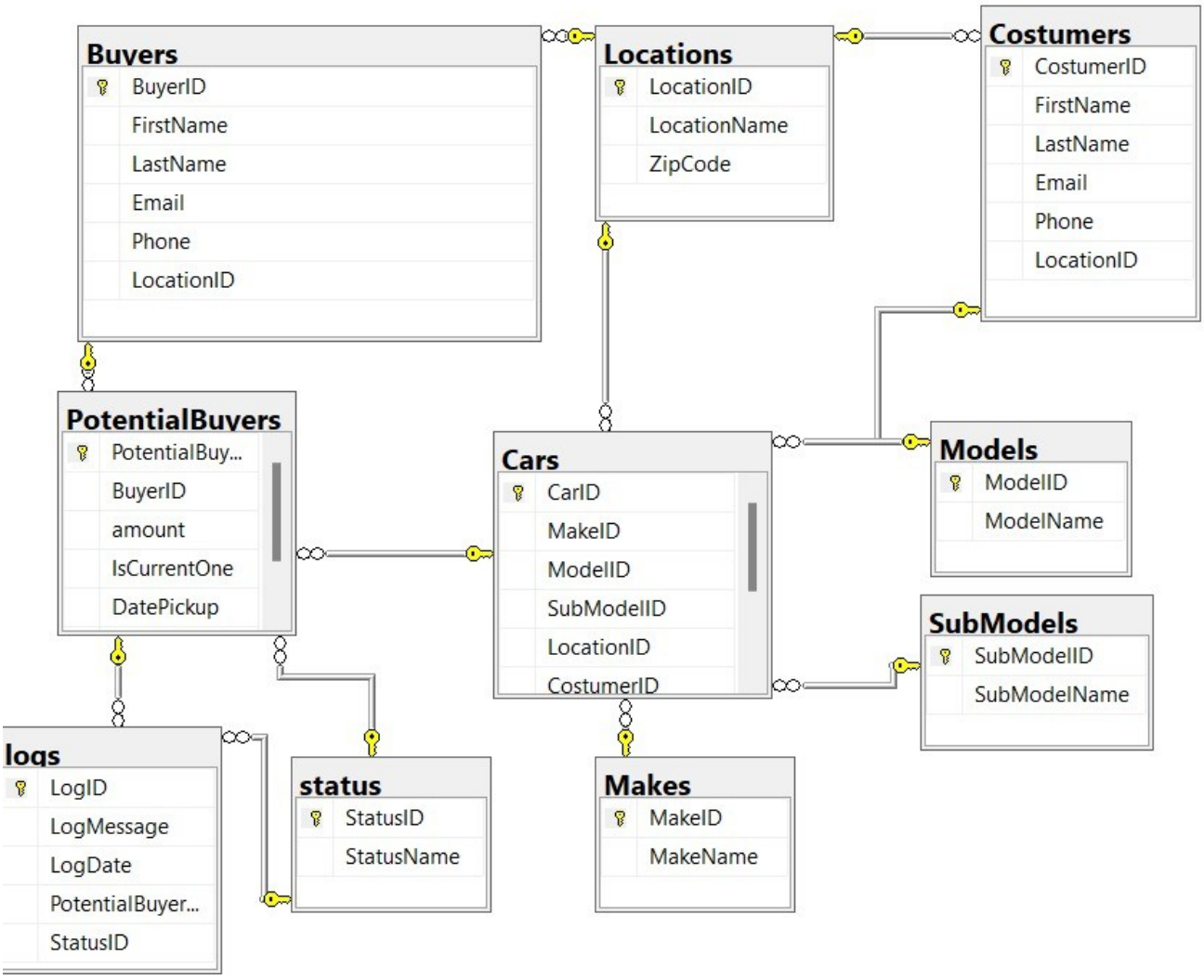
1) **Define the database tables with its corresponding data types and relationships for the following case: A customer wants to sell his car so we need the car information (car year, make, model, and submodel), the location of the car (zip code) and the list of buyers that are willing to buy the car in that zip code. Each buyer will have a quote (amount) and only one buyer will be marked as the current one (not necessarily the one with the highest quote). We also want to track the progress of the case using different statuses (Pending Acceptance, Accepted, Picked Up, etc.) and considering that we care about the current status and the status history (previous statuses indicate when it happened, who changed it, etc.). The status "Picked Up" has a mandatory status date but the rest of the statuses don't.**
Write a SQL query to show the car information, current buyer name with its quote and current status name with its status date. Do the same thing using Entity Framework. Make sure your queries don't have any unnecessary data.



