

ABSTRACT

DATA ACCESS LAYER CODE GENERATOR

GÖKÇE KÜÇÜKEREN

Layering techniques are commonly used in order to supply the operational requirements of today's complex enterprise applications. The codes developed for the Data Access Layer (DAL), which is the foundation of most layered applications, increases the time of projects' development phases, and the bugs in these manually written codes makes the data access unreliable. In this thesis, a tool is developed to generate the required codes for DAL. The automatically generated codes using this tool, Data Access Layer Code Generator, decreases the time wasted for DAL development and makes the data access reliable due to their uniform structure. Data Access Layer Code Generator is able to build the foundation of the layered architectures with its user-friendly interfaces integrated on the development platform and with the functionalities provided to the developers.

ÖZET

VERİ ERİŞİM KATMANI KOD ÜRETİCİ

GÖKÇE KÜÇÜKEREN

Günümüzün kompleks uygulamalarının operasyonel gereksinimlerini karşılamak üzere katmanlama teknikleri yaygın olarak kullanılmaktadır. Katmansal yapıların temelini oluşturan Veri Erişim Katmanı (DAL) için geliştirilen kodlar, projelerin geliştirme sürelerini uzatmakta ve elle yazılan kodlar içerisindeki hatalar, veri erişimini güvensiz kılmaktadır. Bu tezde, DAL için gereksinim duyulan kodları üreten bir araç geliştirilmiştir. Bu Veri Erişim Katmanı Kod Üretici aracı kullanılarak, otomatik olarak üretilen kodlar, DAL geliştirilmesi için harcanan süreleri kısaltmakta ve tek biçimli yapıları sayesinde veri erişimini güvenilir kılmaktadır. Veri Erişim Katmanı Kod Üretici, geliştirme platformuna entegre edilmiş, kullanıcı dostu arayüzleri ve geliştiricilere sağladığı fonksiyonallikler ile katmansal yapıların temelini inşa edecek kabiliyete sahiptir.

ACKNOWLEDGEMENTS

I would like to express my thanks to Prof. Dr. Selahattin Kuru for his comments, help and supervision on the topic; and also I would like to thank all people that support me by giving intelligent ideas, and psychological support.

TABLE OF CONTENTS

ABSTRACT	I
ÖZET	II
ACKNOWLEDGEMENTS	III
TABLE OF CONTENTS.....	IV
LIST OF FIGURES	VI
LIST OF TABLES	VIII
1 INTRODUCTION.....	1
2 CODE GENERATION IN LAYERED DATABASE APPLICATION	3
2.1 Application Layering.....	3
2.2 Code Generation.....	9
2.3 Code Generation For Database Layer	10
3 TECHNOLOGIES USED.....	12
3.1 .Net Framework.....	12
3.1.1 Common Language Runtime.....	12
3.1.2 The .Net Framework Class Library	14
3.2 Overview of C#	15
3.2.1 Classes	16
3.2.2 Attributes And Reflections	17
3.3 Visual Studio 2005	20
3.3.1 Extensibility And Automation in Visual Studio 2005	20
3.4 ADO .Net	24
4 CODED WORK	27
4.1 “DALCodeGeneratorLibrary” Library.....	27
4.1.1 “CSClassGenerator” Class	28
4.1.2 “CSPFClassGenerator” Class	30
4.1.3 “FieldDescription” Class	33
4.1.4 “ClassRelationship” Class	34
4.1.5 “RelationshipFields” Class	34

4.1.6	“CSXMLClassGenerator” Class.....	35
4.2	“DALHelper” Library	35
4.2.1	“BaseDAL” Class.....	36
4.2.2	“DBConnectionHelper” Class	37
4.2.3	“IDBSpecificHandler” Interface.....	39
4.2.4	“DBFieldNameAttribute” Class	40
4.2.5	“DBRelationAttribute” Class.....	40
5	EXAMPLE USE OF PROPOSED TOOL	42
5.1	Presentation Of The Database Used In Demonstration.....	42
5.2	Recommendations To Create DAL Packages	49
5.3	Generation Of DAL Codes.....	50
5.4	Developing An ASP .Net Application	68
5.4.1	Creation Of An ASP .NET Project.....	68
5.4.2	Adding Components To A Web Form.....	69
5.4.3	Preparing Applications For DAL Usage.....	70
5.4.4	Usage Of DAL Classes.....	71
6	CONCLUSSION AND FUTURE RECOMENDATIONS	78
7	REFERENCES.....	79

LIST OF FIGURES

Figure 2.1 General view of a layered application	4
Figure 2.2 OSI 7 Layer Model	5
Figure 2.3 Three Layered Application Architecture	6
Figure 3.1 .Net Framework	14
Figure 5.1 Northwind Database Diagram	44
Figure 5.2 New Database User.....	45
Figure 5.3 New Login Name.....	45
Figure 5.4 SQL Server Login Properties General Tab.....	46
Figure 5.5 SQL Server Login Properties Data Access Tab.....	46
Figure 5.6 Server Explorer Data Connection Menu.....	47
Figure 5.7 Add Connection Wizard	48
Figure 5.8 Add Connection Wizard Advanced Properties.....	48
Figure 5.9 Visual Studio 2005 File Menu	51
Figure 5.10 New Project Wizard.....	51
Figure 5.11 Solution Explorer Menu.....	52
Figure 5.12 Add New Project Wizard.....	53
Figure 5.13 Solution Explorer Menu.....	53
Figure 5.14 Solution Explorer.....	54
Figure 5.15 Solution Explorer.....	54
Figure 5.16 Visual Studio Tools Menu	54
Figure 5.17 DAL Code Generation Wizard	55
Figure 5.18 Connection Test Result For Success.....	55
Figure 5.19 Connection Test Result For Error.....	56
Figure 5.20 DAL Code Generation Wizard Schema Error	56
Figure 5.21 DAL Code Generation Wizard With Loaded Schema.....	57
Figure 5.22 DAL Code Generation Wizard Before Generation Process	60
Figure 5.23 Solution Explorer After Generation.....	61
Figure 5.24 Code Editor After Generation.....	61
Figure 5.25 Solution Explorer After All Generation Process	62
Figure 5.26 Initial View Of Relation Configurator.....	62
Figure 5.27 Relation Configurator For New Relation Definition	63

Figure 5.28 Method To Call Selection Part 1.....	64
Figure 5.29 Method To Call Selection Part 2.....	65
Figure 5.30 Method To Call Selection Parameter Number Error	65
Figure 5.31 Relation Configurator For Update Existent Relations	66
Figure 5.32 Visual Studio 2005 Add New Web Site Wizard.....	69
Figure 5.33 Visual Studio 2005 Toolbox	70
Figure 5.34 Solution Explorer When the Preparation Completed	71
Figure 5.35 Query Section Of Default.aspx.....	72
Figure 5.36 Products Listed On Default.aspx	74
Figure 5.37 The components For Loading The Product	74
Figure 5.38 The Form State After Loading The Product	75
Figure 5.39 The Form State After “Edit Product” Button Pressed	75
Figure 5.40 The Form State For An Insertion.....	76

LIST OF TABLES

Table 3.1 Add-in Installation Locations.....	24
Table 3.2 Data Provider Interfaces.....	26
Table 5.1 Categories Table.....	42
Table 5.2 Supplier Table	43
Table 5.3 Products Table.....	44

1 INTRODUCTION

Today's enterprise applications compose of a large number of components, which makes the systems too complex to comprehend in their entirety. The high complexity of the applications possibly causes serious problems on supporting such operational requirements as maintainability, reusability, scalability, robustness, and security. To deal with these issues, the separation of the components into layers can be the solution. The term, layer, refers to a collection of application components, that focuses on a specific aspect of the application and builds upon the facilities of other layers.

A common practice of layering is dividing the application into three layers, that are the presentation layer, the business layer and the data access layer (DAL). The presentation layer contains elements responsible for providing some form of communication with a human being, such as an element in the user interface. On the other hand, the business layer contains elements responsible for performing some kind of business processing and the application of business rules. Lastly, the data access layer contains elements responsible for providing access to an information source, such as a relational database.

While each layer is responsible of its duty, the communication between the layers is to be provided with the business entities (BE). Business entities are data containers, that carry the necessary data between layers. Each layer accepts an entity, which contains the required data to be processed, and passes the same entity or a new entity to the other, after its process has been completed. For instance, data access components will often return business entities instead of database-specific structures. This helps significantly in isolating database-specific details to the data layer.

In this thesis, we develop and present a tool that helps the developers by building the data access layer in their layered applications. The tool, Data Access Layer Code Generator, is designed to save the time, wasted by developing data access codes, with a generation process provided within a simple wizard. The usage of the tool produces all the required data access functionalities and the business entities of the layer.

A brief description is given below for each achieved work done during the development phase of Data Access Layer Code Generator;

- *The DAL and BE codes generation*

The codes developed for data access purposes may reach thousands of lines, but the tool makes their development process available with a few entries on a wizard.

- *Microsoft Visual Studio 2005 integration*

Data Access Layer Code Generator is a tool integrated on the development platform, Visual Studio 2005. It behaves as a part of the platform, and does not executed externally.

- *DB2 and Microsoft SQL Server support*

The data access support is provided for the two most popular database management systems. The tool's structure is also designed to support more systems with a little modification requirement.

- *Database schemas*

The developers are also able to store the database schemas, which are the attributes of the generated codes. This function decreases the time of the regeneration of the codes, when the database structure is changed.

- *Relationships between generated codes*

Through the tool, the developers are able to relate generated codes to simulate the relationships between databases.

The following chapters will give a detailed information about the key technologies used in development phase, about the builded libraries, and about the usage of the tool.

2 CODE GENERATION IN LAYERED DATABASE APPLICATION

The chapter is prepared to give an overview about the three main concept related to this thesis. The first section informs you about the layering an application and the advantages gained by layering. The second section is an overview of code generation and lists the advantages of the code generation approach. And the last section, Code Generation in Database Application, explains why the database layer is suitable for using code generators.

2.1 Application Layering

Layering is one of the most common techniques that software designers use to break apart a complicated software system. You see it in machine architectures, where layers descend from a programming language with operating system calls into device drivers and CPU instruction sets, and into logic gates inside chips. Networking has FTP layered on top of TCP, which is on top of IP, which is on top of ethernet.

When thinking of a system in terms of layers, you imagine the principal subsystems in the software arranged in some form of layer cake, where each layer rests on a lower layer. In this scheme the higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer. Furthermore, each layer usually hides its lower layers from the layers above, so layer 4 uses the services of layer 3, which uses the services of layer 2, but layer 4 is unaware of layer 2.

Breaking down a system into layers has a number of important benefits.

- You can understand a single layer as a coherent whole without knowing much about the other layers. You can understand how to build an FTP service on top of TCP without knowing the details of how ethernet works.
- You can substitute layers with alternative implementations of the same basic services. An FTP service can run without change over ethernet, PPP, etc..
- You minimize dependencies between layers. If the cable company changes its physical transmission system, providing they make IP work, we don't have to alter our FTP service.
- Layers make good places for standardization. TCP and IP are standards because they define how their layers should operate.

- Once you have a layer built, you can use it for many higher-level services. Thus, TCP/IP is used by FTP, telnet, SSH, and HTTP. Otherwise, all of these higher-level protocols would have to write their own lower-level protocols.

Layering is an important technique, but there are downsides.

- Layers encapsulate some, but not all, things well. As a result you sometimes get cascading changes. The classic example of this in a layered enterprise application is adding a field that needs to display on the UI, must be in the database, and thus must be added to every layer in between.
- Extra layers can harm performance. At every layer things typically need to be transformed from one representation to another. However, the encapsulation of an underlying function often gives you efficiency gains that more than compensate. A layer that controls transactions can be optimized and will then make everything faster.

The number of layers in an application may vary according to the system. A general view of the layer architecture is displayed in Figure 2.1

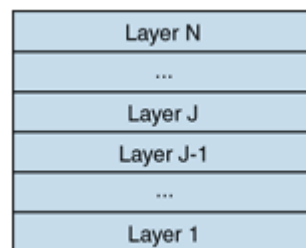


Figure 2.1 General view of a layered application

One of the most familiar models for layering is the OSI 7 Layer Model, defined by the International Organization for Standardization (ISO).

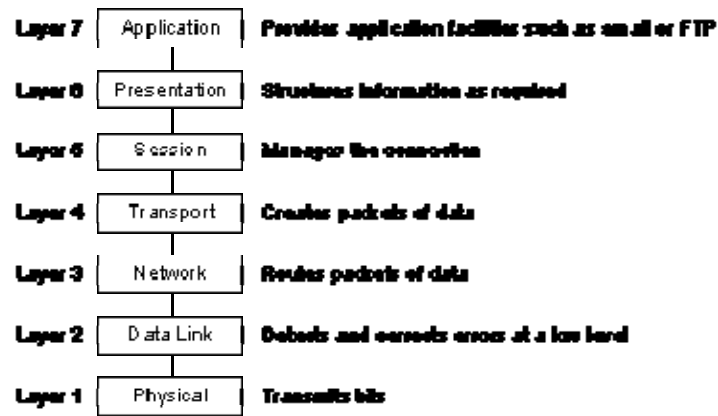


Figure 2.2 OSI 7 Layer Model

Although there is no limit for layering, a common pattern for the enterprise applications has three principle layers.

- *Presentation Layer* :
The presentation layer contains elements responsible for providing some form of communication with a human being, such as an element in the user interface.
- *Business Layer* :
The business layer contains elements responsible for performing some kind of business processing and the application of business rules.
- *Data Access Layer* :
The data access layer contains elements responsible for providing access to an information source, such as a relational database.

The following figure explains dividing an application into three layers. The presentation layer showed with yellow boxes interacts with the user and the layer below that, which is business layer. Business layer with blue colored boxes is the middle layer of the application and contains the business parts, Business Workflow and Business Component, where the business logic is implemented. One more blue box exists in this layer to hold the components, which carries data from data access layer, Business Entities. The green layer, Data Access Layer, is the base of the application that is responsible to collect the data from data sources and services.

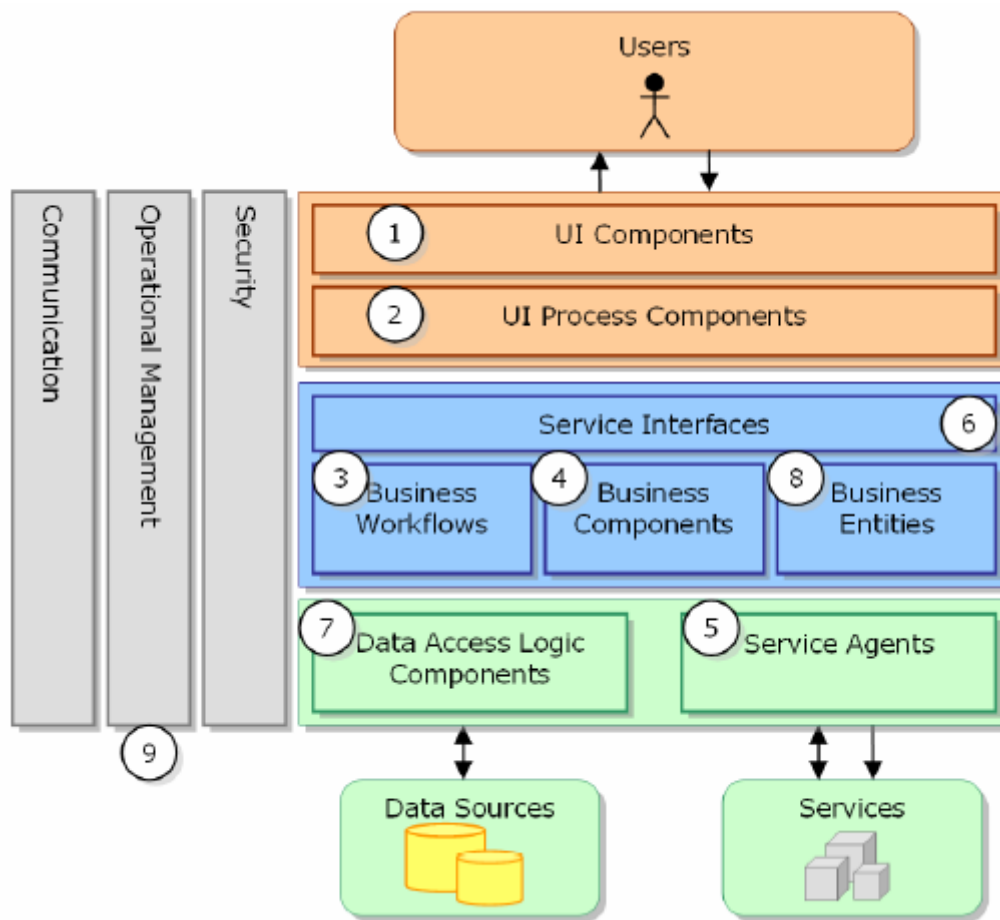


Figure 2.3 Three Layered Application Architecture

The description of the numbered components included in the layers are listed below. Some components may not be exist in every application, but most applications needs these kind of component.

1. *User interface (UI) components :*

Most solutions need to provide a way for users to interact with the application. In the retail application example, a Web site lets customers view products and submit orders, and an application based on the Microsoft Windows operating system lets sales representatives enter order data for customers who have telephoned the company. User interfaces are implemented using Windows Forms, Microsoft ASP.NET pages, controls, or any other technology you use to render and format data for users and to acquire and validate data coming in from them.

