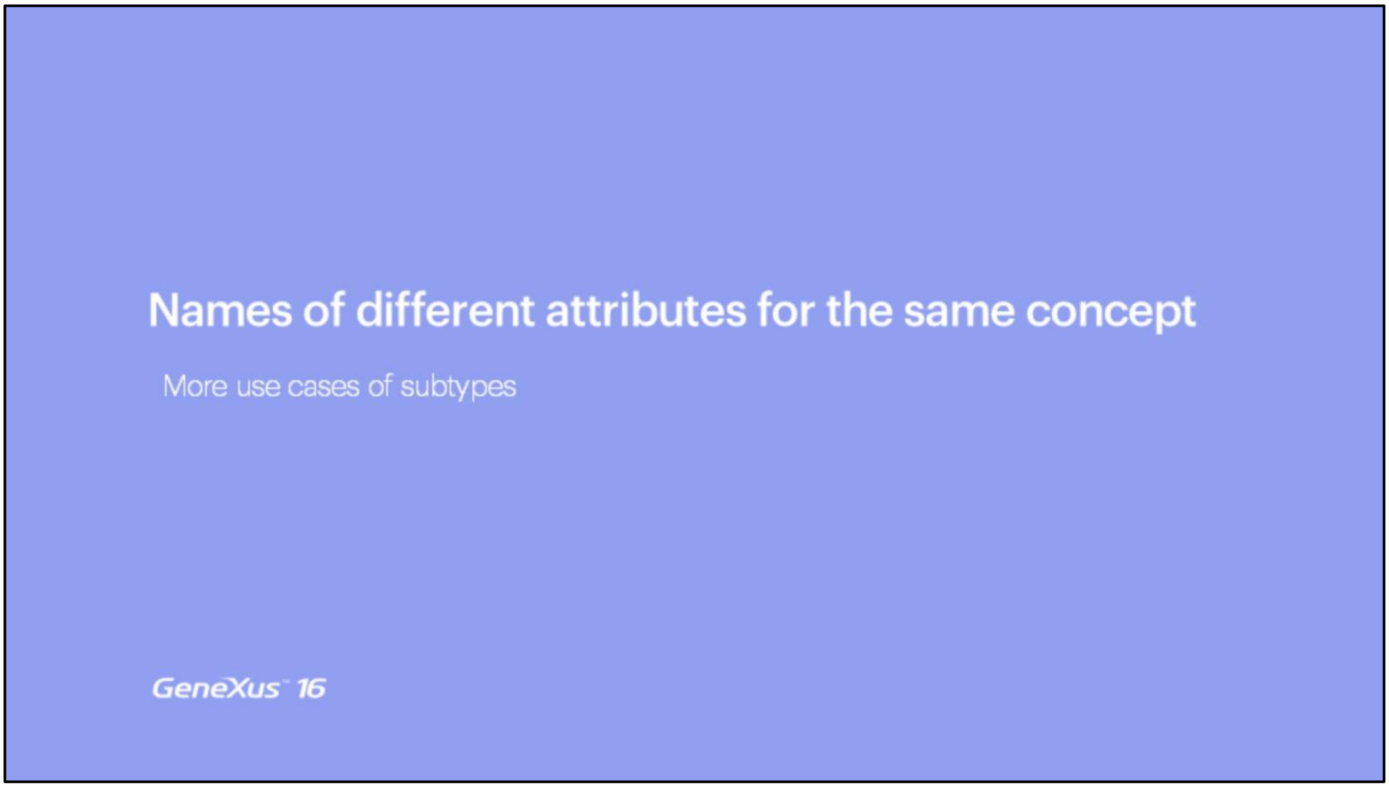
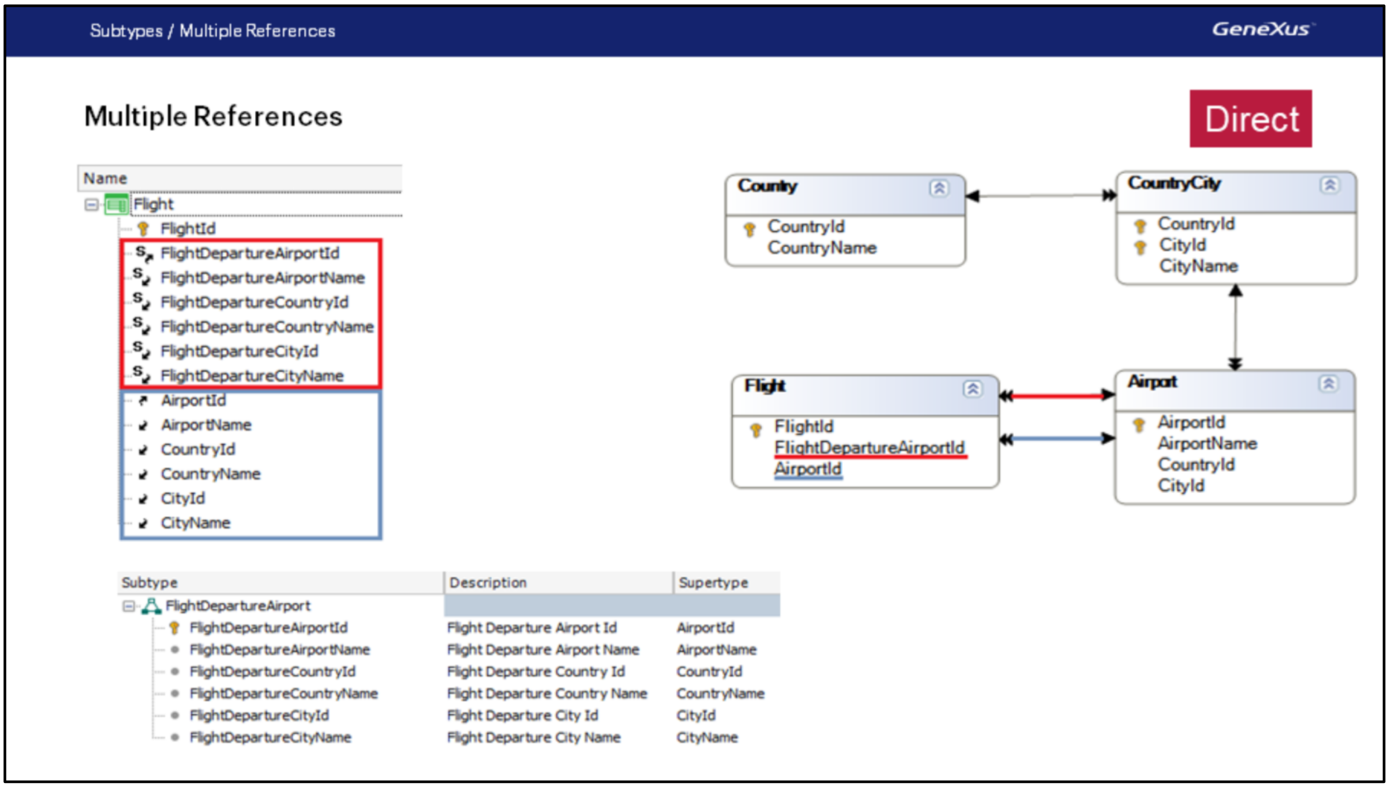


