N-Tier Architecture and Tips

## Contents

* [Overview](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#Overview)
* [**N-Tier Architecture Introduction**](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#N-Tier Architecture Introduction)
  + [Some Terminology's Difference and Relationship](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#Some Terminology's Difference and Relationships)
    - [**Tier and Layer**](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#Tier and Layer Relationship)
    - [Tier And Process](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#Tier And Process Relationship)
    - [Layer and Process](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#Layer and Process Relationship)
  + [**3-Tier Architecture**](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#3-Tier Architecture)
  + [**1, 2, 3 or More Tier Architectures**](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#1, 2, 3 or More Tier Architecture)
* [**Advantages and Disadvantages of Different Tier Architectures**](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#Advantages and Disadvantages of Different Tier Architectures)
  + [**1 or 2-Tier Architecture**](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#1 or 2-Tier)
    - [Advantages](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#12Advantages)
    - [Disadvantages](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#12Disadvantages)
  + [**N-Tier Architecture**](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#N-Tier Architecture)
    - [Advantages](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#nAdvantages)
    - [Disadvantages](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#The Disadvantage of the N-Tier Deployment)
* [Business Data Validation in N-Tier Architecture](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#Business Data Validation in N-Tier Architecture)
* [**How to Deploy N-Tier Application Correctly**](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#How to Deploy N-Tier Application Correctly)
* [**How to Achieve N-Tier Deployment Capability by Software Technology**](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#How to Achieve N-Tier Deployment Capability by Software Technologies)
* [Some Practical Tips on N-Tier Architecture Development](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#Some Practical Tips on N-Tier Architecture Development)
* [Conclusions](http://www.codeproject.com/Articles/430014/N-Tier-Architecture-and-Tips?display=Print#Conclusions)

Overview

## N-Tier architecture is an industry-proved software architecture model, suitable to support enterprise-level client/server applications by resolving issues like scalability, security, fault tolerance and etc. .NET has many tools and features, but .NET doesn’t have pre-defined ways to guard how to implement N-Tier architecture. Therefore, in order to achieve good design and implementation of N-Tier architecture in .NET, understanding fully its concepts is very important. However, many of us may hear, read or use N-Tier architecture for many years but still misunderstand its concepts more or less. This article tries to clarify many basic concepts in N-Tier architecture from all aspects, and also provide some practical ****tips****. The tips in this article are based on the assumption that a team has a full control over all layers of the N-Tier architecture. We have another article to elaborate a N-Tier architecture sample in .NET:

## N-Tier Architecture Introduction

### Some Terminology's Difference and Relationship

#### Tier and Layer

Firstly we need to clarify the difference between two terms in N-Tier architecture: tier and layer. Tier usually means the physical deployment computer. Usually an individual running server is one tier. Several servers may also be counted as one tier, such as server failover clustering. By contrast, layer usually means logic software component group mainly by functionality; layer is used for software development purpose. Layer software implementation has many advantages and is a good way to achieve N-Tier architecture. Layer and tier may or may not exactly match each other. Each layer may run in an individual tier. However, multiple layers may also be able to run in one tier.

A layer may also be able to run in multiple tiers. For example, in Diagram 2 below, the persistence layer in .NET can include two parts: persistence Lib and WCF data service, the persistence lib in the persistence layer always runs in the same process as business layer to adapt the business layer to the WCF data service. However, the WCF data service in persistence layer can run in a separate individual tier. Here is another example: we may extract the data validation in business layer into a separate library (but still kept in business layer), which can be called by client presenter layer directly for a better client-side interactive performance. If this occurs, then data validation part of the business layer runs in the same process of the client presenter layer, the rest of business layer runs in a separate tier.

#### Tier And Process

If a layer can run in an individual process, usually it will also be able to run in an individual computer (tier), hence it can be considered capable for an individual tier in N-Tier architecture. However, this isn’t always true. For example, assume that there are two layers which are implemented to run in two individual processes; they communicate with each other too. However, if these two layers are implemented in a such way that their IPC (inter-process communication) is solely based on a non-distributed way, such as the local shared memory, then these two layers can run in two different processes only in the same computer, not in two different computers. Unless there is another alternative distributed IPC way (such as socket) available for these two layers, these two layers will be considered capable for only one tier even though they can run in two different processes of the same computer.