2. *User process components :*

In many cases, a user interaction with the system follows a predictable process. For example, in the retail application you could implement a procedure for viewing product data that has the user select a category from a list of available product

categories and then select an individual product in the chosen category to view its details. Similarly, when the user makes a purchase, the interaction follows a predictable process of gathering data from the user, in which the user first supplies details of the products to be purchased, then provides payment details, and then enters delivery details. To help synchronize and orchestrate these user interactions, it can be useful to drive the process using separate user process components. This way the process flow and state management logic is not hard-coded in the user interface elements themselves, and the same basic user interaction “engine” can be reused by multiple user interfaces.

3. *Business workflows* :

After the required data is collected by a user process, the data can be used to perform a business process. For example, after the product, payment, and delivery details are submitted to the retail application, the process of taking payment and arranging delivery can begin. Many business processes involve multiple steps that must be performed in the correct order and orchestrated. For example, the retail system would need to calculate the total value of the order, validate the credit card details, process the credit card payment, and arrange delivery of the goods.

4. *Business components* :

Regardless of whether a business process consists of a single step or an orchestrated workflow, your application will probably require components that implement business rules and perform business tasks. For example, in the retail application, you would need to implement the functionality that calculates the total price of the goods ordered and adds the appropriate delivery charge. Business components implement the business logic of the application.

5. *Service agents* :

When a business component needs to use functionality provided in an external service, you may need to provide some code to manage the semantics of communicating with that particular service. For example, the business components of the retail application described earlier could use a service agent to manage communication with the credit card authorization service, and use a second service agent to handle conversations with the courier service. Service agents isolate the idiosyncrasies of calling diverse services from your application, and can provide additional services, such as basic mapping between the format of the data exposed by the service and the format your application requires.

6. *Service interfaces* :

To expose business logic as a service, you must create service interfaces that support the communication contracts (message-based communication, formats, protocols, security, exceptions, and so on) its different consumers require. For example, the credit card authorization service must expose a service interface that describes the functionality offered by the service and the required communication semantics for calling it.

7. *Data access logic components* :

Most applications and services will need to access a data store at some point during a business process. For example, the retail application needs to retrieve product data from a database to display product details to the user, and it needs to insert order details into the database when a user places an order. It makes sense to abstract the logic necessary to access data in a separate layer of data access logic components. Doing so centralizes data access functionality and makes it easier to configure and maintain.

8. *Business entity components* :

Most applications require data to be passed between components. For example, in the retail application a list of products must be passed from the data access logic components to the user interface components so that the product list can be displayed to the users. The data is used to represent real-world business entities, such as products or orders. The business entities that are used internally in the application are usually data structures, such as DataSets, DataReaders, or Extensible Markup Language (XML) streams, but they can also be implemented using custom object-oriented classes that represent the real-world entities your application has to work with, such as a product or an order.

9. *Components for security, operational management, and communication* :

The application will probably also use components to perform exception management, to authorize users to perform certain tasks, and to communicate with other services and applications.

2.2 Code Generation

Code generation is the use of a program, a code generator, to write your programs for you. There is nothing magical about the code generator itself. Just like any program it takes some input and creates something new as output that is high level code (e.g. C, C++, C#, Java, Perl, Python, Ruby, etc.). These tools range in size and complexity from simple code parsers to fully featured class and layer builders.

Code generation has four key advantages;

- *Quality* :
The code created is of uniform quality across the entire output code base. The higher the quality of the templates the better the resultant code. So investments in the templates are rewarded quickly. In addition, when the templates are changed to fix a bug that fix is propagated by design across all of the maintained code. Because of this, code eneration does not suffer from the maintainability problem of copy and paste coding.
- *Consistency* :
The use of a code generator to build the code means that the design of the interfaces and classes that are produced are completely uniform. This makes writing client code much easier.
- *Productivity* :
Having a generator produce code is undeniably faster than handcoding, but that is not where the productivity gain really lies. Gains become immediately apparent when you can alter your design input and easily generate new code to match the new requirements.
- *Abstraction* :
Perhaps the most important benefit is the abstraction provided by some generator models. When the design of the database access layer is abstracted into an external form e.g. templates, you have extracted the core portion of your business logic the implementation. This means that you are not bound to any particular technology decisions, such as language or platform. This is a tangible portability benefit.

Code generation is often considered just a speed-up tool. It is important to think beyond the speed and about the advantages for all aspects of software design and implementation. With all of these advantages it is hard to imagine the downsides.

2.3 Code Generation For Database Layer

The foundation of most applications is the database access layer, which bundles the queries and other operations performed upon the database. The business objects and the user interface use the database access layer to read and write the database. As with any structure, the strength of an application lies within the strength of its foundation. The more solid and robust the database access layer, the foundation, the stronger the application built on top of it will be.

The database access layer takes responsibility for:

- Marshalling database types and application types
- Wrapping SELECT queries
- Wrapping the INSERT, UPDATE and DELETE operations
- Wrapping any stored procedures, if those are being used
- Doing validations of the arguments being passed to the queries and operations
- Handling errors coming from the database

Most of this code in the database access layer is fairly simple and repetitive. A query method, for example, follows this process: It accepts and validates the parameters for the query. It then establishes the database connection and executes the query. As a final step it places the resultant data in a memory structure.

The types of bugs associated with this type of method are usually argument ordering mismatches and inadequate error handling. Other more disturbing sources of error are copy and paste errors, where fixes are applied inconsistently across a number of methods where the source was literally copied and pasted as an implementation technique. Database access code is particularly vulnerable to these issues because of the volume of critical work required to implement the code. As a result, we would like it to contain zero bugs with uniformly high quality code throughout. The layer should have an interface with predictable naming and

consistent argument ordering. In addition the methods should not contain any surprising behavior.

As a conclusion, due to the critical importance of the codes included in data access layer, and due to their repetitive structure, it is necessary and suitable to create data access layer codes through a code generator, so the resulted codes will be reliable, much more readable and uniform.

This thesis offers a code generator to automatically generate the codes of data access layer, which prevents the bugs produced by developers.

3 TECHNOLOGIES USED

This chapter covers the key technologies used in the development phase of Data Access Layer Code Generator tool. In the following pages, these technologies will be discussed just with their important functionalities that have an important role in building this project.

3.1 .Net Framework

The .NET Framework is an integral Windows component that supports building and running the next generation of applications and XML Web services. The .NET Framework is designed to fulfill the following objectives: [1]

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework has two main components:

- The common language runtime
- The .NET Framework class library

3.1.1 Common Language Runtime

The common language runtime is the foundation of the .NET Framework. You can think of the runtime as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that promote security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the

runtime is known as managed code, while code that does not target the runtime is known as unmanaged code.

With regards to security, managed components are awarded varying degrees of trust, depending on a number of factors that include their origin (such as the Internet, enterprise network, or local computer). This means that a managed component might or might not be able to perform file-access operations, registry-access operations, or other sensitive functions, even if it is being used in the same active application.

The runtime enforces code access security. For example, users can trust that an executable embedded in a Web page can play an animation on screen or sing a song, but cannot access their personal data, file system, or network. The security features of the runtime thus enable legitimate Internet-deployed software to be exceptionally feature rich.

In addition, the managed environment of the runtime eliminates many common software issues. For example, the runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. This automatic memory management resolves the two most common application errors, memory leaks and invalid memory references.

The runtime also accelerates developer productivity. For example, programmers can write applications in their development language of choice, yet take full advantage of the runtime, the class library, and components written in other languages by other developers. Any compiler vendor who chooses to target the runtime can do so. Language compilers that target the .NET Framework make the features of the .NET Framework available to existing code written in that language, greatly easing the migration process for existing applications.

While the runtime is designed for the software of the future, it also supports software of today and yesterday. Interoperability between managed and unmanaged code enables developers to continue to use necessary common object model (COM) components and dynamic link libraries (DLL).

The runtime is designed to enhance performance. Although the common language runtime provides many standard runtime services, managed code is never interpreted. A feature called just-in-time (JIT) compiling enables all managed code to run in the native machine language

of the system on which it is executing. Meanwhile, the memory manager removes the possibilities of fragmented memory and increases memory locality-of-reference to further increase performance. [2]

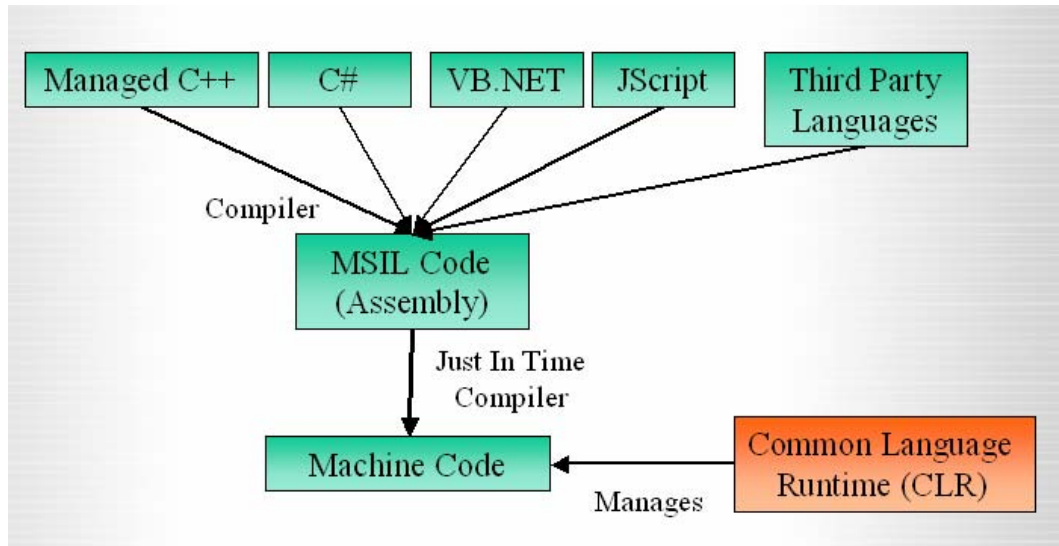


Figure 3.1 .Net Framework

3.1.2 The .Net Framework Class Library

The .NET Framework class library is a collection of reusable types that tightly integrate with the common language runtime. The class library is object oriented, providing types from which your own managed code can derive functionality. This not only makes the .NET Framework types easy to use, but also reduces the time associated with learning new features of the .NET Framework. In addition, third-party components can integrate seamlessly with classes in the .NET Framework.

For example, the .NET Framework collection classes implement a set of interfaces that you can use to develop your own collection classes. Your collection classes will blend seamlessly with the classes in the .NET Framework.

As you would expect from an object-oriented class library, the .NET Framework types enable you to accomplish a range of common programming tasks, including tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized

development scenarios. For example, you can use the .NET Framework to develop the following types of applications and services:

- Console applications.
- Windows GUI applications (Windows Forms).
- ASP.NET applications.
- XML Web services.
- Windows services.

For example, the Windows Forms classes are a comprehensive set of reusable types that vastly simplify Windows Graphical User Interface (GUI) development. If you write an ASP.NET Web Form application, you can use the Web Forms classes.

3.2 Overview of C#

For the past two decades, C and C++ have been the most widely used languages for developing commercial and business software. While both languages provide the programmer with a tremendous amount of fine-grained control, this flexibility comes at a cost to productivity. Compared with a language such as Microsoft Visual Basic, equivalent C and C++ applications often take longer to develop. Due to the complexity and long cycle times associated with these languages, many C and C++ programmers have been searching for a language offering better balance between power and productivity.

The ideal solution for C and C++ programmers would be rapid development combined with the power to access all the functionality of the underlying platform. They want an environment that is completely in sync with emerging Web standards and one that provides easy integration with existing applications. Additionally, C and C++ developers would like the ability to code at a low level when and if the need arises.

The Microsoft solution to this problem is a language called C#. C# is a modern, object-oriented language that enables programmers to quickly build a wide range of applications for the new Microsoft .NET platform, which provides tools and services that fully exploit both computing and communications.

Because of its elegant object-oriented design, C# is a great choice for architecting a wide range of components - from high-level business objects to system-level applications. Using simple C# language constructs, these components can be converted into XML Web services, allowing them to be invoked across the Internet, from any language running on any operating system.

More than anything else, C# is designed to bring rapid development to the C++ programmer without sacrificing the power and control that have been a hallmark of C and C++. Because of this heritage, C# has a high degree of fidelity with C and C++. Developers familiar with these languages can quickly become productive in C#. [3]

3.2.1 Classes

Classes are types, but are far more powerful than the simple types like int and float. Not only you can customize your data storage using classes, but you can also add methods to classes. That kind of compartmentalization, where data and methods are rolled up into a single class, is the entire reason that object oriented programming (OOP) was introduced in the first place. It enables the programmers to deal with larger programs. The process of wrapping related data and methods into a class (and so preventing them from cluttering up the rest of the program) to create a single entity is called encapsulation.

Classes enable you to develop applications using OOP techniques . Classes are templates that define objects. When you create a new form in a C# project, you are actually creating a class that defines a form; forms instantiated at runtime are derived from the class. Using objects derived from predefined classes, such as a C# Form class, is just the start of enjoying the benefits of object-oriented programming—to truly realize the benefits of OOP, you must create your own classes.

The philosophy of programming with classes is considerably different from that of traditional programming. Proper class-programming techniques can make your programs better, both in structure and in reliability. Class programming forces you to consider the logistics of your code and data more thoroughly, causing you to create more reusable and extensible object-based code.

Classes consist of members. Everything defined within the class is considered to be a member of that class. The list of the various types, that can be defined as members of a C# class, comes next.

- *Fields :*
A field is a member variable used to hold a value.
- *Methods :*
A method is the actual code that acts on the objects's data (or field value).
- *Properties :*
A property is actually a method, that looks like a field to the class's clients. The properties allows the client a greater degree of abstraction, because the client does not know whether it is accessing the field directly or whether an accessor method is being called.
- *Constants :*
As the name suggests, a constant is a field with a value that can not be changed.

3.2.2 Attributes And Reflections

Attributes provide a powerful method of associating declarative information with C# code (types, methods, properties, and so forth). Once associated with a program entity, the attribute can be queried at run time using a technique called Reflection. [4]

The attributes can be used to define design-time information (such as documentation), run-time information (such as the name of a database column for a field), or even run-time behavioral characteristics. The possibilities are endless, which is the point. In as sense, this association of information follows the same principles used in the development of extensible markup language (XML). Because you can create an attribute based on any information you want, a standard mechanism exists for defining the attributes themselves and for querying the member or type at run time about its attached attributes.

Attributes exist in two forms: attributes that are defined in the Common Language Runtime's base class library and custom attributes, that you can create, to add extra information to your code. This information can later be retrieved programmatically. [5]

An example will better illustrate how to use this powerful feature. Let assume that we have a class, whose fields hold the values of a record in a database table, and we need to keep the features of these columns for some reason. One practical way to deal with this requirement is attaching an attribute to each field in the class.

The following code is the illustration of an attribute attachment.

```
[DBFieldName("ProductID", true, false, true, DbType.Int32)]  
private int productID;
```

The field, "productID", has an attribute which holds its corresponding column features. Below, you can see the attribute class, "DBFieldName", that explains which features are kept as metadata of the field.

```
public class DBFieldNameAttribute : Attribute  
{  
    private string fieldName;  
    public string FieldName  
    {  
        get  
        {  
            return fieldName;  
        }  
    }  
  
    private bool isKey;  
    public bool IsKey  
    {  
        get  
        {  
            return isKey;  
        }  
    }  
  
    private bool isDecimal;  
    public bool IsDecimal  
    {  
        get  
        {  
            return isDecimal;  
        }  
    }  
  
    private bool isAutoIncrement;  
    public bool IsAutoIncrement  
    {  
        get  
        {  
            return isAutoIncrement;  
        }  
    }  
}
```

```

private DbType fieldDbType;
public DbType FieldDbType
{
    get
    {
        return fieldDbType;
    }
}

public DBFieldNameAttribute(string p_FieldName, bool p_IsKey, bool
p_IsDecimal, bool p_IsAutoIncrement, DbType p_DbType)
{
    this.fieldName = p_FieldName;
    this.isKey = p_IsKey;
    this.isDecimal = p_IsDecimal;
    this.isAutoIncrement = p_IsAutoIncrement;
    this.fieldDbType = p_DbType;
}
}

```

The code above is the content of `DBFieldNameAttribute` class. The properties of this class are explained below.

- *FieldName* stands for the corresponding column's name.
- *IsKey* indicates whether the column is a key of the database table or not.
- *IsDecimal* holds the metadata indicating whether the column's type is decimal or not.
- *IsAutoIncrement* holds the metadata indicating whether the column is an identity column or not.
- *FieldDbType* holds the type of the column.

Until now, we only see how to define an attribute by deriving it from `System.Attribute`, and how to attach it to a type or member. The answer of the question, how we can use attributes in code, is Reflections.

