Patterns for Factoring Responsibilities when Working with Objects and Relational Databases

Leon Welicki
ONO (Cableuropa S.A.)
lwelicki@acm.org

1. Introduction

Data access layers are an ubiquitous issue in enterprise applications. In a great number of companies a relational database coexists with an object oriented middleware and the applications have to deal with that. The impedance mismatch between both models is very well known among the software development community. In order to cope with it several approaches have emerged (ORMs, DMT, mappings, custom DAO layers, etc.)

The patterns presented in this paper try to address some of the most common problems that arise when working with data access layers. These common problems are simple to solve, but the solutions have significant implications. This paper contains the following patterns:

- **Data Source Object**: factoring of complex SQL based queries (mainly for UI generation) in object oriented models.
- **Abstract Cursor**: abstraction of cursors when more than one approach for retrieving data exists (and the existing approaches are in conflict).
- **Base Data Access Object**: factoring of common behavior of custom made data access layers.

These patterns can be used in scenarios where the DAO pattern is used (since they are complementary), in scenarios where ORM (Object Relational Mapping) and DMT (Data Mapping Techniques) are used, and in scenarios where the data access layer is a custom component written specially for a specific application. The patterns are complementary with the ones presented in [Fowler02] and [Nilsson05].

These patterns cover some very specific situations that may arise when working with object oriented middle-ware and relational databases. However, they may be not be useful (neither applicable) when using other data back-ends such as flat files, object databases, etc.

The pattern catalog presented in this paper is by not any mean complete: the patterns in it only handle some very concrete situations. They can be used in conjunction with some other well known patterns as Dao [CJP02] or any of the existing ORM patterns [Fowler03] and technologies. They have been extracted from experience developing .NET applications, although they may fit in J2EE applications as well.

1.1 Target Audience

The patterns included in this paper are targeted to developers and architects that work in the creation of object oriented middleware or presentation layers on top of a relational database. Since they cover very concise, clear but high impact issues, the experience range of the target audience may cover from beginner to expert. The beginners will find some useful ideas for factoring responsibilities in database-driven applications to solve common problems. The advanced users may surely already know or applied the ideas expressed in these patterns but they hopefully would help them to reflect about their every day practices.

In order to take advantage of these patterns the reader must have experience with using relational databases, a basic knowledge of SQL and using object oriented middleware with relation persistence. Additionally, it would be beneficial to have knowledge of the issues that arise when combining relational persistence with object oriented consumers. Finally, some experience with .NET framework may also be useful (although not indispensable), since the patterns have been mainly discovered and applied in .NET scenarios.

Data Source Object

Context

We are developing a web application with an object oriented platform (e.g. NET, J2EE, etc.) and using a relational back-end (e.g. Oracle, Sql Server, etc.) and we want to show information in a UI grid that comes from several tables (the product of a complex SQL query).

We have correctly mapped the responsibility of CRUD operations to our persistence layer (which may be based on public ORM framework or have been written for the app) and now have to decide on an architectural level how the code receives access to this data.

Problem

How can we abstract the retrieval of complex data so it is correctly factored and can be located and reused as a cohesive unit when we can't see clearly to which DAO object nor domain object attach this responsibility?

Example

The persistence layer is managed using NHibernate [NHibernate]. CRUD operations are correctly factored, but there is a problem when complex queries need to be placed somewhere in the code.

Since the data to be shown is the result of complex SQL JOINS between several tables it is not clear where to put it (there isn't any object that can be a candidate for hosting it). Additionally, retrieving a complex object graph for showing tabular data is not a good implementation idea, since it makes a bad usage of resources: in environments with automatic memory management, unnecessary domain objects creation can lead to excessive resource consumption (this becomes a significant problem in web apps, since lots of short lived instances should be created to satisfy a single request).

For example, if we need to show tabular data in a web-based application having a mechanism for directly displaying the retrieved data from the database would probably improve scalability, since it avoids creating (maybe) hundreds of domain objects (with their respective graph). But we need to be careful with this solution: if the retrieval and usage of this information is not normalized through out the system, it can lead to hard to extend and to maintain convoluted applications. Here is DataSource enters, helping developers to factor and abstract the retrieval of complex presentation oriented information.

