



H-Prog

Design Architecture Document

Business Object Model

Author: **Mohammad S. Hefny**
Creation Date: **Thursday, February 26, 2004**
Last Update: **Wednesday, April 20, 2005**
Version: **3.0.0**



Copyright & Disclaimer

The authors and publisher have made every effort in the preparation of this article and samples to ensure the accuracy of the information. However, the information contained in this article and samples is available without warranty, either express or implied. Neither the authors, H-Prog nor publishers will be held liable for any damages caused or alleged to be caused either directly or indirectly by this article.

All copyrights are reserved for H-Prog



Overview

This document discusses design model of Business & Data-Tier used by H-Prog in developing data tier and business tier. This method was reached by collecting information and sharing knowledge with many parties to get enough experience to define such a model. We are happy to continue sharing this knowledge and getting feedback in order to enhance that model.



Three Tiers Architecture

Three tier architecture means to separate application objects into three major groups, data-tier, business-tier and presentation-tier. Data-Tier represents the group of objects responsible for data manipulation including adding, modifying, retrieving and deleting operations.

Data-tier is aware of the nature of the data storage media, i.e. the data tier "*knows*" that it weather to use a MS-SQL, an Oracle database or a simple text file as a storage media. It is also aware of the database schema¹. Data-tier objects "*blindly*" access the storage media that means it does not make any verification on the data given from the business tier; only minor validations such as field types are done. Keeping the logical consistent of data is the business object job.

Business-tier objects are completely unaware of the nature of the storage media, it is not important for them where to store the data as long as the data is properly stored and handled. The main function for this tier is to implement application business rules.

Presentation-tier is the only visible part of the application; its function is to provide GUI that enables human-application interaction.

An important point to notice here is that the tiers are logical boundaries and in many cases they have no relation with the physical deployment of the application. One can make a three tier application that is deployed by XCOPY one executable file only. For enterprise application different parts of tiers can be put together in different ways, the only rule is that presentation objects access business objects only business objects access data objects; this is the only rule we have to stick with.

¹ Stored procedures can hide table names and relations from the data tier. The data-tier section will discuss this in details.



Database-Independent Data Tier

In this section we are going to give generic architecture of data-tier. As previously discussed, data-tier is aware of the nature of the data storage media, i.e. the data tier "*knows*" that it weather to use a MS-SQL, an Oracle database or a simple text file as a storage media. It is also aware of the database schema. It is recommended to call stored procedures rather than directly access the database tables. The benefits of using stored procedures are:

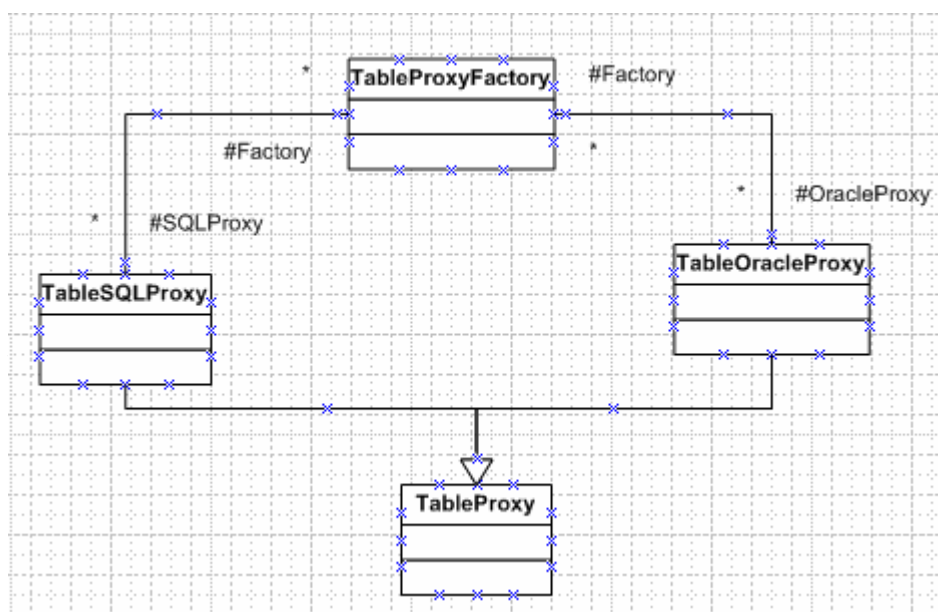
- Provide flexibility to change table's internal structures as well as foreign keys relation without the need of changing the code at all. Only internal implementation of stored procedures will be changed and that is all.
- Efficiency; MS-SQL server caches stored procedures, so it does not have to recompile T-SQL once again before re-executing it.

