, loading them manually into two variables, so that they are not confused with those of the CityTour.

We will have this problem every time we are doing something with a record of the CityTourAttraction table and we need to obtain the country - city pair. Because the question is, which one? The CityTour pair or the Attraction pair?
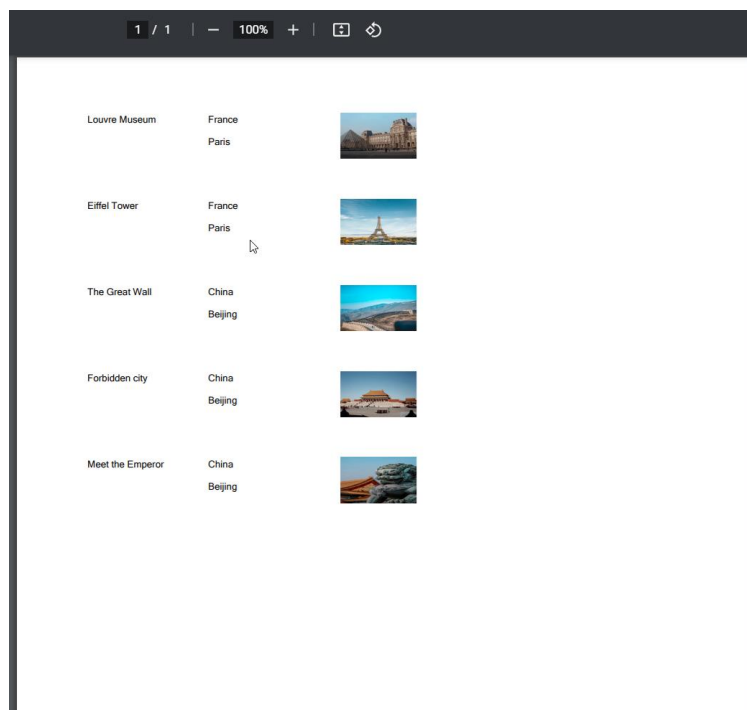
For example, let's suppose we want to run through the table of the Attraction level of CityTour and show the name of the attraction, its country and city, and its photo. How do we know if it will take the CountryName and CityName attributes from the CountryId and CityId of the Attraction table or if it will take them from those of the CityTour table? There is an ambiguity here. It will take them from either of them. To find out specifically which one it chose, we read the navigation list. The navigation list indicates that from CityTourAttraction it will access CityTour, just to bring the country and city ID, and Attraction to bring the other attributes that we want to list, which are the photo and the name. In short, we can see that it didn't choose to show the country and city of the attraction, but the country and city of the CityTour.

In this case, this ambiguity doesn't matter because every time an attraction is entered we check that these values match. And that is why in the list it seems that we are looking at the country and city of the attraction and not of the city tour.
However, as soon as we override that data check, for example, by going to the Attraction transaction and changing the city of the Eiffel Tower to Nice instead of Paris, we see that the list still shows Paris, because it is the one of the CityTour where the Eiffel Tower is, and not the one of the attraction itself. It allowed us to break our rule because it is defined only in the CityTour transaction, and we made the change in Attraction, and even opening the transaction does not cause the error because we are not doing anything with the line.
Therefore, we clearly see that we would need to check everywhere this data can be modified.
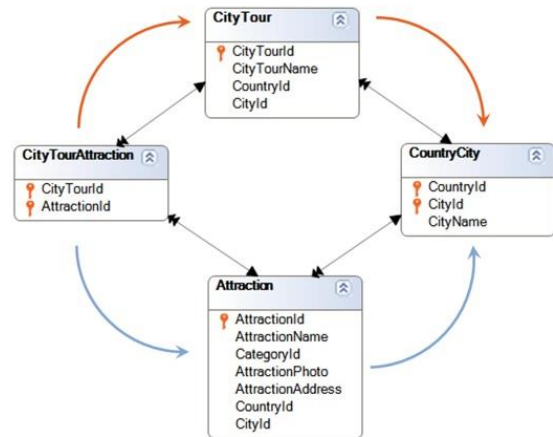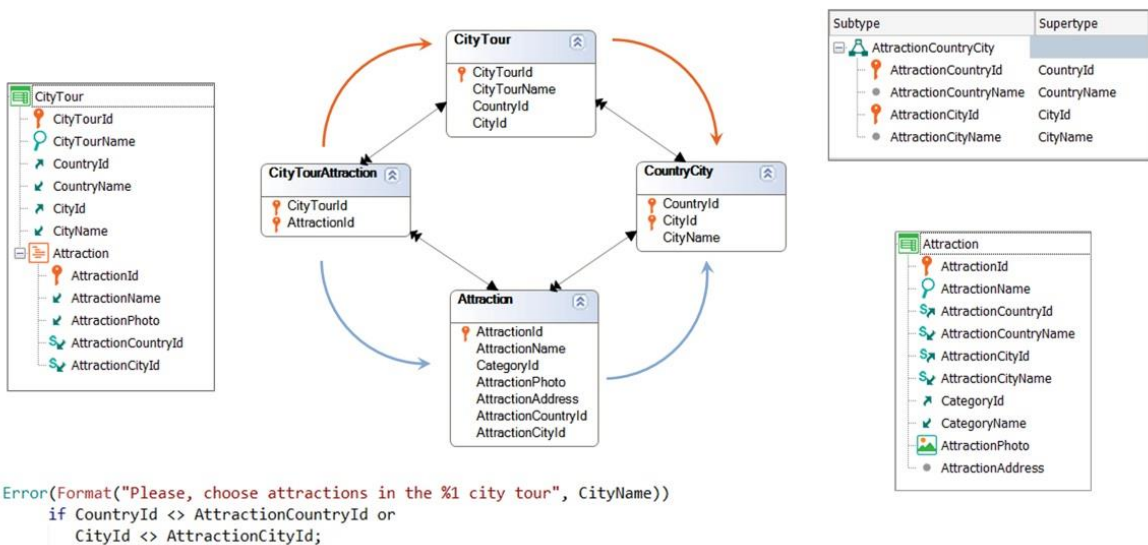
If we are sure that the check will be performed everywhere and therefore the data on both paths will always match, then we may not care which path is chosen to retrieve it. Although sometimes it does, for performance reasons. In the example we saw of the list of attractions of the city tours, having to access only Attraction to go to CountryCity and Country is more efficient than having to go to Attraction to retrieve its name and photo and then to CityTour to go to CountryCity and Country.

If we need to be able to indicate at a given moment one of the two paths, because they are not the same, subtypes can be used.

We will analyze three possibilities, starting with two obvious ones and ending with the least obvious one.

```
Error(Format("Please, choose attractions in the %1 city tour", CityName))
    if CountryId <> AttractionCountryId or
    CityId <> AttractionCityId;
```

The first will be to modify the name of the country-city attributes in this path, in order to be able to identify it.

Thus, we define a group of subtypes for the country and city of attraction.

Note that this group has two primary attributes: AttractionCountryId and AttractionCityId, which correspond to the primary key of the CountryCity table, according to the supertypes indicated: {CountryId, CityId}.

And that we replaced the supertypes with these subtypes in the Attraction transaction.