#### Layer and Process

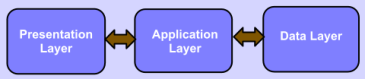
A layer may run in an individual process; several layer may also run in an individual process; a layer may run in several processes too. If you read above section “Tier and Layer Relationship”, you can understand here easily.

### 3-Tier Architecture

We introduce the 3-Tier concept first so that we can understand other tier concepts later easily. The simplest of N-Tier architecture is 3-Tier which typically contain following software component layers listed from the top level to the low level: presentation layer, application layer and data layer, which are depicted in Diagram 1.

A layer can access directly only the public components of its directly-below layer. For example, presentation layer can only access the public components in application layer, but not in data layer. Application layer can only access the public components in data layer, but not in presentation layer. Doing so can minimize the dependencies of one layer on other layers. This dependency minimization will bring benefits for layer development/maintenance, upgrading, scaling and etc. Doing so also makes the tier security enforcement possible. For example, the client layer cannot access the data layer directly but through the application layer, so data layer has a higher security guarding. Finally, doing so can also avoid cyclic dependencies among software components.

In order to claim a complete 3-Tier architecture, all three layers should be able to run in separate computers.



***Diagram 1: 3-Tier Architecture***

These three layers are briefly described as below:

**Presentation layer**: a layer that users can access directly, such as desktop UI, web page and etc. Also called client.

**Application layer**: this layer encapsulates the business logic (such as business rules and data validation), domain concept, data access logic and etc. Also called middle layer.

**Data layer**: the external data source to store the application data, such as database server, CRM system, ERP system, mainframe or other legacy systems and etc. The one we meet often today is database server. For N-Tier architecture, we need to use the non-embedded database server, such as SQL server, Oracle, DB2, MySQL or PostgreSQL. The non-embedded database server can be run in an individual computer. Whereas, the embedded type databases, such as Microsoft access, dbase and etc, cannot run in an individual computer, and then cannot be used as the data layer of the 3-Tier architecture.

### 1, 2, 3 or More Tier Architecture

**1-Tier**: all above layers can only run in one computer. In order to achieve 1-Tier, we need to use the embedded database system, which cannot run in an individual process. Otherwise, there will be at

least 2-Tier because non-embedded databases usually can run in an individual computer (tier).

**2-Tier**: either presentation layer and application layer can only run in one computer, or application layer and data layer can only run in one computer. The whole application cannot run in more than 2 computers.

**3-Tier**: the simplest case of N-Tier architecture; all above three layers are able to run in three separate computers. Practically, these three layers can also be deployed in one computer (3-Tier architecture, but deployed as 1-Tier).

**N-Tier**: 3 or more tiers architecture. Diagram 2 below depicts a typical N-Tier architecture. Some layers in 3-Tier can be broken further into more layers. These broken layers may be able to run in more tiers. For example, application layer can be broken into business layer, persistence layer or more. Presentation layer can be broken into client layer and client presenter layer. In diagram 2, in order to claim a complete N-Tier architecture, client presenter layer, business layer and data layer should be able to run in three separate computers (tiers). Practically, all these layers can also be deployed in one compute (tier).



***Diagram 2: N-Tier Architecture***

Below are brief summaries on all layers in Diagram 2:

***Client layer***: this layer is involved with users directly. There may be several different types of clients coexisting, such as WPF, Window form, HTML web page and etc.

***Client presenter layer***: contains the presentation logic needed by clients, such as ASP .NET MVC in IIS web server. Also it adapts different clients to the business layer.

**Business layer**: handles and encapsulates all of business domains and logics; also called domain layer.

**Persistence layer**: handles the read/write of the business data to the data layer, also called data access layer (DAL).