Data-tier objects "*blindly*" access the storage media that means it does not make any verification on the data given from the business tier; only minor validations such as field types are done. Keeping the logical consistent of data is the business object job.

In the following we will discuss how to design a database-independent data-tier. The data tier should have the following features:

- It should provide the business tier with the same interface regardless of the storage media.
- It should be easy to support new storage media or databases.
- It should be easy to make major changes into ERD model without affecting upper tiers.

The following is the UML represent a general way to access the database:





We make use of two design pattern object, the proxy and the factory classes. Each table or collection of tables that store a certain type of data, e.g. user data, security data ...etc. has a TableProxy abstract class, where TableProxy can be CustomerProxy, SecurityProxy, ...etc.

This abstract class contains virtual "overridable" functions that define different operations that can be performed on that object or object group. Such functions can be AddCustomer, ModifyCustomer and DeleteCustomer. These functions are primitive function that implements no business logic. Only validation regarding to database are performed.

The abstract class is inherited by another class that its function is to implement the abstract class on a certain database. These classes have the same methods but in this case it contains the SQL statements and name of stored procedures to perform these functions.

The factory class is called from business tier to return TableProxy. The factory class checks for the actual database type and returns a correspondent class such as TableSQLProxy and TableOracleProxy.

Another issue here is that Proxy classes never access the database directly. It uses DAL "data access layer" classes to access the database. DAL classes provide consistent way to access database.

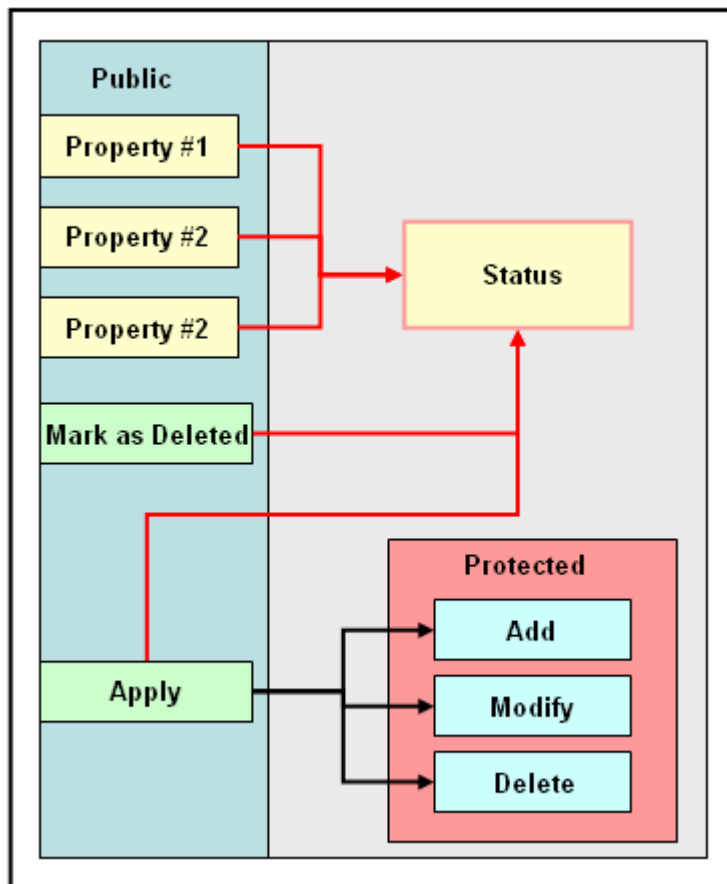
The ConnectionString class function is to retrieve the connection string of the working database, as well as returning the type of the database. This class is used by factory classes as well as proxy classes. The ConnectionString class can store this information in the registry or any suitable file. This is a static class with static "shared" members.



Business Tier

The business-tier separates the data tier from the presentation tier; it contains all business logic and forms the logical representation of the application while data-tier views the system from a database perspective.

The parent object of all business objects is [BusinessObject] class. The class contains a built-in state that decides what action should be applied at any given time. The user of this class has to call Apply() function only, he/she never calls add, modify or delete, in fact he/she never knows what operation will be done by calling apply(), it is fully determined by internal state only.



