Software Abstraction Layer

Danut Prisacaru

March 2008

Summary: The N-tier architecture has been with us for many years. It helped us separate the Presentation tier from the Business tier and that, in turn, from the Data tier. This provides a higher degree of decoupling when compared to the old one-tier or two-tier architecture. Although the N-tier architecture is helping us make applications more flexible and easier to change, there is still an important potential for coupling between the different technologies and logical layers that exist in each physical tier.

This article tries to identify the points of coupling. It also proposes a solution that increases the separation of concerns; this allows the N-tier applications to be easier to change. Although the article makes its point using Microsoft® .NET Framework technologies and the C# language, the solutions can also apply to other platforms and languages.

We will also provide a proof-of-concept prototype that illustrates the points we are trying to make in this article.

Table of Contents

Scenario

For the purpose of this article, we will imagine an application that uses the three-tier architecture. The Presentation tier is implemented using Windows Forms, the Middle tier uses Internet

Information Services (IIS) and exposes Web services, and the Data tier is provided by SQL Server®.

Figure 1 illustrates a simplified view of the three-tier architecture.

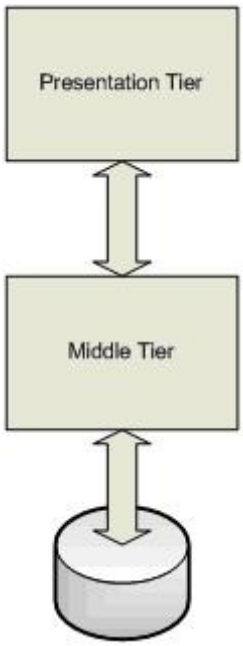

Figure 1. The classic three-tier architecture

Looking deeper at what happens in each physical tier, we may discover that each tier is a mix of several logical layers and, in many cases, these layers depend on each other.

The Complexity of Mixing Code

If we zoom in to the Presentation tier and the Middle tier, we discover that they are not monolithic and homogenous layers. Each tier is actually made of several logical layers. For example, the Presentation tier has a UI part and a logical layer that is composed of proxies that help the UI communicate with the Middle tier. This tier also includes the layer that communicates with the database, such as ADO.NET, the business logic modules, and the layer that makes the service provider. Figure 2 illustrates a closer look at each tier.

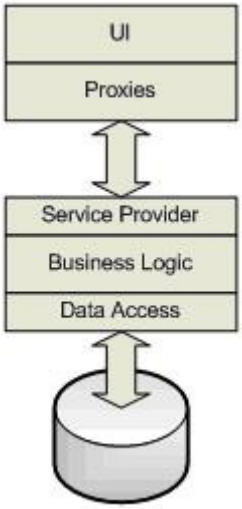

Figure 2. A closer look at each tier

In the ideal situation, each logical layer is separated from the others. In reality, many applications have a mix-and-match between them, as illustrated in Figure 3.

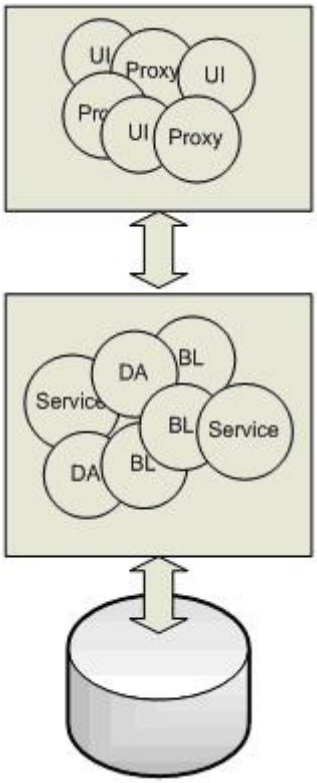

Figure 3. Tiers of a typical application

When tiers are not properly designed and implemented, all the benefits of separating the application into tiers are lost. Although the Presentation tier is physically or logically separated

from the Middle tier, the mix-and-match between UI code and the proxies make the code difficult and costly to maintain. The same thing happens in the Middle tier where we see the service provider (Service) layer, the business logic (BL) layer, and the data access (DA) layer connected with each other. In cases like this, it is virtually impossible to replace one logical layer with a new technology.

Let's say we currently use the .NET Framework 2.0 Web services on the Middle tier. On the Presentation tier, we had the proxies/references generated by the Microsoft Visual Studio® development system. We plan to move to Windows Communication Foundation (WCF). How are we going to do that if the implementation looks like Figure 3?

We all know it is easy to build a new application from scratch. The most expensive part is to maintain it, and if the architecture and design of the modules that make the application do not consider the cost of maintenance, transition to new technologies is very expensive; in some cases, it requires a total rewrite of the application. How do we build an application so we reduce the cost of the maintenance?

We also know that the complexity of any software system grows over time. How do we make it so the parts that make the system stay simple, although the complexity grows, and can be easily changed or even replaced?

N-Tier Flexibility Relies on Layers

Potentially, the business may require one day that the N from the "N Tier" to vary, such as from three to two or another number.

We need a solution that can easily accommodate the business requirements, as illustrated in Figure 4.

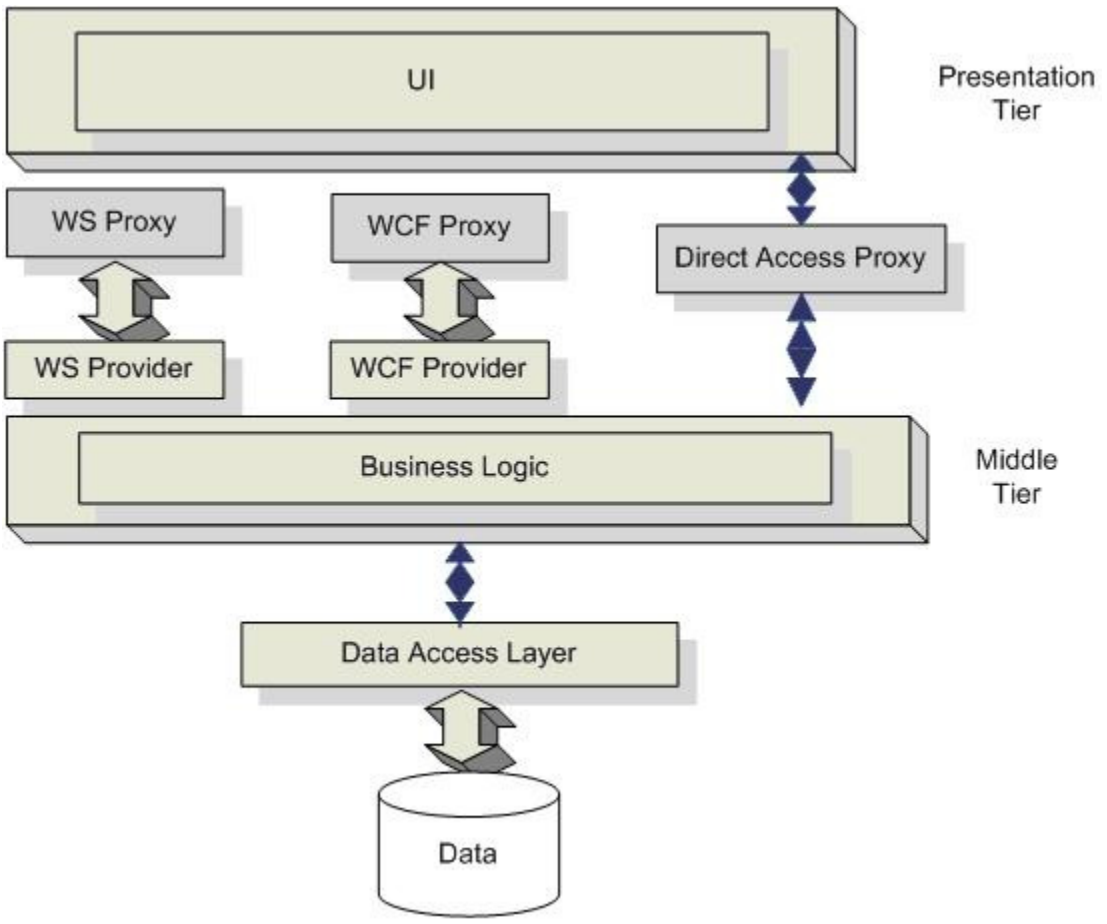

Figure 4. Flexible architecture allows N to change from 3 to 2

Moving Toward a Better Solution

Each of the Presentation and Middle tiers are composed from several logical layers, so we can try to find a solution that will keep the layers apart from each other.

So far, we know the following:

- We need to avoid mixing and matching code from different technologies or bringing code from one layer into another layer. We can do that by abstracting the interfaces between layers, making them unaware of each other's specifics.
- The layers need to exchange data, and this data is the only thing they need to know about each other.

We will start by building a data container layer that has only data types/classes generic enough to go through different layers without affecting the decoupling.

The data containers layer must contain only primitive .NET Framework types, types from System.Data (for example, DataSet), or user-defined types that can contain only the preceding types or other user-defined types that obey these rules.

Abstraction in the Presentation Tier

Question: How can we make the UI layer unaware of the transport type and minimize the changes we need to make to replace a communication layer with another?

Answer: We design and code by interface and use factories to create the concrete classes. The UI "speaks" only the "interfaces language."

Pros and Cons of the Proposed Solution

Pros: Abstracting the communication between tiers provides a system that is easier to change in the future. The changes can be required by the business or they can come as a need to replace an older technology with newer and better technology. We can build a software system with "building blocks" that evolves easier than earlier applications.

Cons: The initial setup requires more work, but that is the same as any approach, such as the Composition UI Application Block or the Smart Client Software Factory. It requires some training and discipline from all the developers. It may require code review once in a while to ensure that no one takes shortcuts and accesses one layer from another by bypassing the Abstract layer.

First Attempt: Factory Method

We try to use the Factory Method pattern to implement the separation between the creation of the concrete classes and the caller.

Factory Method - Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. (Source: Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides)

The code will look like the following:

```
public class SALFactoryMethod
{
```

```
    public enum CommunicationMethod
    {
        WS,
        WCF,
        DirectAccess
    }
    public static IAccountSettings GetAccountSettings(
                                    CommunicationMethod comMethod )
    {
        if( CommunicationMethod.WS == comMethod )
                return new WSAccountSettings();
        else if( CommunicationMethod.WCF == comMethod )
                return new WCFAccountSettings();
        else if( CommunicationMethod.DirectAccess == comMethod )
                return new DirectAccessAccountSettings();
    }
}
```

To use the preceding Factory Method, the client code will call the static method to get the concrete class for each communication method, as shown in the following code.

```
IAccountSettings accSettings =
SALFactoryMethod.GetAccountSettings(
                                    SALFactoryMethod.CommunicationMetho
            d.WS);
```

Cons: We need to pass the Communication Method type each time we need to create an object. This breaks the rule of the open-closed principle (described first by Bertrand Meyer): *"A good design has to assure that the implementation is closed to modification (in ways that affects clients) but open for extension (adaptable)."*

A Better Way: Abstract Factory

Abstract Factory - Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes. (Source: Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides)

The following code shows an example of an abstract class that is also a singleton. Each abstract method will have to be implemented in the concrete factory and return an object that implements the expected interface; in this case, it is **IWrapperAccountInformation**.

```
public abstract class GeneralWrapperFactory
{
      protected GeneralWrapperFactory() { }
      public static GeneralWrapperFactory Instance = null;
      public abstract IWrapperAccountInformation
                              GetWrapperAccountInformation();
```

The following code example illustrates implementation of a concrete factory; in this case, for Web services.

```
public class WSWrapperFactory : GeneralWrapperFactory
{
      public override IWrapperAccountInformation
                              GetWrapperAccountInformation();
      {
            return new WSWrapperAccountInformation();
      }
```

The following code includes an object of the concrete factory to initialize the singleton. This is done only once when the application initializes.

```
* For WS
      GeneralWrapperFactory.Instance = new WSWrapperFactory()
```

```
* For WCF
        GeneralWrapperFactory.Instance = new WCFWrapperFactory()
```

After the instance is initialized with the concrete factory, the client code can start calling the methods that create the objects, as shown in the following code.

```
IWrapperAccountInfomation accInfo =
        GeneralWrapperFactory.Instance.GetWrapperAccountInformatio
        n();
```

Note: The client code is unaware of any kind of underlying technology used, no Web services, no WCF, and no ADO.NET–specific code here!!!