Reflection is the process by which a program can read its own metadata. A program is said to reflect on itself, extracting metadata from its assembly and using that metadata either to inform the user or to modify its own behavior. In other words, it is the way to query a type or member about its attached attributes. [5]

In the previous example, we defined the `DBFieldNameAttribute` attribute. Now we will see, how these attributes can be reached with a simple routine. The following lines of code gets the type of an object and finds the fields defined in this type. After that, it gets the custom

attributes of each field and checks, if the attribute is *DBFieldNameAttribute*. Finally, if any attribute is found in our type, then the place comes to execute any operation using the metadata.

```
Type t = myObject.GetType();
foreach (FieldInfo fi in t.GetFields(BindingFlags.NonPublic |
BindingFlags.Instance))
{
    foreach (Object ca in fi.GetCustomAttributes(false))
    {
        if (myAttribute is DBFieldNameAttribute)
        {
            //Here is the place, where the metadata of a field is reached.
        }
    }
}
}
```

3.3 Visual Studio 2005

Microsoft Visual Studio is a complete set of development tools for building ASP.NET Web applications, XML Web services, desktop applications, and mobile applications. Visual Basic, Visual C++, Visual C#, and Visual J# all use the same integrated development environment (IDE), which allows them to share tools and facilitates in the creation of mixed-language solutions.

3.3.1 Extensibility And Automation in Visual Studio 2005

Visual Studio features a number of targeted, programmable object models. By using these models, you can access the underlying components and events inside the Visual Studio integrated development environment (IDE) and its projects. Each model contains types and members that represent solutions, projects, tool windows, code editors, debuggers, code objects, documents, events, and more. Consequently, you can extend the functionality of the IDE, automate repetitive tasks, and integrate other applications into the IDE. The object models can be accessed through any of four methods: macros, add-ins, wizards, and the Visual Studio Industry Partner (VSIP) program.

This is an important part of Visual Studio because it lets developers easily tailor the tool to their personal working style and enables them to accommodate team practices. You can simply capture several steps in a regular process (for example, for check-ins, creating new

projects or forms, or updating code) and make that process available as a single command to invoke. Independent software vendors can implement entirely new features (including groupware, profiling tools, work flow, or life-cycle tools) that fit into Visual Studio 2005 as seamlessly as if they were built into the shipping Visual Studio product. [6]

With this automation model, you can do the following:

- Intercept commands when they are invoked, and either provide preprocessing or implement the command yourself.
- Control the solution, projects, and project items by adding or removing them.
- Control the build configurations mechanisms and hook various build events.
- Control a large portion of the text editor.
- Implement commands that help you debug with the debugger objects.
- Control the Windows Forms Designer.
- Create tool windows that behave just like the built-in tool windows for docking and floating.
- Provide content to the Property Browser when items are selected in tool windows.
- Control several of the built-in tool windows (including Task List, Toolbox, Command Window, and Output Window).

Add-in Creation in Visual Studio 2005

An add-in is a tool that you create programmatically by using objects, methods, properties, collections in .NET's extensibility model. This compiled application enables you to automate the difficult and tedious tasks within the visual studio integrated development environment (IDE). These tasks can be accomplished in response to an event, such as the mouse being clicked, forms being added to the project or control being added to the form. The actions may not be visible to the developer. An add-in extends the functionality of the Visual Studio IDE. Extensibility is the mechanism exposed to the developer that provides the ability to enhance and extend the functionality of the IDE. It basically exposed IDE's internal functions to add-in developer. [7]

Add-in's can be invoked in many ways, such as

- Through Add-in Manager
- Toolbars command or buttons

- Development environment (devenv) Command Line
- Events such as IDE start-up

An add-in is a compiled DLL that runs inside the Visual Studio integrated development environment (IDE). The fact that it is compiled protects your intellectual property and improves performance. While you can create add-ins manually, it is far easier to use the Add-In Wizard. The Add-In Wizard creates an add-in with a fully functional but basic framework that you can run immediately after you create it. After the Add-In Wizard generates the basic framework, you can add code to it and customize it. [8]

The Add-In Wizard can be started by completing the following steps.

- Open new project and select Other Projects - Extensibility Projects in Project Types and select Visual-Studio .Net add-in in templates.
- Enter Name of the Add-in Project and select the location where you want you add-in to be placed using “Browse” button.
- After clicking ok, you will get Add-in Welcome wizard.

The Add-In Wizard lets you supply a display name and description for the add-in, both of which appear in the Add-In Manager dialog box. Optionally, you can choose to have the wizard generate code that adds a command to the Tools menu to load and invoke the add-in. You can also choose to display a custom About Box for your add-in. When the wizard completes, you have a new project with a single class named “Connect” that implements the add-in. The project includes also a file with the “.AddIn” extension to be used in Add-in registration.

The “Connect” class created by the wizard has methods that are used in the project and briefly explained below.

- “*Connect()*” constructor :
The constructor of the class, that is called first when the class is initialized.
- “*OnConnection()*” method :
OnConnection method is the method that is first called by the IDE when it starts the add-in. It is the obvious point to place your UI (menus, toolbars, tool buttons) through which the user will communicate to the add-in. This method is also the place to put

your validation code if you are licensing the add-in. You would normally do this before putting up the UI. If the user is not a valid user, you would not want to put the UI into the IDE.

- *“OnDisconnection()” method :*
This event occurs when the add-in is unloaded.
- *“QueryStatus()” method :*
When the user clicks a command (menu or tool button), the QueryStatus event is fired. The QueryStatus event returns the current status of the specified named command, whether it is enabled, disabled, or hidden in the vsCommandStatus parameter, which is passed to the event by reference.
- *“Exec()” method :*
The Exec event is fired after the QueryStatus event is fired, assuming that the return to the statusOption parameter of QueryStatus is supported and enabled. This is the event where you place the actual code for handling the response to the user click on the command.

Add-in Registration in Visual Studio 2005

After an add-in is created, you must register it with Visual Studio before it becomes available for activation in the Add-In Manager. This was done in previous versions of Visual Studio by using registry keys, but this is now accomplished by using an XML file, that is created by the Add-In Wizard with an extension “.AddIn”.

You no longer need to register the .NET assemblies with Windows. Instead, you simply place the assembly .DLL file into a specific directory along with an XML file that has an .Addin file extension. This XML file describes the information that Visual Studio requires to display the add-in in the Add-In Manager. When Visual Studio starts, it looks in the .Addin File location for any available .Addin files. If it finds any, it reads the XML file and provides the Add-In Manager with the information needed to start the add-in when it is clicked.

The locations, where the .AddIn XML file should be located to be processed by Visual Studio 2005, is listed below.

Table 3.1 Add-in Installation Locations

.Addin File Location	.DLL File Location	Description
Add-in folder (for example, \Documents and Settings\All Users\My Documents\Visual Studio 2005\Addins) -or- (\Documents and Settings\ <user name="">\My Documents\Visual Studio 2005\Addins)</user>	Project debug folder (for example, \My Documents\Visual Studio Projects\MyAddin1\MyAddin1\bin)	Used for running the add-in in the debugging environment. Should always point to the output path of the current build configuration.
Root project folder (for example, \My Documents\Visual Studio\Projects\MyAddin1)	Local path (MyAddin1.dll)	Used for deployment of the add-in project. It is included in the project for ease of editing and is set up with the local path for XCopy-style deployment.

3.4 ADO .Net

ADO.NET is the data access model for .NET-based applications, that is an object-oriented set of libraries that allows you to interact with data sources. Commonly, the data source is a data base, but it could also be a text file, an Excel spread sheet, or an XML file.

Although ADO.NET allows us to interact with different types of data sources and different types of data bases, there is not a single set of classes that allow you to accomplish this universally. Since different data sources expose different protocols, we need a way to communicate with the right data source using the right protocol. Some older data sources use the Open Database Connectivity (ODBC) protocol, many newer data sources use the Object Linking and Embedding Database (OLE DB) protocol, and there are more data sources every day that allow you to communicate with them directly through .NET ADO.NET class libraries. [9]

ADO.NET provides a relatively common way to interact with data sources, but comes in different sets of libraries for each way you can talk to a data source. These libraries are called Data Providers and are usually named for the protocol or data source type they allow you to interact with. Currently, ADO.NET ships with two categories of providers: bridge providers and native providers. Bridge providers, such as those supplied for OLE DB and ODBC, allow you to use data libraries designed for earlier data access technologies. Native providers, such as the SQL Server and Oracle providers, typically offer performance improvements due, in part, to the fact that there is one less layer of abstraction. [10] The available data providers in ADO.NET are listed below.

- *The SQL Server .NET Data Provider :*

This is a provider for Microsoft SQL Server 7.0 and later databases. It is optimized for accessing SQL Server, and it communicates directly with SQL Server by using the native data transfer protocol of SQL Server.

- *The Oracle .NET Data Provider :*

The .NET Framework Data Provider for Oracle enables data access to Oracle data sources through Oracle client connectivity software. The data provider supports Oracle client software version 8.1.7 and later.

- *The OLE DB .NET Data Provider :*

This is a managed provider for OLE DB data sources. It is slightly less efficient than the SQL Server .NET Data Provider, because it calls through the OLE DB layer when communicating with the database.

- *The ODBC .NET Data Provider :*

The .NET Framework Data Provider for ODBC uses native ODBC Driver Manager (DM) to enable data access by means of COM interoperability.

All the data providers listed above implement the interfaces provided by the System.Data namespace. These interfaces makes it possible to program provider independent data access layer codes. Next table shows the interfaces implemented by the providers. [11]

Table 3.2 Data Provider Interfaces

Interface	Description
IDbConnection	Represents a connection to the database.
IDbDataAdapter	Represents a set of command related properties that are used to work with a DataSet.
IDbCommand	Represents a SQL statement (or command) to be executed against a data source.
IDataParameter	Represents a parameter to the command object.
IDataReader	Represents one or more result sets which can be accessed in read only, forward only manner.
IDbTransaction	Represents a transaction to be performed against a data source.

4 CODED WORK

During the development phase of Data Access Layer Code Generator, some code libraries have been build for different purposes. While writing each line of code, it has been taken care of their readabilities. In other words, an effort has been given to make the codes as clear as a developer can easily understand, when he/she goes into it. To increase the readability of the code, most of the properties, the methods and the classes have been also commented. Despite of the clarity of the codes, this section was prepared to explain the functionalities of the classes and the relationships between these classes, that has been developed for this project.

DAL Code Generator libraries can be divided into two parts according to their service levels. The first and the main function of the tool is to provide an interface, that accepts the attribute entry of the generated classes, and that generates these classes. The second service level contains the functions, that helps the first level, the generated classes and the developer by accessing the desired data. These parts are listed below respectively.

- DALCodeGeneratorLibrary
- DALHelper

On the rest of this chapter, these code libraries will be discusses deeply.

4.1 “DALCodeGeneratorLibrary” Library

This library is used by the DAL Code Generator add-in project, that provides an user interface for the generation process. Although this library is allmost loose coupled, it also uses DALHelper library for database accesses. The main responsibilities of this class collection is listed below.

- Getting and holding the schema of the specified database table. A schema is the collection of the properties of the table. For example, columns of the table, types of its columns, corresponding type of the columns in C#, etc. are included in as schema.
- Provide methods to modify the fields of the schema, each of which is correspond to a column of the table.
- Provide classes and methods to define relationships between generated classes.
- Loading the schema back from XML files, that are stored by the add-in.
- Includes the templates of the DAL classes, that are used during the generation proces.

- Generation of the DAL class and BE class.

The library has 6 main classes to provide these functionalities, that are to be known.

- “CSCClassGenerator” class.
- “CSPFClassGenerator” class.
- “FieldDescription” class.
- “ClassRelationships” class.
- “RelationshipsFields” class.
- “CSXMLClassGenerator” class.

4.1.1 “CSCClassGenerator” Class

As its name explains, this class is responsible to generate classes. Actually, the class is the base class of the other class generators, that expand and customize its content by inheriting CSCClassGenerator. In the scope of this project, just one inherited generator class is developed, CSPFClassGenerator, but CSCClassGenerator has been designed as basic as possible for future developments. It only contains main properties, that a generator class should have, and it includes virtual methods, that an inherited class has to implement.

Moreover, this is the class, whose instance holds the schema info of the related table and the relationship definitions. The variables, properties and methods, that are necessary to be known are explained below.

- “*CSCClassGenerator()*” *constructor* :
The constructor of the class. Although it does not contain any code, all inherited classes should call it for future modifications.
- “*fields*” *variable* :
The protected variable is an arraylist, that holds the fields of the schema as FieldDescription objects.
- “*UpdateField()*” *method* :
The method enables the modification on the attributes of fields.
- “*DeleteField()*” *method* :
The method deletes a field from the field list.
- “*buffer*” *variable* :

The private variable, that holds the generated code.

- *“Relations” property :*

The property that holds all the relation definitions. This hashtable contains ClassRelationship objects with the key of the names of the relations.

- *“AddRelation()” method :*

The method adds a new relation definition to the relation list.

- *“RemoveRelation()” method :*

The method deletes an existent relation from the relation list.

- *“ClassNamespace” property :*

The public variable holds the namespace of the generated class.

- *“ClassName” property :*

The public variable holds the name of the generated class.

- *“CreateClass()” method :*

The method fills the buffer with the code of the generated class by calling the virtual methods, that may be implemented by the inherited generator classes. After the creation of the DAL class is completed, the code in the buffer is returned by this method.

- *“WriteUsingLines()” virtual method :*

The virtual method adds the “using” statements to the buffer.

- *“WriteFields()” virtual method :*

The virtual method adds the property codes, each of which is to hold the values of the database columns (fields).

- *“WriteRelations()” virtual method :*

The virtual method adds the property codes, each of which is to hold an object of the child class of the relation.

- *“WriteConstructors()” virtual method :*

The virtual method adds the constructors’ code of the generated class. The number of constructors may vary, so these codes should also implemented by the inherited generator classes.

- *“WriteOtherMethods” virtual methods :*

The virtual method generates the methods’ code blocks, that are special to the generator classes.

4.1.2 “CSPFClassGenerator” Class

This class is derived from CClassGenerator, so it includes the content of CClassGenerator and extends the capabilities to generate classes from a given physical file in a database to manage the data access to this file. This generator class prepares two types of class. First type is the Data Access Layer class, and the Business Entity class is generated as the second type, that is derived from the DAL class.

The generation process starts by calling “CreateClass()” method of the base class, CClassGenerator, and it ends after calling the implemented virtual methods. In the following lines, these implemented methods and their results will be discussed.

- *“CSPFClassGenerator()” constructor :*
The constructor takes two arguments. First argument indicates on which database the physical file locates. The name of the file is taken as the second argument. After assigning its arguments to the related properties, it calls LoadSchema() method.
- *“Database” property :*
The property holds the database definition, which is assigned by the constructor.
- *“FileName” property :*
The property holds the name of the physical file, which is assigned by the constructor.
- *“ClassBECODE” property :*
The property, that is initially empty, contains the code generated for the business entity class.
- *“LoadSchema()” method :*
The method is responsible to get the column info of the file. It queries the specified database, and fills the “fields” list with the result set that comes from the execution of the query. As you remember, “fields” is the protected variable on the base class. If the method does not find the specified file on the database, it throws PossibleFileNotFoundException().
- *“WriteUsingLines()” override method :*
The method adds the first lines of the generated class, that are “using” statements.
- *“WriteFields” override method :*
The method adds a variable, a property and attributes for each field of the schema. The example given below shows the code generated for a field, that corresponds to the ProductID field in the database.

```
[DBFieldName("ProductID", true, false, true, DbType.Int32)]
private int productID;
private FieldRunTimeInfo _productID;
public int ProductID
{
    get
    {
        return productID;
    }
    set
    {
        UpdateState(ref _productID);
        productID = value;
    }
}
```

- “WriteConstructors()” *override method* :

The method adds the constructors’ code of the generated class. It adds five constructors for a DAL class for different usages, whose templates are given below with an example of a file with two key fields. Let assume that the class name is “Classname” and the names of the keyfiels are “Keyname1” and “Keyname2”.

```
public ClassnameDAL() {...}
public ClassnameDAL(IDataReader p_Reader) {...}
public ClassnameDAL(key1type p_Keyname1, key2type p_Keyname2) {...}
public ClassnameDAL(IDbConnection p_Connection, key1type p_Keyname1,
key2type p_Keyname2) {...}
public ClassnameDAL(IDbTransaction p_Transaction, key1type p_Keyname1,
key2type p_Keyname2) {...}
```

- “WriteOtherMethods()” *override method* :

The method that calls the methods, that are specific to this class. These called methods are listed below in the call order, each of which adds different code blocks to the generated class.