**Data layer**: the external data source, such as a database.

Sometimes, the number of tiers is able to be equal or more than 3, but client presenter layer, business layer and data layer cannot run in three separate computers (tiers). Is this a N-Tier architecture? we categorize this N-Tier as an incomplete N-Tier architecture because its client presenter layer, business layer and data layer cannot run in three separate computers (tiers).

If we use the modem non-embedded database such as Sql Server, Oracle and etc, these databases will always be able to run in an individual computer. Therefore, for this case in Diagram 1, the criteria of a 2-Tier architecture is that presentation layer and application layer can run in only one computer; the criteria of a complete 3-Tier architecture is that presentation layer and application layer can run in different computers. A complete N-Tier architecture has the same criteria as 3-Tier.

## Advantages and Disadvantages of Different Tier Architectures

### 1 or 2-Tier Architecture

**Advantages**: simple and fast for a lower number of users due to fewer processes and fewer tiers; low cost for hardware, network, maintenance and deployment due to less hardware and network bandwidth needed.

**Disadvantages**: will have issues when the number of users gets big; has limitation to solve issues like security, scalability, fault tolerance and etc because it can be deployed in only 1 or 2 computes.

### N-Tier Architecture

**Advantages**: there are following general advantages:

1. Scalable: this is due to its capability of multiple tier deployment and the tier decoupling it brought. For example, the data tier can be scaled up by database clustering without other tiers involving. The web client side can be scaled up by load-balancer easily without affecting other tiers. Windows server can be clustered easily for load balancing and failover. In addition, business tier server can also be clustered to scale up the application, such as Weblogic cluster in J2EE.
2. Better and finer security control to the whole system: we can enforce the security differently for each tier if the security requirement is different for each tier. For example, business tier and data tier usually need higher security level than presentation tier does, then we can put these two high security tiers behind firewall for protection. 1 or 2 tiers architecture cannot fully achieve this purpose because of a limited number of tiers. Also, for N-Tier architecture, users cannot access business layer and data layer directly, all requests from users are routed by client presenter layer to business layer, then to data layer. Therefore, client presenter layer also serves as a proxy-like layer for business layer, and business layer serves as a proxy-like layer for data layer. These proxy-like layers provides further protection for their layers below.
3. Better fault tolerance ability: for example, the databases in data layer can be clustered for failover or load balance purpose without affecting other layers.
4. Independent tier upgrading and changing without affecting other tiers: in object-oriented world, Interface-dependency implementation can decouples all layers very well so that each layer can change individually without affecting other layers too much. Interface-dependency means a layer depends on another layer by interfaces only, not concrete classes. Also, the dependency of a layer only on its directly-below layer also minimizes the side effect of a layer’s change on the whole system. For example, if keep the interfaces unchanged, we can update or replace the implementation of any layer independently without affecting the whole system. Due to the changing of business requirement and technology, changing the implementation of a layer to another totally different one does happen often. For example, originally we use Windows Form mainly, now we use WPF mainly. If our original system is implemented as the decoupled layer structure, then we will only need to update the client side from Windows Form to WPF without the need to change the server side layers.
5. Friendly and efficient for development: the decoupled layers are logic software component groups mainly by functionality, they are very software development friendly and efficient. Each layer can be assigned individually to a team who specializes in the specific functional area; a specialized team can handle the relevant task better and more efficiently.
6. Friendly for maintenance: N-Tier architecture groups different things together mainly by functionality and then makes things clear, easily understandable and manageable.
7. Friendly for new feature addition: due to the logical grouped components and the decoupling brought by N-Tier architecture, new features can be added easily without affecting too much on the whole system.
8. Better reusability: this is due to the logically grouped components and the loose couplings among layers. Loosely-coupled component groups are usually implemented in more general ways, so they can be reused by more other applications.

#### The Disadvantages of the N-Tier Deployment

1. The performance of the whole application may be slow if the hardware and network bandwidth aren’t good enough because more networks, computers and processes are involved.
2. More cost for hardware, network, maintenance and deployment because more hardware and better network bandwidth are needed.