Rules for Building Each Tier/Layer

This section describes the rules for each layer:

- **Data containers layer**. This layer contains classes for data exchange between layers. Examples include **AccountID**, **AccountConfigurationInfo**, and other classes that are now in the **MyCompany.Common.Tools.DataContainer** namespace.
  These classes only depend on .NET Framework primitive types (such as Int32 or String) or other types/classes from this layer.
  These classes must not depend on UI, Web services, ADO.NET, or other layers.
- **Data wrappers layer**. This layer contains classes that encapsulate the ADO.NET calls.
  These classes must not depend on UI, Web services, or other layers except ADO.NET.
- **Abstract communication layer**. This layer contains an abstract layer and an Abstract Factory that may have to be part of a separate assembly/library.
  This layer depends on types defined by the data connections layer and .NET Framework types (such as Int32 or String).
  These classes must not depend on UI, Web services, ADO.NET, or other layers.
  These classes will be sub-classed (inherited) by concrete layers. We could have three separate sub-layers, depending on the communication type:
  - Web services
  - WCF
  - Data access
- **Communication layer (consumer/proxy/reference) - client side**. This layer depends on the data containers layer and abstract communication layer. It also depends on the communication-specific types; for example, if the layer uses Web services, it will depend on Web services–specific types and proxies.
- **Communication layer (provider) - server side**. This layer contains the server-side communication classes and types. For example, if Web services are used, the provider has Web service definition

classes, such as .asmx files.
This layer does not call the data wrappers layer directly; instead, it calls the communication layer that subclasses the abstract communication layer and calls the database through the data wrappers layer. These classes must not depend on UI or ADO.NET.

The data containers layer and the Abstract Factory layer need to be very stable because almost all the components involved will use them; therefore, there will be a lot of coupling with these two.

That is why we must be very careful when choosing the types that make the data containers module. The same holds true for the interfaces exposed by the Abstract Factory layer.

The proposed solution offers the following advantages:

- **It decreases the complexity and the coupling between modules**. This leads to more efficiency during the maintenance of the code.
- **It eases the transition to new technologies**. For example, Web services to WCF, Windows Forms to Windows Presentation Foundation, and ADO.NET to Language Integrated Query (LINQ).
- **It forces the developer to create reusable components instead of monolithic applications**. The explicit separation between layers through the abstraction layer will prevent the developer from taking shortcuts and mixing the code.
- **It allows for independent testing**. Because we separate the logical layers and provide a high degree of decoupling each logical layer/module can be tested independently or reused by other applications.
- **It opens the door for dynamic loading of modules**. This will be addressed in a future article.

Figure 5 illustrates the modules dependency diagram.