Thus, we see that the path below can now be identified.
We could even add to the CityTour transaction the country and city attributes inferred through AttractionId, in order to implement the check directly through the error rule.
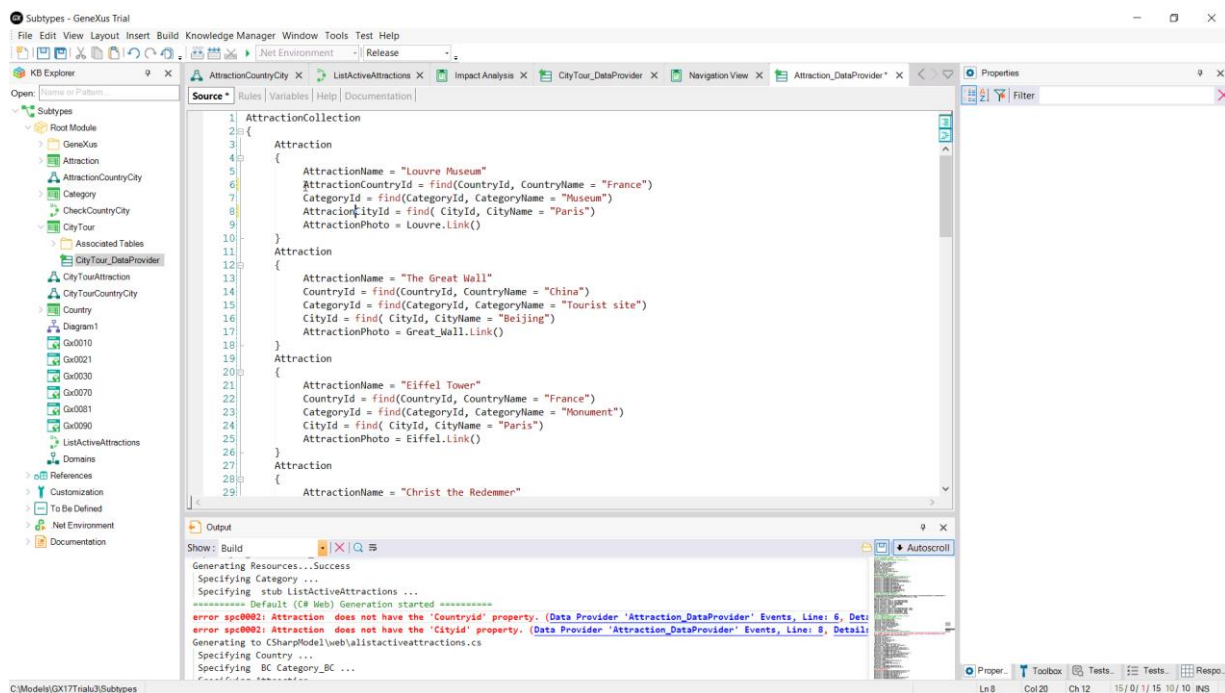
Note that after making these changes it is easy to remove the ambiguity from the list we had before. It is enough to change here CountryName and CityName for the subtypes.

However, since we already had data in the tables, it indicates that it should reorganize, in particular, the Attraction table. It must place the new attributes – the subtypes – and delete the old ones – the supertypes. And it seems that it will correctly transfer the data from the old attribute, the supertype, to the new one, the subtype.

However, when reorganizing after completing the specification we find errors. In particular, it indicates that in the DataProvider to populate the table with attractions, a Data Provider that we had before, the attribute CountryId is being used, but it is no longer in Attraction. The same goes for CityId. Here we clearly see a disadvantage of this solution. We will have to replace one by one the old attributes with the new ones in all the objects that already had access to the Attraction table before.

In fact, if we had already been doing development work on the application where CityTour was not considered, we had not yet encountered any ambiguity problem, so presumably we already had many other objects working on CountryId and CityId in Attraction.

If we choose this solution, we will have to solve all these pitfalls.

Leaving that aside, if we now look at the navigation list of the list we were interested in, we now see that to get country and city for each CityTourAttraction record, it is doing it through the Attraction table, as we wanted. We removed the ambiguity and explicitly chose the path we wanted.
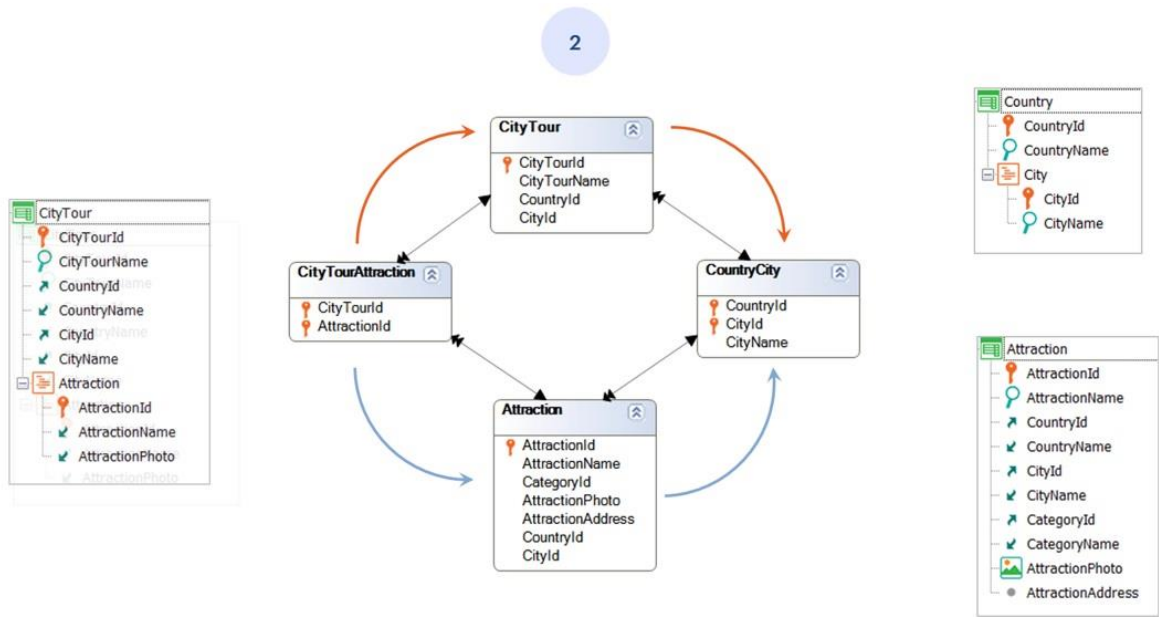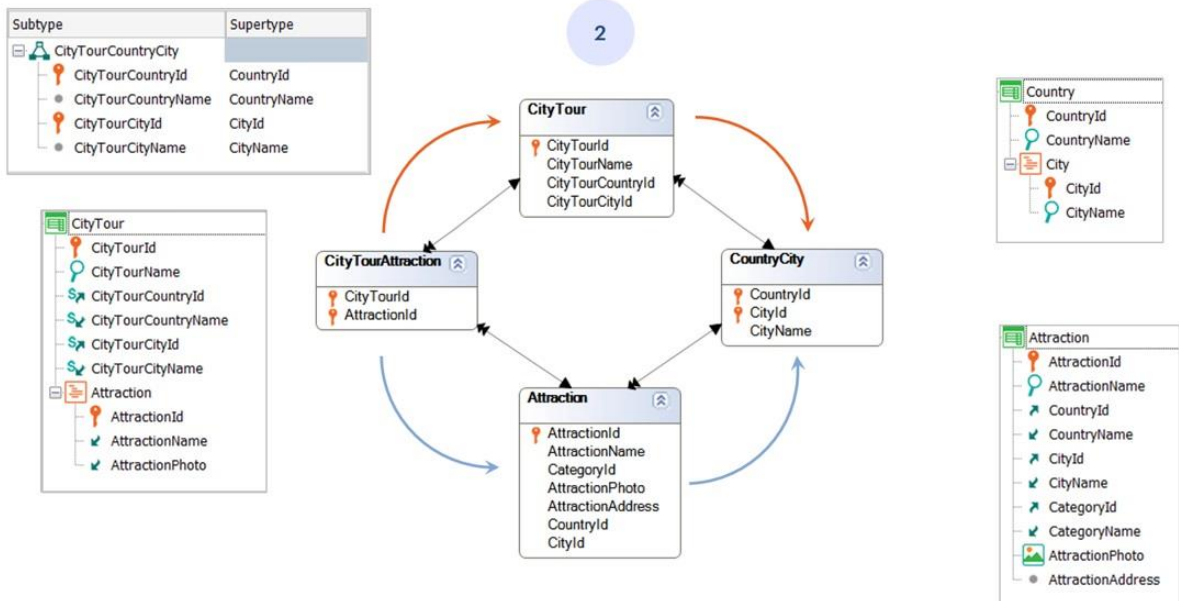
Now let's think of another alternative that is less complicated in relation to objects that already existed.