The effects of N-Tier deployment on the application performance are a double edge issue. In one side, if the number of uses isn’t big enough, the performance may be slow due to more computers, process and network involved. Namely, if put everything in one tier or one process, performance will be better for a small number of users. However, if the number of user gets big, then the scalability brought by N-Tier will improve the overall performance, such as load balancing and database clustering which all improve the performance of N-Tier architecture. Why the performance result of the small number of users and the big number of users are different? this is because the bottleneck of the whole application are different for these two situations. For the case with the small number users, the bottleneck is the time for data to communicate among different processes. If more computer, more processes and longer network, then costs longer, then performance is bad. However, when the number of user gets big, the bottleneck shift to other things because of server’s capacity, such as cpu and memory resource contentions in one computer, database threshold in server, horsepower limitation of a web server and etc. Only the scalability of N-Tier architecture can solve these bottlenecks existing with a big number of users; usually load balancing by server clustering is used for N-Tier architecture’s scalability. With more computers are scaled up to share the task for big number of users, then performance is improved.  Besides gaining performance by scalability of N-Tier architecture, we can also improve the performance with better hardware and better network bandwidth to meet our business requirement.

## Business Data Validation in N-Tier Architecture

Data validation is important and a MUST in N-Tier architecture in order to keep the whole business system healthy and integral. The first question for business data validation will be: where or which layer should handle the data validation? There are some rules and facts as below for the business data validation, which will give us some tips and can answer this question too:

1. Data validation can be checked in any layer. Usually, the closer to the client layer the validation is, the more efficient the performance is. The farther to the client layer the validation is, the more reliable and robust the application is. When the validation is checked in business layer or persistence layer, it is guaranteed that every type client will get the validation regardless whether or not the client side will check the validation.
2. When we decide which layer should do the validation, we need to achieve a balance result between performance, reliability and robustness, also we need to make decisions based on the actual situation. If we have full control of the all layers, we can let all validations happen in client/client presenter layers only to gain performance. However, if the business layer is also exposed to some client/client presenter layers which is out of our control, then the business or lower layer must do all validations to gain reliability regardless of whether or not our clients do the same validation.
3. Client side validation is efficient, such as Javascript validation in a web page. However, users may bypass the client side validation easily and intentionally, such as webpage hacking. Therefore, it is needed to do data validation in both client side and server side to achieve both performance and reliability. The business layer and other further lower layers usually belong to the server side. Client presenter layer may or may not be in the server side; a webserver client presenter layer such as ASP.NET is in the server side. Client presenter layer for WPF may not be in the server side.
4. A more practical way is to do the simple data validation in client side for performance and further do a full validation in server side for reliability. The simple data validation is mainly the single property checking of an entity instance. The full validation includes the simple data validation and some complex data validation. The complex data validation can include class-level data validation which crosses multiple properties of an entity instance and the data validation which crosses multiple entity instances of similar or different types.
5. For some interactive client application, we need to do client side validation anyway for acceptable interactive performance, regardless of whether or not we will do the validation in server side.  Some game applications belong to this category.
6. We should implement and maintain one version of validation logics in one place, regardless of where the validation will be checked. All layers should share this one version of validation logics. Why? Doing so has better reusability; it can avoid duplicate and conflicting validation logic in many places and makes the development, maintenance and deployment easier; it keeps the whole validation logics consistent throughout the whole application. In addition, the places to check validation may change with the changing and growing of the business, so validation should be kept in one place with one version but flexible to be called by any layer if needed.

## How to Deploy N-Tier Application Correctly

More tiers brings extra complexity, extra deployment/maintenance effort and extra cost. Therefore, the number of tiers should be kept as minimal as enough to solve issues like the scalability, security, failover and etc. If these issues are solved as needed, don’t deploy more tiers further. But, in order to solve these issues as better as possible, usually 3-Tier will be needed at least. If these issues aren’t concerned at all in certain cases, then we can select 1 or 2-Tier architecture or 1 or 2-Tier deployment of N-Tier architecture to gain performance. What is the best number of tiers? no fixed answer. In order to meet our business requirement, we need to select the number of tiers to achieve a best balance result among those good and bad things of N-Tier architecture.