Names of different attributes for the same concept

More use cases of subtypes

GeneXus 16

Multiple References



Multiple References

Direct

Name
- Flight
  - FlightId
  - FlightDepartureAirportId
  - FlightDepartureAirportName
  - FlightDepartureCountryId
  - FlightDepartureCountryName
  - FlightDepartureCityId
  - FlightDepartureCityName
  - AirportId
  - AirportName
  - CountryId
  - CountryName
  - CityId
  - CityName

Subtype	Description	Supertype
FlightDepartureAirport		
FlightDepartureAirportId	Flight Departure Airport Id	AirportId
FlightDepartureAirportName	Flight Departure Airport Name	AirportName
FlightDepartureCountryId	Flight Departure Country Id	CountryId
FlightDepartureCountryName	Flight Departure Country Name	CountryName
FlightDepartureCityId	Flight Departure City Id	CityId
FlightDepartureCityName	Flight Departure City Name	CityName

We saw a case where we needed to define a group of subtypes because, in a transaction, we had a **double reference** to the same actor from reality. It was the case of the Flight transaction, where we had a departure airport and an arrival airport. We could not include the same attribute -AirportId- two times in the structure of the transaction. And that's why we decided to leave that attribute for the arrival airport role and defined a subtype of AirportId which we called FlightDepartureAirportId in order to identify the departure airport (we defined it within a group that we called "FlightDepartureAirport").

Since we wanted to infer the country and city of that airport, in the "FlightDepartureAirport" group we also defined subtypes of the attributes corresponding to the country and city, in addition to a subtype of the airport name.

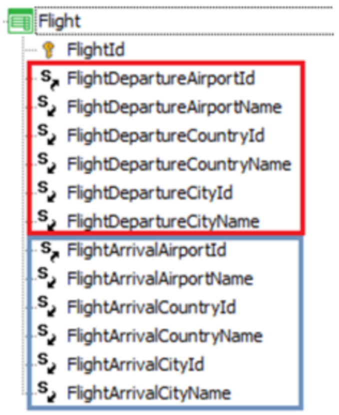
Therefore, when in the Flight transaction we name FlightDepartureCountryName, we know that it will be a CountryName inferred through the departure airport FlightDepartureAirportId. These subtypes were defined within the same group, so an association and relation have been established between them.

And when in the Flight transaction we name CountryName, we know that it will be inferred through the AirporId attribute.

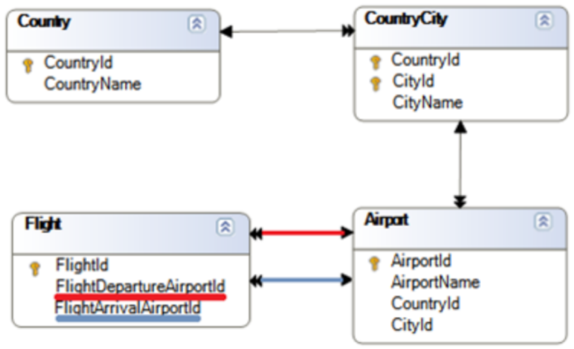
So, there are no ambiguities. We have two perfectly differentiated ways to reach Country, from Flight.

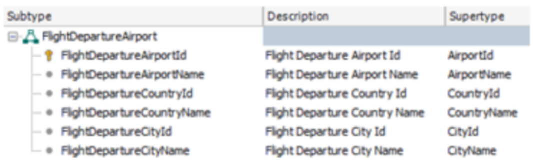
Multiple References

Direct

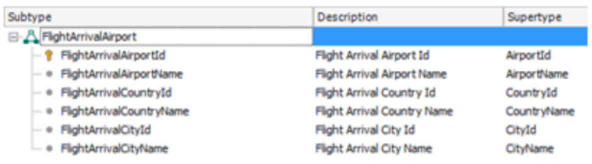


```
Flight
  FlightId
  FlightDepartureAirportId
  FlightDepartureAirportName
  FlightDepartureCountryId
  FlightDepartureCountryName
  FlightDepartureCityId
  FlightDepartureCityName
  FlightArrivalAirportId
  FlightArrivalAirportName
  FlightArrivalCountryId
  FlightArrivalCountryName
  FlightArrivalCityId
  FlightArrivalCityName
```