Forces

- The responsibility of getting the requested data doesn't fit in any class of the domain.

- Resource management is an important issue: we want to avoid deserializing a complex object graph that will be used only for presenting data in a single request.

- Correct factoring of responsibilities makes maintenance and evolution easier. (Because of this we want to follow the Single Responsibility Principle [Martin02])

- The queries may vary independently of our domain model according to end-users needs. The requested changes may have nothing to do with the domain model (for example, a new join or filter may be added).

- When a query is changed, the perceived responsible may change (for instance, if a join is removed the responsibility of fetching the data may "move" to other entity). Therefore, complex presentation queries for reporting that don't fit clearly with any domain abstraction should not be linked to domain objects.

- We don't want to pollute our domain model with presentation-oriented queries.

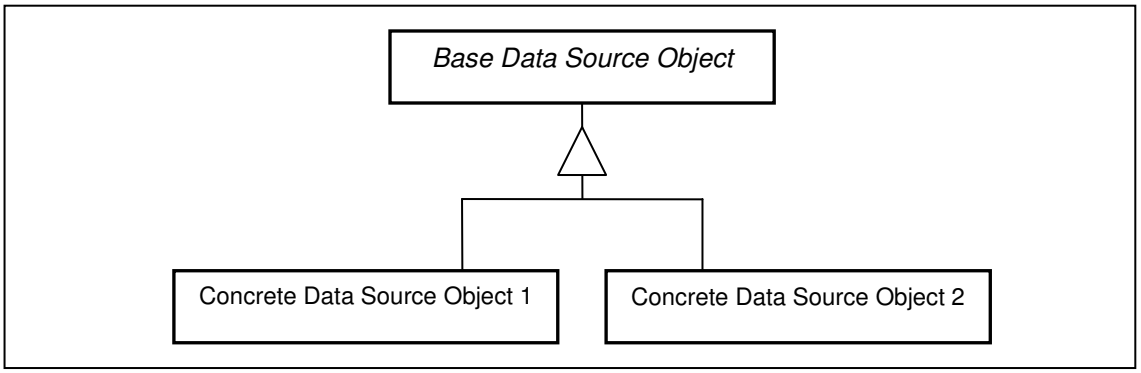
- We want to have the flexibility and power of SQL queries without affecting the maintainability, modularity and extensibility of our code.

Solution

Create data source objects with the responsibility of retrieving this information. The only purpose of these kinds of objects is to retrieve data from a repository by encapsulating a complex query. Therefore, Data Source Objects abstract the retrieval of concrete sets of information from the actual data source. As a consequence, they factor this responsibility cohesively and univocally in a concrete entity with a very clear and an explicit purpose.

The Data Source Objects may implement a well defined interface so they can be used generically in a presentation framework. Moreover, they can extend a base class that may contain all the primitives for retrieving data and handling parameters.

The Data Source Object returns a cursor to the user - in an appropriated format - that may be used to create output in the UI level.



This pattern may be used as a complement of the DAO pattern [CJP02]. We suggest using the DAO pattern for CRUD operations or for retrieving data for loading object models and using the Data Source Object pattern for factoring complex presentation-oriented or report generation queries. Data Source Object can be seen as a special case of DAO that is not linked to any domain entity and is specialized for retrieving complex queries (for reporting or presentation purposes).

Data Source Object can be also very helpful when using OR-Mappers: we can use the OR-Mapper for CRUD operations and for any domain related task that needs to use persisted domain objects. Whenever we need to generate a report or present tabular data from the database (based on several joins) to the user we can use the Data Source Object to simplify the querying and improve the performance and resource usage to achieve this task. Combining both concepts (ORM and DSO) can help us to leverage the best of both approaches.

