Re-usability

If application code can be re-used within itself, or for some other external application, then not only do we save development and maintenance costs, but we also avoid code replication and make our code componented. For example, assume that you are developing an **Order Management System** (**OMS**) for a company. Now the company wants to re-use the business logic and data access code in their own small applications that might not be a part of the OMS you are developing. If you do not separate your code into separate physical assemblies, they won't be able to re-use your code easily. It would be too cumbersome to make a copy of your code as it is and then use it in a third-party application. Sometimes, it is not even possible to make this copy, like the other system is using a different .NET language than the one used by your OMS. So a physical separation is inevitable in such scenarios. Once this is done, we can give them individual assemblies, such as a data access code wrapped inside a DLL, so that they don't have to write DAL of their own and can use our assembly instead. This method of programming is called developing an **Application Programming Interface** (**API**). It is an important principle in software programming to remove rigidity and make your application components re-usable in other applications.

Application Programming Interface, or API development, is considered a very important principle in software design and development. Each method you write in code, if written properly, can be considered an API. The more loosely-coupled and flexible your code is, the more API-like it becomes. This helps you distribute your code and make it re-usable.

Loose-Coupling

If the code you write in one layer is highly-dependent on the code in some other layer, then your code is tightly-coupled, which means changing any part of it might break the other parts on which it is dependent. For example, in a 1-tier 2-layer system that we studied in Chapter 3, the UI code was calling the data access layer from the code-behind classes. This means that if the data access method has any error, the UI code will break. So UI and DAL are tightly-coupled. In the same chapter, we learnt the 3-layer model, where the UI interacted with business logic (BL) classes, which in turn called DAL methods. So if the DAL method breaks, the UI may not break as easily as in the first case, because we have made the layers loosely-coupled by bringing in a third layer (BL).

If we can make the higher-level components of our application independent of each other, then our application will become loosely-coupled. This means that changing one of the layers or tiers should not break the other layers. For example, if your DAL code is not properly abstracted but is tightly wound with the other layers above or below it, then it would be difficult to re-use it in any other application. So if someone makes a mistake in the DAL, the entire application will break down.

To avoid rigidity in large software systems, the concept of "implementation of loose coupling" is very important. N-tier architecture makes it possible to bring in loose coupling into our applications.

Plug and Play

To understand the Plug and Play functionality, consider the OMS example given earlier. Consider the requirement of making the application database agnostic. This means that the same application should work with MS SQL Server as well as with Oracle or any other database. So we need to make our DAL code capable of switching between databases. To achieve this, we create different DAL assemblies each having the code targeted to each database type, and we load a specific DAL assembly at runtime based on a key value in a `config` file. This means that our application is Plug and Play.

Another simple example can be a file encryption program that needs to support multiple encryption algorithms. The user should be able to select a particular algorithm from among all of the choices for encrypting files. These are two simple examples of where a Plug and Play type architecture would be required. The application would be developed from ground up to be able to support a Plug and Play based style. Also, this would give the developer the opportunity to write code to support other databases or algorithms later on without changing any code in the main application. A well-known design pattern that can be used in such cases is known as **Dependency Injection** (**DI**, which we will learn in the coming chapters).

All of the above points are major reasons to consider an n-tier architecture and make applications more scalable, robust, and loosely-coupled. In the coming sections, we will learn how to break layers into physical tiers and implement an n-tier architecture in our Order Management System, addressing what possible options we have, and which ones to select depending on the project needs.