We should differ the following two cases: a) all layers run in one process of a computer, and b) all layers run in different processes of a computer. The 1st case is actually 1 tier architecture; the 2nd case is usually N-Tier architecture but deployed in just 1 tier.  Even both are in one computer, the 1 tier architecture will have better performance because of fewer processes involved.  Compared to one process, communication crossing process boundaries is more complex and slower, regardless what type of IPC (inter-process communication) technique is used: TCP/IP, named pipe, message queue or shared memory and etc. Therefore, in a deployed computer, we need to keep the number of the application processes as few as possible to gain performance. How to achieve this? the N-Tier architecture can be implemented in such a way that switching among different tier architectures is as easy as updating the configuration file only. This is introduced in detail in our sample application article: [A N-Tier Architecture Sample with ASP.NET MVC3, WCF and Entity Framework](http://www.codeproject.com/Articles/434282/A-N-Tier-Architecture-Sample-with-ASP-NET-MVC3-WCF).

Practically, there are many variations of N-Tier architecture; they are existing for reasons. For example, one possibility is to put the client presenter layer and business layer in the same process to achieve a better interactive performance. You can explore and research this topic further by yourself if you are interested in it.

In addition, 3-Tier architecture can be deployed as 3 or fewer tiers. But a 2-Tier architecture cannot be deployed as 3-Tier; otherwise it should be called 3-Tier architecture, not 2-Tier.

## How to Achieve N-Tier Deployment Capability by Software Technologies

The main characteristic of N-Tier is the ability to deploy one or more layers in different computers to deal with issues like scalability, security, fault tolerance and etc; two related tiers need to communicate with each other. How to achieve this? Application running in a tier is in processes. So, communication between two tiers actually comes down to IPC (Inter-process Communication) issue. A distributed IPC method can support two processes in two different computers to communicate with each other, such as socket, distributed message queue and etc. So, natively, these distributed IPC methods can support N-Tier deployment capability. For example, if two layers are implemented with TCP/IP socket as the communication way, these two layers can be deployed in two different computers (tiers) to communicate with each other. In .NET, WCF can achieve N-Tier deployment requirement easily since WCF supports communication among processes - either on the same computer or different computer; WCF is built on top of the basic IPC methods. Another advantage to use WCF for N-Tier architecture is that WCF is able to achieve SOA (service oriented architecture) result with a very loosely-coupled dependency among layers.

## Some Practical Tips on N-Tier Architecture Development

Design, implement, deploy and maintain a N-Tier architecture are daunting tasks. If you don’t have clear top thoughts in the beginning, you probably end up with a lot of time wasting due to detouring and twisting here and there. We already talks some tips on deployment and data validation above, here, we will provide some extra practical tips on N-Tier architecture development. The tips in this article are based on the assumption that a team has a full control over all layers of the N-Tier architecture.