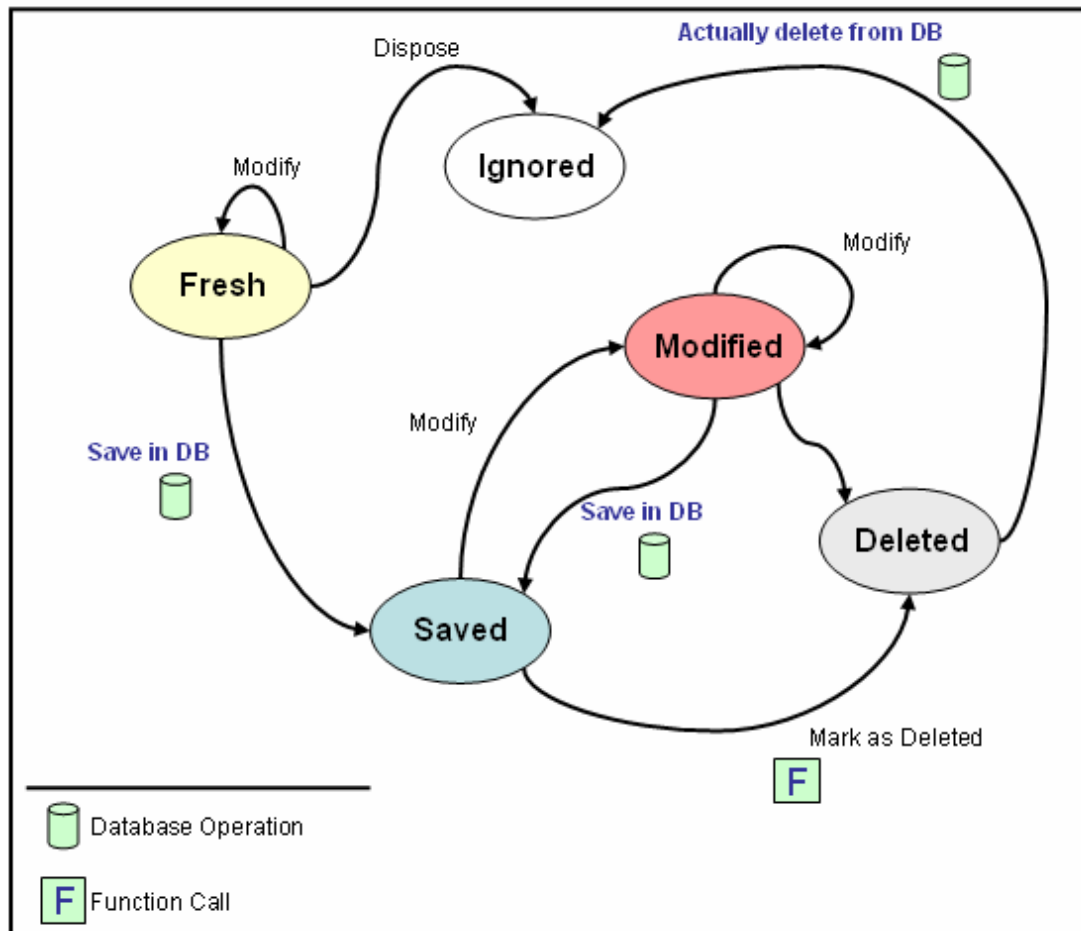
The above figure shows the main idea to achieve this is to make a *closed* object that is only accessible from its properties, the properties handles object status internally. The object has two main public functions; MarkAsDeleted() function which is called to *tell* the object to delete itself from DB next time Apply() is called, and Apply() function which is called to make actual Add, modify, or delete. Object user will call apply function that will decide the proper operation and will update object status.



Business Object State Diagram

The object has a private member called mStatus, mStatus is an enumeration that can take one of the following values:

- Fresh
- Saved
- Modified
- Deleted
- Ignored



Fresh: When created a new object its status is set to "Fresh". This means that this is a new object that is not stored in the database yet. The object stays into this state until it is "Saved" or "Ignored"; if the object is marked as deleted it goes to state "Ignored". Any access to object properties will not change its "Fresh" status as whatever data is defined or changed the object is still new and not originally fetched from database, only can Apply() or MarkAsDeleted() change its status.