Example Solution

In our previous application, NHibernate did a great job with CRUD operations and reading and writing objects, but fell short when we needed to perform complex queries: it needs to create lots of objects and since these objects properties change very often in the database, NHibernate's caching facilities are not very reliable nor useful. Additionally, since the tabular data must be presented very often, the application's memory footprint and resource usage (for creating and setting all the necessary instances) is higher than it should be (lots of graphs of objects are created just for showing a grid of data in a web page).

In order to fix this situation data source objects were created with the purpose of hosting the queries to the relational data store. By doing this we can combine the best of both worlds (ORM for giving persistence ignorance [Nilsson05] to our domain object and custom written SQL queries for presenting data from complex SQL queries to the users).

Resulting Context

- Responsibility for retrieving data is correctly factored in "data-retrieval-oriented" objects and is not mixed with the domain models.
- We have the benefits of domain model for factoring behaviour and the power of relational databases for querying data.
- All queries are grouped and can evolve independently of the domain model.
- Querying infrastructure is normalized (is in the base class of the Data Source Object).
- The data is fetched in a format optimized for presentation or UI code generation.
- The relational database is used with optimized relational semantics.

- A good control of Data Source Objects is needed. Otherwise the system can be polluted with unused or out of date data source objects.

- Data Source Objects are semantically weaker that domain objects. They are just object oriented adapters (wrappers) on complex SQL queries while domain objects represent domain abstractions.

- Avoids creation of unnecessary domain objects for showing sets of data, making a better usage of resource (this is especially true in automatic memory management environments as .NET and J2EE).

Implementation

The implementation of this pattern is very simple. The first step is to create an abstract base class with the data access primitives: this class will define the interface of the data source objects and will provide the primitive functions for handling the queries.

Once the abstract base class is created (the AbstractDataSourceObject) the concrete data source objects may be created. These objects must redefine the methods where the query is created and the parameters are retrieved.

Related Patterns

DATA SOURCE OBJECT may return an ABSTRACT CURSOR.

DATA SOURCE OBJECT can be seen as a variant of combination of STRATEGY and a FAÇADE

Data Source Object is a complement of the DAO pattern. The first may be used for CRUD operations or for retrieving data for hydrating object models and the last for retrieving complex presentation-oriented (based on SQL queries) or report generation queries.

Abstract Cursor

Context

We are using a data access framework such as ADO.NET that makes a big distinction between several ways of retrieving information (in this case, connected and disconnected modes). The framework's semantics for information retrieval and consuming varies significantly according to the selected method. Therefore, once a method is chosen is quite difficult (or time consuming and error prone) to refactor to the other (the choice of one of the available approaches is a high entropy decision).

Problem

How can our data retrieval and consuming code be flexible when we have several approaches for retrieving and iterating through sets of data and each of them have very different semantics?

Example

In .NET applications there is a big distinction between the connected mode and disconnected mode for retrieving data [Microsoft]. In the former case there is a link with the data source and in the last the data is stored in memory. In each case the semantics for walking through the data is different, and once an approach is chosen a lot of effort needs to be done to refactor to the other

