Data Abstraction Layer

Independent from Database

By
Nguyễn, M. Hải - Chief Architect

Updated 07/08/2008

THN Solutions (Thiên Hải Ngọc manufacturing & trading company)
website: www.thnsolutions.com

Contents

Abstract

Developers are trained to think in OOP (Object-Oriented-Programming), but work with relational data. Impedance mismatch is inevitable and counter-productive. This problem is so serious that many programmers think all applications consist of three layers, in which the data layer requires the most significant effort. In fact, many programmers are more proficient at writing SQL queries and stored procedures than understanding what an interface or abstract class is.

Sadly, this phenomenon is widespread. The author has spoken to different technical managers in several large software companies. Many of them fanatically believe the only correct way to program is to write manual data access methods using SQL. They simply don't believe in anything else, even if their method is proven to be less productive than others 10 - 20 times.

In order to save efforts and resources, one needs to treat data exactly as what they are: data, not to be dictated by them. Data Abstraction Layer is one of several methodologies designed to do that. Using DAL, programmers can forget tables, rows, SQL altogether to work comfortably in OOP environment. This design blends in so well that programmers can use the same code for different relational databases, different data source technologies, and even for different tiers within one application or inter-applications.

Data Access Object

Writing manual SQL codes is a proven nightmare. As the result, in the popular 3-layer architecture, programmers are required to dedicate a data layer to abstract away the SQL codes. However, when it comes down to implementing the data layer, there are many different schools.

Manual Data Access Object

This school attracts the most followers. These programmers know SQL syntax by heart. Many of them even master the dark art of debugging stored procedures and query execution plan optimization.

Typically, for each and every table in the database, there will be at least one Data Access Object[1] (DAO) class that contains the plumbing codes to invoke SQL commands. The programmers of this school has been trained to believe that this is the best method. Their beliefs are based on the following false premises:

- Stored procedure is faster than dynamic query
- Dynamically generated code is slower than hand-code
- Easier to understand when you are in control of your code
- Business entities are best represented by relational tables

Code 1.1: a typical DAO class

```
    /// <summary>
    /// Data access object for Customer table
    /// </summary>
    public class CustomerDAO : ICustomerDAO
    {
        public IDbConnection GetConnection() {}

        public int Insert(string Name) {}
        public int Insert(string name, DateTime dob, string address){}

        public void Update(int id, string name) {}
        public void Update(int id, string name, DateTime dob, string
                          address) {}

        public void Delete(int id) {}

        public DataSet Select(string sqlQuery) {}
        public DataSet SelectAll() {}
        public Customer SelectByID(int id) {}

        public DataSet SelectProductPurchased(int id, DateTime date) {}
    }
```

Code 1.2: a typical DAO method

```
        public int Insert(string name, DateTime dob, string address)
        {
            IDbConnection con = GetConnection();

            //build SQL query
            string sql = "Insert Into Customer Values({0}, {1}, {2})";
            sql = string.Format(sql, name, dob, address);

            //build command & execute
            IDbCommand cmd = new SqlCommand(sql, con);
            cmd.ExecuteNonQuery();
        }
```

The Data Access Object is a well-known, well-used pattern. However, it brings too many problems along with it.

Redundant codes
For each table, there is at least one DAO class. Frequently, a DAO class and a DTO (Data Transfer Object[2]) class are required.
For each data operation, at least one method is needed in DAO class. For example, select all products and select a single product by its identifier are two distinct operations, each requires its own DAO method.

Redundant tests
Unit testing for DAO layer is a painful task for developers. For each DAO method, at least one testcase is required. The more methods there are, the more test there will be.

Dependent on database
Once the code is written for a specific database management system (DBMS), it is not wise to switch to a completely different DBMS. Switching frequently means rewriting and retesting codes.

Intolerant to changes
If either the database schema or the business entity structure is changed, codes must be rewritten. Not only the DAO class itself, but classes that use the DAO class must be revised as well.

Code Generation

Automatic code generation shines where there are repetitive and similar codes. Since the glorious days of Delphi, code generators and transformers have popped up like mushroom after rain. Generators are indeed very useful tool to generate data layer. One can either ask a generator to generate the database from business objects, or generate the business objects from the database's structure.

However, once you have a ticking bomb on your hand, it does not matter whether you are walking on foot or riding a plane. Once you have a bad design and a faulty methodology, automatically generated code only make the problems come to you faster. As long as the code is still based on Data Access Object pattern, it inherits all the problems of this pattern.

THN's Data Abstraction Layer

Introduction

THN's **Data Abstraction Layer** frees developers from dealing with data persistence mechanism. Except for extreme performance scenarios, developers will no longer have to care about where the data come from and how they will be stored. Data are presented to developers as pure OOP objects. It doesn't matter whether the underlying data storage is MS SQL, Access, Oracle, or MySQL, etc. It doesn't matter whether the storage methodology is relational database, object database, hierarchical file, web service, xml, or even queued messages. To a developer's view, the data are the objects he codes. He doesn't have to master T-SQL to store his objects.