Saved: Any object that is created by retrieving its data from a storage media such as database has status "Saved". It is a clean copy of the data stored in database. The object can move to state "modified" if any property has been accesses in write mode, any change to any object properties means that it is no longer the same copy of the database so it has been modified and need to be copied to database before return to "Saved" status again. Calling "MarkAsDeleted" function will change current status of object either it is "Saved"



or "Modified" to "Deleted". Status "Deleted" does not mean the object is deleted. It means that the object *wants* to be deleted next time Apply() function is called.

Deleted: This state means a "Saved" or "Modified" object wants to be deleted, that means when Apply() is called the Delete function should be invoked to *actually* delete the object. The object status then changes to ignored. There is no point to convert a "Fresh" object to "Deleted" object as it already has no copy in database, that is why the a "Fresh" object directly moves to "Ignored" status.

Ignored: This is the status where actual delete exists. An ignored object means that the object is no longer existed in database and no longer considered as active object. It is a kind of garbage object. This state is very important if you need to provide an UNDO or UNDELETE feature.

The object pseudo-code

In the following we will discuss BusinessObject actual source code:

```
Public Enum Object Status
    Fresh
    Ignored
    Saved
    Modified
    Deleted
End Enum

Class BusinessObject

    Protected mStatus As ObjectStatus

    Public Overridable Sub Apply ()
        Select case mStatus
            Case "Fresh"
                Add()
                mStatus = Saved
            Case "Modified"
                Modify()
                mStatus = Saved
            Case "Deleted"
                Delete()
                mStatus = Ignored
        End case
    End Sub
```



```
Public Overridable Sub MarkAsDeleted
    mStatus = "Deleted"
End Sub
```

```
Protected Overridable Sub Add()

End Sub
```

```
Protected Overridable Sub Modify()

End Sub
```

```
Protected Overridable Sub Delete()

End Sub
```

```
End Class
```

As we can see from the previous sample, the class internally handles its state. Apply() function determines which operation to make using depending on current status and after operation completed it updates the status.

My First Business Object

Now it is time to create our first Object using this model. We will create a very simple object which represents a Person with PersonName & Age

```
Class Person
    Inherits BusinessObject

    ' Attributes

    mPersonName as String
    mAge as Integer

    Public Property PersonName as String
        Get
            Return mPersonName
        End Get

        Set (ByVal Value as String)
            PersonName = Value
            MyBase.ChangeStatus (ObjectStatus.Modified)
```



```
        End Set
    End Property

    Public Property Age as Integer
        Get
            Return mAge
        End Get

        Set (ByVal Value as String)
            Age = Value
            MyBase.ChangeStatus (ObjectStatus.Modified)
        End Set
    End Property

    Protected Overridable Sub Add()
        ID = DataTier.Person.Add (PersonName, Age)

    End Sub

    Protected Overridable Sub Modify()
        DataTier.Person.Modify (ID, PersonName, Age)
    End Sub

    Protected Overridable Sub Delete()
        DataTier.Person.Delete (ID)
    End Sub

End Class
```

A business object should not have any public attribute, all inputs should be done through methods and properties, and logic should be applied there to change the object status according to the current object status and the given input values.

The major mistake is to forget to put `ChangeStatus()` at any write access property, failure to put it will make the business object status inconsistent and `Apply()` function will take unpredictable behavior accordingly.



Other Issues with Business Object

If a business object e.g. "TeamLeader" has an attribute called "MyTeam" which is a collection of Developers business objects. TeamLeader object should override Apply() and call apply of each Developer in the collection when Apply() is called.

```
Public Overrides Sub Apply ()  
  
    MyBase.Apply()  
  
    For each oDev as Developer in Me.MyTeam  
        oDev.Apply()  
    Next  
  
End Property
```

Now, why do we do all of that? Assume you define a developer of title team leader, and you selected his team from a list of all developers, you also defined some more developers who were not on the list while you still in the definition of that team leader!! That means you created many objects that has no existence in database and you also linked them to TeamLeader developer by including them in MyTeam collection of the team leader which he is not saved yet in the database. Using this model you only have to call TeamLeader.Apply() which will stored the team leader and all fresh developers and link them to him as well in one step. This makes presentation tier fully independent and handling business objects is easy and trivial.

Extra Features

H-Prog business object is not only what this document mentioned. The object has extra features to enable avoid loading unnecessary data to keep network bandwidth reasonable in case it access a web service or a remote data-tier. Also other helper features such as attached status which is used to help linking objects that has one-to-many or many-to-many relations between them. We will explain these features later, although source code of it already published.



Feedback

Your feedback is highly appreciated. Please send your feedback to:

info@h-prog.com