1. Decouple a layer from another layer as much as possible by some loosely coupling techniques, such as soap xml and interface etc. In object-oriented world, each layer should depend on its immediate lower layer by interfaces only, not by concrete classes. By doing so, we can achieve the maximum decoupling between two layers, this decoupling will bring many benefits for development, unit test, maintenance, upgrading, inter-changeability, reusability and etc.
2. Try as much as possible to auto-generate and maintain only one version poco business entity classes, which can be reused in the whole application, why? Business entity classes are the base of the N-Tier architecture, they convey the info from the topest layer to the lowest layer. Modern application tends to continue to grow bigger and bigger, therefore, manually creating the mass entity classes is daunting and error-prone, particularly for different versions in different layers as some people prefer. So, we suggest that in the whole application, we should try to use only one lightweight POCO version of entity classes, which should be auto-generated by code generators. Doing so will save us a lot of efforts, also wipe out the headache of the mappings and the inconsistency of different versions of entity classes. There are many code generators available nowadays for doing this, such as code generators for Entity Framework. Certainly, different layers may have different requirement on entity classes, if so, we can use annotation feature in System.ComponentModel.Annotation and partial class feature in C# to restrict or expand these auto-generated entity classes for our special needs in a specific layer.
3. Use some auto tools or packages to generate the mappings between business entity classes and a traditional relational database (data layer). Modem business and database get bigger and bigger, so manually create these mappings isn’t easy and is error-prone. There are many existing packages or tools to help us,  such as Entity Framework and NHibernate for .NET, and Hibernate for Java.
4. Use code generators as much as possible further for other massive and style-similar code. If you cannot find any existing code generator for your purpose, develop one for yourself. By using an object-oriented programming  languages (like C# or Java) together with XSLT (Extensible Stylesheet Language Transformations), it won’t be harder to develop a code generator for your special need. XSLT is xml-based, very helpful and flexible for code generator purpose because it can convert any xml document into any text document easily.
5. Business layer is easily prone to be tightly coupled with persistence layer; we should avoid this. For example, in .NET, a WCF business service may access Entity Framework directly. This situation is very popular. However, doing so has issue that the business layer and the persistence layer are tightly coupled. This tightly coupling will bring many issues on layers for unit-testing, upgrading, inter-changeability and etc. Usually we need an adapter layer between these two so that they will be loosely coupled by interface only. There is a physical sample for this in our N-Tier sample article: [A N-Tier Architecture Sample with ASP.NET MVC3, WCF and Entity Framework](http://www.codeproject.com/Articles/434282/A-N-Tier-Architecture-Sample-with-ASP-NET-MVC3-WCF).
6. In client presenter layer, we should put all common code for all clients to a separate library as much as possible to maximize the code reusability for all types of clients.
7. A cache layer can be added into any existing layer to speed up performance. For example, Varnish accelerator can be used for ASP .Net, Drupal or other web applications as a cache layer sitting between client layer and client presenter layer to speed up performance. Memcached and APC cache are PHP cache packages which can be added in many PHP layers as extra layer for caching business data. Requests will go to cache layer first, if valid data is in cache, then request will return and won’t go to the lower layer further, then performance is improved. Usually updating or expiration of data will invalidate the old data in a cache. In .NET 4, namespace System.Web.Caching can be used for caching in ASP.NET; namespace System.Runtime.Caching can be used for caching in any place; the Caching Application Block in Enterprise Library is also an option for caching.
8. In order to adapt to the changeable business requirements and technologies, it is good to implement N-Tier architecture in such a way that it can be flexible to any type of deployment easily, including 2-Tier architecture deployment. For example, the N-Tier architecture can be implemented in such a way that different tier architectures can be switched simply by some parameter-value updating in configuration files. There is a physical sample implementation for this in our N-Tier sample article: [A N-Tier Architecture Sample with ASP.NET MVC3, WCF and Entity Framework](http://www.codeproject.com/Articles/434282/A-N-Tier-Architecture-Sample-with-ASP-NET-MVC3-WCF).