This second solution removes the ambiguity by modifying the name of the country-city attributes in this other path.
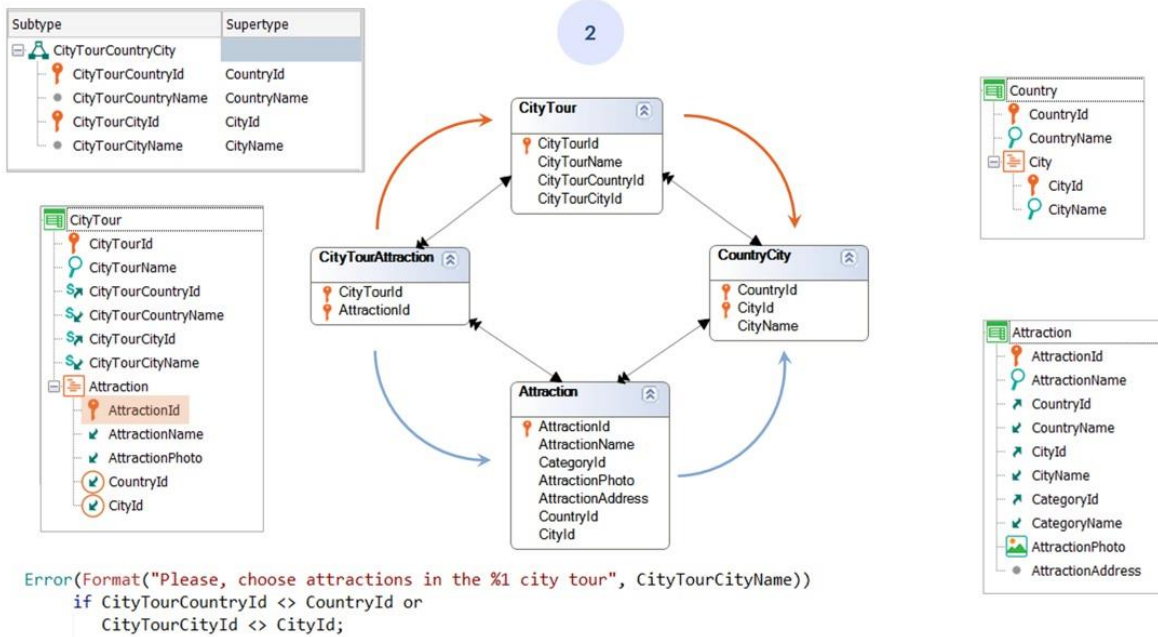
To do this we define the subtype group for country and city using them directly in the CityTour transaction header. The result is this change in the CityTour table (from supertypes to subtypes).

The difference between this solution and the previous one is that it is less likely that the flat transaction was built first, without the second level – which is the one that introduces the two paths to CountryCity. So it is not to be expected that there were other objects navigating CityTour before we thought of adding the second level, and then having to change attributes for subtypes in the first one.

In other words, it is to be expected that the CityTour and CityTourAttraction tables are created at the same time, instead of creating CityTour first, loading data to it and only later realizing that we will also need a second level (which is the one that generates the two paths and this solution).

With this solution we can see that the path above is now identifiable, so that...

Indirect Multiple references


```
Error(Format("Please, choose attractions in the %1 city tour", CityTourCityName))
    if CityTourCountryId <> CountryId or
    CityTourCityId <> CityId;
```

...for example, we can now implement the match check directly through the error rule, with no need for the procedure.

