Here is the code for the `Customer` class:

```
using OMS.CodeTier.DAL;
namespace OMS.CodeTier.BL
{
    public class Customer
    {

        private int _ID;
        private string _name;
        private string _address;
        //more private members go here
        public int ID
        {
            get { return _ID; }
            set { _ID = value; }
        }
        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
        //more properties go here
        public void Add()
        {
            CustomerDAL.AddCustomer(this);
        }
        public void Delete(int customerID)
        {
            CustomerDAL.DeleteCustomer(this.ID);
        }
        //other methods..
    }
}
```

We have already seen all of these methods in Chapter 3. The only difference in this chapter is that instead of being in a logical layer inside the main web project, the BL classes are physically separated into another tier, and will compile into a different assembly.

The same thing goes for the DAL code. It is exactly the same as in Chapter 3, but is now under a different project and a new assembly.

Note that the BL code and the DAL code are still not physically separated. They are logically partitioned under different namespaces but under the same assembly (which means that they are under the same tier). But the GUI tier is now different from the BL and DAL tiers. This gives us the flexibility to change the BL and DAL assembly without re-compiling the GUI tier. Also, this structure gives us the

flexibility to use the current **OMS.CodeTier** assembly in other GUIs in addition to this one. For example, we can refer to and add this assembly to a Windows-based console application for our Order Management System, making our code more re-usable.

So we have achieved a greater degree of loose-coupling than the 1-tier solution architecture we studied in the previous chapters. In the next section, we will further de-couple the BL and DAL into separate tiers, and understand the 5-tier architectural approach.

5-Tier Architecture

With a 5-tier system, we introduce more redundancy into the application as a whole, along with separating the BL and DAL code into physical assemblies. This is how a sample 5-tier system would look like:

- Presentation tier
- UI tier
- Logical tier containing business logic (BL tier)
- Data access tier (DAL tier)
- Data tier (physical database)

Now why do we need to separate the logical layer and data layer into physical tiers? There can be many reasons to go for this architectural configuration. Some of them are listed here:

- You want further decoupling of the layers to introduce a flexible architecture for your project. Let me explain this further. When we have business and data access code in the same assembly (but logically separated in different files or using namespaces), we cannot distribute the code separately. In most enterprise applications, there is a greater need for code re-use. Some third-party applications might want to use our applications' business logic code (such as consuming an API) and some might want to use our data access code. In such cases, if we go for a layered solution, then changing anything in DAL would necessitate re-compiling the whole assembly, which also includes business logic assembly too. And this unnecessary change might create ripple effects in the application as the same DLL might be used in other third-party applications. So this should be avoided for large enterprise applications. For flexible adaptability, it is better to separate BL and DAL into their own assemblies.