QUERY:

```
SELECT
        B.FirstName + ' ' +B.LastName AS BuyerName,
        PB.amount AS Amount,
        S.StatusName,
        C.RegistrationNumber,
        M.ModelName,
        MK.MakeName
FROM PotentialBuyers AS PB
INNER JOIN Buyers AS B ON PB.BuyerID = B.BuyerID
INNER JOIN Status AS S ON PB.StatusID = S.StatusID
INNER JOIN Cars AS C ON PB.CarID = C.CarID
INNER JOIN Models AS M ON C.ModelID = M.ModelID
INNER JOIN Makes AS MK ON C.MakeID = MK.MakeID
```

Entity-Framework

```
public List<ResultPotentialBuyer> GetBuyerDetails()
{
   var detailBuyerQuery = carSalesContext.PotentialBuyers
      .Join(carSalesContext.Buyers, pb => pb.BuyerId, b => b.BuyerId, (pb, b) => new { pb, b })
      .Join(carSalesContext.Statuses, temp => temp.pb.StatusId, s => s.StatusId, (temp, s) => new { temp.pb, temp.b, s })
      .Join(carSalesContext.Cars, temp => temp.pb.CarId, c => c.CarId, (temp, c) => new { temp.pb, temp.b, temp.s, c })
      .Join(carSalesContext.Models, temp => temp.c.ModelId, m => m.ModelId, (temp, m) => new { temp.pb, temp.b,
temp.s, temp.c, m })
      .Join(carSalesContext.Makes, temp => temp.c.MakeId, mk => mk.MakeId, (temp, mk) => new ResultPotentialBuyer
      {
         BuyerName = temp.b.FirstName + " " + temp.b.LastName,
         Amount = temp.pb.Amount,
         StatusName = temp.s.StatusName,
         RegistrationNumber = temp.c.RegistrationNumber,
         ModelName = temp.m.ModelName,
         MakeName = mk.MakeName
      });

   return detailBuyerQuery.ToList();
}
```

2) What would you do if you had data that doesn’t change often but it’s used pretty much all the time?

Depending the of issue we can resolve :

- We might use some kind of cache when the data is using regularly in this case it could be information of product.
- If is a data that take a long time to change like a url of API external we can use the file setting to maintain those data.

3) Analyze the following method and make changes to make it better. Explain your changes.

```
public void UpdateCustomersBalanceByInvoices(List<Invoice> invoices)
{
        foreach (var invoice in invoices)
        {
                var customer =
                dbContext.Customers.SingleOrDefault(invoice.CustomerId.Value);
                customer.Balance -= invoice.Total;
                dbContext.SaveChanges();
        }
}
```

```
public void UpdateCustomersBalanceByInvoices(List<Invoice> invoices)
{

    try
    {

                        //get ids the of customers
                        var customerIds = invoices.Select(r => r.CustomerId ).ToList();

                        //Searching all customers by Customer Id so this way optimize the query because the search by id always are fasters
                        var customerToUpdate = dbContext. Customers
                                                .Where(r => customerIds.Contains(r.CustomerId ))
                                                .ToList();

                        //Update each customer but don't save in database until all records are updated
                        foreach (var customer in customerToUpdate)
                        {
                                var totalInvoice = invoices.First(r => r.CustomerId == customer.CustomerId).Total;
                                customer.Balance -= totalInvoice;
                        }

                        //Save all records updated in database
                        dbContext.SaveChanges();

              }
    catch (Exception ex)
    {
       // Handle the exception here
       Console.WriteLine("An error occurred while updating the Customers Balance: " + ex.Message);
    }
}
```

4) Implement the following method using Entity Framework, making sure your query is efficient in all the cases (when all the parameters are set, when some of them are or when none of them are). If a “filter” is not set it means that it will not apply any filtering over that field (no ids provided for customer ids it means we don’t want to filter by customer).

```
public async Task<List<OrderDTO>> GetOrders(DateTime dateFrom, DateTime dateTo, List<int>
customerIds, List<int> statusIds, bool? isActive)
{
   var query = carSalesContext.Orders.AsQueryable();

   // Apply filters based on the provided parameters
   if (dateFrom != default(DateTime))
   {
      query = query.Where(o => o.OrderDate >= dateFrom);
   }

   if (dateTo != default(DateTime))
   {
      query = query.Where(o => o.OrderDate <= dateTo);
   }

   if (customerIds != null && customerIds.Any())
   {
      query = query.Where(o => customerIds.Contains(o.CustomerID));
```

```
    }

    if (statusIds != null && statusIds.Any())
    {
        query = query.Where(o => statusIds.Contains(o.StatuID));
    }

    if (isActive.HasValue)
    {
        query = query.Where(o => o.IsActive == isActive.Value);
    }

    // Execute the query and return the result
    var orders = await query.ToListAsync();

    // Map the result to the OrderDTO model if needed
    var orderDTOs = orders.Select(o => new OrderDTO
    {
        CustomerID = o.CustomerID,
        IsActive = o.IsActive,
        OrderDate = o.OrderDate,
        OrderID = o.OrderID,
        StatuID = o.StatuID
    }).ToList();

    return orderDTOs;
}
```

5) Bill, from the QA Department, assigned you a high priority task indicating there's a bug when someone changes the status from "Accepted" to "Picked Up". Define how you would proceed, step by step, until you create the Pull Request.

1. Understand the bug: Review the bug report or any additional information provided by Bill to understand the specific issue.
2. Reproduce the bug: Try to reproduce the bug locally by following the steps provided in the bug report or by creating a test scenario that triggers the issue.
3. Fix the bug: Once you have identified the cause of the bug, make the necessary code changes to fix it.
4. Create a new branch: Create a new branch in your version control system (e.g., Git) to isolate your changes.
5. Commit your changes: Make a commit with your code changes
6. Push the branch: Push your branch to the remote repository
7. Finally Create a Pull Request: Open a Pull Request (PR)