To do so, we must add CountryId and CityId to the structure, in order to use them in the rule. Note that we are clearly told that they are being inferred from AttractionId.
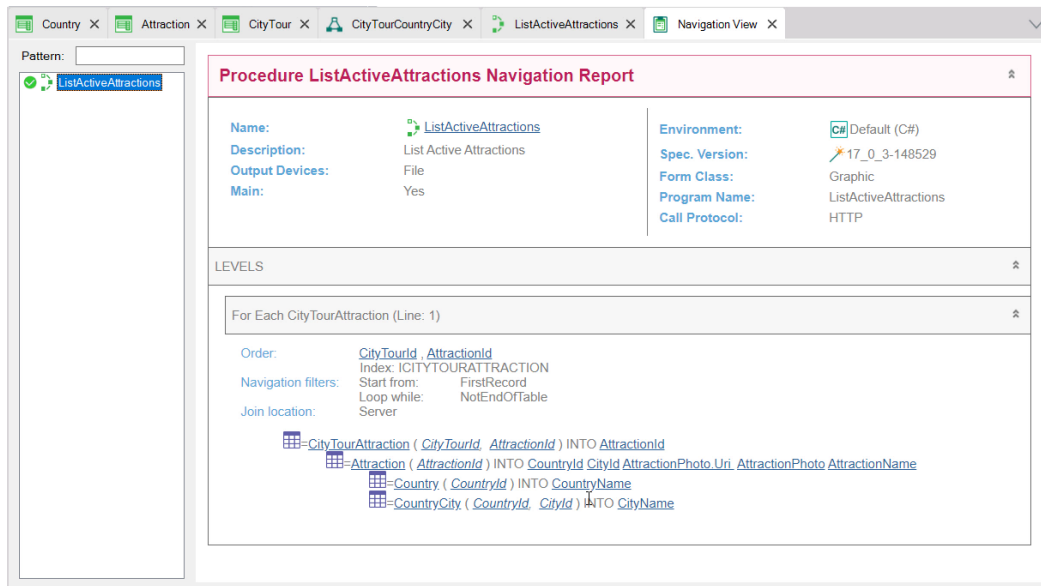
Now we have, then, this solution implemented in GeneXus; clearly if in the list we were analyzing we leave the supertypes CountryName and CityName, they will be taken from CountryId and CityId of the Attraction table. And no longer through the other path, that of CityTour. Let's confirm this in the navigation list. It shows what we expected. There is no longer any ambiguity here.
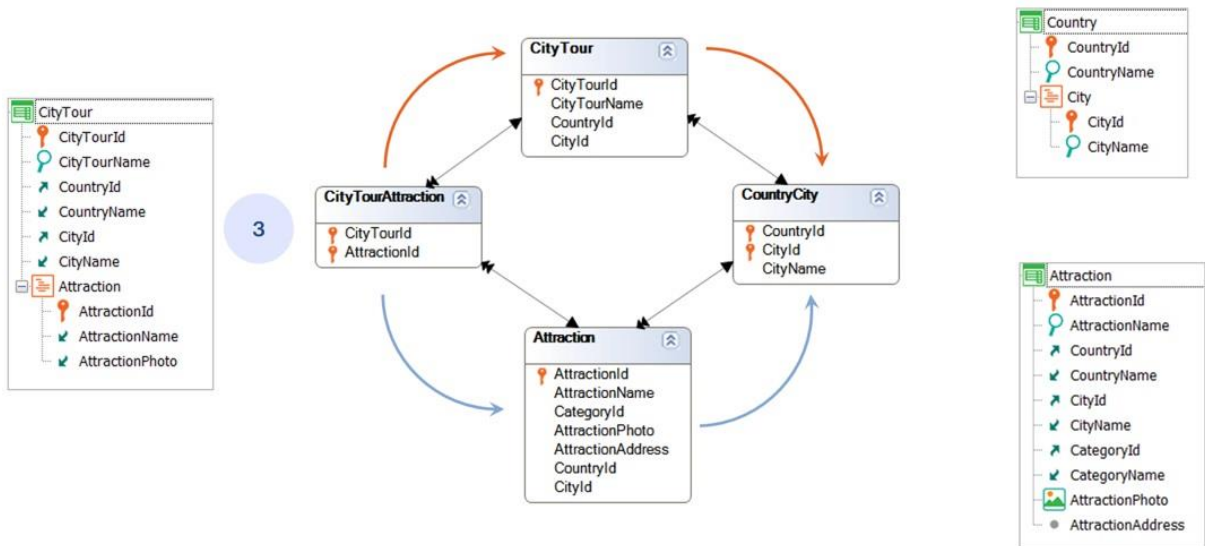
We now come to the last alternative with subtypes, which at first seems less intuitive, but has a great advantage: it provides disambiguation in the table itself in which the ambiguity occurs. That is to say, the table that originates the two paths.

If we think about the previous solutions, we are changing the names of the Country and City attributes in tables that, when looking at their extended table, have no ambiguity.
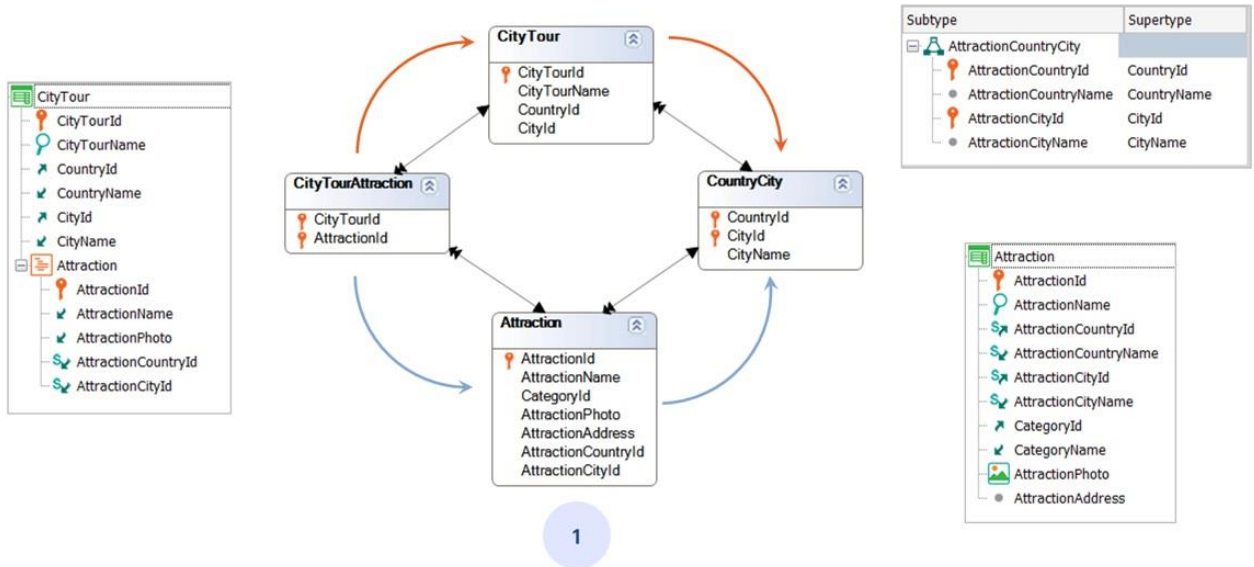
So, if we look at solution 1, and we think, for example, that we want to develop a Web Panel that shows the tourist attractions with their country and city information, we do not understand why there are subtypes in that table instead of supertypes. Looking from Attraction there is no need to define them.

Indirect Multiple references
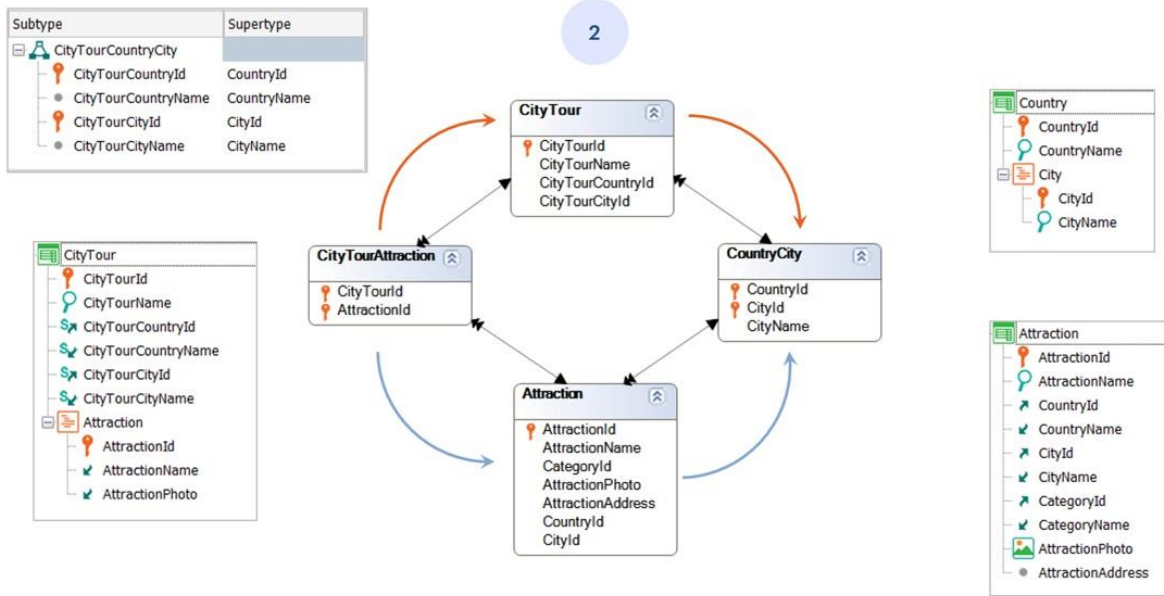
The same is true if we consider solution 2 and stand on CityTour.