Features

- Works with MS SQL, Oracle, Access, MySQL, Firebird
- Works with practically any RDBMS
- Works with ODBMS and unconventional DBMS (web service, xml, email, custom, etc.) using simple adapters
- Support type safety
- Flexible query object
- Flexible conditional object
- Serializable query commands
- Multi-level and deferred transaction
- Atomic transaction by design
- Automatic SQL-injection attack prevention
- 100% pure managed code
- POCO (plain old C# object). Conform to CLS 1.1

Quick Tutorials

Since most programmers are familiar to T-SQL, the DAL is designed to use a query object pattern similar to T-SQL. But the similarity is superficial. Regardless of the actual storage mechanism, developers only need to be aware that he is working with his own data objects.

Connect to Database
Connecting to the actual physical storage can be done by manual code, or via an xml config file.

Code 2.1 Connect to Access Database by manual code

```
        IDatabase db = new AccessDatabase();
        db.Initialize("connection string");
```

Code 2.2 Connect using an xml config file

```
        IDatabase db = null;
        DataConfigurator config = new DataConfigurator();
        db = config.Load("Data.config", "Access");
```

The IDatabase is the abstraction of any data storage. The interface IDatabase is used for all data-related activities such as insert, update, delete, select, transaction. Using this interface, consumer codes are protected from coupling with the concrete database implementation.

Create Structures

Data programming usually starts with designing a database first, then create object classes that are tightly based on the database design. One can continue to do so with DAL. But if we do not need to care how our objects are stored, why should we be concerned with database designing? Why not have someone else do that for us?

Assuming we have a blank database and a nicely designed Product class, the following codes will create the Product table to the database for us:

Code 2.3: Create Product table from Product class

```
        db.Register(typeof(Product));
        db.CreateStructure(false);
```

Insert Data

Insert, Update, Delete object is truly simple with DAL. The following codes demonstrate how to insert object. Update and Delete are done similarly.

Code 2.4: Insert a product instance to database

```
        Product p = new Product();
        p.Name = "Apple";
        p.Price = 9.99;

        db.Insert(p).Execute();
```

The first three lines create a new product instance and populate its properties. The last line save the product into the underlying database. As you can clearly see, there is no SQL involved at all.

Not only that, notice how the product instance is populated. Since Product is an OOP class, we can enjoy the comfort of type-safety. The DAL will handle all the type conversion from OOP types to database types automatically.

Furthermore, SQL-injection attack prevention is already built-in in the DAL. We do not need to deal with that. With the above simple lines, our code are already protected.

As a side bonus, the DAL's flexible API allows us to insert object in bulk easily:

Code 2.5: Insert an array and a list of products

```
        //insert an array of products
        Product[] products;
        db.Insert(products).Execute();

        //insert a list of products
        IList productList;
        db.Insert(productList).Execute();
```

Transaction

Encapsulating database activities within an ACID transaction is where many programmers make designing errors. Without careful planning, wrong transaction design may render the system unscalable or even worse, create random data artifacts that are very hard to track and debug.

Most database management system does not support nested (multi-scope) transactions and certainly not deferred (long-execution) transactions. The DAL, fortunately, encapsulate the transaction concept and make it totally independent from actual database implementation.

The transaction can be done manually by relying on the physical DBMS's transaction support like the following:

Code 2.6: Manual transaction:

```
        object context = db.BeginTransaction();

        db.InsertObjects(p);
        db.InsertObjects(p2);

        db.CommitTransaction(context);
```

Manual transaction can support multi-level transaction, even if the physical DBMS does not support it, by the notion of transaction context. However, this model does not support deferred transaction. Most importantly, this model is not thread-safe and the developer must well understand what he is doing.

The DAL supports multi-scope and deferred transaction natively. It does not matter whether the physical DMBS supports multi-level or deferred transaction or not. In fact, even if the underlying DBMS does not support transaction, DAL can still supports transaction by transaction object model.

Code 2.7: Example of using transaction:

```
        db.Begin
                .Insert(p)
                        .Begin
                            .Insert(p2)
                        .End
          .End
          .Execute();
```

Query commands such as insert, update, delete will be encapsulated in transaction scope bounded by corresponding Begin and End pair. All commands will not be executed until method Execute() is invoked.

Using this model, all activities are practically atomic by design. The query object can have as many levels and as many commands as necessary. That means the query can be built dynamically in various tiers of the system. It can be stored, replayed, transferred over the network, etc. With all of that nice stuff, the developer's codes are still encapsulated in atomic transaction automatically.

Select Data
Selecting data is similar to SQL (for the sake of developers' learning). But instead of DataSet or DataRows, DAL return actual objects for us.

Code 2.8: Select product Apple

```
        Product p = db
                .Select
                .From(typeof(Product))
                .Where
                    .Equal("Name", "Apple")
                .Execute()
                .AsSingleObject as Product;
```

Code 2.9: Select all product with price > 10 and sort by name

```
        IList products = db
                .Select
                .From(typeof(Product))
                .Where
                    .Greater("Price", 10)
                .Execute()
                .AsSortedList("Name");
```

As the above codes indicate, results are returned as strongly-typed objects. We can also apply transformation to the returned objects such as sorting them.

Benefits

- Reduced coding time
- Reduced testing time
- Reduced designing time
- Reduced time to market
- Independent from physical database
- Protected from changes in DBMS or database design
- Protected from SQL-injection attack

In simple words, Data Abstraction Layer from THN saves time and efforts.

DAL Compared to DLINQ

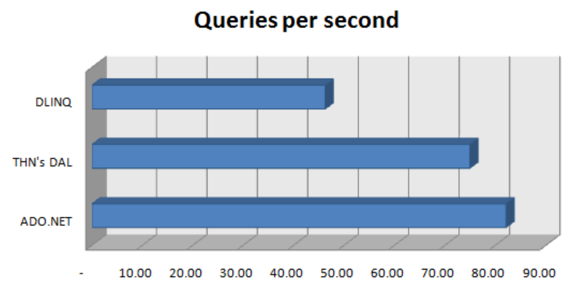
DLINQ[3] from Microsoft is a very powerful and nice query technology. However, DLINQ may not be a very attractive solution when it comes to data abstraction.

Portability
THN's DAL conforms to CLS version 1.1. That means it can be ported to Linux, Mac and other environment easily, as long as .Net framework version 1.1 is supported. In the rare cases, if porting to languages other than .Net is required, POCO conformant means DAL can be easily ported because it uses no fancy features or syntactic sugar of .net language. On the other hand, DLINQ requires .Net 3.0.

Database Independence
THN's DAL is designed on the principle of database independency. While DLINQ only works with MS SQL (3rd-party wrappers required for other DBMS), DAL can work with most relational DBMS and other DBMS methodologies.

Learning curve
DAL is built for simplicity. DLINQ is overly complex. To learn DAL, a developer needs about 15 minutes. To learn DLINQ, a developer need to master both SQL, ADO.NET, LINQ, and DLINQ.

Performance



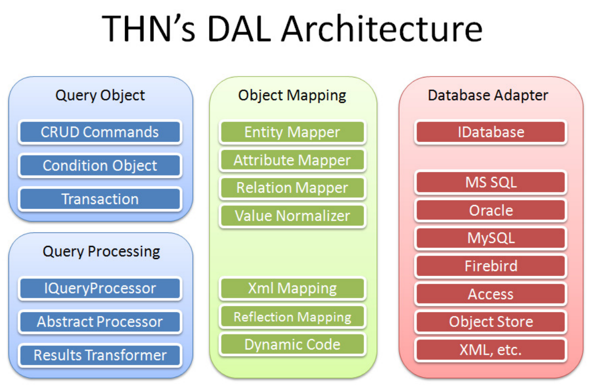
The above diagram shows that DAL imposes roughly 9% overhead over ADO.NET. Whereas, DLINQ imposes 46% overhead.

POCO data objects
DLINQ and many O/R mappers require data objects to use special techniques or, at the least, must be inherited from special base classes. On contrary, THN's DAL can work with .Net objects natively. No base class to inherit from, no interface to implement, no special tricks to handle objects.

How Extensible is DAL?

THN's DAL is designed to be decoupling from data storage as much as possible. Even though O/R (object-to-relational) mapping is a major feature, it is not the only capability. DAL can be used for non-relational databases such as object database, object collection, xml database, etc. Wherever there are objects and they need to be managed, DAL can be used. In fact, our FlexObj technology (a business object layer) uses parts of DAL to handle object querying. The FlexObj, as a business layer, does not know anything of the physical database. Not only that, since FlexObj can be used in standalone as well as client-server architecture, developers should be able to use the same code for both local and remote scenarios. Using DAL, FlexObj allows developers to use the same flexible API for local, remote, and physical data store transparently and automatically.

This diagram depicts the overall architecture of DAL:



Thanks to its open architecture, DAL can be customized to support new databases, can be optimized for better object mapping performance, or can be plugged in to a totally different architecture that does not know anything about databases (as in our FlexObj case).

Is object mapping by XML definition file or attribute-based not your taste? Extend the mapping component. Is a different database not supported yet? Write a new, simple database adapter. Need to intercept query processing? Override or implement a new IQueryProcessor. Need faster, smarter querying? Put a cache to query processor. The freedom to extend is great!

Final Thoughts

Since the last redesign of THN’s Data Abstraction Layer a couple years ago, the author has almost forgotten SQL completely. Even connection string has slipped away in his mind.

To be objective, THN’s DAL is an abstraction layer that aims at freeing developers from dealing with the nuts and bolts of data management. For a tiny bit of performance sacrifice, DAL dramatically improve the productivity and quality of developing data-aware applications. Not only that, DAL can also be integrated to other tiers to open new possibilities.

References

[1] **Alur, Crupi, and Malks.**, *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2001, Data Access Object

[2] **Martin Fowler et. al.**, *Patterns of Enterprise Application Architecture*, Data Transfer Object, p306

[3] **DLINQ**, *Language Integrated Query,* LINQ for SQL
http://msdn.microsoft.com/