## Conclusions

1. A complete 3-Tier architecture should be able to run its presentation layer, application layer and data layer in 3 separate computers (refer to Diagram 1). A complete N-Tier architecture should be able to run its client presenter layer, business layer and data layer in at least 3 separate computers (refer to Diagram 2). A layer can only access directly the public components of its directly-below layer; doing so can minimize the dependencies of one layer on other layers and also makes the tier security enforcement possible. The complete N-Tier architecture has the best capability to handle issues like scalability, security, fault tolerance and etc.
2. Tier usually means physical deployment computer; layer usually means logic software component group mainly by functionality. Layer implementation is a common and best way to achieve a N-Tier architecture. Tier and layer may or may not exactly match. A layer may run as an individual tier; a layer may also be able to run in multiple tiers; multiple layers may also run as one tier only.
3. If a layer can run in an individual process, usually it will also be able to run in an individual computer (tier). However, this isn’t always true. If this process communicates with other layer processes only by a non-distributed IPC (inter-process communication) method, this process won't be able to run in an individual computer (tier).
4. N-Tier architecture has following advantages: better scalability, better and finer security control, better fault tolerance ability, independent tier upgrading and changing ability without affecting other tiers, friendly and efficient development, friendly maintenance, friendly new feature addition, better reusability and etc.
5. If a modem non-embedded database (such as Sql Server, Oracle and etc) is used as the data layer, this database will always be able to run in an individual tier. For this case in Diagram 1, the criteria of 2-Tier architecture is that presentation layer and application layer can only run in one computer; the criteria of a complete 3-Tier or other complete N-Tier architecture is that presentation layer and application layer can run in separate computers.
6. More tiers brings extra complexity, extra deployment/maintenance effort and extra cost. Therefore, the number of tiers should be kept as minimal as enough to solve issues like the scalability, security, failover and etc. If these issues are solved as needed, don’t deploy more tiers further. In addition, in a deployed computer, we need to keep the number of processes as few as possible to gain performance; good architecture design can achieve this easily.
7. Distributed IPC (inter-process communication) methods (such as socket) can make a layer’s process deployable in a tier. In .NET, WCF is a good way to achieve the deployment capability of N-Tier architecture.
8. There are following facts and rules on data validation in N-Tier architecture:
   1. Data validation can be checked in any layer. Usually, the closer to the client layer the validation is, the more efficient the performance is. The farther to the client layer the validation is, the more reliable and robust the application is. When the validation is checked in business layer or persistence layer, it is guaranteed that every type client will get the validation regardless whether or not the client side will check the validation.
   2. When we decide which layer should do the validation, we need to achieve a balance result between performance, reliability and robustness, also we need to make decisions based on the actual situation.
   3. Client side validation is efficient, such as Javascript validation in a web page. However, users may bypass the client side validation easily and intentionally, such as webpage hacking. Therefore, it is needed to do data validation in both client side and server side to achieve both performance and reliability.
   4. A more practical way is to do the simple data validation in client side for performance and further do a full validation in server side for reliability. The simple data validation is mainly the single property checking of an entity instance. The full validation includes the simple data validation and some complex data validation crossing multiple properties of an entity instance or crossing multiple entity instances of similar or different types.
   5. For some interactive client application, we need to do client side validation anyway for acceptable interactive performance, regardless of whether or not we will do the validation in server side.  Some game applications belong to this category.
   6. We should implement and maintain one version of validation logics in one place, regardless of where the validation will be checked. All layers should share this one version of validation logics. Doing so has better reusability and makes the development, maintenance and deployment easier, and also keeps the validation logics consistent throughout the whole application.
9. Below are some extra practical tips on N-Tier architecture development, based on the assumption that a team has its full control over all layers:
   1. Decouple a layer from another layer as much as possible by some loosely coupling techniques, such as soap xml and interface etc. In object-oriented world, each layer should depend on its immediate lower layer by interfaces only, not  by concrete classes. By doing so, we can achieve the maximum decoupling between two layers, this decoupling will bring many benefits for development, unit test, maintenance, upgrading, inter-changeability, reusability and etc.
   2. Try as much as possible to auto-generate and maintain only one version poco business entity classes, which can be used in the whole application. This will reduce all hassles for version conflicting, mapping, errors from manual coding and etc.
   3. Use some auto tools or packages to generate the mappings between business entity classes and a traditional relational database (data layer), such as packages Entity Framework and NHibernate for .NET, and Hibernate for Java.
   4. Use code generators as much as possible further for other massive and style-similar code. If you cannot find any existing code generator for your purpose, develop one for yourself. C#, java, XSLT and etc can help to develop code generators easily.
   5. Business layer is easily prone to be tightly coupled with persistence layer; we should avoid this by an adapter layer between these two layers so that they will be loosely coupled by interface only, such as a persistence adapter between WCF business layer and Entity Framework.
   6. In client presenter layer, we should put all common code for all clients to a separate library as much as possible to maximize the code reusability for all types of clients.
   7. A cache layer can be added into any existing layer to speed up performance.
   8. In order to adapt to the changeable business requirements and technologies, it is good to implement N-Tier architecture in such a way that different tier architectures can be switched simply by some parameter-value updating in configuration files.