- CreateArrayListMethods()
- CreateStaticArrayListAllRecordsMethod()
- CreateStaticArrayListAllRecordsWithWhereMethod()
- CreateStaticArrayListAllRecordsWithWhereMethodAndConnection()
- CreateSearchMethods()

- *CreateArrayListMethods() method :*

The method adds some functions to the generated class, that runs a SELECT query on the database and returns the results in an arraylist, which contains generated BE objects. The condition of the query is determined inside the methods using the given parameters. It adds N-1 methods to the class, where N is the number of the key fields in the schema. For a schema having three key fields, two methods will be generated.

```
public ArrayList LoadAll_Key1(int p_Key1){...}
public ArrayList LoadAll_Key1_Key2(int p_Key1, string p_Key2){...}
```

- *“CreateStaticArrayListAllRecordsMethod()” method:*

The methods adds a single method, Load(), to the generated code, which returns an arraylist including all the records of the related table.

```
public override ArrayList LoadAll(){...}
```

- *“CreateStaticArrayListAllRecordsWithWhereMethod()” method:*

The method adds a method to the code, which takes a condition and queries the database using this where clause. The obtained result is returned by the generated method in an arraylist.

```
public override ArrayList LoadAll(string p_Where, int p_MaxRecord){...}
```

- *“CreateStaticArrayListAllRecordsWithWhereAndConnectionMethod()” method:*

This method adds a very similar code to the method above, except that its generated routine accept an database connection and queries the database over this connection.

```
public ArrayList LoadAll(IDbConnection p_Connection, string p_Where, int
p_MaxRecord){...}
```

- *“CreateSearchMethods()” method :*

The method adds two FindFirst() methods to the generated code. The added methods queries the database as the LoadAll() methods do, but FindFirst() methods returns just the first found record in the database.


```
public BEObject FindFirst(string p_Where){...}
public BEObject FindFirst(IDbConnection p_Connection, string p_Where){...}
```

4.1.3 “FieldDescription” Class

This class is designed to hold all the necessary data about a field of a database column. The instances of this class are initialized in the LoadSchema() method of CSPFClassGenerator, because they are the main elements of the table schemas. Although it is basically the container of column attributes, the decision about the .Net type of the field is taken here. This type will also be the types of the related local variable and the related property in the generated class.

The important parts of the class structure are listed below.

- “FieldDescription()” constructor :

The constructor assigns the variables of the class.

- “FieldDbType” property:

The property holds the type of the column on the database.

- “FieldType” property:

The property holds the converted .Net type of the field.

- “Name” property :

The property holds the column name in the database file.

- “FieldName” property :

The property holds the name of the field, which will be the name of a property in the generated class.

- “IsKey” property :

The property indicates whether the field is a key of the database table.

- “IsAutoIncrement” property :

The property indicates whether the field is an autoincrement column, in other words identity column, or not.

- “IsDecimal” property :

The property indicates whether the corresponding column is decimal or not.

4.1.4 “ClassRelationship” Class

DALCodeGeneratorLibrary provides a class to define a relationship between the generated class and the class, that is already generated. Basically, a relation means that the parent class has a property, that returns an instance or a list of instances of the child class by calling a method of the child class. If the relation type is 1 to 1, than the property returns just one instance of the child class, but in a 1 to N relationship, the property returns an arraylist.

The following list describes the necessary structure to define a relationship.

- “*ClassRelationship()*” *constructor* :
The constructor initializes the object, by the way, assigns the local variables.
- “*Name*” *property* :
The name of the relationship, that is also the name of the discussed property in the main class.
- “*ChildClassName*” *property* :
The name of the child class.
- “*Relationship*” *property* :
The property holds the type of the relation. It can be 1 to1 or 1 to N.
- “*MethodToCall*” *property* :
The method name of the child class, which is called in the main’s property to realize the relationship.
- “*MethodParameters*” *property* :
The property holds the RelationshipFields objects, that are basically the mapping of which variable of the main class is passed to MethodToCall as which method argument.

4.1.5 “RelationshipFields” Class

As dicussed above, a child’s method is called by the parent class to realize a relationship. If the method has any arguments, that are to be passed by calling it, we should also specify which variables in the parent class are used for each argument. The specification is made by RelationshipFields objects.

- “*RelationshipFields*” *constructor* :
The constructor of the RelationshipFields class.
- “*ParentField*” *property* :

The property holds the parent field name, that located in the parent class.

- “*ChildField*” property :

The property holds the name of the method parameter.

4.1.6 “CSXMLClassGenerator” Class

This class is a derivation of CSPFClassGenerator. It expands its base class with a single method, LoadFieldsFromXML(), that provides to load its attributes from a XML file.

4.2 “DALHelper” Library

This library is designed to help the data access, so it is used by the elements of data access layer. DALHelper contains a collection of classes, each of which provides methods to manage the interactions with the databases. The main responsibilities of this class collection is listed below.

- Contains the base class of the generated DAL classes.
- Contains the definitions for the production databases, that are permitted to access.
- Contains the definitions for the test databases, that are permitted to access.
- Handling the behaviors, that are specific to the database providers.
- Managing database connections and transactions.
- Managing the Create, Read, Update and Delete (CRUD) operation for the generated DAL classes.

The list of the classes, included in this library, to implement the responsibilities is as the following.

- BaseDAL class.
- DBConnectionHelper class.
- The classes, that implements IDBSpecificHandler interface.
 - SQLClientHandler class.
 - OLEDBHandler class.
 - MSDB2Handler class.
- DBFieldNameAttribute class.
- DBRelationAttribute class.

These classes will be discussed deeply in the following pages.

4.2.1 “BaseDAL” Class

This class is the base of the generated classes. Its main responsibility is to contain the methods, that every DAL class should have. Of course, these mandatory methods is about dealing with the CRUD operations. As a result of being inherited BaseDAL by DAL classes, the critical operations are centralized, and it is provided, that the management of the code is made in one place. The variables, properties and methods, that are necessary to be known are explained below.

- *“Load()” methods :*
The library has four overloaded methods, each of which is responsible to query the database and to assign the fields with the related column values of the found records.
- *“Insert()” methods :*
Eight overloaded methods are to insert a new record to the database. Each overload applies different insertion operations.
- *“Update()” methods :*
The methods are to update an existent record on the database. Four overloaded methods are provided.
- *“Delete()” methods :*
An existent record can be deleted using on of the three overloaded Delete() methods.
- *“Save()” methods :*
The methods insert or update a record. The decision, whether the record is inserted or updated, is taken in the methods according to the state of BaseDAL object.
- *“CurrentState” property :*
The property holds the state of the BaseDAL object. An object can be in Empty, Loaded, Inserted or Modified state.
- *“IsAutoIncrementFieldExists” property :*
The property is the flag indicating whether the database file has an identity column or not.

- *“AutoRefreshDataAfterInsert” property :*
The property is the flag indicating whether the object should be loaded after an insertion. If its value is true, a select query will be executed after the insertion, so it affects the performance.
- *“InternalVersion” property :*
The property holds the version of the DAL class.
- *“DBFileName” property :*
The property holds the name of the database file, on which CRUD operations occur.

4.2.2 “DBConnectionHelper” Class

This class is designed to help the generated classes and the developer by writing their custom DAL classes. The main purpose of DBConnectionHelper is to provide the methods, that contains the routines to establish a database connection, to get the result sets and to close the connection. The definitions for each permitted database are also located here. The properties and the methods to accomplish the functionalities are listed in the following.

- *“DBConnectionHelper()” constructor :*
The constructor of the class takes the destination database as parameter. The passed argument is assigned to ConnectionType property.
- *“ConnectionType” property :*
The property holds the index of the database definition list. During the operations, this index will be used to get the appropriate connection provider, connection string, etc.
- *“Connections” variable :*
The variable holds the array of providers defined for each database.
- *“ConnectionStrings” variable :*
The variable holds the array of connection strings defined for each production database.
- *“TestConnectionStrings” variable :*
The variable holds the array of connection strings defined for each test database.
- *“DBConnectionHandlers” variable :*
The variable holds the array of specific handlers.
- *“DBHandler” variable :*
The variable holds the appropriate specific handler for the destination database.

- *“Test” property :*
This property holds whether the connection will be established with the test database or with the production database.
- *“productionWebServers” variable :*
The variable holds the list of the production server names. The list is for the web applications. If the web server is in the list, Test property will be set to false automatically.
- *“GetConnection()” method :*
The method opens a connection with the destination database using the appropriate connection string and database provider. It check Test property and decides whether the connection will be established with the production database or with the test database.
- *“BeginTransaction()” method :*
The method begins a transaction with the destination database.
- *“GetCommand()” method :*
The method creates a IDbCommand object and returns it.
- *“CreateCommandParameter()” method :*
The methods adds the specified parameter to the specified command object.
- *“GetReader()” method :*
The methods returns an IDataReader object, which is obtained after creating a command and executing the command.
- *“CloseConnection()” method :*
The method closes and disposes the specified connection object. Before these actions, the object is also checked if it is null or not.
- *“CloseReader()” method :*
The method closes and disposes the specified reader object. Before these actions, the object is also checked if it is null or not.
- *“CloseAll()” method :*
The method does the operations of both CloseReader() and CloseConnection().
- *“Commit()” method :*
The method commits the specified transaction. After the commit operation, the connection of the transaction is terminated.

- *“Rollback()” method :*
The method rolls back the specified transaction. After the rollback operation, the connection of the transaction is terminated.

4.2.3 “IDBSpecificHandler” Interface

The database providers used during database operations may have different implementations from each other. If it is needed to make use of a variety of providers, these differences should be handled separately. For this reason, IDBSpecificHandler interface is added to DALHelper library. For each used provider, a class is created, that implements this interface, such as SQLClientHandler, OleDbHandler and MSDB2Handler. The following methods belong to the interface, that are to be implemented.

- *“GetParameterPlaceholder()” method :*
The placeholder used in queries for command parameter may differ provider by provider. This method returns the appropriate placeholder. Two query examples are given below to show the placeholders of SQLClient provider and OleDb provider respectively.

```
SELECT * FROM DbFile WHERE @ParameterName = 10
SELECT * FROM DbFile WHERE ? = 10
```

- *“InsertAndGetIdentityValue()” method :*
If a database file has an identity column, the methods of getting the value of this column after an insertion can differ. Each handler should implement its own way in this method.
- *“GetDbTypeFromProviderType()” method :*
While getting the schema from the database, a provider returns a column type as integer values. The mapping between these integers and .Net DbType should be done in this method. For example, SQLClient provider returns 8 for Int32 type, on the other hand, OleDb provider returns 3.
- *“GetLargeTextFieldSelectStatementConversion()” method :*
If a column is AnsiString, the select query to read its value may differ between the providers. For example, the column should be cast to character type, if you are accessing a DB2 database.

```
SELECT CHAR(ColumnName, 256) FROM DatabaseFile
```

- “*LockTable()*” method :

To lock a database table, also different implementation may be needed. This method separates these implementations.

4.2.4 “DBFieldNameAttribute” Class

This class is designed to hold the attributes of each database column on the generated classes. The fields on the generated classes holds the values of the corresponding column. Moreover, each of fields has also an DBFieldNameAttribute attribute to keep the metadata of its column. These attributes are needed during CRUD operations. In the following lines of code, you can see an example of DBFieldNameAttribute usage.

```
[DBFieldName("CategoryID", true, false, true, DbType.Int32)]  
private int categoryID;
```

The attribute indicates that the name of the corresponding column of “categoryID” field is “CategoryID”. Moreover,

- this column is a key of the table (first `true`),
- the type of the column is not Decimal (`false`),
- this column is an autoincrement column (second `true`),
- the type of the column is `DbType.Int32`.

4.2.5 “DBRelationAttribute” Class

The attribute class to indicate the relationship metadata of a parent field. As you remember, the generator adds an extra property for the child object to the generated class. The methods, that provide the relation integrity, need the metadata which relates the variable and the property of the parent field with the child property. An example is given below.


```
[DBRelation("categorie", "CategoryID")]
private int categoryID;
public int CategoryID
{
    get{...}
    set{...}
}
...
private Categories categorie;
```

Where “categorie” is the name of the property name for the child object, and “CategoryID” is the name of the parent property.

5 EXAMPLE USE OF PROPOSED TOOL

Data Access Layer Code Generator tool gives the developers the ability of building the layer, that is responsible of accessing data on the defined databases, without giving huge effort. In other words, instead of wasting time to deal with data access, the developer will concentrate much more on the business part of his/her projects.

This chapter aims to view the earnings by using Data Access Layer Code Generator. The best way to show the usage of the generator is developing a simple application, so an ASP .Net program has been prepared, whose development phases you will see in the following pages.

5.1 Presentation Of The Database Used In Demonstration

During the demonstration of Data Access Layer Code Generator tool, Misrosoft SQL Server 2000 will be used as DBMS, that is running on the same machine with the development environment. While installing the server, the sample database – Northwind – has been also installed, so I had the chance to test developed codes on that database. As the same, it is planned that the application developed in the following pages also will try to access this database.

Northwind database is a sample database that comes with the setup of Misrosoft SQL Server 2000. It has many tables that have relations with eachother and have columns with variaty of types, so Northwind is suitable to realize a database that is build in any project. In our application, three Northwind tables are choosen to demonstrate the DAL Generator Tool.

- Categories Table :

The table with four colums to define a category has records each of which logically corresponds to a group of products.

Table 5.1 Categories Table

Column Name	Data Type	Is Nullable	Description
CategoryID	int	False	Category key.
CategoryName	nvarchar	False	Name of category.
Description	ntext	True	Description of category.
Picture	image	true	Picture of category.

- Supplier Table :

The table contains data about product suppliers.

Table 5.2 Supplier Table

Column Name	Data Type	Is Nullable	Description
SupplierID	int	False	Supplier key.
CompanyName	nvarchar	False	Company name of supplier.
ContactName	nvarchar	True	Contact name of supplier.
ContactTitle	nvarchar	True	Contact title of supplier.
Address	nvarchar	True	Address of supplier.
City	nvarchar	True	City of supplier.
Region	nvarchar	True	Region of supplier.
PostalCode	nvarchar	True	Postal Code of supplier.
Country	nvarchar	True	Country of supplier.
Phone	nvarchar	True	Phone of supplier.
Fax	nvarchar	True	Fax of supplier.
HomePage	ntext	True	HomePage of supplier.

- Products Table :

The table holds the information about the products. It has two relations with the Categories table and the Suppliers table on CategoryID, SupplierID columns respectively. In other words, a product record can be included just in one category and can have just one supplier. On the other hand, a category can include and a supplier can supply more than one product.

Table 5.3 Products Table

Column Name	Data Type	Is Nullable	Description
ProductID	int	False	Product key.
ProductName	nvarchar	False	Name of product.
SupplierID	int	True	Supplier's key of product.
CategoryID	int	True	Category's key of product.
QuantityPerUnit	nvarchar	True	Quantity per unit of product.
UnitPrice	money	True	Unit price of product.
UnitsInStock	smallint	True	Units in stock of product.
UnitsOnOrder	smallint	True	Units on order of product.
ReorderLevel	smallint	True	Reorder level of product.
Discontinued	bit	True	Discontinue status of product.

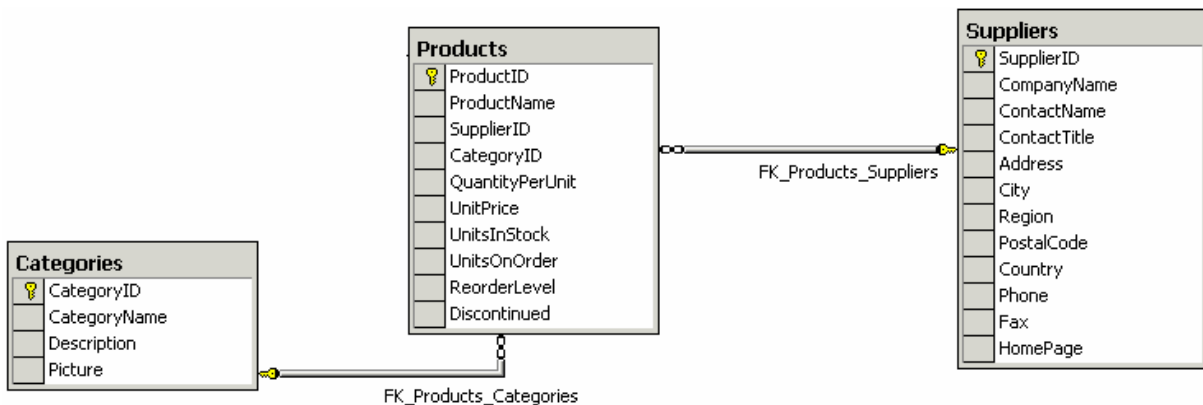


Figure 5.1 Northwind Database Diagram

Before we develop our application, we should prepare the database to establish a connection. The first step is to define a user on the database which will be used during data access. The procedure listed below is adequate for this example.

- Open Enterprise Manager of MSSQL 2000.
- Expand Northwind database and right-click on “Users”. Select “New Database User...” menu item.

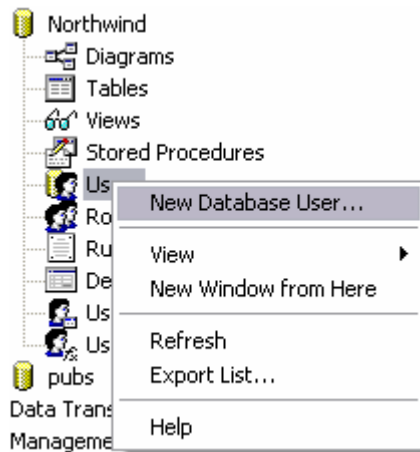


Figure 5.2 New Database User

- Select “<new>” item for Login name on the up coming window.

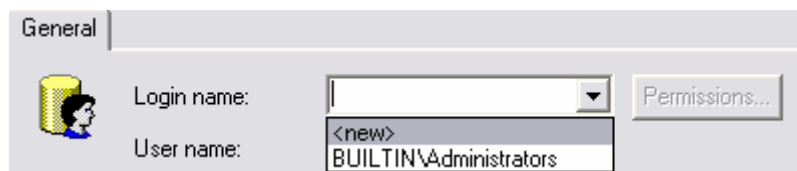


Figure 5.3 New Login Name

- Enter the login properties as...