Forces

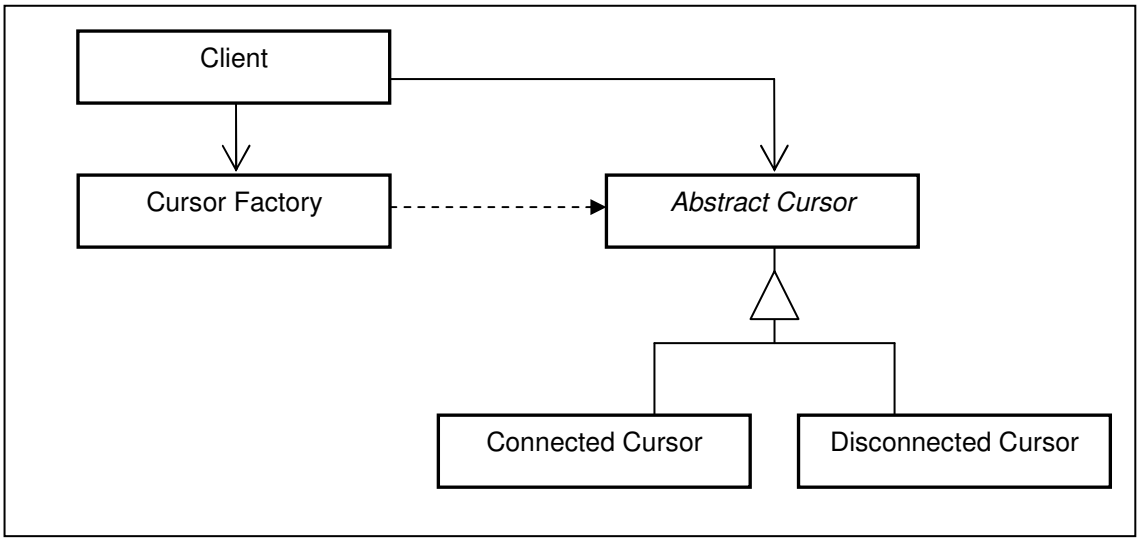
- Selecting one of multiple approaches to retrieve data shouldn't be a high entropy decision
- We want to avoid the situation that once an approach is selected, is very difficult to refactor to other. Refactoring from one data retrieval approach to other should be trivial
- Simplify software maintenance and evolution
- Changing the way data is retrieved should not produce changes in the data access components. When the method is changed binary compatibility should be maintained.
- Code for walking through a cursor should be uniform throughout all the application
- A developer's decision on the selected approach for retrieving data should have minimum impact in the application.

Solution

Create an abstraction to represent a cursor. This abstraction may contain the necessary behaviour for iterating through a cursor. Internally it will have a reference to a data source that implements one of the available data access approaches.

The Abstract Cursor Pattern implementation will be typically placed in a data access framework or toolkit that will be used in our applications. This framework should include concrete implementations for each kind of method. The end users will interact with these concrete implementations through the high level abstraction (the abstract cursor interface). The instances of the concrete cursors may be created using a Cursor Factory (which may be implemented for example with a Creational pattern [GoF]), abstracting the creation process from the client code.

Users of the Abstract Cursor Pattern will focus on an abstraction (the Abstract Cursor, following the Dependency Inversion Principle that states that "a*bstractions should not depend upon details; details should depend upon abstractions*" [Martin02]), delegate the creation in the Cursor Factory (that will handle the creation of the instances) and then focus on the interface. Therefore, choosing any of the available implementations on the abstract cursor will not have any impact on the client code that consumes the cursor.



Example Solution

In our .NET application, we can create an abstraction for representing the cursor and a factory for creating the instances. This abstraction may expose a comprehensive interface for walking through sets of data.

Once the abstraction is created two implementations must be created, for dealing with the connected and disconnected modes. These implementations will be part of our data access framework or toolkit and will be used by client applications in place of DataReaders and DataSets. This will allow us to maintain the flexibility of both approaches using the same semantics. Both implementations will expose the interface created in the previous step, but will hold internally an instance of a DataReader or a DataSet (according to the selected mode). Therefore, the users won't be using explicitly

any of the concrete implementations, but interact with them through the abstract interface.

After the concrete cursors have been created a Cursor Factory may be created for helping the user to abstract the creation of instances (concrete implementations) of the Abstract Cursor.

Resulting Context

- All code in the application for retrieving and walking through cursors is identical.
- Obtaining a cursor and walking through its contents is normalized.
- Selecting one approach of the available approaches for retrieving data in an application is no longer a determinant decision. Is very easy to refactor from one approach to other.
- Unique semantics for manipulating data.
- Simplified maintenance and evolution of data access related code.
- There is no code in an application that makes explicit reference to a specific for retrieving data (the code is more generic).
- The creation of the whole infrastructure (abstract cursor interface and concrete implementations) may add be relatively complex task.
- Once the infrastructure is implemented it can be reused in several projects, leveraging its creation cost.
- All existing approaches are equalized to a lower common denominator.