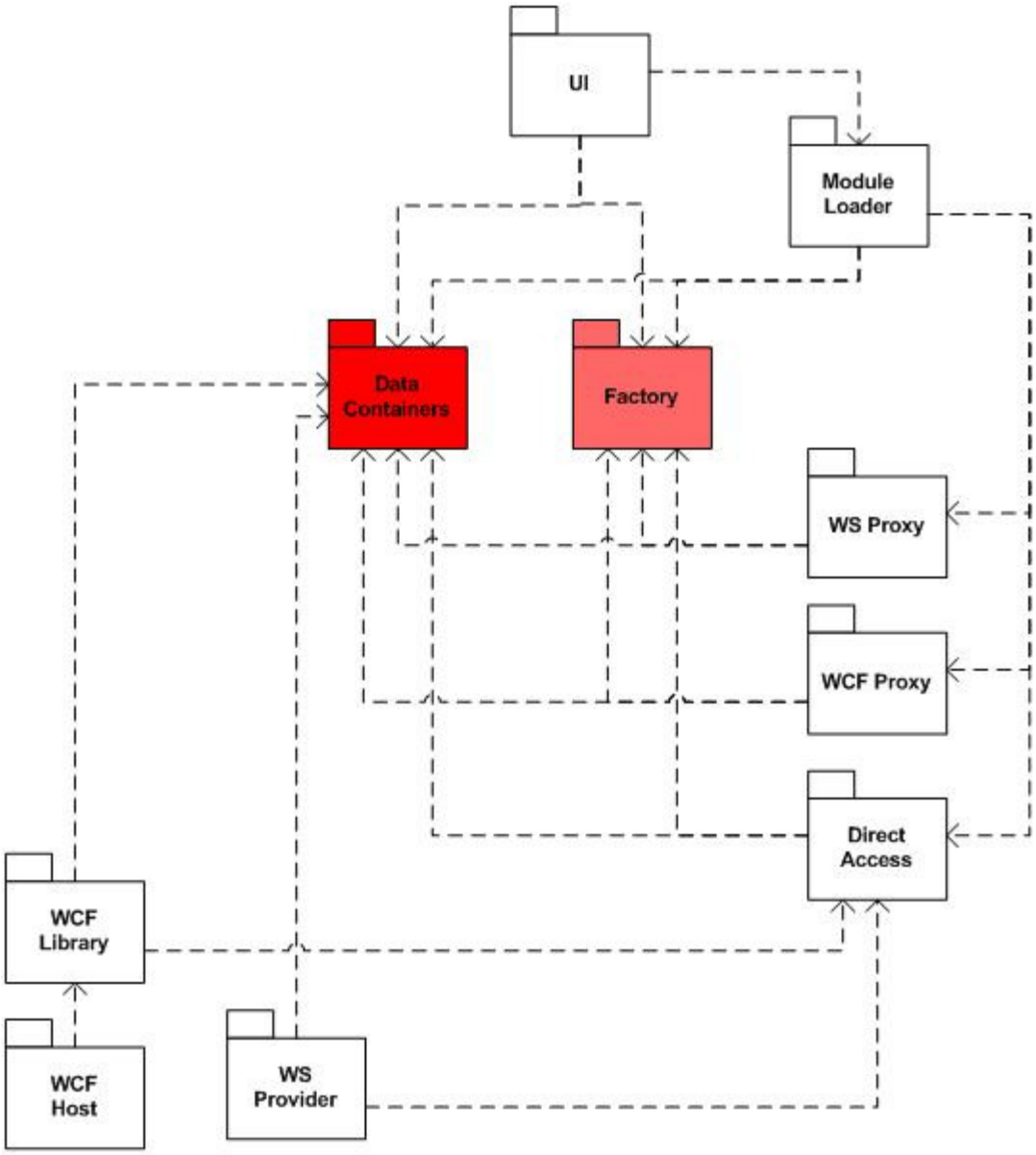

Figure 5. Modules dependency diagram

As we can see in the Figure 5, the UI no longer depends on the proxies/references, and neither depends on the direct access layer.

The Data Containers module and the Factory module have a high degree of coupling because almost all the other modules depend on them. The Data Containers module contains only data; it does not include any code. The Factory module contains only interfaces and abstract classes.

These two modules need to be very stable. Because many other modules depend on them, any change in these two modules may require expensive changes in many places. Therefore, a good

solution is to adopt an old COM practice: after an interface (and a Data Containers type) is published, it does not change; if new features are required, we create new interfaces, add new types, or add data members to the existing types. This way, old code that is already using these two modules does not have to change, and it can coexist with new client code that requires the new features.

The solution presented here was implemented in a real three-tier application that has more than 50,000 lines of code. Although we did not plan from the beginning to have it easily become a two-tier application, we performed the exercise of trying to make it a two-tier application and discovered that it was a job that took less than four hours. The exercise also showed other points of coupling we did not cover in our solution, such as static initializations performed when the ASP.NET Web services application starts.

Proof-of-Concept Prototype

We provide a proof-of-concept prototype that shows the points we made in this article. The simple application has a Windows Forms UI that can connect to the same data source in three different ways: by using Web services, by using WCF, or by direct connection. This change happens at run time by one line of code that just sets the concrete factory in each case.

The prototype can be found here on the CodePlex Web site.

About the Author

Danut Prisacaru is an application architect working in the financial industry. He has more than 17 years of experience in developing software applications. In the last few years, he architected, designed, and implemented several two-tier and three-tier applications using object-oriented design principles, the .NET Framework, SQL Server, the Smart Client Applications, and service-oriented architecture (SOA). Danut likes to find solutions that can make an application easy to maintain. He likes the world of object-oriented design and is also passionate about taking business needs, analyzing them, and turning them into solutions that can improve the efficiency of software applications. His favorite period in world history is Renaissance and his favorite historical figure is Michelangelo di Lodovico Buonarroti Simoni. Danut can be contacted at danut@computer.org.