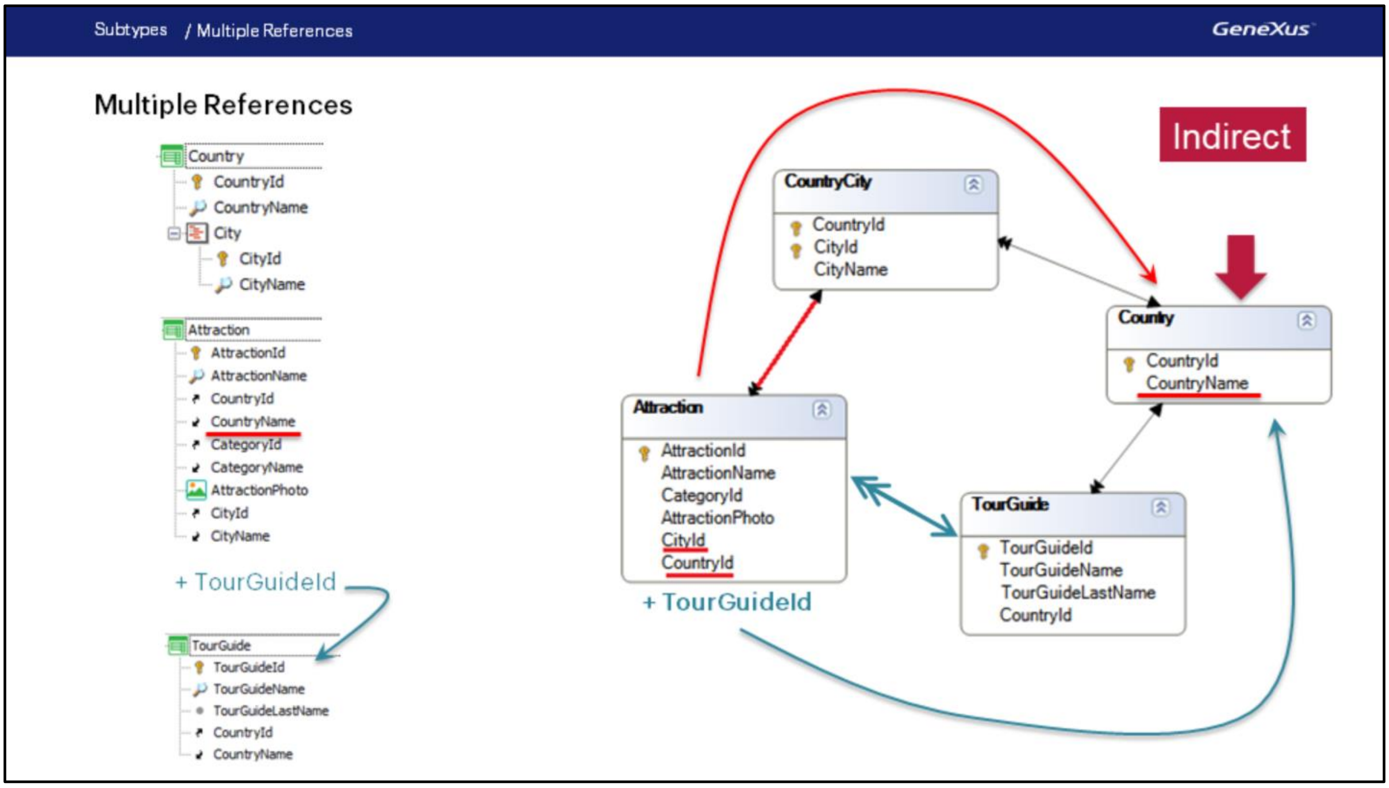


Subtype	Description	Supertype
FlightDepartureAirport		
FlightDepartureAirportId	Flight Departure Airport Id	AirportId
FlightDepartureAirportName	Flight Departure Airport Name	AirportName
FlightDepartureCountryId	Flight Departure Country Id	CountryId
FlightDepartureCountryName	Flight Departure Country Name	CountryName
FlightDepartureCityId	Flight Departure City Id	CityId
FlightDepartureCityName	Flight Departure City Name	CityName



Subtype	Description	Supertype
FlightArrivalAirport		
FlightArrivalAirportId	Flight Arrival Airport Id	AirportId
FlightArrivalAirportName	Flight Arrival Airport Name	AirportName
FlightArrivalCountryId	Flight Arrival Country Id	CountryId
FlightArrivalCountryName	Flight Arrival Country Name	CountryName
FlightArrivalCityId	Flight Arrival City Id	CityId
FlightArrivalCityName	Flight Arrival City Name	CityName

Another option was to also define a group of subtypes for the arrival airport role. The data model will reflect the same relations as in the previous solutions.



In the previous case we had a double reference from one table to another table that was directly related to it. We will now see the case of an indirect relation.

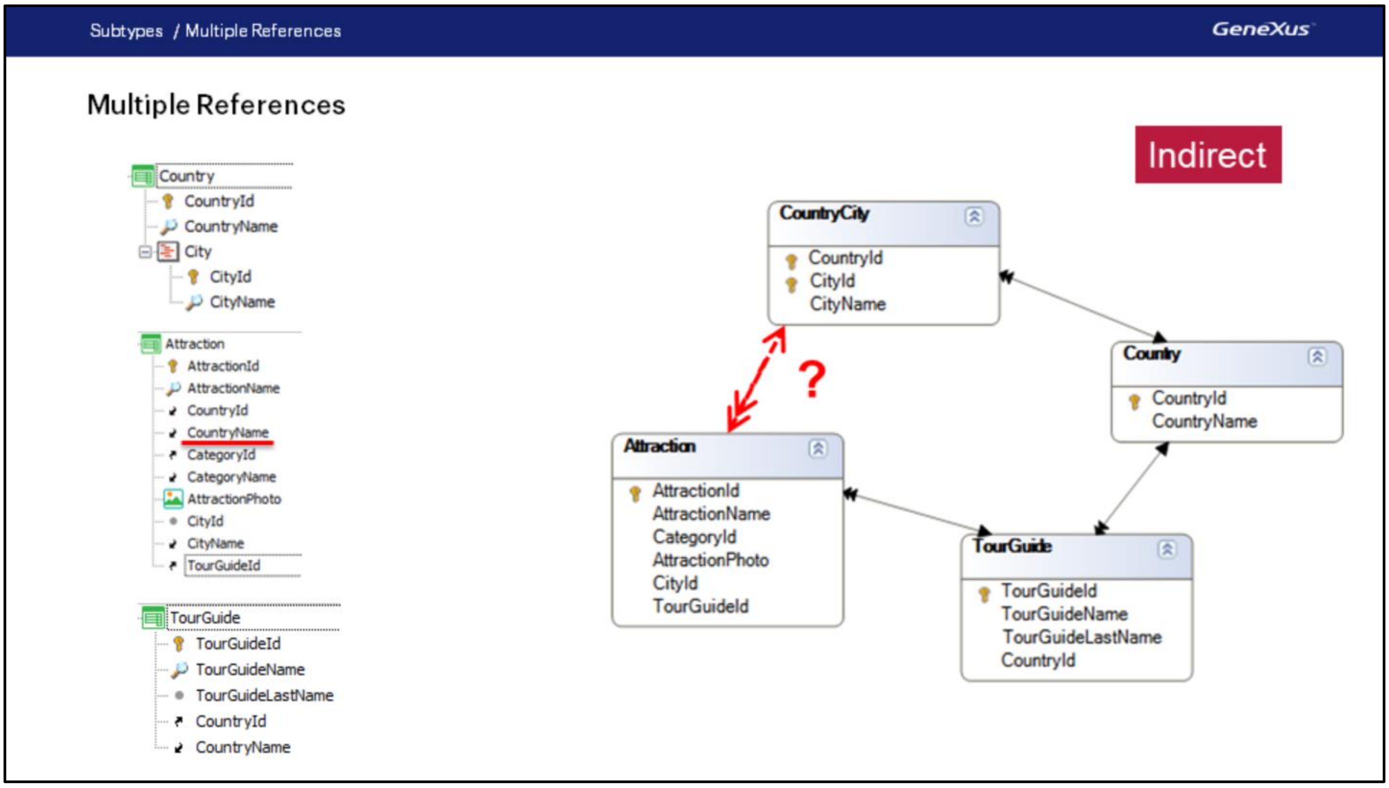
Imagine that we add a transaction to record the tourist guides' information. Each guide has a specific nationality, so we have added the CountryId attribute to the corresponding transaction structure.