Implementation

1. *Create a generic interface for the abstract cursor*: design a generic interface that all cursors must implement. This interface may contain methods for walking through the data and strongly typed primitives for fetching the values of columns.
2. *Create the concrete cursors*: create implementations for the interface created in the previous step. Each implementation will wrap a resultset, controlling access to its contents. Each concrete cursor will implement a specific method for retrieving and using data through a generic interface.
3. *Create a Factory*: create a factory for creating instances of the concrete cursors. The end user should never create directly instances of the concrete cursors. She should create the instances through the Cursor Factory.

4. *Implement Advanced Featured*: implement features to allow advanced scenarios. For example, a payload property that exposes the wrapped the resulted may be included ("simple should be simple and complex should be possible" [CA05]).

Related Patterns

ABSTRACT CURSOR can be seen as a variant of a combination of the ITERATOR and ADAPTER patterns [GoF95].

Base Data Access Object

Context

We are writing an application where due to client requirements the data access must be written in plain SQL. We want our data access layer code to be consistent and simple in order to simplify the maintenance and evolution of the application. We want also to remove data access code from domain objects (moving a step further from persistence ignorance [Nilsson05]). Our data access code should be uniform through all our application.

Problem

How can we abstract the developers of the details of the plumbing of data access objects?

If each developer writes its very own basic data access code we would end with a system that uses several distinct approached for a very critical task. Problems as various connections management, parameter handling and results retrieval strategies may arise, all of them with different results. This difficult application maintenance and evolution, since bugs are hard to find: as there isn't a unique strategy for developing data access objects each developer invents its very own and when problems arise we first need to check the data access code and then the application's code. Another negative outcome is that something so basic and critical is not standardized through the project, having data access objects with various quality and functional characteristics.

Example

We have an application where the data access layer should be written in plain SQL. Each programmer in the team has his very own way of writing data access code. Therefore, the data access code in the application is not normalized and depends exclusively on the skills of the developer that writes it.

Forces

- Data access code should be uniform through all the application
- Data access code should be easy to write and maintain
- Developers should focus on the query and the parameters type, but not in the plumbing of data access and let the infrastructure do the rest for them
- We want to minimize errors related with data access code
- We want to minimize complexity regarding to developing common data access tasks
- Lots of code for handling connections, executing queries and retrieving information will need to be written.

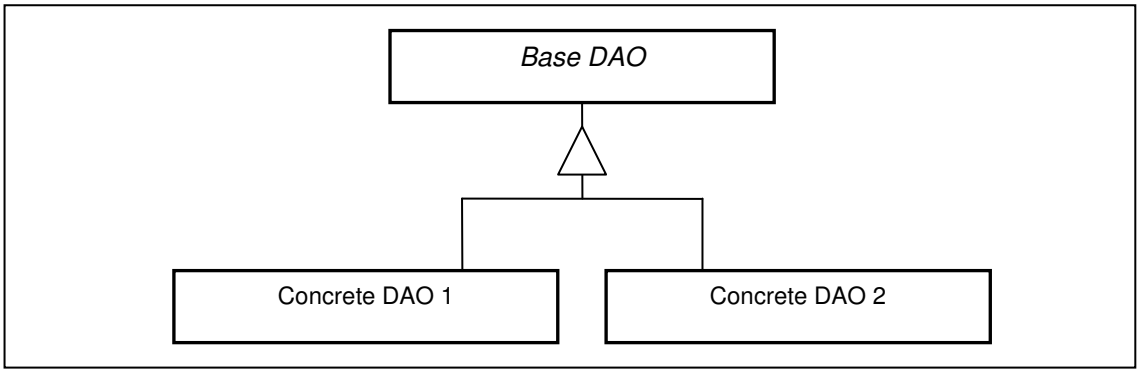
- We want to avoid that each developer writes his own data access code, since each programmer can handle this task differently, making difficult the future maintenance and evolution of the application

Solution

Create an abstract base type for all data access objects. This abstract type must contain all the primitives (properties and methods) for data access plumbing (connecting to a database, executing queries, retrieving data, parameter passing, etc.). Each concrete data access class must extend the base abstract data access object.

