```
                c.PhoneNo = reader["PhoneNo"];
                c.UserName = reader["UserName"];
                c.Password = reader["Password"];
                c.Email = reader["Email"];
            }
             list.Add(c);
        }
        }
    }
    return list;
}
```

In this DAL method, I am lazy loading all customers, which means that I check the load status (which is being passed as an argument), and then, if the status is loaded, I load fetch all properties from the database; otherwise I fetch only `CustomerID` and `Name` as these properties will be displayed in the front end, the `CustomerList.aspx` page. So we are not loading other properties that are not required on that particular page. But if we need to load them all at once, then we can pass in the `Load Status` value as `Loaded` to indicate to our DAL that it has to load all properties.

Performance note: Note the use of data readers for better performance. Data readers are read-only, one-way pointers to the database. Hence, they are much lighter and faster than data sets and data adapters (which in turn use data readers to map data). It is always better to use data readers for filling in custom entities so that we have a fine level of control over the data access process, in addition to gaining the performance advantage of a data reader.

Architectural note: Note that instead of returning `List<Customer>` we are returning `Collection<Customer>` from the DAL methods due to the fact that `List<T>` should be used for internal use only, not in public APIs. Now our DAL is made in an API-like fashion. To make sure that our API is extendable, we need to use `Collection<T>` as this is extendable, unlike `List<T>` where we cannot override any member. For example, we can override the `SetItem` protected method in `Collection<T>` to get notified when the collection is changed (such as adding a new item and so on). Besides this, `List<T>` has too much extra stuff that is useful for internal use only, and not as a return type to an API.

Updating Business Objects

Now let us see how we can update a `Customer` business object in the `Customer.cs` business class. Here is the code:

```
public bool Update()
      {
         try
         {
         ///<remark>
            ///Check for the load status to make sure that update is
            ///not being called on disconnected/'ghost' loaded objects.
            ///</remark>
            if( _customerDTO.loadStatus == LoadStatus.Loaded)
            {
               CustomerDAL.UpdateCustomer(_customerDTO);
               ///<remarks>
              ///Set load status to 'ghost' to mark that it needs to
              ///be fully loaded again
                ///</remarks>
               _customerDTO.loadStatus=LoadStatus.Ghost;
               return true;
            }
            else
            {
                return false;
            }
         }
      }
```

In this `Update()` method, we first check if the `Customer` object to be updated is fully loaded or not, as there is no use performing an update action on a partially-loaded (or ghost loaded) `Customer` object (with only `Name` and `ID` fields). If the object is ghost loaded, we return the method `Call`, else we call the `Update` method of the DAL class to update the `Customer`.

After the DAL's `Update` method is called, we make sure that the `LoadStatus` of the `Customer` object is set to `Ghost`. This is done to make sure that all consumers of this object need to get the latest properties from the database, as it has been recently updated.