In the table diagram we can see that, from the Attraction transaction, we may infer the CountryName corresponding to that transactions because it is in the extended table. Likewise, from TourGuide we also infer its own CountryName, that is: the guide's country.

If we now relate the Attraction and TourGuide entities by adding the TourGuideId attribute to the first one –the guide's identifier– representing that a tourist attraction will only have one guide assigned, then how will the tables be related?

TourGuideId will be a foreign key in Attraction to the TourGuide table, so the relation will appear marked with a blue arrow. This may lead us to think that now, from Attraction, there are two ways for inferring CountryName, and that means that there is an ambiguity. In other words, since GeneXus has the CountryName attribute in Attraction, where does it infer it from?, from the city of the attraction or from the attraction's tourist guide? If the two values match, this will not matter. But in this case, they will not necessarily match. The country where the attraction is located will not necessarily match the country of origin of the tourist guide.

We will then need subtypes in order to differentiate the two roles of CountryName.

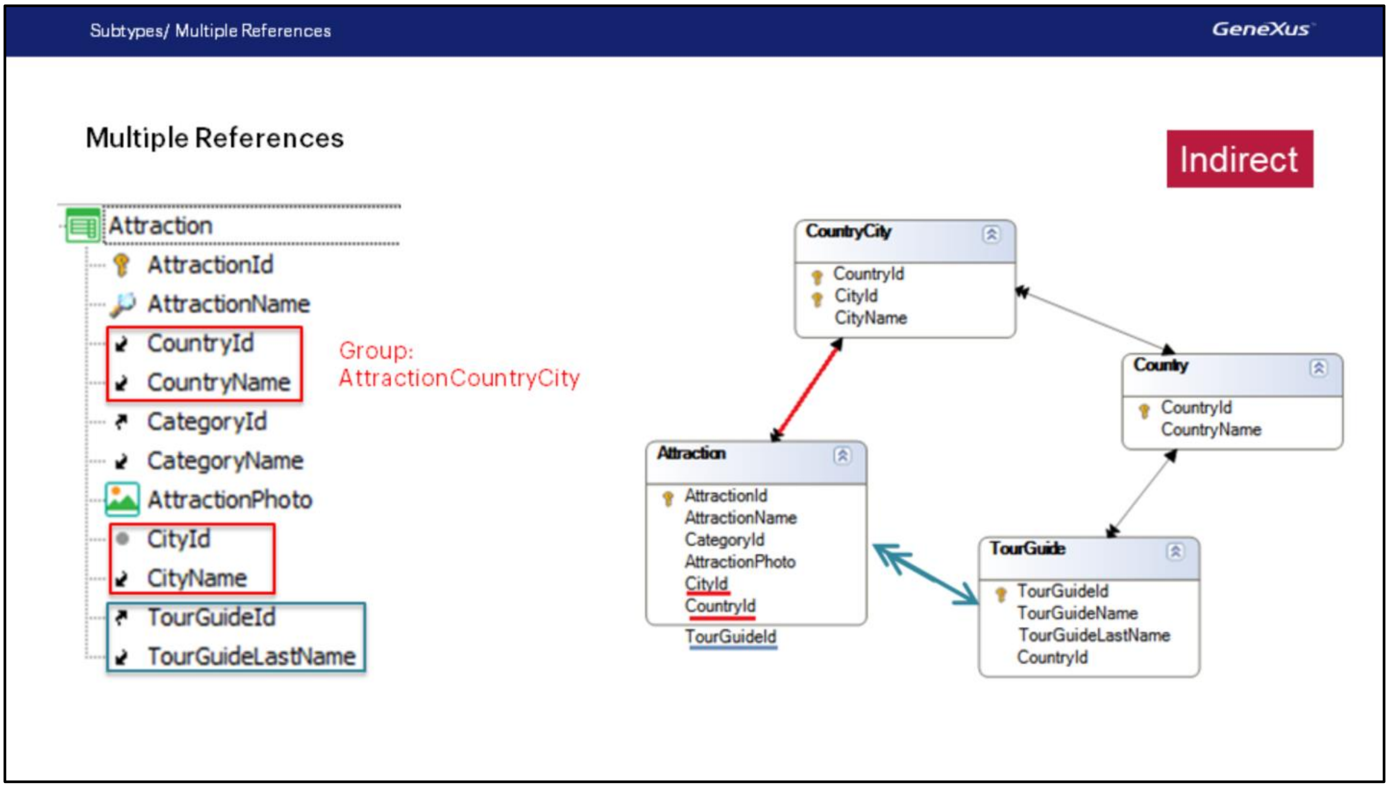


But in this case we will need them, not only to distinguish the roles. If we look at the table diagram after the TourGuideId attribute has been added to Attraction, we will see that the relation between Attraction and CountryCity has disappeared, meaning that CountryId is no longer a foreign key in Attraction. Note that it is not in the table anymore, for it will be an inferred attribute. But, inferred from what? From TourGuideId!

We should recall that, in the first place, GeneXus normalizes tables. This means that, based on the names of attributes, along with the identifiers, it determines which attribute is placed in each table, as well as the relations between the tables. Since TourGuideId –which is identifier of TourGuide– appears in Attraction, and, in turn, CountryId –which is identifier of Country– appears in TourGuide, it is then understood that upon a given TourGuideId in Attraction, the CountryName is inferred from it, passing through the TourGuide intermediate table.

So, with this transaction design, **we have lost** the possibility of indicating which is **the country of the attraction.** The CountryName will be that of the tourist guide.

We have no choice but to use subtypes in order for Attraction to have its own country, regardless of the tourist guide's country.