If we wanted to list the city tours with their country and city, positioned on that base table, we would not understand, either, why subtypes are being used. Although in this case, as the CityTourAttraction table comes from a second level of the transaction that generates the CityTour table, they are more closely related and the rationale for having these subtypes can be seen more clearly.
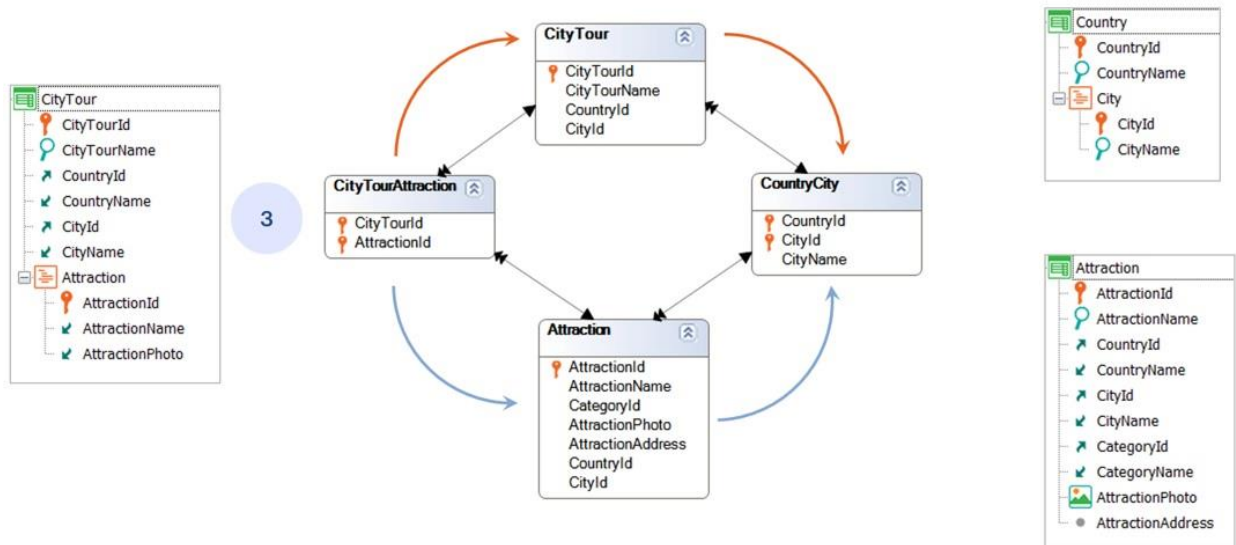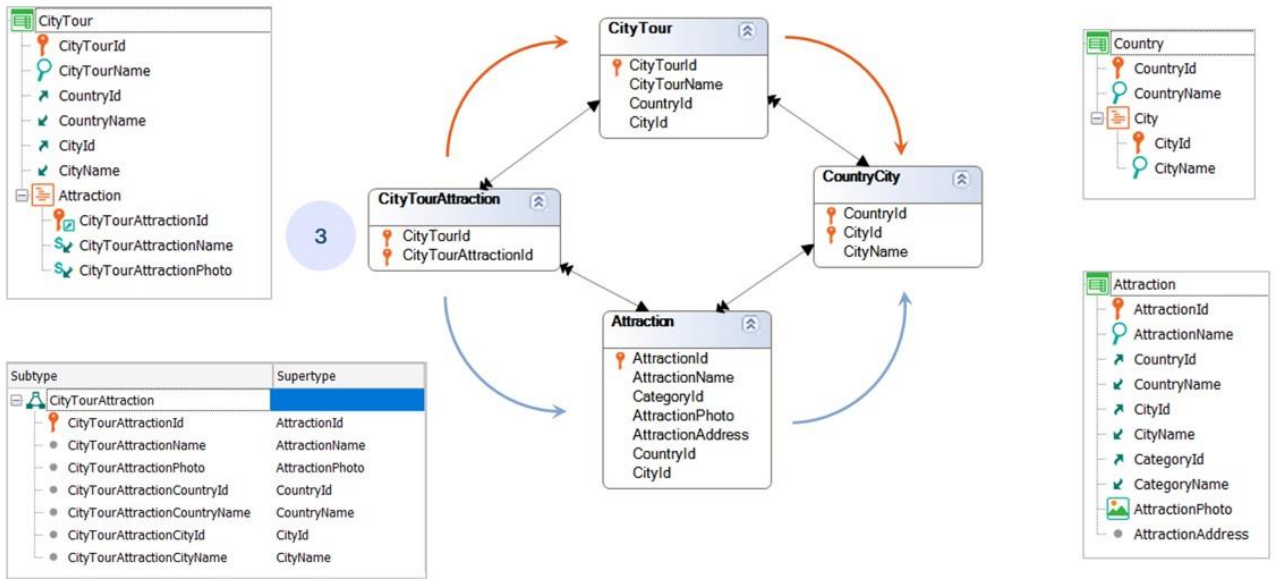
To perform a disambiguation in the table that causes the two paths seems to be an advantage. At least in the sense that ambiguity is inherent to this table, so it will never be unjustified for a subtype to appear there.

Now, how do we perform disambiguation in the table that makes those two paths appear? In the example, in the CityTourAttraction table itself.

What if we define a group of subtypes that allow us to modify the name of AttractionId when it appears as a foreign key, and all the attributes that are inferred from it and are of interest to us?

And then in CityTour instead of using the AttractionId attribute, we use that subtype? And, of course, we must now infer attraction name and photo also in subtypes of that group.

