In the first step, we simplify the types as much as possible and reuse them where feasible. This approach led us to create the Person type, which comprises three other types: Address, Phone, and Name. We use nested tables in the Phone type due to Varray's limitation, which must be declared in its declaration. Specifically, the elements of a Varray cannot be deleted individually.

One alternative to creating the Person class was to combine all the values into one class. Although this alternative makes data access much easier and eliminates the need to traverse several layers to access data such as name or phone number, the problem is that all the data would be in one place, making it disorganized and untidy. Moreover, in practice, the design is to create the Person class with three different classes, namely Name, Phone, and Address. The other reason to declare the Address type was because we were using it in the Branch table. The reason for not creating any more classes is that we cannot use them in at least two different subclasses.

One of the classes we have is the Job type, which includes the Position and Salary attributes. The Position attribute is unique and is used to reference the job from the employee table. In the Job table, the Position is also unique, as we assume that for the same position across all branches, the salary is the same.

An alternative approach would be to include both the Position and Salary attributes directly in the Employee class. However, we chose to use the Job table and reference it through the Employee table in order to reduce the amount of data that needs to be saved in the database and make it easier to manage the tables.

The Branch type includes the bID, Address type, and phone number as a varchar2. In the table, the bID is the primary key, and the phone number is unique. This is because each branch has a unique branch ID and phone number.

The next type is the Account type, where we save the account information in its own table. All the values are the same as in relational modeling, but here the bID is used as a reference to the rows of the Branch table. In the Account table, the accNum is the primary key, and the accType value can only be equal to either "current" or "savings".

The Customer type uses the Person type as the superclass, with CustID as the primary key, and niNum as a unique identifier to prevent duplicate values for customers in the database.

The Employee type is a subclass of Person, with supervisorID, E\_Job, and bID as references to the Employee, Job, and Branch tables, respectively. The empID serves as the primary key. Although we could have combined the three references into one type, this would have increased the complexity of accessing supervisor, job, or branch information.

The purpose of the customer account is to establish a many-to-many relationship between Accounts and Customers. This is achieved using custID as a reference to the Customer and accNum as a reference to the Account. While nested tables could have been used for the many-to-many relationships, this would have added complexity to finding and retrieving the necessary information.

The rationale for using object-relational features in the design is to create a more structured and organized database schema. The use of object types, nested tables, and object tables allows for the creation of more complex and structured data types that can be used to define the various attributes of the entities in the system.

Alternative possible object-relational representations were considered, including using inheritance and subtypes. However, these representations were rejected because they would have made the database schema more complex and harder to manage. The use of object types and nested tables provides a more straightforward and more efficient way to manage the database schema while still capturing the semantics of the application.