The methods in the concrete data access classes will contain a query, a set of parameters and will pass messages to the base class to perform any task related with manipulating the database, leaving the "hard work" to the abstract base class and allowing the concrete DAO or DSO classes to focus on the SQL queries. This pattern is only aimed to factor common data access logic and normalizing the way data access code is written, but does not impose any restriction on how the resulting DAOs or DSOs should be used.

Developers should not decide individually how to handle database objects (connections, etc) in regular scenarios (these should be normalized except for special situations). This decision should be made for them at an architectural level. The base data access object abstracts all these decisions.



As an example we can have the following functionalities in our Base Dao class: connecting to the database, preparing queries, executing queries, data retrieval, connection lifecycle management, general error handling and logging (for common situations), parameters management and transactions. This class will provide this entire infrastructure. The derived classes will only contain data focused methods that will have the purpose of modifying or retrieving data (they might only contain SQL sentences and parameter management code)

Example Solution

We use a base data access object in our application. Therefore the developers that write data access code don't focus on the data access plumbing and focus on the SQL queries.

Resulting Context

- Simplified DAOs

- Developers don't need to be aware of the data access plumbing. The Base DAO manages the underlying connection and serializes the parameters

- All DAO code in the application is uniform

- Unique semantics for creating DAO

- Simplified maintenance and evolution of data access related code

- All data access code is placed in a common place and is not distributed through the application.

- Some scenarios may not fit within the normalized structure that is proposed in this pattern. In these cases special data access code must be written to cope with those scenarios.

- A change in the Base Data Access Object may affect all derived classes.

- Data Access Objects can only inherit from the Base Data Access Object

Implementation

1. Create an abstract base Data Access Object: Create an abstract base class for the data access objects. This class will have a set of protected members and methods aimed to manage data base connections for retrieve and modify data

    1.1. It is a good idea having very task oriented methods that hide the internals of data access code to the developers. For example, the Base Data Access Object can have a method called ExecuteSql that would create a connection and execute a query. By using this method the developer does not have to worry about dealing with the underlying connection.

2. *Create the concrete Data Access Objects*: create data access objects to manage data from specific entities. Each data access class should extend the abstract base Data Access Object. Each method in the concrete DAO will be responsible for performing a query. The methods will contain the SQL and the parameter passing, but at the time to perform the query they will pass a message to the base class.

Related Patterns

BASE DATA ACCESS OBJECT uses TEMPLATE METHOD

BASE DATA ACCESS OBJECT may have a method for creating an ABSTRACT CURSOR

DATA ACCESS OBJECTS must extend the BASE DATA ACCESS OBJECT

Appendix: Glossary

- DAO: Data Access Object
- SQL: Structured Query Language
- ORM: Object Relational Mapping
- DMT: Data Mapping Technique
- ADO.NET: Microsoft's data access technology for the .NET framework

Acknowledgements

I would like to thank Marina Haase, my shepherd for EuroPLoP 2007, for her great help and support during the shepherding phase.

References

[Fowler03] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003

[GoF95] Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 1995.

[HT00] Hunt, Andrew; David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. 2000.

[POSA1] Buschman, Frank et al. *Pattern Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley & Sons. 1996

[POSA3] Kircher, Michael; Prashant Jain. *Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley & Sons. 2004.

[Martin02] Martin, Robert C. *Agile Software Development*.

[Nilsson05] Nilsson, Jimmy. *Applying Domain Driven Design and Patterns*.

[NHibernate] NHibernate. http://www.hibernate.org/

[Microsoft] Microsoft Corporation. .NET Framework Developer Center. http://msdn2.microsoft.com/en-us/netframework/default.aspx
[CA05] Czwalina, Kystof; Brad Abrams. *Framework design guidelines*.

[CJP02] Sun Microsystems - Core J2EE Patterns - Data Access Object - http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html