By doing this, the table diagram becomes like this. Note that the other tables should not be modified at all, so that all objects that were previously working on these other tables will continue to do so without any problem.

```
Error(Format("Please, choose attractions in the %1 city tour", CityName))
    if CountryId <> CityTourAttractionCountryId or
       CityId <> CityTourAttractionCityId;
```
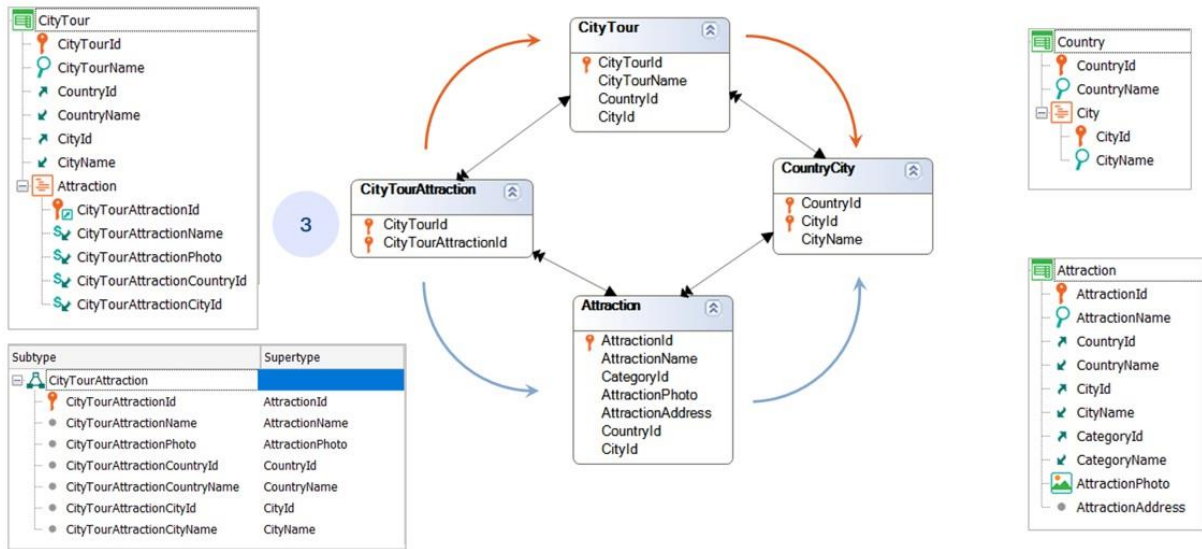
If now we would like to check that the country and city of the city tour are identical to the country and city of the attraction, it is enough to add to the transaction structure the two inferred attributes CityTourAttractionCountryId and CityTourAttractionCityId, and write the error rule as we see it.
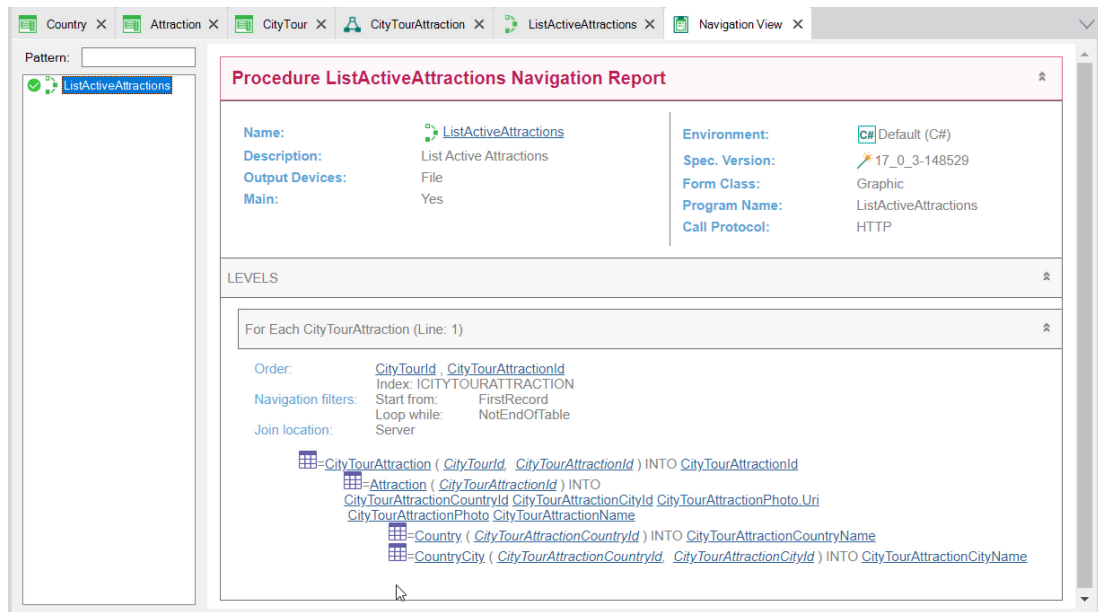
With this solution, the list we wanted to disambiguate from the beginning is very clear.

We run through the Attraction level of the CityTour transaction with a For each. And here we have to modify the attributes, which are all inferred from the attraction, but now it is no longer AttractionId, but its subtype. We change, then, all the attributes for their subtypes. In particular, the country name, and the city name. But when we look at the navigation list, it says that precisely these two attributes cannot be reached.

Why? Because while we have these two subtypes defined within the group, we have not specified them at the level of the transaction we are running through with the For each. If we now add them –which will have no negative effect since they are inferred–, and have the list navigated again, we find that there is no longer any problem.

And in fact, we see how it is retrieving the information correctly, from the subtype in the navigated table, accessing the Attraction table and from there CountryCity and Country.

However, having to add every time all the inferred information that you want to use in any navigation to the transaction structure can be a bit annoying.

Indirect Multiple references
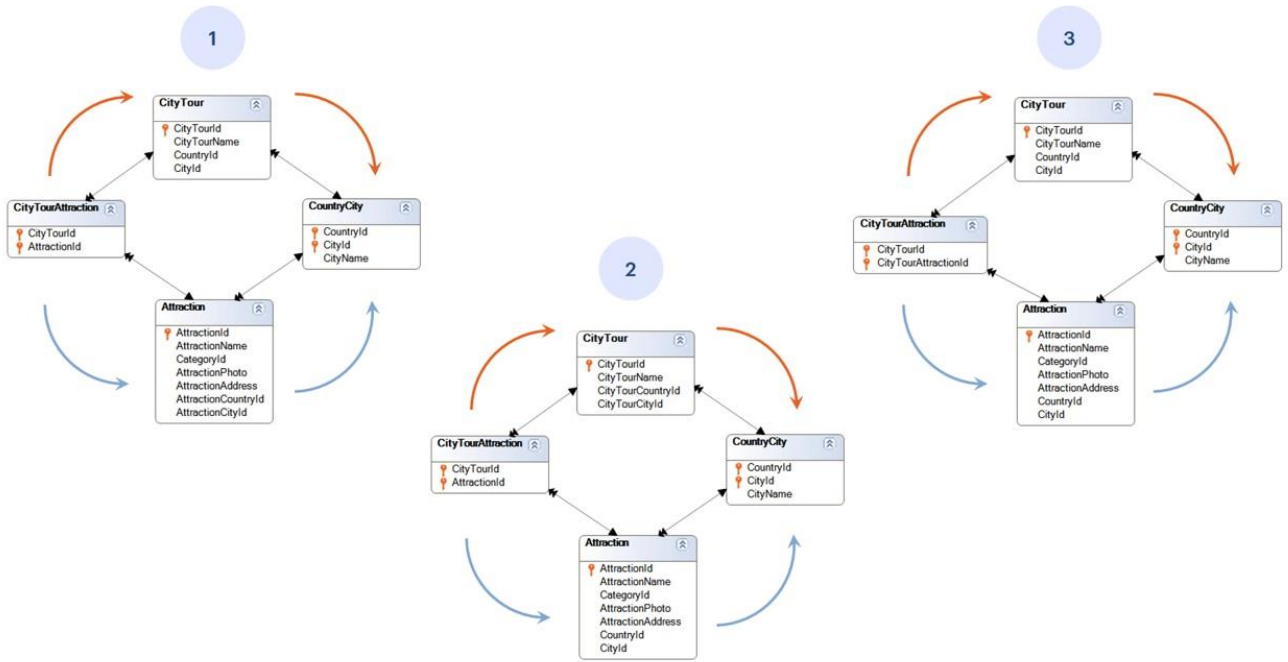
Lastly, every solution has its pros and cons and this depends on the particular case to which it is being applied.