General Tab

Name : webuser
 Authetication : SQL Server Authentication
 Password : webuser
 Database : Northwind

Database Access Tab

*Select Northwind on the database list to permit the user access.

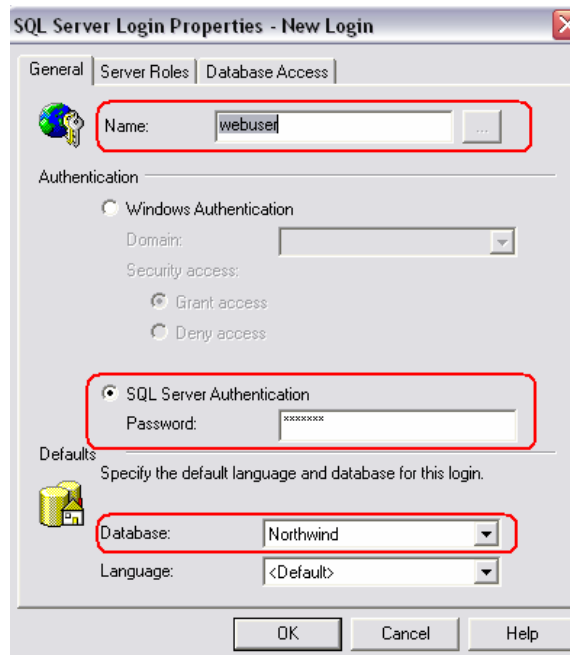


Figure 5.4 SQL Server Login Properties General Tab

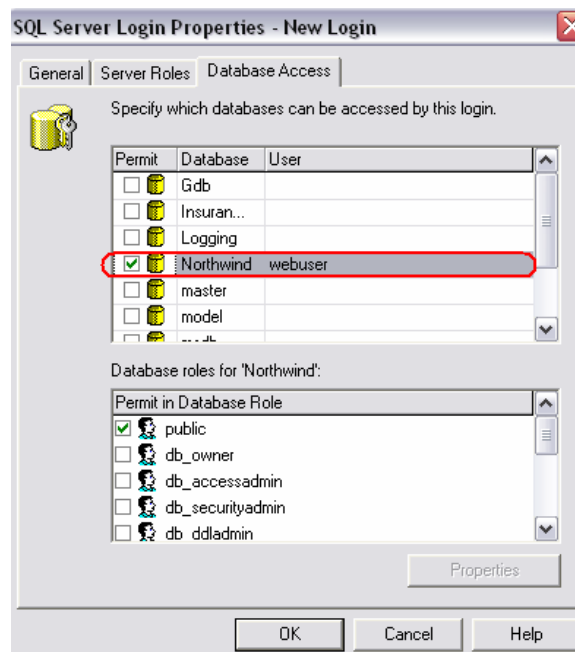


Figure 5.5 SQL Server Login Properties Data Access Tab

Until now a user is defined that has right to access Northwind database. On the next step, we will test the connection to the database using MS Visual Studio 2005 and obtain a basic connection string. The work to be done for this step comes...

- Open VS 2005.
- View the Server Explorer pane of VS 2005.
- Select “Add Connection...” item on the appeared menu by right-clicked on Server Explorer.

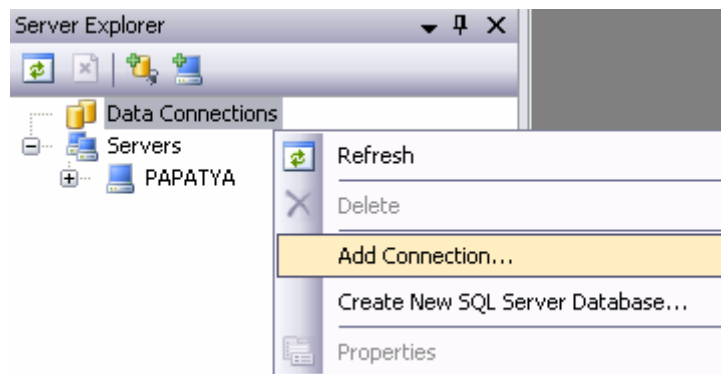


Figure 5.6 Server Explorer Data Connection Menu

- Enter the connection parameters and test it. If test is successful, get the connection string from the Advanced window.

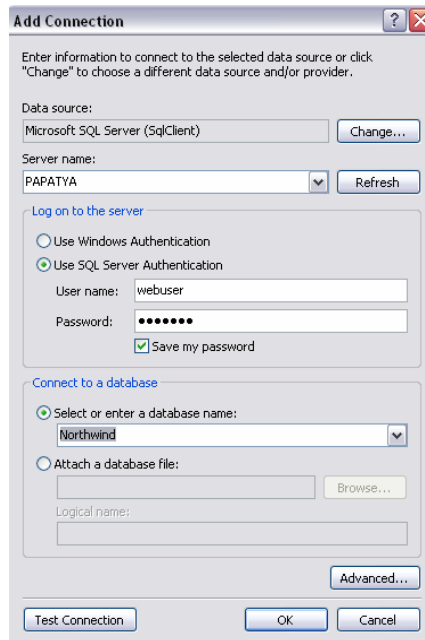


Figure 5.7 Add Connection Wizard



Figure 5.8 Add Connection Wizard Advanced Properties

The obtained connection string is one of the parameters used by DAL Generator to establish database connections, so it has to be defined in DALHelper.DBConnectionHelper class. As explained in earlier chapters, the connection string value should be added to ConnectionStrings and TestConnectionStrings arrays. A brief list of actions to define a new database for DAL Generator is listed below.

- Add an explanatory name for the connection to the DBConnectionType enum on DALHelper.DBConnectionHelper.
- Add the connection string to ConnectionStrings and TestConnectionStrings arrays on DALHelper.DBConnectionHelper.
- Add the SqlConnection type to the Connections array on DALHelper.DBConnectionHelper.
- Add SQLClientHandler type to the DBSpecificHandlers array on DALHelper.DBConnectionHelper.
- Build and distribute the DAL Generator to the developers, who will need to use the newly defined connection.

After these procedures were completed successfully, we are ready to build the data access layer of the applications that works on Northwind database's records.

5.2 Recommendations To Create DAL Packages

A DAL package is the class library project in which a group of DAL class reside. This part contains some recommendations that can be useful for developers when they generate DAL packages. Usage of this written experience is not a mandatory rule of DAL Generator, but it gives some clue about basic decisions.

Before generating any DAL code, the first thing to do should be creating a blank solution which will contain all the DAL packages and the DALHelper project. After having a solution for DAL codes, any generated class should be added to this solution. With this decision, a developer will know where the data access layer classes locates, and whether any needed class has already been generated or not. If the searched DAL class is found, it can be reused without generating new one. In other words, management of data access layer gets easier and reusability of classes increases.

The DALHelper project is the class library that is using by all of the DAL classes. Although this library has static attributes, some needs may arise to add extra properties and functionalities to this library. In such a case, compiling the DAL classes with new functionalities against undetermined errors will be useful. Instead of determining the locations of each DAL code, collecting all classes in the same solution makes the life easier.

Another point, that the developer should think about it, is how he/she will group the DAL classes, because using one well grouped package can be enough for an application. On the other hand, using more than one DAL package for a purpose can make programming difficult. For example, the DAL classes related to accounting of a company can be put in the same package. The grouping criterias differ developer by developer or according to the requirements of the applications. Possible package decisions;

- Grouping DAL classes according on which database they access.
- Grouping DAL classes according to the application they used in.
- Grouping DAL classes according to the usage frequencies of the classes.

While grouping the classes, the package names should also be as clear as every developer can understand/guess the content of it.

DAL Code Generator tool provides to save the schema of the generated classes as XML files. With this ability, the developer can regenerate the classes after some modifications with a little effort. It is important to locate the schemas on the same project with the related DAL class, so it can be found easily by the developer.

5.3 Generation Of DAL Codes

Until this step, we prepared the connectivity to the database and set the connection parameters of the DAL Code Generator tool. The following pages will explain the usage of the tool and its capabilities.

As written in previous part, a solution will be created, into which a DAL package will be added to be used in the example application, before generating any class.

In the following lines, the creation of a solution will be described.

- After opening VS 2005, select “Project...” item under File-New menu. This begins new project dialog.

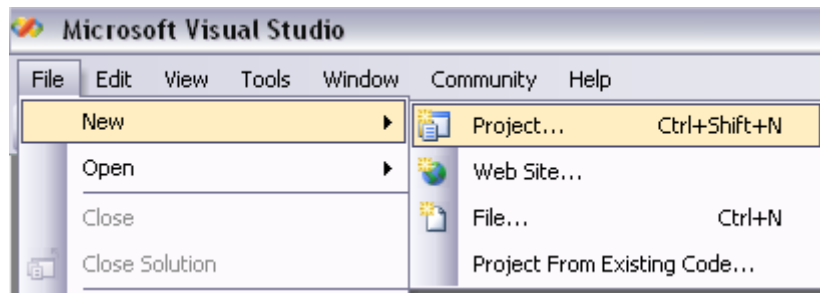


Figure 5.9 Visual Studio 2005 File Menu

- The entered parameters on the New Project dialog are listed below.
Project Type : Visual Studio Solutions under Other Project Types tree node
Visual Studio Installed Templates : Blank Solution
Name : DALSoln
Location : C:\Projects\Packages

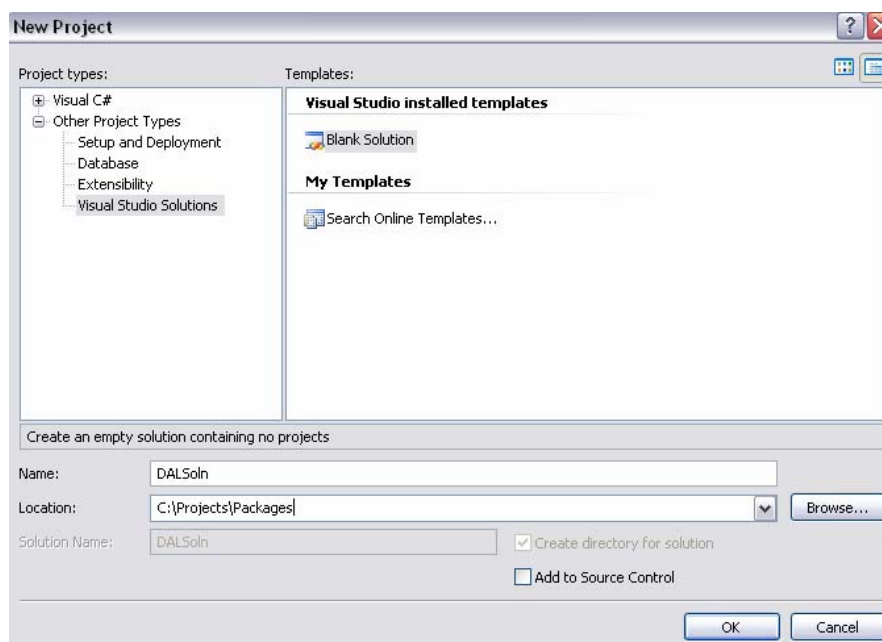


Figure 5.10 New Project Wizard

- The dialog creates and opens a blank solution with the given parameters.

DALSoln is the solution where all the dal packages are included, so the class library for the Northwind database will be genared on it.

- When righth-clicked over the DALSoln on the Solution Explorer, “New Project...” item should be selected under “Add” menu. As the result, “Add New Project” dialog will appear.

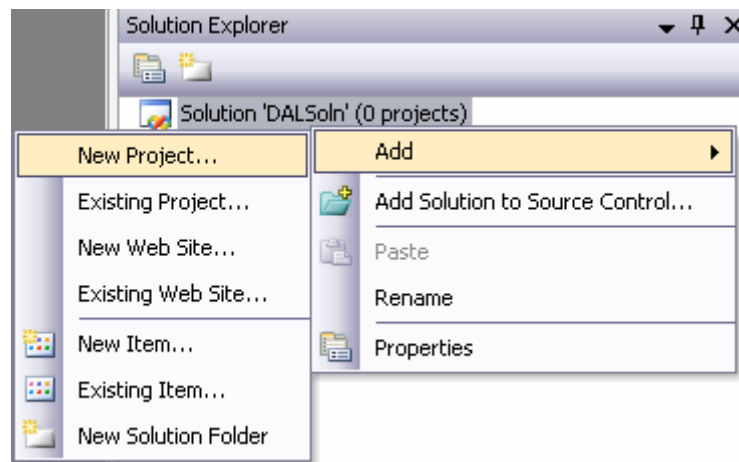


Figure 5.11 Solution Explorer Menu

- The entered parameters on the dialog are listed below.
Project Type : Windows under Visual C# tree node
Visual Studio Installed Templates : Class Library
Name : NorthwindDAL
Location : C:\Projects\Packages\DALSoln

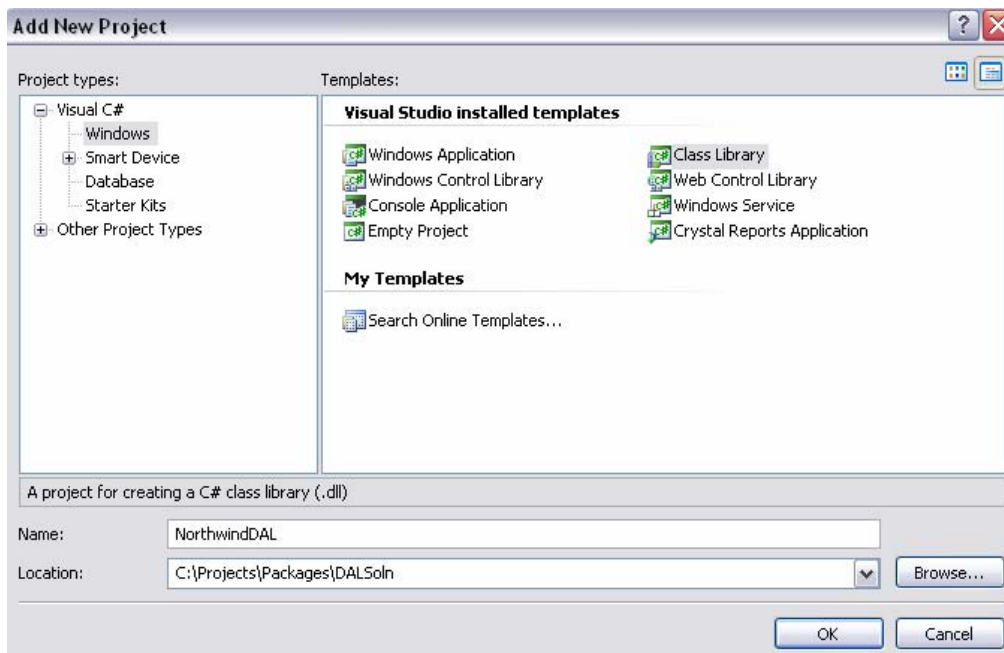


Figure 5.12 Add New Project Wizard

- This dialog creates a class library that contains an empty class, Class1. It should be deleted, because no code will be written on that.

As written before, all the generated classes use the DALHelper library. Due to that reason, the library has to be added to the solution by selecting “Existing Project...” appearing when right-clicked over the DALSoln.

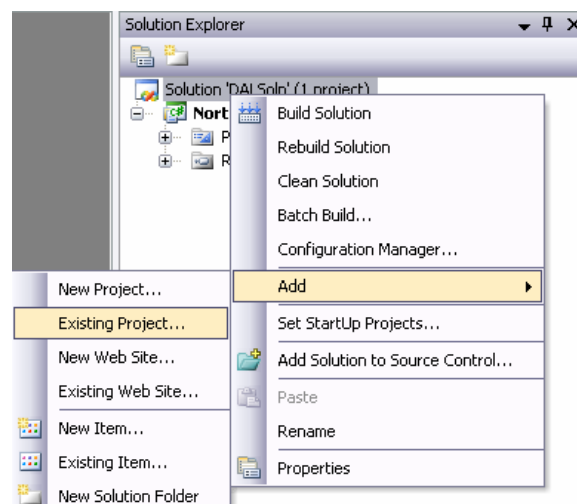


Figure 5.13 Solution Explorer Menu

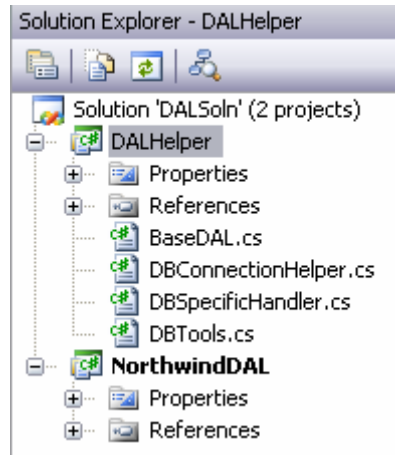


Figure 5.14 Solution Explorer

We are now ready to execute the DAL Code Generator tool and to generate our first class. The generator decides, on which project it generates code, by getting the selected project on the solution explorer, so firstly, the project (DAL Package) or a member of the project is to be selected on the solution explorer. After the project selection, it can be run by clicking on “Generate DAL Code” item under “Tools” menu of Visual Studio.