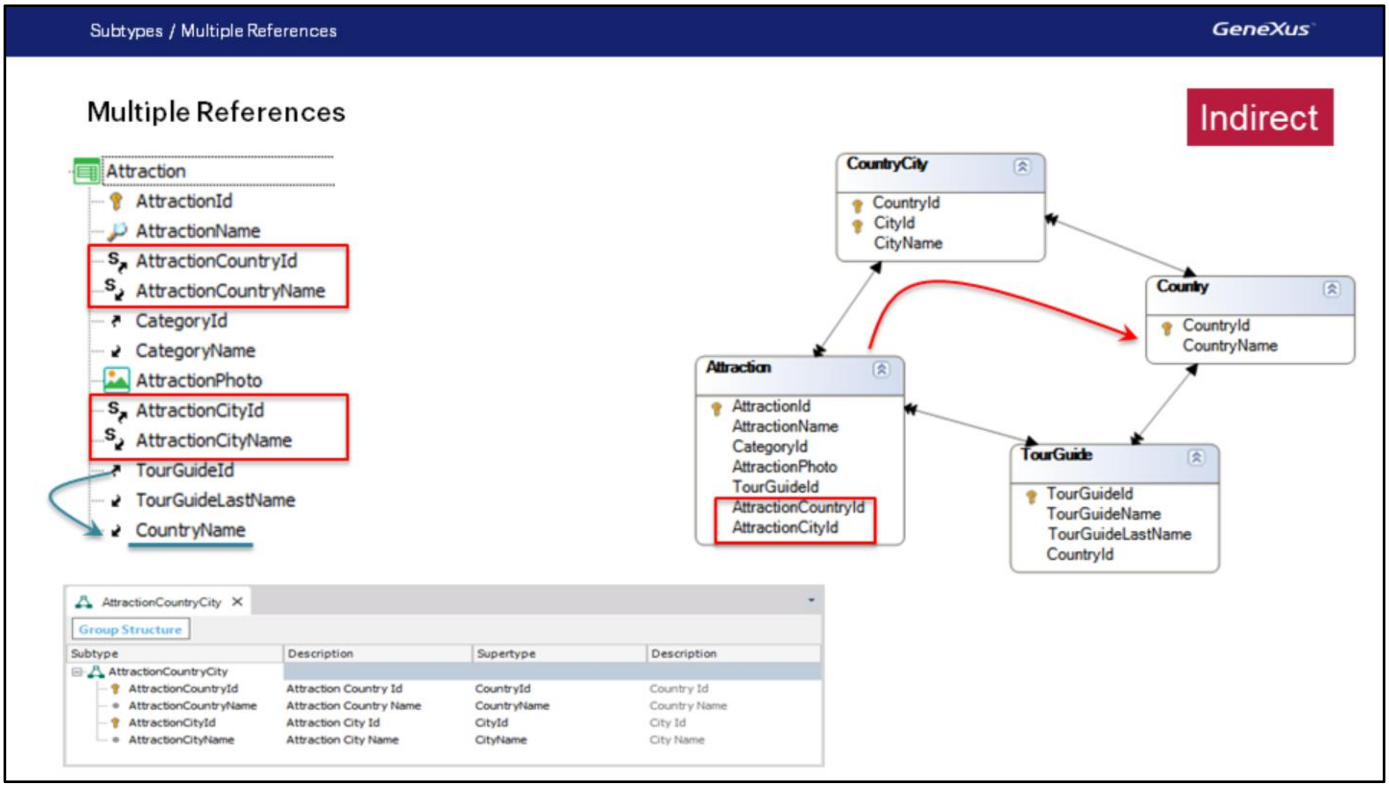


GeneXus

Multiple References

Indirect

Again, we are facing several alternatives. The most evident one is to set up a group of subtypes for the attraction country/city.



Multiple References

```
Attraction
  AttractionId
  AttractionName
  AttractionCountryId
  AttractionCountryName
  CategoryId
  CategoryName
  AttractionPhoto
  AttractionCityId
  AttractionCityName
  TourGuideId
  TourGuideLastName
  CountryName
```

AttractionCountryCity

Group Structure

Subtype	Description	Supertype	Description
AttractionCountryCity			
AttractionCountryId	Attraction Country Id	CountryId	Country Id
AttractionCountryName	Attraction Country Name	CountryName	Country Name
AttractionCityId	Attraction City Id	CityId	City Id
AttractionCityName	Attraction City Name	CityName	City Name

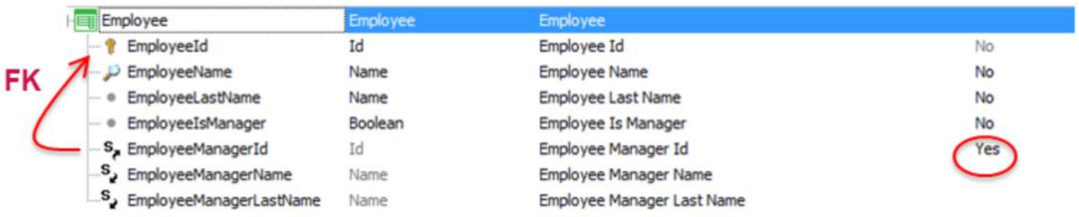
Here we have defined a group of subtypes to represent the country and city of the attraction. Note that now GeneXus will correctly represent the relations, in addition to inferring the CountryName attribute from TourGuideId with no ambiguity. The attribute that represented by and in which the country of the attraction is inferred will be the one called AttractionCountryName, subtype of CountryName, which belongs to the AttractionCountryCity group.

Also note that this group has two primary attributes: AttractionCountryId and AttractionCityId, which correspond to the primary key of the CountryCity table, due to the subtypes we indicated: {CountryId, CityId}.

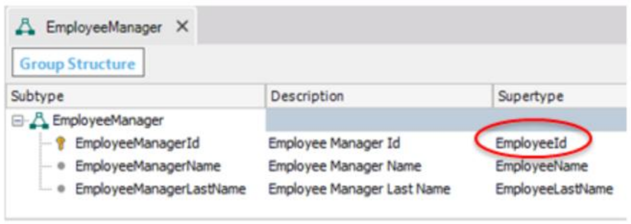
Recursive Subtypes

Recursive Subtypes

- When auto-referencing from an entity is required:



Employee	Employee	Employee	
EmployeeId	Id	Employee Id	No
EmployeeName	Name	Employee Name	No
EmployeeLastName	Name	Employee Last Name	No
EmployeeIsManager	Boolean	Employee Is Manager	No
EmployeeManagerId	Id	Employee Manager Id	Yes
EmployeeManagerName	Name	Employee Manager Name	
EmployeeManagerLastName	Name	Employee Manager Last Name	