For example, if all transactions are created together, then the problem of having other objects using the old attributes from before disappears, and options 1 and 2 are no longer problematic in that sense. However, they do not in themselves justify the use of subtypes.

In this case it doesn't seem too much of a problem, because, for solution 1, while standing in Attraction it is enough to look at the directly superordinate table to find out why. And the same can be said about solution 2, while positioned on CityTour.
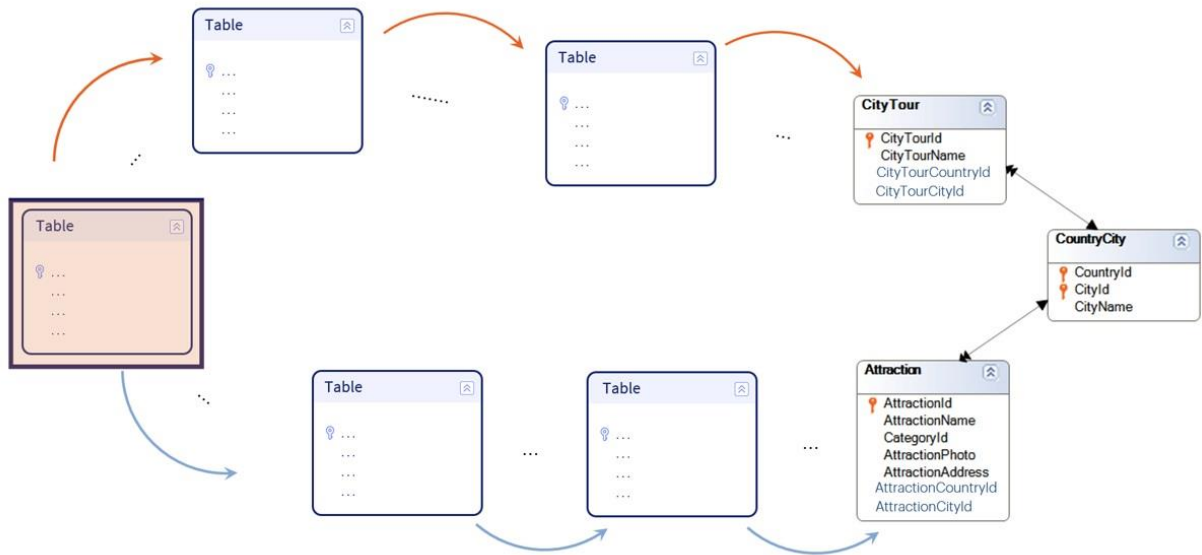
But, if in solution 1 we are positioned on Attraction and there we define subtypes for CountryId and CityId, and the table that causes the path ambiguity is far away, then it gets a bit more confusing.

Or if, for solution 2, we are positioned on CityTour, the same happens. It is no longer so easy to understand what these subtypes are doing there.

On the other hand, if subtypes are created in the table with the ambiguity, defining one of the foreign keys as a subtype, it is very clear. It is enough to look at its extended table to find the table that can be reached through several paths.

Indirect Multiple references
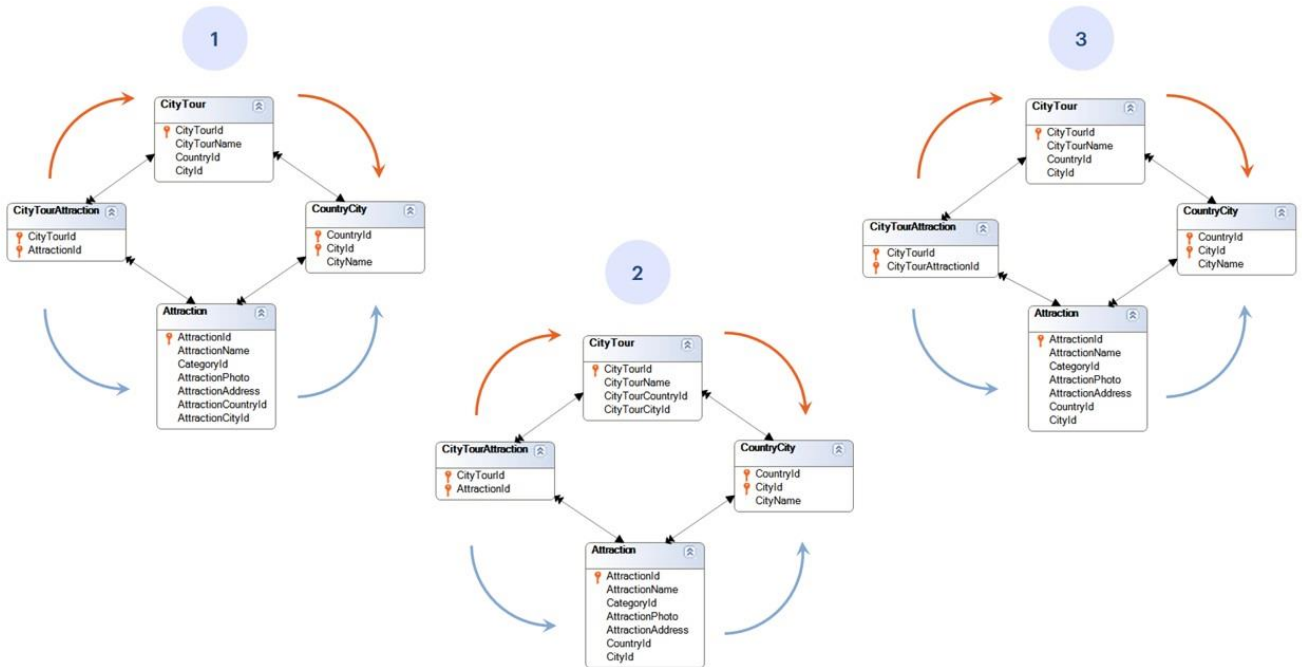
This is the third solution. The disadvantage of this solution is that in any object where we must use attributes that are obtained from the foreign key subtype, we must add them, of course, to the subtype group. There they will be marked as inferred, but in addition, we must add them to the transaction structure.