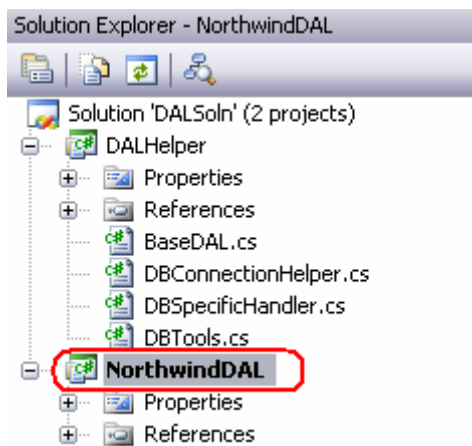


Figure 5.15 Solution Explorer

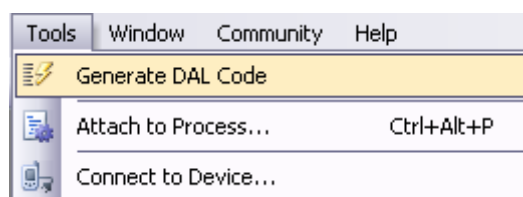


Figure 5.16 Visual Studio Tools Menu

The generator wizard comes to the screen.

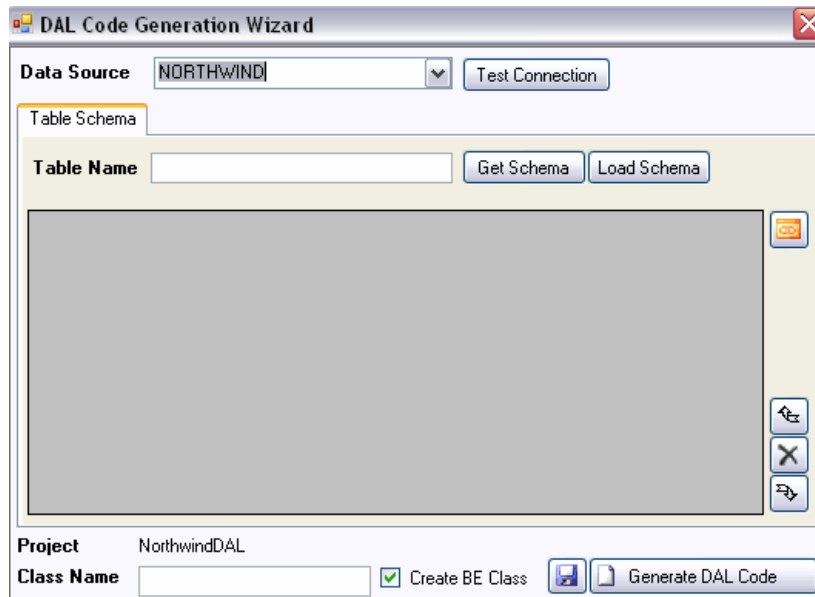


Figure 5.17 DAL Code Generation Wizard

The tool has three main parts. On the top of the wizard, the first part exists for the data source selection. The data sources listed on a combobox component are the defined sources on DALHelper.DBConnectionHelper class. The developer should choose the right one according to the table he/she wants to access. This wizard also contains a button to test the connection to the selected data source. As the connection test result, the developer is informed with a messagebox which indicates a successfully established connection or the exception thrown during connection attempt.

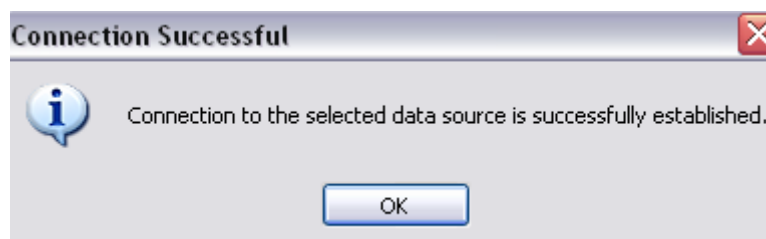


Figure 5.18 Connection Test Result For Success



Figure 5.19 Connection Test Result For Error

The second part of the wizard is the area where the table selection is made and where the attributes of the DAL class is set. On this part, the developer should enter the table name, whose DAL code will be generated. After that, pressing to the “Get Schema” button fills the Gridview component with the schema of the table by accessing the data source. If the table name is miss written or the database can not be accessed, the wizard will warn the developer.

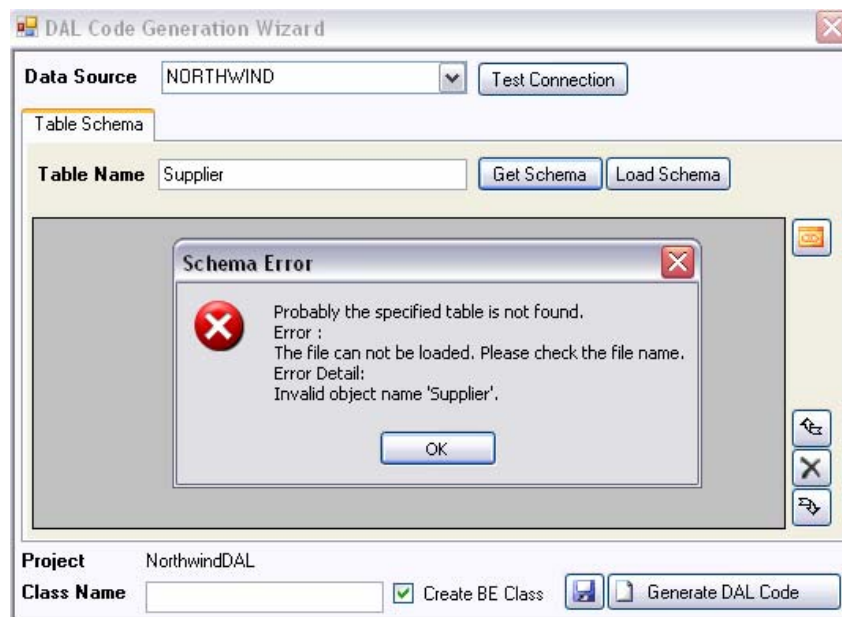


Figure 5.20 DAL Code Generation Wizard Schema Error

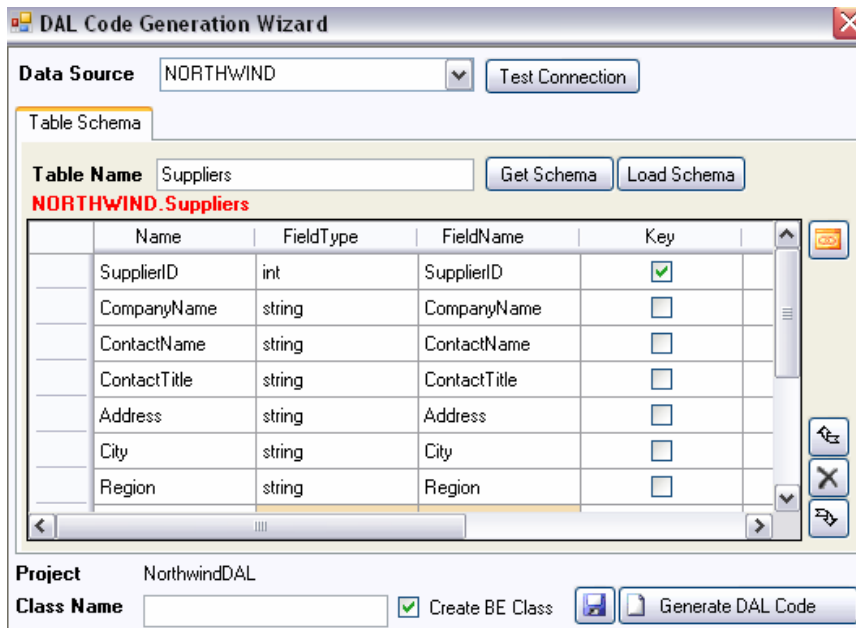


Figure 5.21 DAL Code Generation Wizard With Loaded Schema

The main schema of the table is loaded to the grid. Now, the developer can modify the fields of the schema, if any changes is needed. The grid columns and the circumstances where the changes is needed is explained below.

- **Name Column :**

The names listed under this column is the column names of the database table. This values should not be changed, otherwise an exception will be thrown when accessing that column using the generated class.

- **FieldType Column :**

The generated class will have properties for each table column. FieldTypes indicates the .Net types of these properties. The values listed under this column can be modified by the developer under such circumstances;

- The generator decides the appropriate .Net type according to an algorithm, but it does not always choose the preferred data type by the developer. Mostly, the reason is that not all the database types have the exact match to a .Net type. For example, DB2 database does not have the boolean type, so SHORT INT type is generally is used to hold true/false values. In this case, the developer can modify FieldTypes value with “bool” string. As a result, the property will be in boolean type, on the other hand it sets its value into a SHORT INT database column. As an another example, the database column can be double, but the

programmer can prefer to access its values through a decimal property, so he/she should change the FieldTypes from double to decimal.

- **FieldName Column :**

As it is explained, the generator will create properties on the generated class for each database column. The name of these properties will be the values of FieldName column. The developer can edit these values, but he/she should watch out the uniqueness of the names in the schema.

- **Key Column :**

For the key fields of the table, the corresponding checkboxes will be checked by the generator while getting the schema from the database. Some tables may have a key column, which does not have a logical meaning. For example, a student table can have an autoincrement column as key, but it can also include a column that holds the student numbers, which is known as unique for every student record. In this case, the programmer may prefer to set student number as key, because it is much more logically useful in applications than an identity column.

- **Decimal Column :**

If a database column is decimal, and the generator decides to use non-decimal .Net type for this column, the checkbox is checked while getting the schema. A checked checkbox indicates that the value on the database should be covered as decimal. If the developer preferred to use double type, the value on the database is to be cast to double during a read operation, and for an update operation, the object's value is to be cast to decimal.

- **AutoIncrement Column :**

An autoincrement (or identity) column is the column whose value is given by the database during insertion of the record. If any identity column is found while loading the schema, the corresponding checkbox will be checked automatically. The developer should be aware of that reading an identity column decreases the performance, so if such a column is not needed, it should be deleted from the loaded schema, or its AutoIncrement checkbox should be unchecked.

- **DbType Column :**

This column specifies the type of the corresponding field on the database. The value is not to be changed.

Moreover, the wizard has buttons to change the order of the schema's rows and to delete a row from the schema. A deleted row is not reflected to the generated schema.

This part contains buttons to load previously saved schemas and to define relationships, but these functionalities will be covered after our first DAL class will be generated.

The last section of the wizard is the last step before the class generation. Here, the developer enters for the final parameters of the wizard that are explained below.

- **Project Information :**

The project label shows on which project the generated class will be created, so the developer can be sure that he/she did not select the wrong project on the solution explorer of Visual Studio.

- **Class Name :**

The name written on this textbox will be the name of the generated DAL classes. For the class that accesses to the database, the generator appends "DAL" extension to the name specified on the textbox. By the way, for the business entity class, that inherits the DAL class, will have the same name as specified.

- **"Create BE Class" Checkbox :**

The generator creates also a business entity class, if this checkbox is checked. Because the business entities may contain custom properties and methods, that are added by the developer, he/she may not want to create a new business entity and lose these modifications when a regeneration is in process.

- **"Save Schema" Button :**

The DAL Code Generator gives the ability to save the loaded and modified schema in an XML file. At this point, the developer can press this button and save the schema after a save dialog. The location of the saved xml file may be critical for the future regeneration needs. It is recommended, that the schema should be kept in the project folder and, also, should be included in the project, so it can be easily found by other developers.

- **"Generate DAL Code" Button :**

This button is the last action of the wizard. The generation process begins by pressing this button.

Until now, the main functions of the tool are explained. At this point, we can resume to build the data access layer of our demonstration application.

Lastly, the generator is executed, and the schema of the Supplier table is loaded to the generator. The last view is shown below, after the class name was specified the same name as the table, Suppliers.

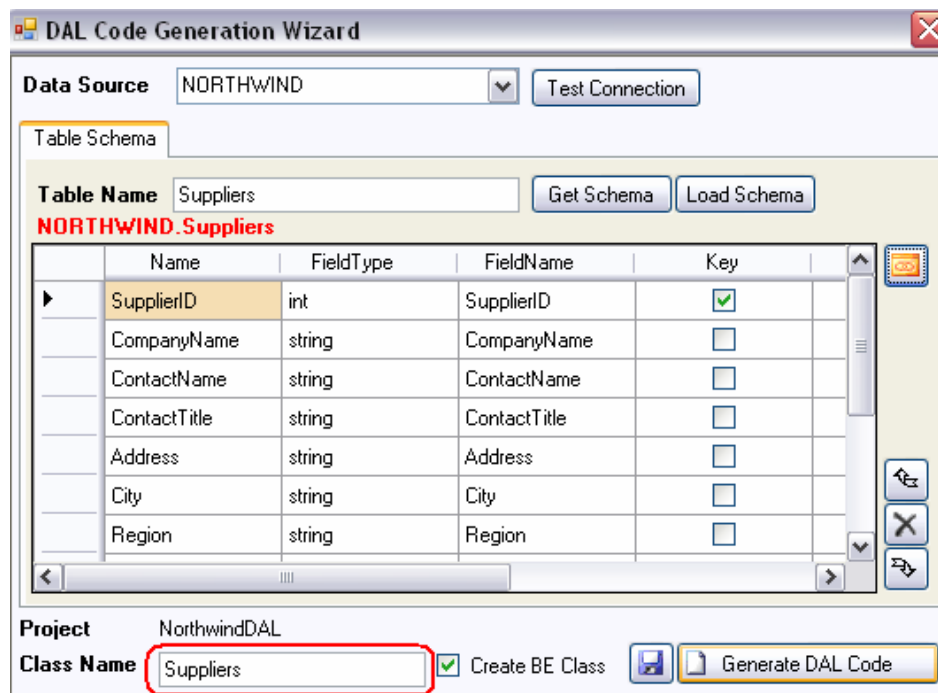


Figure 5.22 DAL Code Generation Wizard Before Generation Process

After the “Generate DAL Code” button is pressed, the generator creates two files and opens them on the Visual Studio. The created files for the Suppliers table are;

- **Suppliers.cs :**
Business entity class file, that is open for the developer’s modifications.
- **SuppliersDAL.cs :**
The class that is actually responsible of the data access to the Suppliers table.

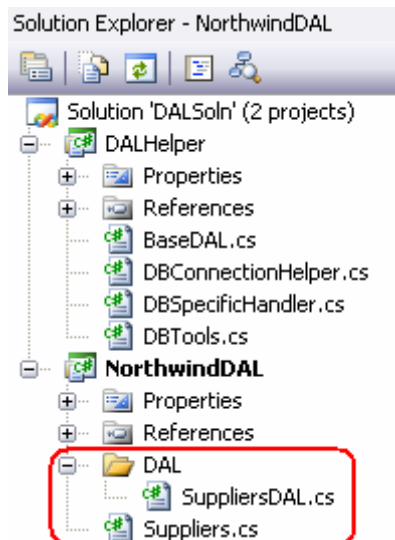


Figure 5.23 Solution Explorer After Generation

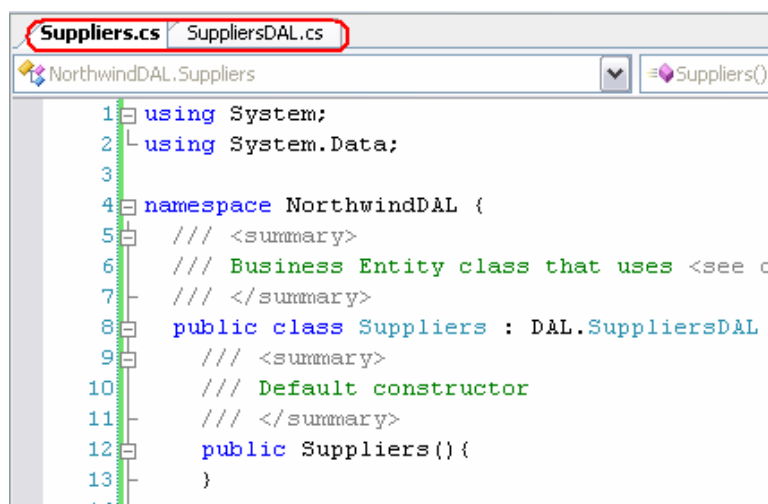


Figure 5.24 Code Editor After Generation

As a result, hundreds of lines of codes are generated with a little effort in a very short time. Now, the same generation process will be repeated for the Categories table of Northwind database. After the generation process is completed, the server explorer will have a view displayed below.

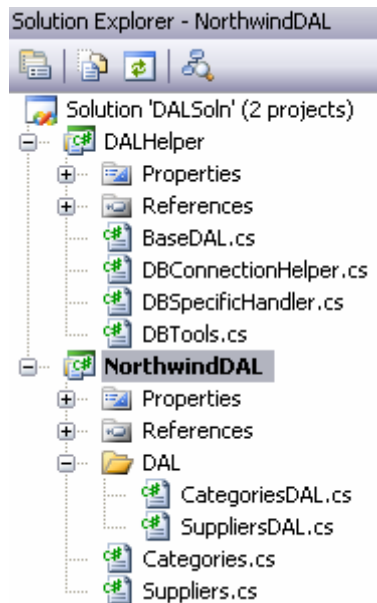



Figure 5.25 Solution Explorer After All Generation Process

There is one more table left to generate its DAL class on which the demonstration application aims to work. This table is the Products table, that has relations with Suppliers and Categories tables. The steps taken to generate ProductsDAL file will show how a relationship can be defined using the DAL Code Generator tool.