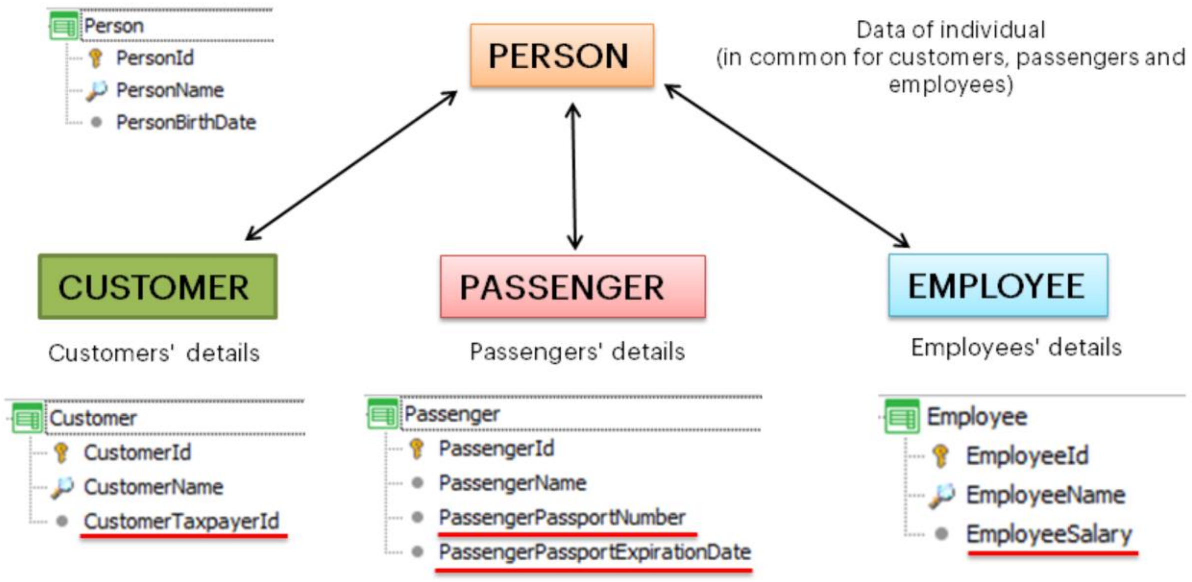
Subtype	Description	Supertype
EmployeeManager		
EmployeeManagerId	Employee Manager Id	EmployeeId
EmployeeManagerName	Employee Manager Name	EmployeeName
EmployeeManagerLastName	Employee Manager Last Name	EmployeeLastName

In our example, we are representing the data on the travel agency's employees. Each employee may also be the manager to another employee or employees. From all employees who have a manager, we must indicate who the manager is. Specifically, the manager is also an employee. So, there is a relation between the employees' table with itself. To do this we must create a group of subtypes to represent the data relative to the employees' manager.

The EmployeeManagerId attribute will then be taken, to all effects, as an EmployeeId. Therefore, it will constitute a foreign key for the Employee table itself. So, when we enter data on one employee through the transaction, upon the user's selection of a value for the EmployeeManagerId field, GeneXus will control referential integrity. This means that it will control the existence of a record in the employees' table, with that value for the EmployeeId attribute.

Specialization

Specialization



What do we mean by **Specialization**?

Suppose that the travel agency needs to handle specific information about its customers who acquire travel tickets and tour packages (for instance, the customer's Taxpayers' Registry Number, if any), in addition to having to record particular information on the passengers (such as passport numbers), as well as specific data on the agency's employees such as their salaries.

In other words, the travel agency will issue invoices to customers, it will record seats on flights assigned to passengers, and it will issue wage receipts for employees.

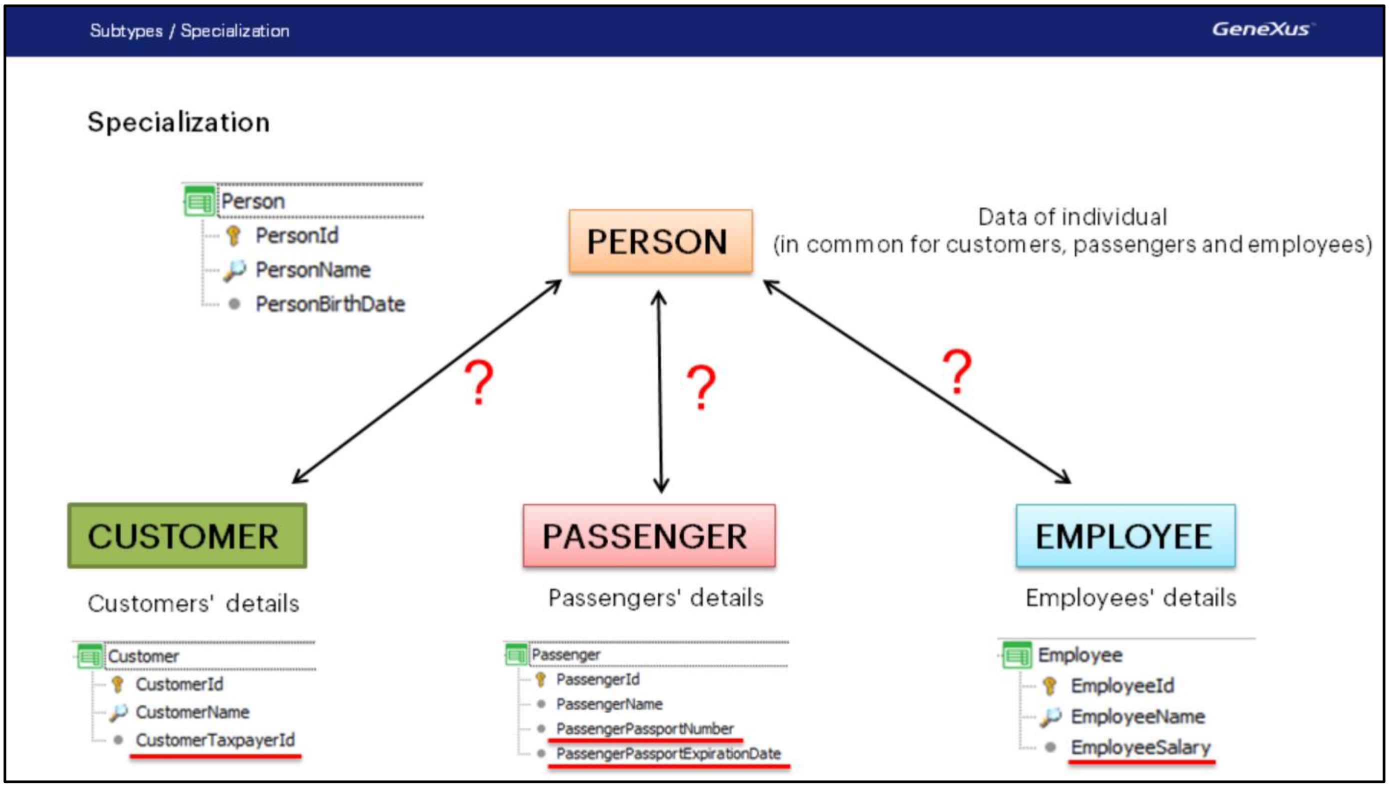
Instead of what previously was just a transaction –Customer–, we could now define what is proposed here above. Customers, as well as passengers and employees, are all persons, to they will have certain information in common such as name, birthdate, etc.

