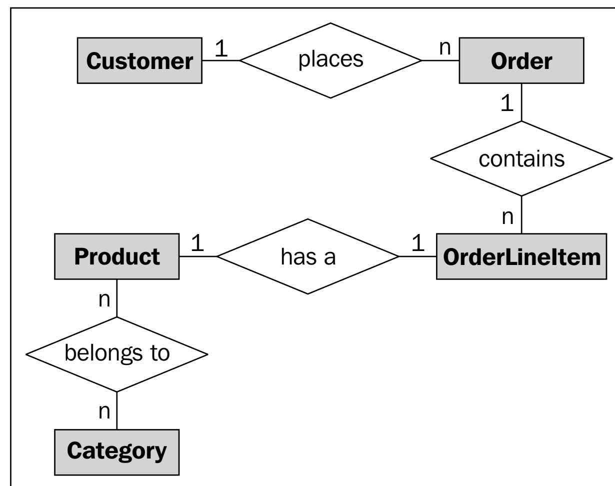
- **Many-to-many**: Depicted as n:m

  Example: One Product can be included in multiple Categories and one Category can contain multiple Products; therefore the Product and Category entities share a many-to-many relationship

After adding the cardinality of the relationships to our ER diagram, here is how it will look:



This basic ER diagrams tells us a lot about how the different entities in the system are related to each other, and can help new programmers to quickly understand the logic and the relationships of the system they are working on. Each entity will be a unique table in the database.

OMS Project using 2-Layer

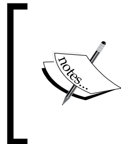
We know that the default coding style in ASP.NET 2.0 already supports the 1-tier 1-layer style, with two sub-layers in the main UI layer as follows:

- Designer code files: ASPX markup files
- Code behind files: Files containing C# or VB.NET code

Because both of these layers contain the UI code, we can include them as a part of the UI layer. These two layers help us to separate the markup and the code from each other. However, it is still not advisable to have logical code, such as data access or business logic, directly in these code-behind files.

Now, one way to create an ASP.NET web application for our Order Management System (OMS) in just one layer is by using a DataSet (or DataReader) to fill the front-end UI elements directly in the code-behind classes. This will involve writing data access code in the UI layer (code-behind), and will tightly bind this UI layer with the data access logic, making the application rigid (inflexible), harder to maintain, and less scalable. We have already seen this approach in the guestbook example in Chapter 2 (2-layered systems), and we know the drawbacks of this approach.

In order to have greater flexibility, and to keep the UI layer completely independent of the data access and business logic code, we need to put these elements in separate files. So we will now try and introduce some loose-coupling by following a 2-layer approach this time. What we will do is, write all data access code in separate class files instead of using the code-behind files of the UI layer. This will make the UI layer independent of the data-access code.



We are assuming that we do not have any specific business logic code at this point, or else we would have put that under another layer with its own namespace, making it a 3-layered architecture. We will examine this in the upcoming sections of this chapter.

Sample Project

Let us see how we can move from this 1-tier 1-layer style to a 1-tier 2-layer style. Using the ER diagram above as reference, we can create a 2-Layer architecture for our OMS with these layers:

- UI-layer with ASPX and code-behind classes
- Data access classes under a different namespace but in the same project

So let's start with a new VS 2008 project. We will create a new ASP.NET Web Project in C#, and add a new web form, `ProductList.aspx`, which will simply display a list of all the products using a Repeater control. The purpose of this project is to show how we can logically break up the UI layer further by separating the data access code into another class file.

The following is the ASPX markup of the ProductList page (unnecessary elements and tags have been removed to keep things simple):

```
<asp:Repeater ID="prodRepeater" runat="server">
     <ItemTemplate>
          Product Code: <%# Eval("Code")%>
               <br>
          Name:         <%# Eval("Name")%>
```