- As it is done on the previous generation procedures, the NorthwindDAL project should be selected on the solution explorer, and the generator wizard should be executed by pressing “Generate DAL Code” item on Visual Studio’s tools menu.
- After loading the schema from the database using “Get Schema” button, as described earlier, the relation button should be clicked. () As a result, the Relation Configurator will appear on the center of the screen.

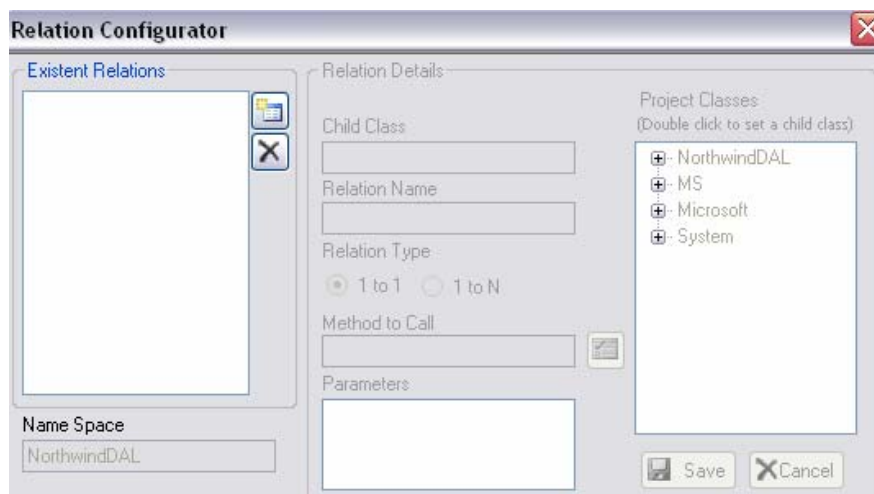


Figure 5.26 Initial View Of Relation Configurator

- The Relation Configurator has two main parts. The first part, located on the left side of the form, is to display defined relations. Because the schema does not have any existent relation yet, the list is empty. Moreover, this part contains two buttons to add a new relation and to delete an existent one, that is selected on the list. The second part, “Relation Details”, makes the developer to be able to enter the attributes of a new relation or to modify an existent relation. To add our first relation, the button, “Add New Relation”, should be pressed.
- This action enables “Relation Details” part. As written on the top of the part, the status became “Define a new relation”, so the developer is aware of the action, that he/she is doing. Moreover, it is also seen, that there is a tree, Project Classes, which is displaying the classes in the project and the classes in the references of the project. The treenode, we intrested in, is the NorthwindDAL project. This treenode has two subnodes, that are familiar for us, because Categories and Suppliers are the classes, that have been generated earlier. Next action should be double-clicking on the Categories subnode to begin defining a relation with Categories class.

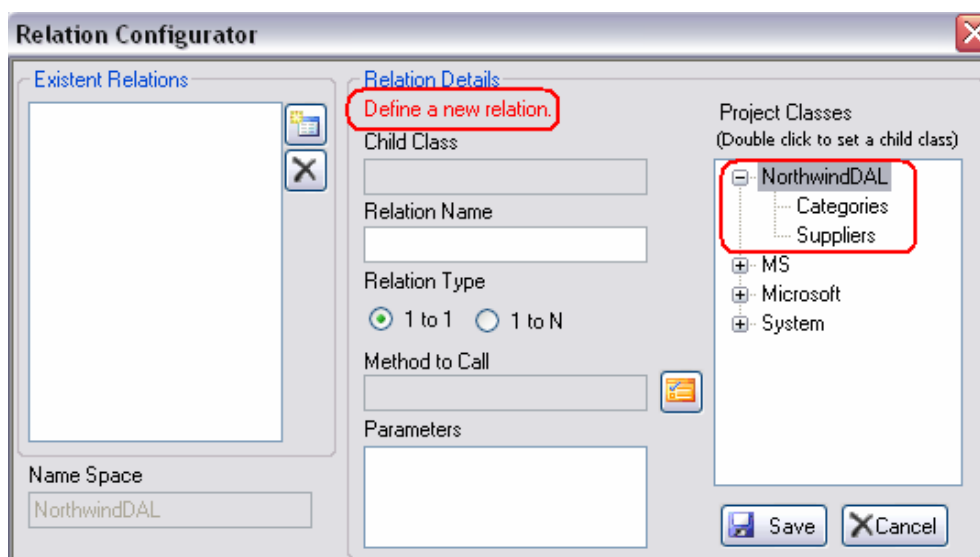


Figure 5.27 Relation Configurator For New Relation Definition

- By default, “Relation Name” entry has the same name with the relation class. In our case, it will be “Categories”. The developer can change it to a more explanatory name for him/her, because the generated class will have a property with this name. In general, I rename it with a singular word, if the relation is 1 to 1. If not, a plural word

is chosen by me. For this relation, the name, “Category”, is used, because each product can have at most one category, that means the relation type is 1 to 1.

- The relation between Products and Categories classes is provided with calling a method of Categories by Products, so the right method should be specified. Additionally, the parameters, that is to be passes to the method, should be chosen in the property list of Products. To start these selections, “Method to Call” button should be use.
- The first dialog lists the constructors of Catogories. The developer selects one of them on single selection listbox. We choose the construnctor, that takes just the category id as parameter.
- The second dialog of “Method To Call Selection” gives a multi selection list of Product’s properties. As mentioned in previous step, the method to call takes just one argument, so the developer has to select as many properties as the number of the selected method’s arguments, 1. The “CategoryID” property is the one, that is to be passes to the constructor. If the developer select wrong number of items on the list, he/she will be warned.

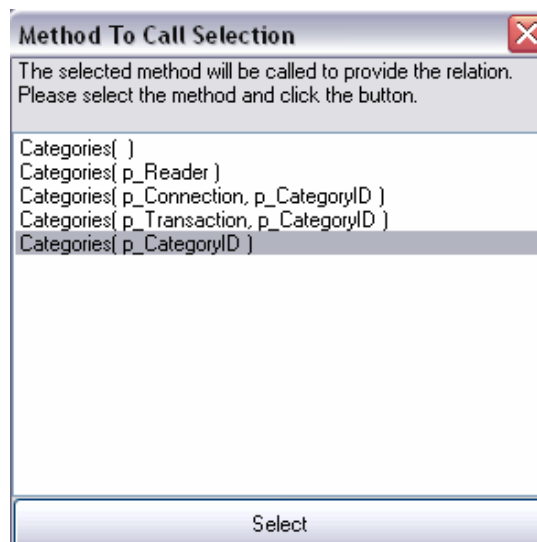


Figure 5.28 Method To Call Selection Part 1

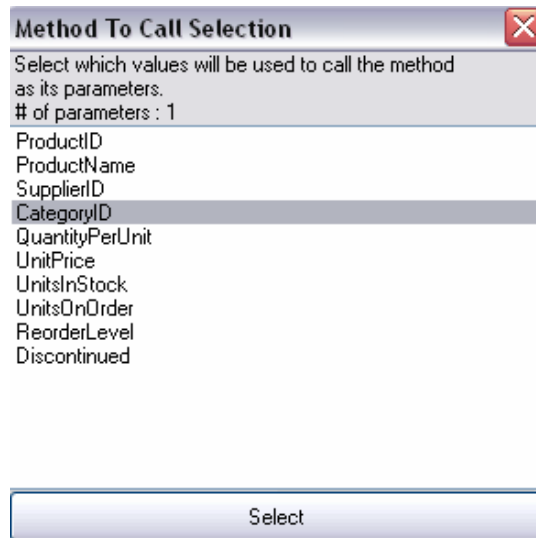


Figure 5.29 Method To Call Selection Part 2

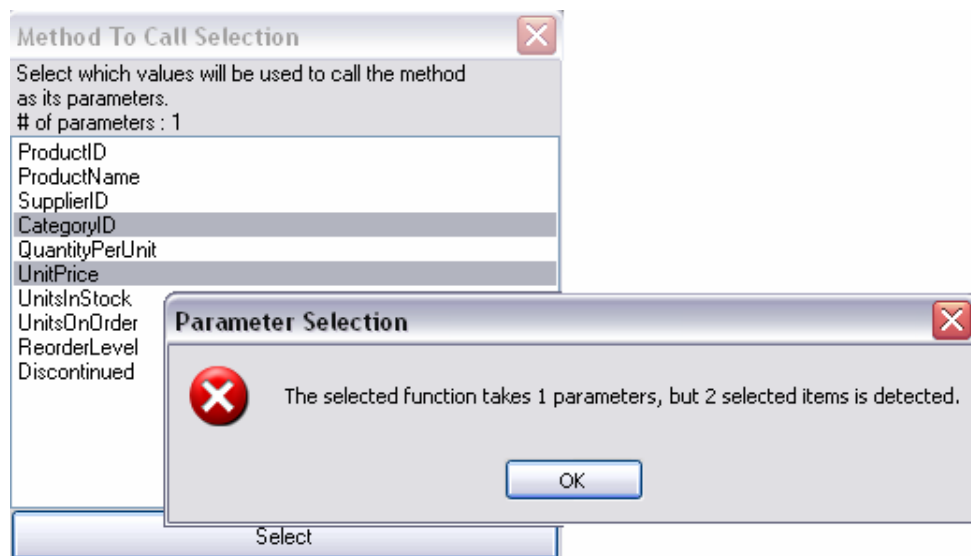


Figure 5.30 Method To Call Selection Parameter Number Error

- This relation definition ends with saving the relation. The same steps will be repeated for the relation with Suppliers. The last screen is as shown below.

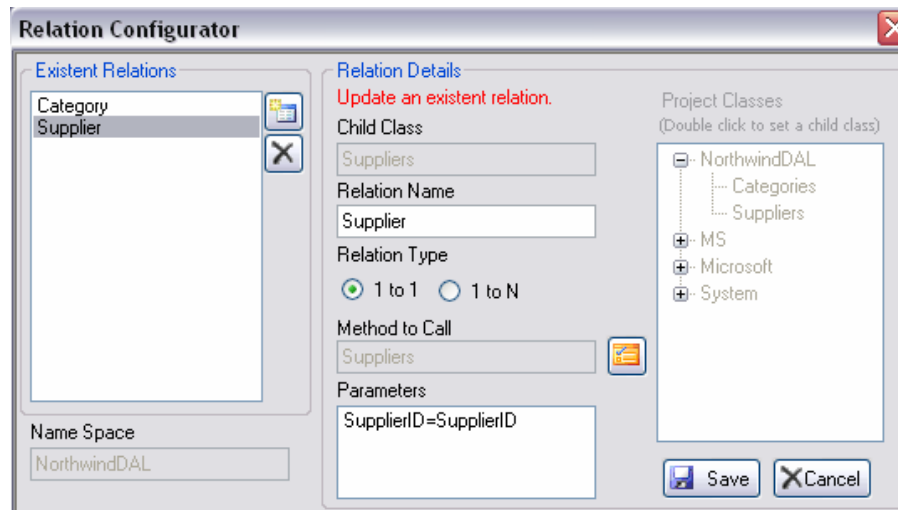


Figure 5.31 Relation Configurator For Update Existent Relations

With closing the relation configurator, the wizard form appears again. The first thing, that the developer should do after defining the relations, is to save the schema changes. The XML file, produced with save operation, will also contain the relation definitions, so if a regeneration of the Products DAL class is needed, the relations info can also be obtained.

```
<ClassRelations>
  <Name>Category</Name>
  <ChildClassName>Categories</ChildClassName>
  <Type>0</Type>
  <MethodToCall>Categories</MethodToCall>
</ClassRelations>

<ClassRelations>
  <Name>Supplier</Name>
  <ChildClassName>Suppliers</ChildClassName>
  <Type>0</Type>
  <MethodToCall>Suppliers</MethodToCall>
</ClassRelations>

<Parameters>
  <Name>Category</Name>
  <MethodToCall>Categories</MethodToCall>
  <ParamName>CategoryID=CategoryID</ParamName>
</Parameters>
```

```
<Parameters>
  <Name>Supplier</Name>
  <MethodToCall>Suppliers</MethodToCall>
  <ParamName>SupplierID=SupplierID</ParamName>
</Parameters>
```

After the generation process of Products class, we will observe two more files added to the NorthwindDAL project like previous generation processes did. On the other hand, when we take a close look in ProductsDAL.cs, which is one of the produced files, we will see that the properties, Supplier and Category, was added to provide relationships.

```
private Suppliers supplier; //Local variable.
public Suppliers Supplier //Property as the given relation name.
{
    get
    {
        if (supplier == null)
        {
            try
            {
                //The selected methods to call is loacted in the property.
                supplier = new Suppliers(SupplierID);
            }
            catch (RecordNotFoundException)
            {
                supplier = null;
            }
        }
        return supplier;
    }
    set
    {
        supplier = value;
        UpdateRelationBackwards("supplier", supplier);
    }
}

private Categories category;
public Categories Category
{
    get
    {
        if (category == null)
        {
            try
            {
                category = new Categories(CategoryID);
            }
            catch (RecordNotFoundException)
            {
                category = null;
            }
        }
        return category;
    }
    set
```

```
{
    category = value;
    UpdateRelationBackwards("category", category);
}
```

5.4 Developing An ASP .Net Application

Before developing an application, the developer generally decides the data sources, which the application will access, according to the requirements. After the determination of the sources, the DAL Code Generator tool helps to building data access layer just in minutes, so the developer can quickly begin to focus on the presentation and the business layers. Similarly, we can start to develop our demonstration application.

This section will mostly focus on the usage of the generated classes, but the basic steps of creating an ASP.Net application and the main attributes of the used component will be also explained.

5.4.1 Creation Of An ASP .NET Project

Visual Studio .Net has a very basic wizard to create an ASP.Net Application. The “Add New Web Site” wizard can be viewed by using “File – New – Web Site” menu item of Visual Studio 2005. On the opened form, the developer have the choices to select the template of the site, the location, where the project will reside, and the language, in which the application will be written. The selection made for this example is listed below.

Visual Studio installed templates :	ASP.Net Web Site
Location Type :	File System
Location :	“C:\Projects\WebProjects\DALUsageDemonstrationSoln\DALUsageDemonstration”
Language :	Visual C#

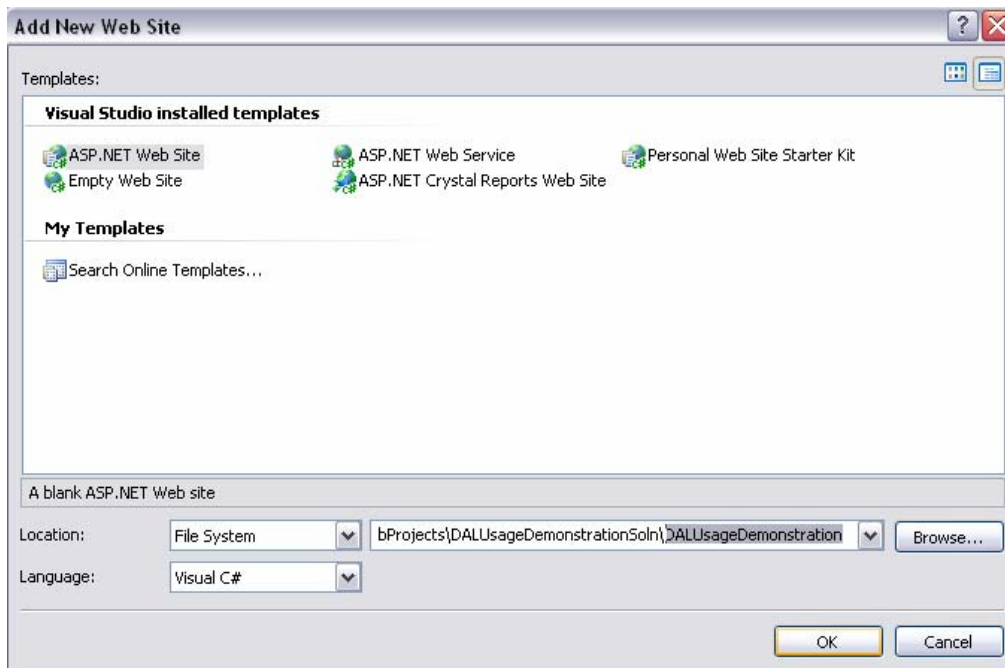


Figure 5.32 Visual Studio 2005 Add New Web Site Wizard

This wizard creates an ASP.Net project according to the given parameters and adds it to a solution. The project contains a default web form, Default.aspx , but the developer can add more items to the project using “Add New Item” wizard, that pops up by right clicking on the project and selecting “Add New Item” item. This wizard shows the available items on Visual Studio. In the project developed in the following pages, we will need to add one more Web Form using this wizard.

5.4.2 Adding Components To A Web Form

Visual Studio includes lots of component for each kind of projects, that can be easily added with drag and drop, and whose attribute can be set in the code behind of the form or in the property pain of Visual Studio.

For the ASP.Net web forms, the available components are also listed on the Toolbox pain, and they can be moved onto the form.