And Customer is a Specialization of Person, just like Passenger and Employee.

We could read the following relation:

- Each Customer **is** Person
- Each Passenger **is** Person
- Each Employee **is** Person

Note that the proposal includes 4 Person transactions with data common to all persons. And then we have the specializations: Customer, Passenger and Employee, each with its specific data (Customer will have the Taxpayer number, while Passenger will have passport number and Employee a salary).

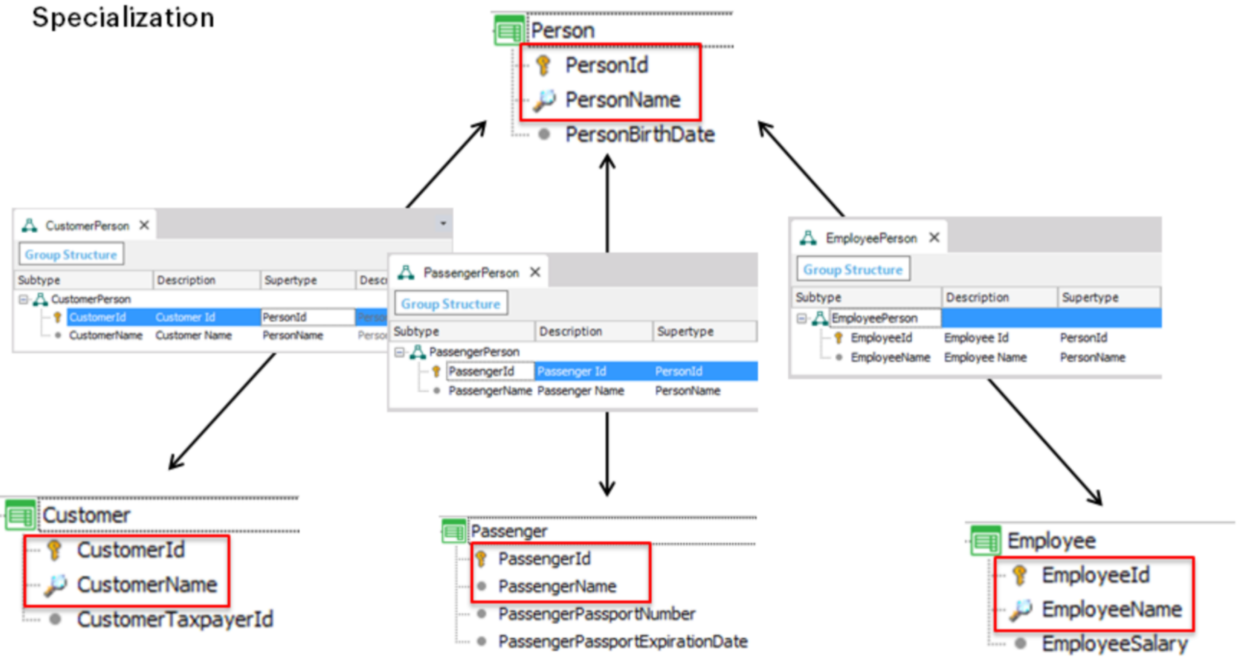


We want the customer identifier to **exactly match** the identifier of a person, to reflect that the <u>customer</u> **is** <u>a person</u>. This means that if the name of the person with ID 8 is Ann Roberts, and she was born on 05/05/1970, when we enter her information as customer, the user must be able to type in ID 8 into the Customer transaction, and the name Ann Roberts must be shown upon exiting the field, to then enter the Taxpayer ID number (CustomerTaxpayerId). Likewise, when we execute the Passenger transaction, we will want that, upon typing in value 8 for PassengerId, Ann Roberts is to be inferred in PassengerName, and the user must be able to assign the passport number and the expiry date in the corresponding attributes (PassengerPassportNumber and PassengerPassportExpirationDate). And the same applies to the case of employees.

If we just define Passenger and Employee as the primary keys of Customer, and the CustomerId, PassengerId and EmployeeId attributes respectively without relating them in any way to PersonId (and the same with CustomerName, PassengerName and EmployeeName, without relating them to PersonName), then we will not reach our objective. To GeneXus, they will all be fully independent transactions.

GeneXus

Specialization



CustomerPerson

Subtype	Description	Supertype	Desc
CustomerPerson			
CustomerId	Customer Id	PersonId	Perso
CustomerName	Customer Name	PersonName	Perso

PassengerPerson

Subtype	Description	Supertype
PassengerPerson		
PassengerId	Passenger Id	PersonId
PassengerName	Passenger Name	PersonName

EmployeePerson

Subtype	Description	Supertype
EmployeePerson		
EmployeeId	Employee Id	PersonId
EmployeeName	Employee Name	PersonName

But we will define groups of subtypes to represent that a customer must be a valid person (that is: a person that has been previously recorded), that a passenger must be a person, and that an employee must be a person as well.

We create a group called CustomerPerson, where we define CustomerId and CustomerName as subtypes, respectively, of PersonId and PersonName.

A second group called PassengerPerson, where we define PassengerId and PassengerName as subtypes, respectively, of PersonId and PersonName.

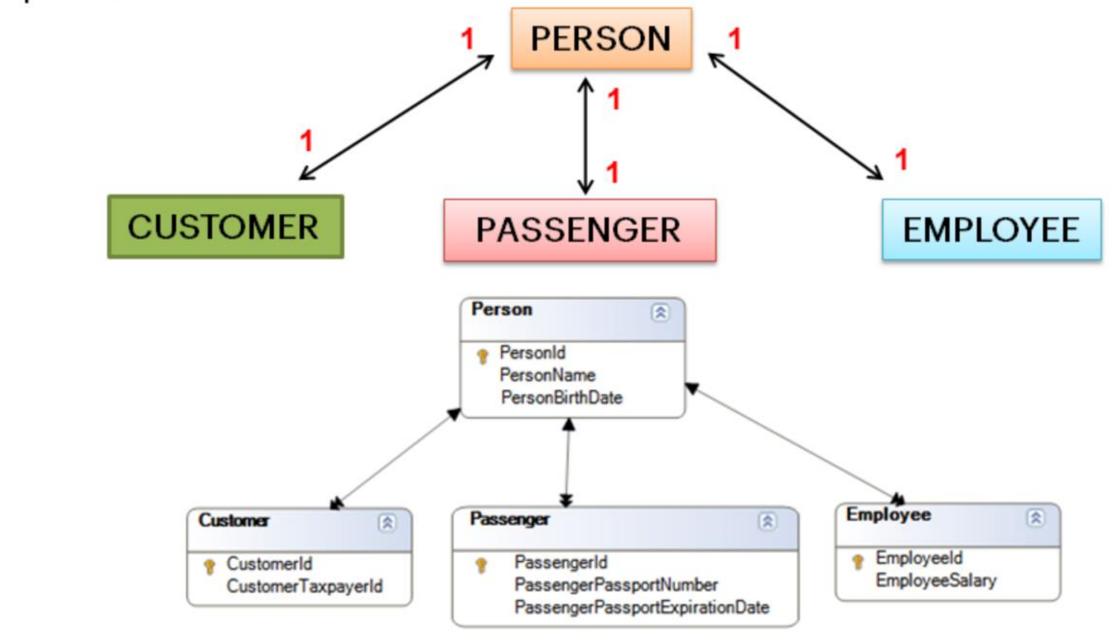
And a group called EmployeePerson, where we define EmployeeId and EmployeeName as subtypes, respectively, of PersonId and PersonName.