And, of course, it is not clear at first glance that the ambiguity is made by Country and City. If we **don't** analyze the extended table we **don't** know by which of the attributes of the subtype group it was defined. It could have been by CategoryId, for example.
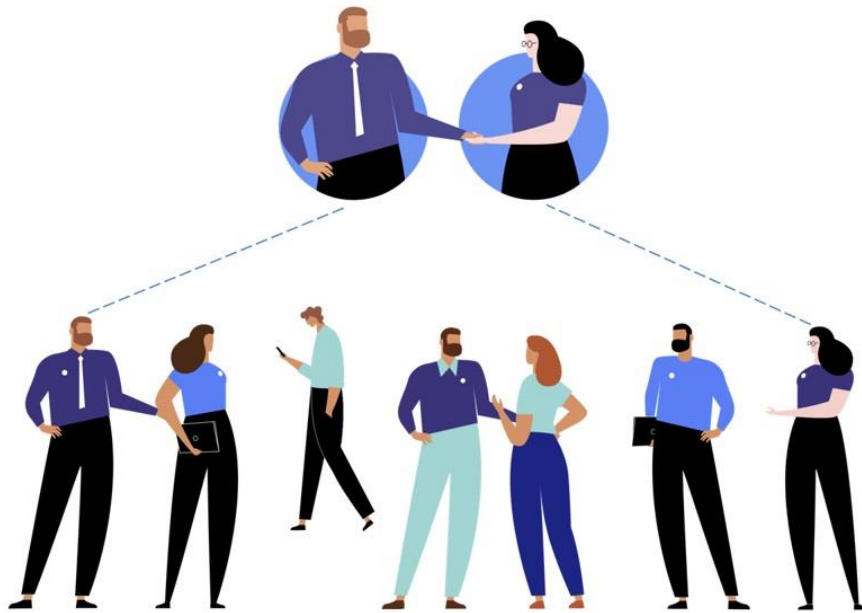
Here we have presented these three solutions. The developer will choose the most suitable one according to his/her reality.
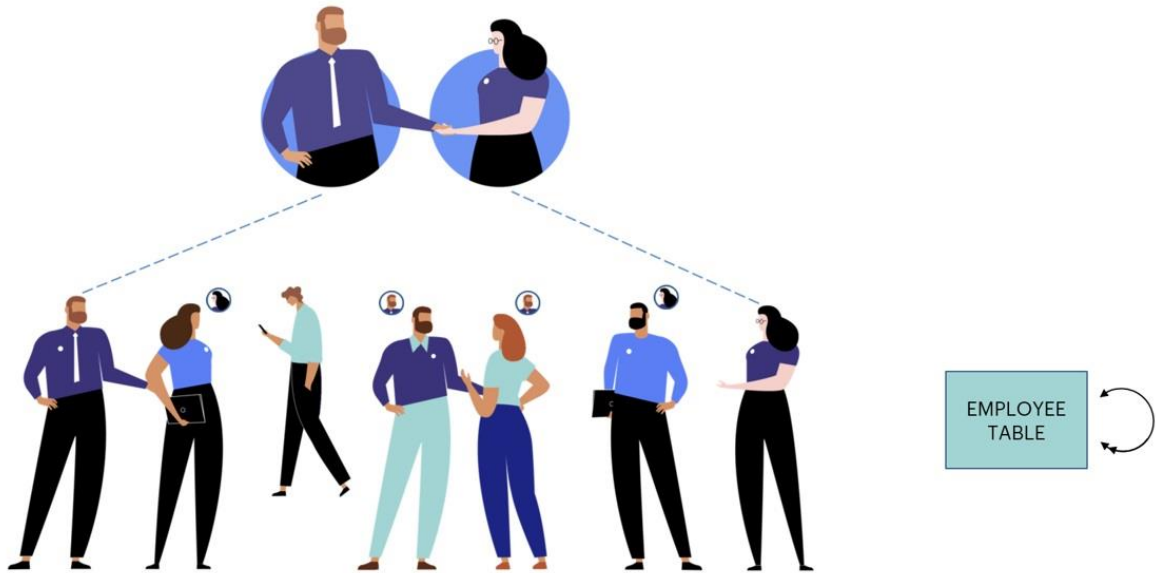
Recursive subtypes
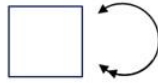
Now let's look at another use case of subtypes, which we call recursive subtypes, where we have an entity that must be self-referential.

To study this case, let's suppose we are representing the information of the travel agency employees. Each employee may, in turn, be a manager of one or more other employees.

Of all employees who have a manager, it is necessary to indicate who that manager is.
This manager is, in particular, an employee. Therefore, a relationship is established in the employee table with itself.
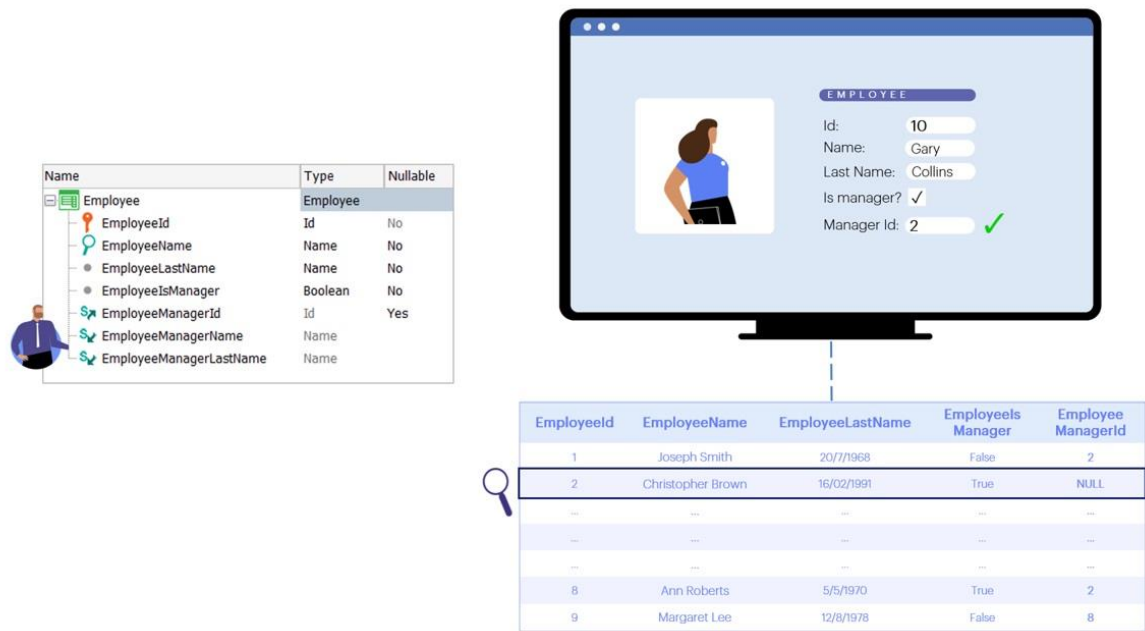
To solve this, we must create a subtype group that represents the information of the employee's manager.

The EmployeeManagerId attribute will be, for all purposes, taken as an EmployeeId, and for this reason, it will form a foreign key to the Employee table itself.

So, when entering the information of an employee through the transaction, when the user chooses a value for the EmployeeManagerId field, GeneXus will check the referential integrity; that is, it will check that there is a record in the employee table with that value for the EmployeeId attribute.

We encourage you to think of real situations in which it is necessary to use the different use cases of subtypes we have seen and to carry out the implementation in GeneXus.

GeneXus™