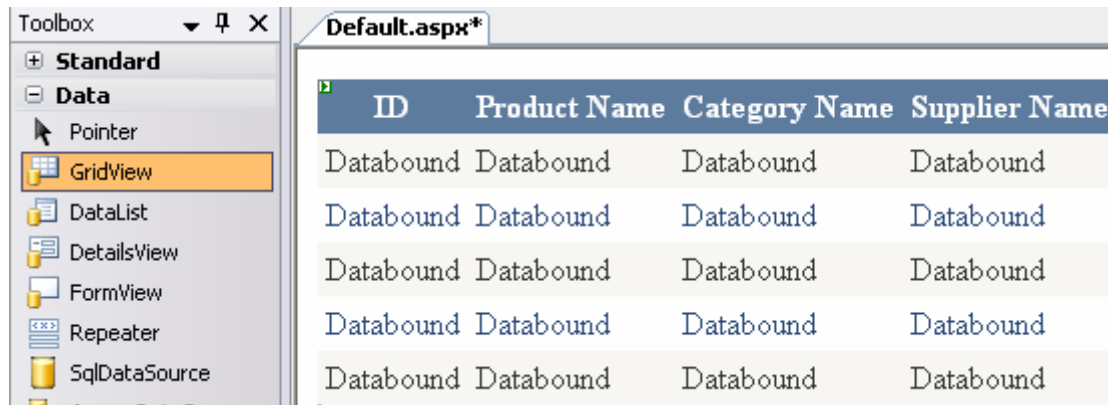


Figure 5.33 Visual Studio 2005 Toolbox

5.4.3 Preparing Applications For DAL Usage

After these informations, we will begin to build our example project, but we should first to complete the requirements of DAL Classes' usage.

In the previous pages, a class library is created, which includes data access codes of Northwind database. Firstly, the class library should be referenced by the project, that will use it. There is two ways to reference a DAL library generated by our tool.

- The reference can be defined by targetting the DLL file of the DAL library. The developer should build the DAL library, so he/she obtains the DLL file.
- The second method is defining the reference from the list of the solution's projects. To practice this method, the developer is to add the related DAL library project to the same solution as the application that will use it.

While the second way is chosen to apply, the NorthwindDAL project is added to the demonstration solution.

The second requirement of DAL usage preparation is adding the reference of DALHelper library to the solution as well, because DALHelper is the library, that is used by every generated DAL class. Moreover, it also contains helper functions, that can be used to access the data sources without using DAL classes, or to obtain parameters to call a DAL class. If DALHelper will be used in the project, it is also to be added to the references of the project.

After these requirements are completed, the view of the solution explorer looks like as the following. The developer is now ready to continue developing his/her application.

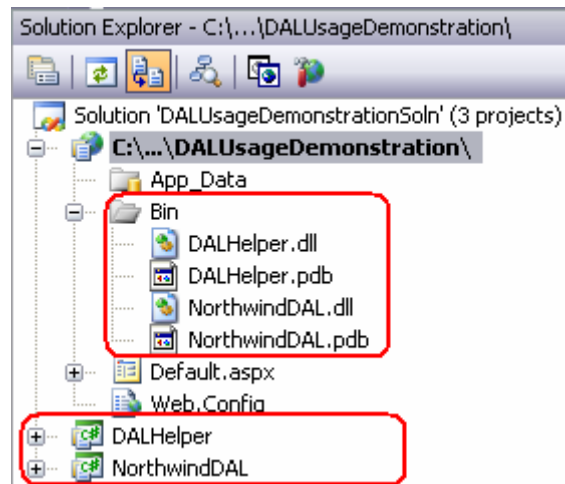


Figure 5.34 Solution Explorer When the Preparation Completed

5.4.4 Usage Of DAL Classes

The first web form of the demonstration application will list the Products' records. It will contain a grid view component which shows the product id, the product name, the category name and the supplier name for each product record.

To deal with this approach, the generated Products class provides the first two data, but it has only the ids of the related category and supplier records, not their name info. What the developer is going to do is to code the properties for them in Products.cs of NorthwindDAL library. The added code should be like this.

```
#region Custom Properties
public string CategoryName
{
    get
    {
        if (this.Category != null)
        {
            return this.Category.CategoryName;
        }
        else
        {
            return null;
        }
    }
}

public string SupplierName
{
    get
    {
        if (this.Supplier != null)
        {
            return this.Supplier.CompanyName;
        }
        else
    }
}
```

```

        {
            return null;
        }
    }
}
#endregion

```

As a result, Products class become able to provide the needed data over its properties.

After we complete the missing data, we can continue to design our first form, Default.aspx.

It is planned to have a form, over which some queries can be executed and the result are displayed on the same form. Due to these requirements, some components are added onto Default.aspx to let the users to enter the query conditions. The designed query section of the form has a view as the following.

Figure 5.35 Query Section Of Default.aspx

As you see, the form has two textboxes for “Product Id”, “Product Name” inputs, and it has two dropdown lists to let the user to select “Supplier Name” and “Category Name”, which have been filled with the existent records on the database. The codes to fill the dropdown lists are given below.

```

private void LoadSupplierList()
{
    Suppliers supplierLoader = new Suppliers();
    ArrayList supplierList = supplierLoader.LoadAll();

    supplierDropDownList.Items.Clear();
    supplierDropDownList.Items.Add(new ListItem());
    foreach (Suppliers supplier in supplierList)
    {
        ListItem listItem = new ListItem(supplier.CompanyName,
supplier.SupplierID.ToString());
        supplierDropDownList.Items.Add(listItem);
    }
}

private void LoadCategoryList()
{
    Categories categoryLoader = new Categories();
    ArrayList categoryList = categoryLoader.LoadAll();
}

```



```

categoryDropDownList.Items.Clear();
categoryDropDownList.Items.Add(new ListItem());
foreach (Categories category in categoryList)
{
    ListItem listItem = new ListItem(category.CategoryName,
category.CategoryID.ToString());
    categoryDropDownList.Items.Add(listItem);
}
}

```

The marked lines shows, how the lists of suppliers and categories are retrieved from the database. As you see, just two lines of code for each are sufficient to access the database records using generated DAL classes.

The query section of the form has also a button, “Load Products”, that is responsible to get the condition inputs, to query database and to bind the resulted query result to a gridview component, that is also be added next to this section.

Because the document focus on the usage of DAL classes more than the usage of form components, the only code block is included, that is responsible with data access of Products results.

```

//p_Condition Examples:
//"SupplierID = 1 AND CategoryID = 2"
//"ProductName LIKE '%Chai%'"
//"ProductID = 5"
private ArrayList LoadProducts(string p_Condition)
{
    Products productLoader = new Products();
    ArrayList products = productLoader.LoadAll(p_Condition, 0);
    return products;
}

```

An example screenshot of Default.aspx has been taken, which list the products, whose Supplier is “Exotic Liquids” and Category is “Beverages”.

Product Id Product Name

Supplier Name Category Name

3 products has been loaded.

ID	Product Name	Category Name	Supplier Name
1	Chai	Beverages	Exotic Liquids
2	Chang	Beverages	Exotic Liquids
98	test 4	Beverages	Exotic Liquids

Figure 5.36 Products Listed On Default.aspx

Obviously, it is very easy to get the records on the database using generated DAL classes, but most of the applications have more requirements than listing a block of records, such as CRUD operations. Next demonstration form will show the way of dealing with CRUD using DAL classes.

The designed form to demonstrate CRUD operations on Products' records includes many sections for each operation. Due to its wide content, each section will be discussed separately. First section of the form is designed to read a single record and to display its content on the form. For the purposes, a combobox is located, which lists the product names, and a button is ready to get the selected product from the database.

Select the product

Figure 5.37 The components For Loading The Product

The code triggered with the button explains the way of a read operation using the generated Products class, where "p_ProductId" parameter is the identity value of the destination record.

```

Products product = new Products(p_ProductId);
productNameLabel.Text = product.ProductName;
supplierIDLabel.Text = product.SupplierID.ToString();
supplierLabel.Text = product.SupplierName;
categoryIDLabel.Text = product.CategoryID.ToString();
categoryLabel.Text = product.CategoryName;
quantityPerUnitLabel.Text = product.QuantityPerUnit;
unitPriceLabel.Text = product.UnitPrice.ToString();
unitsInStockLabel.Text = product.UnitsInStock.ToString();

```

As the result of the assignments of component values, next view will be appeared on the form.

Product Name	Aniseed Syrup
Supplier	1 - Exotic Liquids
Category	2 - Condiments
Quantity Per Unit	12 - 550 ml bottles
Unit Price	10
Units In Stock	13
<input type="button" value="Edit Product"/>	

Figure 5.38 The Form State After Loading The Product

The “Edit Product” button is the gate to an update operation. When its click event is fired, the fields are replaced with the components, that let user to modify the product’s values. Next view displays the instance, after the user make some modification on the record.

Product Name	<input type="text" value="Aniseed Syrup 1"/>
Supplier	<input type="text" value="New Orleans Cajun Delights"/> ▼
Category	<input type="text" value="Beverages"/> ▼
Quantity Per Unit	<input type="text" value="10 - 50 ml bottles"/>
Unit Price	<input type="text" value="20"/>
Units In Stock	<input type="text" value="30"/>
<input type="button" value="Update"/>	

Figure 5.39 The Form State After “Edit Product” Button Pressed

Again, a button, “Update”, is ready to accomplish an action. The name of the action is an update.

```

Products product = new Products(p_ProductId);
product.ProductName = productNameTextBox.Text.Trim();
product.SupplierID = int.Parse(supplierDropDownList.SelectedValue);
product.CategoryID = int.Parse(categoryDropDownList.SelectedValue);
product.QuantityPerUnit = quantityPerUnitTextBox.Text.Trim();
product.UnitPrice = double.Parse(unitPriceTextBox.Text.Trim());
product.UnitsInStock = short.Parse(unitsInStockTextBox.Text.Trim());
productToUpdate.Update();

```

The first line is familiar from the previous part, because an DAL object is to be loaded before an update operation. Each loading means a database access, which decreases our performance. To deal with this issue, caching of the object is recommended after its first load, so it can be

used from the cache and does not need to be loaded again. The last line of the code block is the place, where the update occurs, after replacing the object values with user entries.

The next operation to demonstrate is the insertion. The components used for the update are also used for this one, on which the user enters the values for a new Product record. The user interface and the code fired with the “Insert” button comes next.

Product Name	<input type="text"/>
Supplier	<input type="text" value="▼"/>
Category	<input type="text" value="▼"/>
Quantity Per Unit	<input type="text"/>
Unit Price	<input type="text"/>
Units In Stock	<input type="text"/>
<input type="button" value="Insert"/>	

Figure 5.40 The Form State For An Insertion

```
Products product = new Products();  
product.ProductName = productNameTextBox.Text.Trim();  
product.SupplierID = int.Parse(supplierDropDownList.SelectedValue);  
product.CategoryID = int.Parse(categoryDropDownList.SelectedValue);  
product.QuantityPerUnit = quantityPerUnitTextBox.Text.Trim();  
product.UnitPrice = double.Parse(unitPriceTextBox.Text.Trim());  
product.UnitsInStock = short.Parse(unitsInStockTextBox.Text.Trim());  
productToUpdate.Insert();
```

The code to be executed for an insertion is very similar to the update. The important difference is on the first line, where the object is initialized with the default constructor. A call to the default constructor of a DAL class causes an empty object creation, which also means no database connection is established. It is also logically true, because the record, that will be inserted, is not to exist in the database, in other words, it is an empty object. The code block continues with the assignment statements, and ends with an insert call, where the connection is established and created on the database.

The only operation in CRUD, that is not discussed yet, is the deletion. The demonstration is made by a single button, which triggers the following code block.

```
Products product = new Products(p_ProductId);  
productToUpdate.Insert();
```

As the update operation needs, a deletion is also needs a previously loaded object before its execution.

6 CONCLUSION AND FUTURE RECOMENDATIONS

This thesis is about presenting a solution for the layered applications. Although layered architectures prevent serious problems on supporting such operational requirements as maintainability, reusability, scalability, robustness, and security, the implementation of a planned layered structure probably requires an assigned source of architects, an effort of developers and, most important, a time period of the projects. The tool developed in this thesis offers savings on the sources reserved for the implementation of the data access layer, which is the base layer of any layered architecture. The tool, Data Access Layer Code Generator, generates the DAL components automatically, any of which has a standart structure, are reusable, and are easy to use. The generator supports the generation of codes that are responsible to access data on two most popular database management systems, DB2 and Microsoft SQL Server. On the other hand, it is also designed to provide an easy adaptation of other DBMS'.

The tool produces the DAL classes that are able to successfully process any CRUD operations on the destination databases in multiple ways. Moreover, the developers can define relations between the generated classes to realize the relationships between database tables. Because it is fully integrated on Visual Studio 2005 development platform, the developers can easily adapt to the tool. As much as the usage of the tool increases, the possible errors will be prevented done by the developers on the data access. Additionally, the readability level of the codes will also be improved due to the standardized and commented codes generated by this tool.

As a conclusion, I believe in that the functionalities and the easy usage experience provided by DAL Code Generator tool makes itself one of the first preferences of the developers, where they require access to an information source.

In the future, the tool will be improved with the modules that will generate data access layer components for the sources other than relational databases, such as XML sources. Moreover, some design elements will be builded for the ASP .Net and desktop projects, which will be responsible for adding data access capability to the applications with just a single drag and drop action.

7 REFERENCES

References Cited

- [1] Microsoft, “.NET Framework Conceptual Overview”, URL <http://msdn2.microsoft.com/en-us/library/zw4w595w.aspx>
- [2] Microsoft, “Common Language Runtime Overview”, URL <http://msdn2.microsoft.com/en-us/library/ddk909ch.aspx>
- [3] Netscope, “C# Introduction and Overview”, URL http://www.netscope.co.za/directions/mpnetpro/mpnet_csharpintro.htm
- [4] Microsoft, “Attributes (C# Programming Guide)”, URL [http://msdn2.microsoft.com/en-us/library/z0w1kczw\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/z0w1kczw(VS.80).aspx)
- [5] Jesse Liberty, “Programming C#”, O'REILLY, 2001
- [6] Microsoft, “Visual Studio 2005 Automation Samples”, URL <http://www.microsoft.com/downloads/details.aspx?FamilyId=79C7E038-8768-4E1E-87AE-5BBBE3886DE8&displaylang=en#Instructions>
- [7]. Santhi Maadhaven, “Creating simple Add-in for Visual Studio.NET”, 16 March 2005, URL <http://www.c-sharpcorner.com/Code/2005/March/CreatingAddin.asp>
- [8] Microsoft, “How to: Create an Add-in”, URL <http://msdn2.microsoft.com/en-us/library/80493a3w.aspx>
- [9] Alex Mackman, Chris Brooks, Steve Busby, Ed Jezierski, Jason Hogg, Roberta Leibovitz and Colin Campbell, “.NET Data Access Architecture Guide”, Microsoft Corporation, 2003
- [10] Joe Mayo, “The C# Station ADO.NET Tutorial”, 8 January 2004, URL <http://www.csharp-station.com/Tutorials/AdoDotNet/Lesson01.aspx>
- [11] Manoj G, “Implementing a Provider Independent Data Access Layer in .NET”, 4 November 2003, URL http://www.codeproject.com/vb/net/data_access_layer.asp

References Not Cited

- [12] Microsoft, “Overview of the .NET Framework”, URL <http://msdn2.microsoft.com/en-us/library/a4t23kkt.aspx>
- [13] Microsoft, “Creating Custom Attributes (C# Programming Guide)”, URL <http://msdn2.microsoft.com/en-us/library/sw480ze8.aspx>
- [14] Tom Archer, Andrew Whitechapel, “INSIDE C#”, Arkadaş Yayınları
- [15]. Microsoft, “Custom Add-Ins Help You Maximize the Productivity of Visual Studio .NET”, URL <http://msdn.microsoft.com/msdnmag/issues/02/02/VSIDE/default.aspx>

- [16] Microsoft, “.NET Framework Data Providers”, URL <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconADONETProviders.asp>
- [17] Dino Esposito, “Building Web Solutions With ASP.Net And ADO.Net”, Microsoft Press, 2002
- [18] Mickey Williams, “Microsoft VISUAL C# .Net”, Microsoft Press, 2002
- [19] David Scoppa, “Microsoft ADO.Net”, Microsoft Press, 2002
- [20] Jeff Webb, “Visual Basic .Net And Visual C# .Net”, Microsoft Press, 2003
- [21] Nikhil Kathari, Vandana Datye, “Developing Microsoft ASP.Net Server Controls and Components”, Microsoft Press, 2003
- [22] Martin Fowler, “Pattern of Enterprise Application Architecture, Addison Wesley Professional”, 2003
- [23] Steve McConnell, “CODE COMPLETE”, Microsoft Press, 2004
- [24] Peter Eeles, “Layering Strategies”, 15 October 2001, URL <http://www-128.ibm.com/developerworks/rational/library/4699.html>
- [25] Buschmann, Frank, “Pattern-Oriented Software Architecture”, Wiley & Sons, 1996
- [26] Peter Eeles and Oliver Sims, “Building Business Objects”, John Wiley & Sons, 1998
- [27] Peter Herzum and Oliver Sims, “The Business Component Factory”, John Wiley & Sons, 2000
- [28] Ivar Jacobson, “Software Reuse”, Addison-Wesley, 1997
- [29] David Hayden, “Domain-Driven Design - Layered Applications”, 29 March 2005, URL <http://codebetter.com/blogs/david.hayden/archive/2005/03/29/60806.aspx>