When we do this, the CustomerId, PassengerId and EmployeeId attributes, besides being the respective identifiers of the Customer, Passenger and Employee tables, and thus their primary keys, they will also be foreign keys to the Person table.

Therefore, when the user enters a value in the ID of any of the three transactions (Customer, Passenger or Employee), the search will be made in the Person table for a record whose ID is that same value.

Likewise, when we want to eliminate a person, we will control, through the Person transaction, that there are no records in Customer where CustomerId = PersonId for the person we want to eliminate, and the same for records in Passenger where PassengerId = PersonId, nor records in Employee where EmployeeId = PersonId. If any of these three records exist, then the deletion of that person will not be allowed.

Specialization



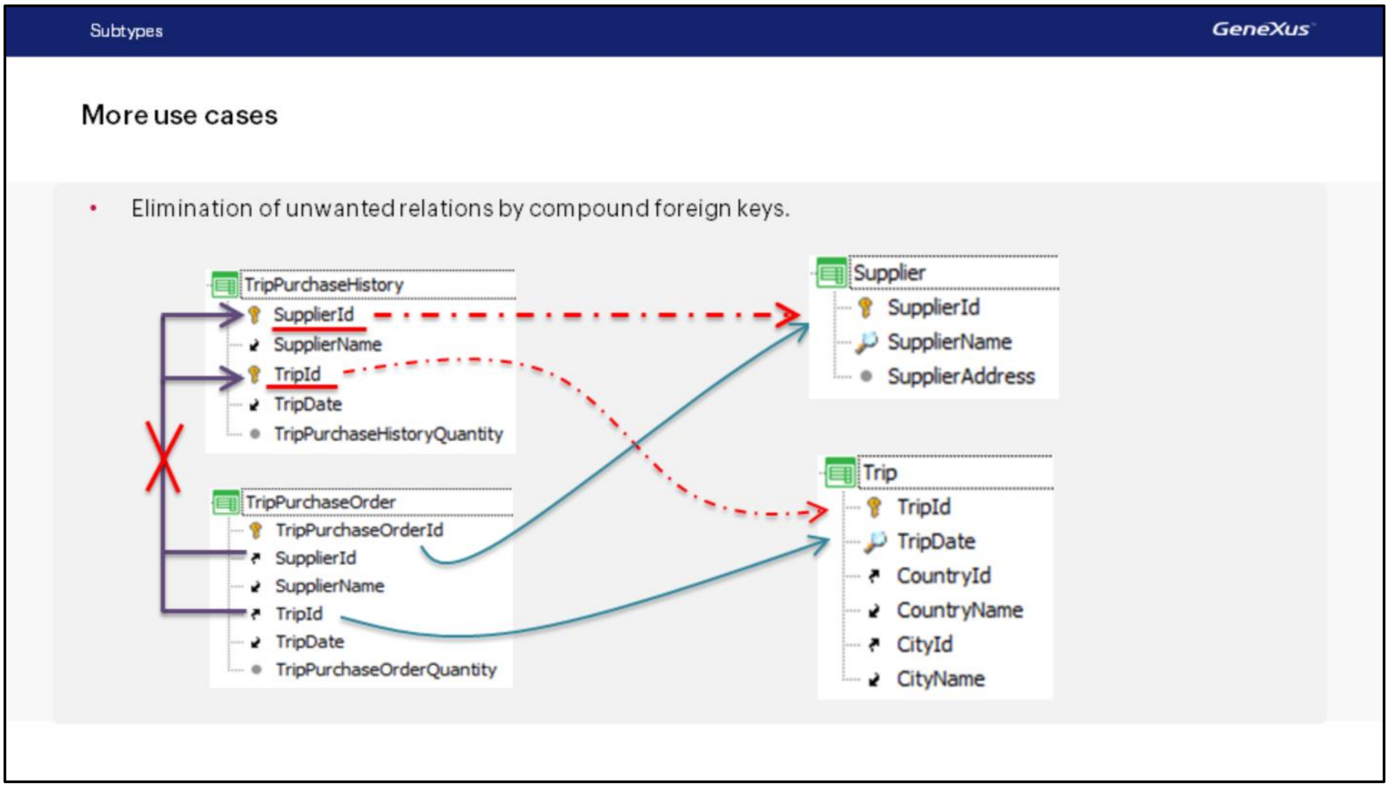
Note that this design represents 1 to 1 relations between the general table and the one corresponding to each specialization.

This means that a person may be recorded only once as customer, because CustomerId is a valid PersonId and it is also primary key. Likewise, a person may be recorded only once as passenger and only once as employee.

A person may have the three roles, or be recorded only as person with no additional data as agency customer or employee or passenger.

Think about the structure of the CUSTOMER, PASSENGER and EMPLOYEE tables. Obviously, CustomerName, PassengerName, and EmployeeName will be attributes inferred (that will not be physically in the respective CUSTOMER, PASSENGER and EMPLOYEE physical tables).

Note: the GeneXus diagrams do not show the 1 to 1 relations. That is why we can see the double arrow on the specialized tables side. With these arrows, GeneXus is just indicating the relations of foreign keys.



GeneXus

More use cases

- Elimination of unwanted relations by compound foreign keys.

And there are more cases of use. For instance, when we have a transaction to record the purchase orders of trips from suppliers of the travel agency, and another one to record the history of purchases by supplier/trip, we will see that, among the foreign keys, one will be defined for relating the purchase order with the historic purchases. However, we do not want it to be generated, because when the first purchase order of a supplier/trip is created, there will be no records in the history.

A way to avoid that relation from being controlled is by modifying the name of some of the attributes in the primary key of TripPurchaseHistory. This implies the definition of subtypes.

But this case will not be studied in detail in this course.

GeneXus™

The power of doing.

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