The Need for a Logical Data Model

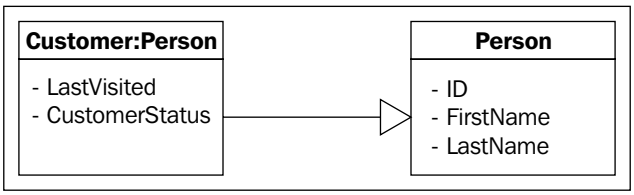
A logical model is very helpful for the following reasons:

- **It helps in understanding project requirements**: A logical data model translates business requirements into simple entity relationship models, making it easy to identify and understand the business logic. Sometimes business requirements can be complex, making it difficult for the developer to understand a system's internal logic. Using a logical data model helps to create a visual representation of the system, from a high level.
- **It helps in creating the actual database**: The logical data model is the blueprint of the actual physical database that needs to be created for the application. Most beginner developers start creating physical tables in the database while starting to work on a project, which can lead to problems, as it might not be possible to foresee all of the physical tables and the relationships between them right at the start of the project. A logical data model helps in mitigating risks by laying out the entity relationships and attributes before the actual physical data modeling starts.
- **It is database independent**: The same logical data model can be used for different databases, as a logical model is independent of the physical database. This saves time when developing applications targeting multiple databases, which helps to reduce development time and cost.

The Domain Model Versus the Logical Data Model

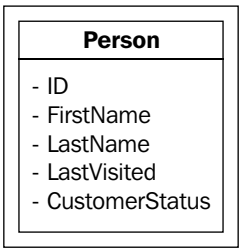
Many developers are unclear of the difference between the domain model and the logical data model, sometimes mistaking one with the other. Even though a logical model might look similar to a domain model in terms of depicting primary entities of a system, there are fundamental differences between the two. A logical data model is more focused on the structure of the data in the entities and on the relationships between different entities, whereas a domain model is focused on encapsulating the entities from an object-oriented perspective. So a logical data model is purely relational in nature, but an object domain model is richer, depicting inheritance, associations, aggregations, and so on, between the different entities involved.

In our object model, there will be some entities that will be the same as in the logical data model, whereas some might not be present in the logical data model at all. For example, if we create a parent class for a `Customer` called `Person`, then in the object model, the entities would look like this:



Here, the `Customer` class inherits the `Person` class, and both are different kinds of objects in the domain model. Even though we can say `Customer` is a type of `Person`, but not every `Person` is a `Customer`. So both of these entities need to be depicted separately in the domain model.

But in the logical data model, we cannot depict inheritance or any other object-oriented feature. In this case, both the `Person` and the `Customer` classes can be persisted in the same table in the database. So there would be a single entity for both of these in the logical data model as shown here:



How, then, do we identify Persons from Customers? One simple way is to add a Boolean attribute: `IsCustomer`. For all `Person` objects who are Customers, this attribute would be true. This is one simple example to show that the relational data model is very different from an object-oriented domain model.

Physical Data Model

Once we have defined the logical data model, we can then create a physical data model for our application. A logical model is closer to the business model whereas a physical model actually mirrors the actual database. In short, the physical data model is the logical data model with: