275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

Stop Hacking, Start Mapping: Object Relational Mapping 101

By: Stephen Forte, Chief Strategist. Telerik Corporation.

Executive Summary

As the industry has moved from a three tier model to n-tier models, the object relational impedance mismatch has become more prevalent. Object Relational Mappers (ORMs) exist to bridge this gap as best as possible. Telerik's OpenAccess ORM is an industrial strength ORM that will meet the needs of all modern applications. It offers a wizard for both Forward and Reverse mapping to all major databases including Microsoft SQL Server, tight Visual Studio integration, LINQ support, transparent persistence and lazy loading, runs in medium trust, as well as a fully scalable architecture via its Fetch Plans and Level 2 cache.

Contents

The Three Tier Model

Business applications today all access data as part of their core functionality. As relational database servers gained in popularity 20 years ago, the industry moved from a one tier (mainframe) model to a client server model where we had a client performing the presentation logic and most of the business logic and the server

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

with the data storage and some business logic in the form of stored queries. By the early 1990s this model broke down due to high maintenance costs and a lack of a separation of concerns and we moved to a three-tier architecture as shown in Figure 1.

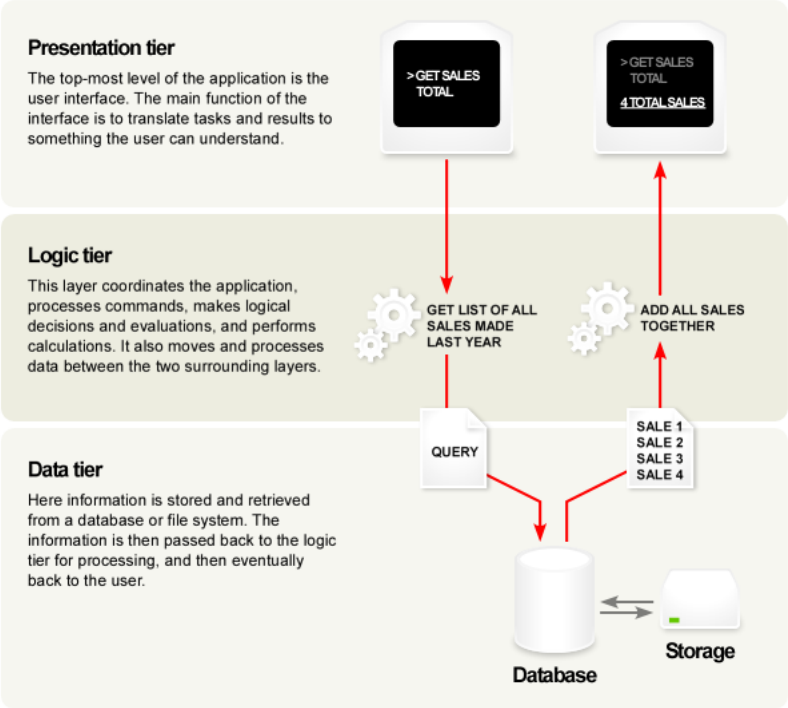

Figure 1. A 3-tier architecture. (Source: Wikipedia)

A three-tier architecture enforces a logical (and sometimes physical) separation of three major concerns:

- *Presentation*-the user interface components.
- *Business Logic*-processes commands requested by the users, makes logical decisions such as calculations and computations. Retrieves data from the data tier.
- *Data storage and retrieval*-storage and retrieval of data for the system and passes the data to the business layer for processing and ultimate rendering to the user by the presentation tier. In theory this tier can be file storage, XML, or a relational database, however, a relational database is by far the most common usage.

The goal of separating the logic is twofold. First there is a performance gain by having the database server focus only on database storage and retrieval. Specific hardware and topologies (such as RAID) are used for database storage and access that are different from an “application server” or middle tier of business objects and logic. In addition with powerful client machines it made sense to push UI processing down to the client.

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

Second was the separation of concerns principle. By separating out the logic of a system, you can easier maintain the overall system, reuse code, and keep associated logic and code in one location.

The n-Tier Model

By the later 1990s, the industry extended the three-tier model to a multi-tier approach. The model is logically the same but what forced it to change was that the Internet became an important part of many applications.

Web services (and later REST data) have become more integrated into applications. As a consequence the data tier usually became split into a data storage tier (database server) and a data access layer or tier (DAL). In very sophisticated systems an additional wrapper tier is added to unify data access to both databases and web services. Web browsers were far less powerful than a traditional client tier application and the user interface logic became split across the browser with JavaScript and the server with web server UI rendering logic such as ASP or PHP.

Tiers started to get blurred ever further with the addition of stored procedures by all the major database vendors and open source databases. This spread some business logic from the business tier to the database tier, creating tiers within tiers. For example a business component inside of Microsoft Transaction Server (an Object Request Broker or ORB) is a logical business tier; however, it most likely calls a stored procedure, which is a logical business tier inside of the database tier.

As tiers got more blurred due to the Internet, technology innovations and services, the three-tier model evolved to the n-tier model as shown in an example in Figure 2.

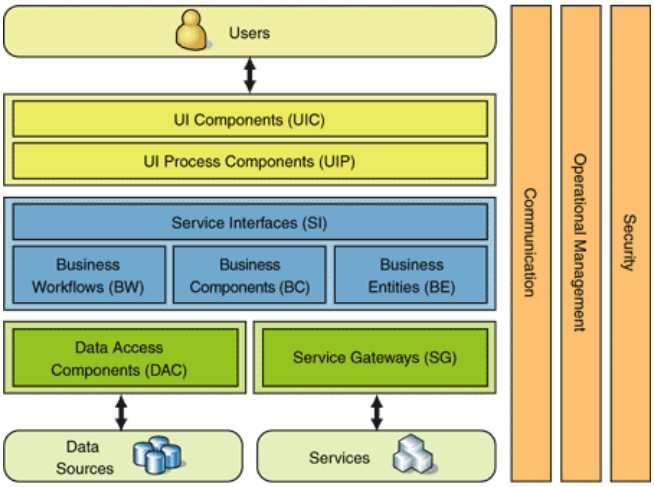

Figure 2. An n-tier architecture. (Source MS Patterns and Practices)

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

The Problem with the n-Tier Architecture

The n-tier architecture has been very successful. Most sophisticated applications today use some form of the n-tier model. Years ago to support this model, enterprises would have as many as five different job titles around the data centric business application. The job titles were:

- Data Modeler
- Database Administrator
- SQL Programmer
- Object Modeler
- Application Developer

The data modeler would design the physical tables and relationships. The DBA would create the tables, maintain them and come up with an index and physical disk strategy and maintenance plan. The object modeler would build and object model (or API) and map behaviors to methods. The SQL programmer would work with the object modeler and application developer and write stored procedures and views under the guidance of the DBA. The application developer would use the object model components and would "glue" the application together.

That was then. While some large organizations still develop this way the advent of RAD tools in the late 1990s, the agile/XP movement in the early part of this decade, the .com boom, and ultimately offshoring and budget cuts, most firms do not organize this way anymore. Many smaller shops have one "database guy" (if at all) and one "code guy." Some only have one person in total. Companies then push most of the modeling and procedure creation to the "db guy" and application developer.

With budget cuts, smaller teams, and the rise of cross-function teams there are not enough "database guys" to go around. Developers complain that they spend over 30% of an application's code on database access code. This only exacerbates the object-relational impedance mismatch.

The Object-Relational Impedance Mismatch

Database normalization theory, based on mathematics, encompasses different strategies than object oriented theory, which is based on software engineering principles. Database tables model data and prescribe storage techniques. Objects model data and behavior. The problem is that there are subtle differences between the way a database is designed compared to the way an object model is designed. The approach of doing straight mapping of database tables to objects leads to the famous "object-relational impedance mismatch."

To demonstrate the impedance mismatch let's take a look at an example. This sample was designed by Scott Ambler in a more detailed discussion of the impedance mismatch. Below in Figure 3 is a simple database model diagram. There are four database tables: a Customer table, an Address table, and a many-to-many table called CustomerAddress, linking them together, representing the notion that a customer can have multiple addresses and addresses are reusable, even between customers. Finally, there is also a support or "lookup" table for States, representing a one-to-many relationship between States and Addresses.

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

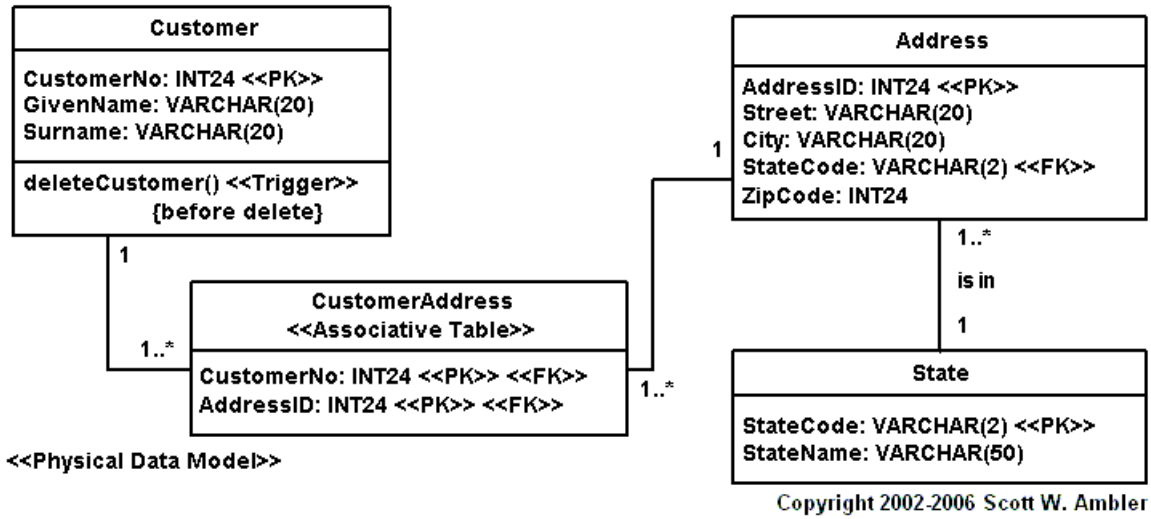

Figure 3. A database model. Source: http://www.agiledata.org/essays/impedanceMismatch.html

Now let's consider an object model that would interact with this data. In Figure 4, we have four objects: a Customer object, an Address object and a supporting State object. Notice that there is no need for the "many to many" database table to be modeled since there is a "lives in" and 1..* arrow notation between the Customer and Address objects. This will indicate that you will have an Address collection associated with the Customer. In addition we have a separate ZipCode Object that does not exist in the database model. This object is used for validation and formatting, behaviors that are not necessary to model in the database.

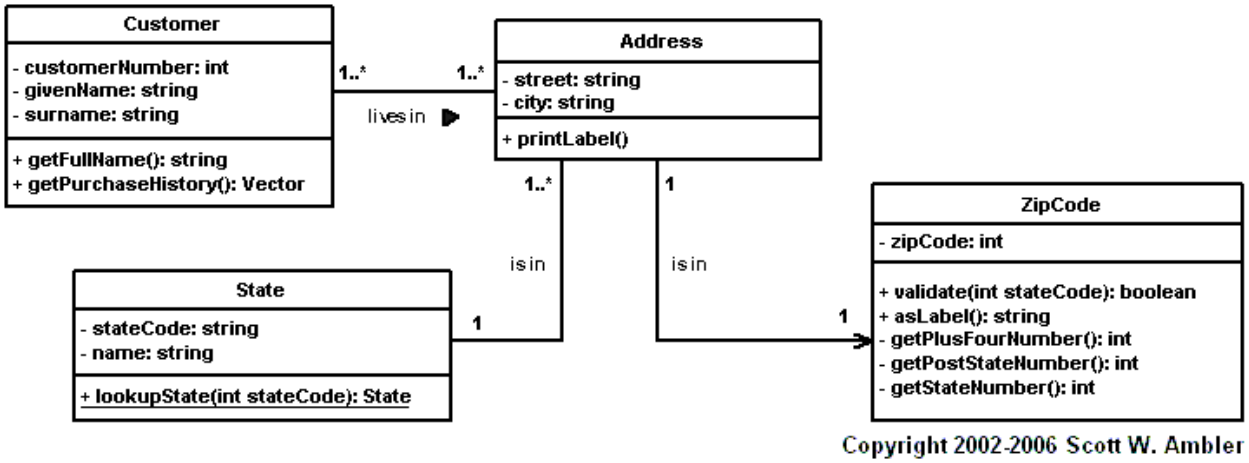

Figure 4. An object model. Source: http://www.agiledata.org/essays/impedanceMismatch.html

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

Notice the subtle difference between Figure 3 and Figure 4? They each have 4 objects, some are identically named (Customer, Address, and State) and look similar, but the ZipCode object is quite different from the ZipCode represented in the database model. This is because object models structure both data and behavior while the database model only models data. Triggers are not considered a behavior, it is more about data updates and flow.

Data Access Layers (DALs)

The impedance mismatch has no easy solution, there are fundamental differences in the database normalization theory and object oriented theory. To bridge the gap as best as you can and to reduce the amount of code you have to write for database access developers have taken to write data access layers (DALs). The goal of a DAL is to decouple the data access code from the object model, allowing the object model to evolve according to its behaviors and the database to evolve based on its specific needs. A DAL should have these characteristics:

- Be completely independent of the object model. In other words the DAL should place no constraints on the object model.
- The DAL should hide all the data access code from the object model. This is sometimes referred to as persistence ignorance. Your object model should not care if you are using raw SQL, stored procedures, or an ORM. If you have the name of a stored procedure or SQL inside of your business (domain) objects then you violate this point.
- The DAL should be able to be ripped out and replaced with minimal to no impact.

The problem with DALs is that they are time consuming to write and not easily reusable from application to application. This is where ORMs come in.

Object Relational Mapping (ORM)

To overcome the impedance mismatch in a DAL you have to understand and implement the process of mapping objects to relational database tables. Class attributes will map to zero, one, or many columns in a database table. If there is proper mapping and persistence ignorance, the developer should only have to worry about working with the objects in the domain model and the DAL takes care of the rest. One way to facilitate this is to use an Object Relational Mapper or ORM.

An ORM is automated way to create your DAL based on your database and object model by mapping domain objects to database tables. Wikipedia defines an ORM as: “a programming technique for converting data between incompatible type systems in relational databases and object-oriented programming languages. This creates, in effect, a "virtual object database," which can be used from within the programming language.” An ORM has to provide a facility to map database tables to domain objects, usually a design surface or wizard. This mapping is in-between your database and domain model, independent from the source code and the database. The ORM runtime then converts the commands issued by the domain model against the mapping

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

into back end database retrieval and SQL statements. Mapping allows an application to deal seamlessly with several different database models, or even databases.

An ORM's mapping eliminates the need for a developer to manually write an entire DAL. A good ORM will eliminate 90% of the DAL code with its mapping and persistent objects. Since developers tend to focus 30% of their code on writing a DAL, an ORM will save over 25% of the total application code. Since the DAL is generated by an ORM and not by hand, it will be error free and always conform to product standards, standards that are easier to change in the future if products change.

When it comes to mapping there are two approaches that an ORM can take: forward mapping and reverse mapping. Forward mapping will take your already existing object model and then create a database schema out of it. Reverse mapping is the process of taking an already existing database and creating a set of objects from the tables. Most ORMs will support either forward or reverse mapping, and some support both methods. An automatic one-to-one mapping may not be preferred due to the differences between the object model and an appropriate data model (as show in Figures 3 and 4), most ORMs will give the user the ability to alter the mappings.

Besides mapping, an ORM has to provide a way for your domain objects to issue database agnostic commands to the DAL, have the DAL translate that command to the appropriate SQL, and then return a status (in the case of an update, etc), an object or object collection to the requesting domain object. This has to be done in a transparent way, the objects in your domain need to be ignorant of what happens behind the scenes in the DAL (persistence ignorance). Popular ways to do this today are using an ORM LINQ provider or an ORM specific object query language (OQL). The most important thing is that you are querying your DAL in an agnostic way. For example consider this simple LINQ Query against a popular ORM:

```
var result = from o in scope.Extent<Order>()
             where o.Customer.CustomerID.Matches("ALFKI")
               select o;
```

This code only concerns itself with the object model and more specifically with the Order object and returns a collection of Orders filtered by the Customer ID "ALFKI". Taking the example further, following code demonstrates using an ORM to filter an ASP.NET GridView based on the results of the user input, notice that we are querying objects (the Order again) and projecting the results into a new object, never worrying about the database, connections, or any SQL code. That was handled by the ORM in the DAL.

```
protected void DropDownList1_SelectedIndexChanged(object sender, EventArgs e)
{
       //get the selected customer id
       string customerid = DropDownList1.SelectedValue.ToString();

       //linq query
       IObjectScope scope = ObjectScopeProvider1.GetNewObjectScope();
       var result = from o in scope.Extent<Order>()
                    where o.Customer.CustomerID.Matches(customerid)
                    select new { o.OrderID, o.ShipName, o.ShipCity, ShipCompany =
       o.Shipper.CompanyName };

       //databind the results
       GridView1.DataSource = result;
       GridView1.DataBind();
```

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

```
}
```

In addition to mapping and commands, ORMs should also integrate with development IDEs such as Microsoft Visual Studio (or Eclipse), provide caching techniques, integrate with source code version control software, and provide a testable framework. This will enable better developer productivity while saving the developer from having to write the DAL from scratch.

Telerik OpenAccess ORM

In November 2008 Telerik released OpenAccess ORM to its customers. OpenAccess is a mature ORM that was developed by Vanatec prior to its acquisition by Telerik in Autumn 2008.

Mapping

OpenAccess has deep integration with Microsoft Visual Studio and provides both forward and reverse mapping to six major databases including Microsoft SQL Server and Oracle with an easy to use Wizard shown in Figure 5. As you can see in Figure 5, the wizard gives you full control over what to map and how to map it (as a class or collection). OpenAccess gives the user complete control over the object and member naming (or database field and table naming in the case of reverse mapping).

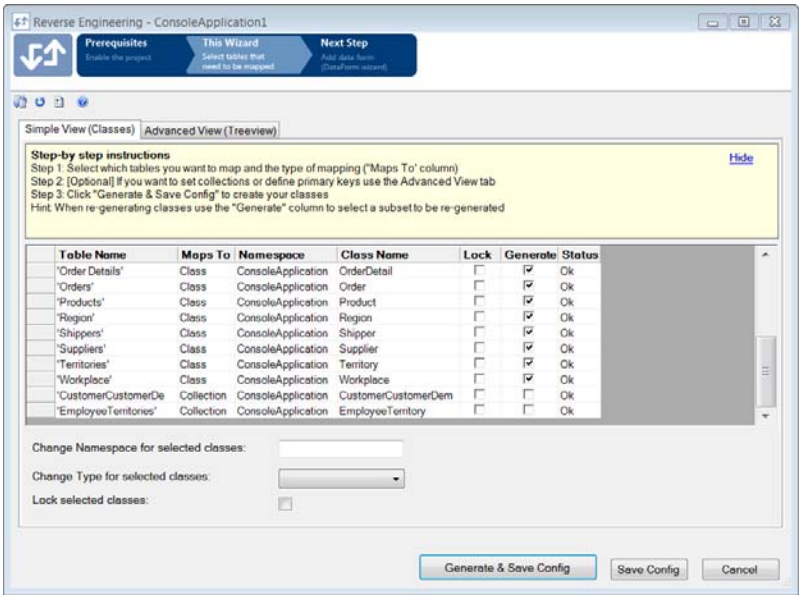

Figure 5. OpenAccess (Reverse) Engineering Wizard.

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

OpenAccess also generates a DAL that is transparent persistence and mostly persistence ignorant[1] and by utilizing partial classes, generating very clean reflection free classes, allowing you to run in a medium trust environment. If you need it, you have the ability to add your own additional logic to the DAL that will not be overwritten if you need to regenerate the class. This also makes integration with source version control software much easier as well as facilitates testability and TDD. Here is a class generated by OpenAccess, it is just like a normal class that you would create:

```
using System;
using System.Collections.Generic;

namespace ConsoleApplication1
{
    //Generated by Telerik OpenAccess
    //NOTE: Field declarations and 'Object ID' class implementation are added to the
'designer' file.
    //      Changes made to the 'designer' file will be overwritten by the wizard.
    public partial class Region
    {
        //The 'no-args' constructor required by OpenAccess.
        public Region()
        {
        }

        [Telerik.OpenAccess.FieldAlias("regionID")]
        public int RegionID
        {
            get { return regionID; }
            set { this.regionID = value; }
        }

        [Telerik.OpenAccess.FieldAlias("regionDescription")]
        public string RegionDescription
        {
            get { return regionDescription; }
            set { this.regionDescription = value; }
        }


    }
}
```

[1] Jimmy Nilsson author of Applying Domain-Driven Design and Patterns, defines the term persistence ignorance as "ordinary classes where you focus on the business problem at hand without adding stuff for infrastructure-related reasons. " He continues on page 184 saying: If you use a PI-based approach for persistent objects, there's no requirement to do any of the following:

- o Inherit from a certain base class (besides object)
- o Only instantiate via a provided factory
- o Use specially provided datatypes, such as for collections
- o Implement a specific interface
- o Provide specific constructors
- o Provide mandatory specific field
- o Avoid certain constructs
- o Write database code, such as calls to stored procedures

We feel that OpenAccess does not violate any of these above rules, even though OpenAccess uses attributes in its persistent classes. Attributes are used so often in the .NET Framework that we feel it warrants an exception and the OpenAccess attributes do not affect the object's domain intention or behaviors, the object is still specific to its domain and independent of the database and infrastructure.

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

Data Access

You can query the objects in the DAL using standard Object Query Language (OQL) and OpenAccess provides an OQL explorer tool shown in Figure 6 where you can test your OQL and even see what SQL OpenAccess generates against the backend datasource.

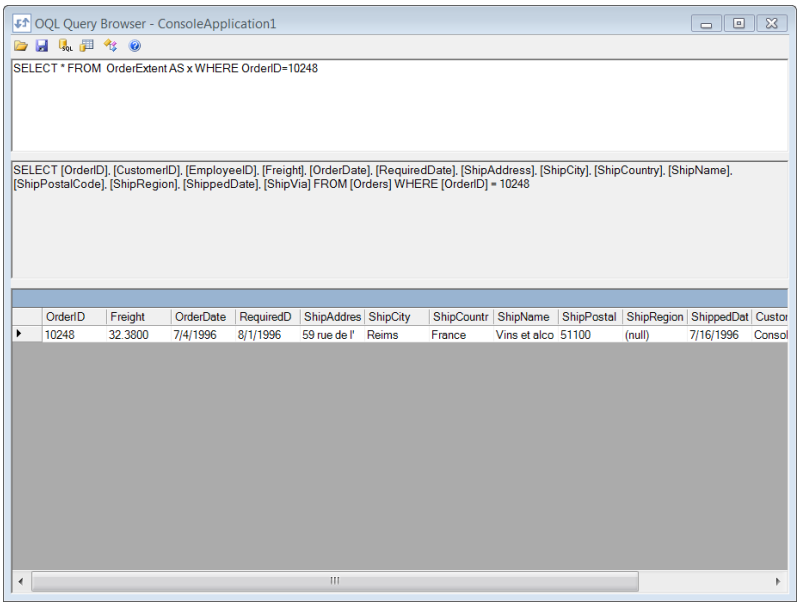

Figure 6. The OQL Query Browser

OpenAccess has first class LINQ support through its automatically generated ObjectScopeProvider. The LINQ support is specific to OpenAccess, not the backend database. An example is shown here, after you get a reference to the ObjectScopeProvider provided by your DAL, you can use the standard LINQ query operators on that reference.

```
IObjectScope scope = ObjectScopeProvider1.GetNewObjectScope();

            var result = from o in scope.Extent<Order>()
                         where o.Customer.CustomerID.Matches("ALFKI")
                         select o;
```

If you are not satisfied with the SQL that OpenAccess generated via LINQ or OQL, you can optimize the query that OpenAccess will generate by defining a fetch plan. You can graphically define a fetch plan using the fetch plan browser as shown in Figure 7.

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

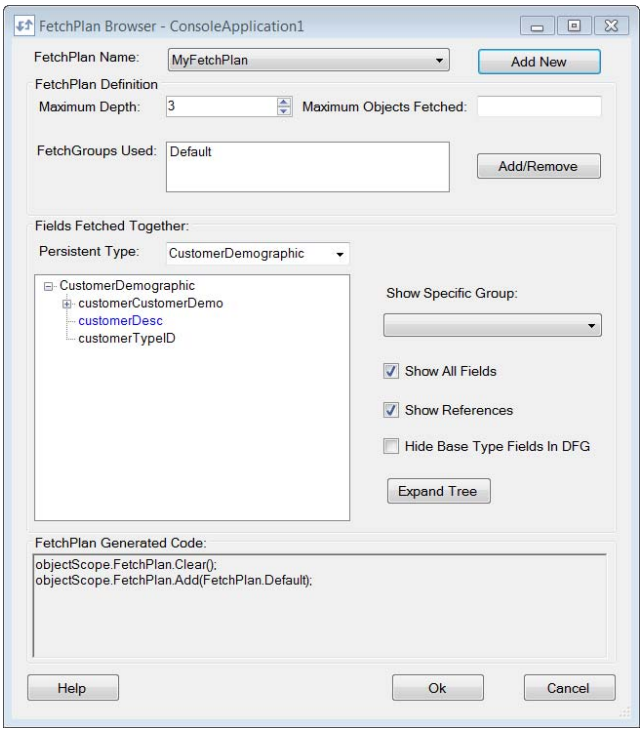

Figure 7. The Fetch Plan Browser

Lazy Loading and Level 2 Cache

OpenAccess also uses “lazy loading” or the ability to load resources as they are needed, increasing performance. OpenAccess also has a very unique caching mechanism, the 2nd level (L2) cache, which reduces the calls made to the backend database server. Database access is therefore necessary only when the retrieving data that is currently not available in the cache. This increases the efficiency and performance of your application and enables you to move effortlessly to a multi-tier application or web-farm environment.

The cache is even more scalable via the 2nd Level Cache Cluster. In an application server farm scenario where more than one application is using the L2 cache for the same database, OpenAccess synchronizes the various caches, even preserving transactional integrity. Every modifying transaction sends its eviction requests to all participants in the L2 cache cluster asynchronously, ensuring that the modified data will be expired and preventing both incorrect and dirty reads.

Distributed Applications

Applications today need to work across multiple tiers even if they are not always connected. Service oriented architectures via WCF, Web applications, mobile application and asynchronous applications still need to deal directly with persistent objects. OpenAccess supports distributed environments by allowing you to work with portions of your data in a disconnected mode via the OpenAccess Object Container. In addition OpenAccess integrates with client side technology such as Silverlight, allowing you to spread your processing across multiple tiers.

275 Grove St, 4th Floor
Newton, MA 02466, USA
phone: 1.866.886.0452, fax: 1.617.249.2116

Conclusion

If you are looking to bridge the object-relational impedance mismatch and increase your productivity and data access performance, you should consider the Telerik OpenAccess ORM. It offers a wizard for both forward and reverse mapping to all major databases including Microsoft SQL Server, tight Visual Studio integration, LINQ support, transparent persistence and lazy loading, runs in medium trust, as well as a fully scalable architecture via its fetch plans and level 2 cache.

For more information please visit: http://www.telerik.com/products/orm.aspx