The Smart Persistence Layer

Mariusz Trzaska
Software Engineering
Polish-Japanese Institute of Information Technology
Warsaw, Poland
mtrzaska@pjwstk.edu.pl

***Abstract*— We present an approach to solve the impedance mismatch problem caused by incompatibility between two models: object-oriented and relational ones. We believe that it cannot be unraveled by creating new Object-Relational Mappers (ORMs) like most of the software industry does. It is caused by some inherent differences between those two worlds. In our method we assume that both a programming language and a data source should be based on the same data model. Thus we propose a persistence layer for native data structures of a programming language. The presented idea is supported by a working prototype called the Smart Persistence Layer, which also supports extent management and bidirectional links. The prototype together with LINQ, the native query language for the .NET platform, formulates an easy-to-use yet powerful solution.**

Keywords-Impedance mismatch; Databases mapping; Object-Relational Mappers; ORMs; Persistence; LINQ.

I. INTRODUCTION

The impedance mismatch is a negative software development phenomenon denoting severe incompatibility between two models: object-oriented and relational ones. It is caused by the fact that most modern software is implemented in object-oriented programming languages, but its data is persisted using relational databases. Such an approach forces the necessity of translating a rich object-oriented universe to a pretty simple relational world and vice versa.

In 2004, Ted Neward coined the phrase "Object/relational mapping is the Vietnam of Computer Science" [1]. His thesis was based on the observation that in the Vietnam and ORM cases there are less and less hope for success and unacceptable consequences of giving up. Two years later the phrase became famous thanks to Jeff Attwood who published the paper [2]. The paper mainly confirmed Neward's observations. One of the most important conclusions is choosing a single model both for the programming and data. Any other options are vulnerable to some level of the impedance mismatch.

This approach might be seen as too radical but in our opinion it is the only right choice. Contrary to the Attwood's preferences [2] we believe that the better choice is to select the object-oriented side rather than the relational one.

Unfortunately, a few years have passed since the phrase was coined, and nothing has changed on the battlefield. Even worse, it seems that nothing will change in the next few years. The software industry focuses on improving ORMs rather than changing the approach to the problem. It looks like a situation where one is looking for a better and better medicine rather than eliminating the source of the illness. We believe that improving ORMs is questionable because there are too big discrepancies between the models and too big risk that attempts to match them will cut a lot from their functionalities. Usually, in such a cases and for large databases the object model is the victim: object-oriented qualities are reduced to minor (mostly syntactic) differences between the object and relational data schemas. The object model becomes a slave of the relational model. It is not possible to create a generic mapper, which will be able to automatically transform object-oriented queries addressing sophisticated object model into relational queries and commands (SQL), and vice versa. The main reason of that is the fact that probably there is no general algorithm that maps object-oriented queries and updates into SQL and still ensures good performance. In typical cases (our experience from other projects [3]) a mapper uses non-standard SQL features (e.g., traversing tables by cursors), thus the SQL query optimizer has no chances to work properly. Hence each case has to be manually designed by the programmer. In fact, it does not even matter how the mapping is to be defined: using a configuration file, a DSL or some other way. The result is still the same: the programmer has to spend his/her valuable time doing some repetitious and error-prone work.

The problem is not only related to mapping definitions by programmers. It is much more extensive and spreads on query languages, different types, semantics, etc.

There are opinions that solving the impedance mismatch problem should employ extending programming languages with declarative specification capabilities like JML [4] or Spec# [5]. Generally we do not agree with such a solution mainly because of the complexity, e.g., Spec# requires a dedicated compiler.

Our proposal is based on replacing both an ORM and a database with a data source native to a programming language. As a result, there is no impedance mismatch at all. The approach is supported by a working prototype for the .NET platform. The prototype provides a persistence layer and extent management for objects of a programming language.

The rest of the paper is organized as follows. To fully understand our motivation and approach some related solutions are presented in Section 2. Section 3 briefly discusses key concepts of our proposal and its

 ISBN: 978-1-61208-165-6

implementation. Section 4 contains sample utilizations of the prototype and simple benchmarks. Section 5 concludes.

II. Related Solutions

As we suggested previously, to reduce completely the impedance mismatch we need to leave the object model and to eliminate another data model. It means that both business logic and data store will be on the programming language's side or the database side. Both approaches have their advantages and disadvantages. We discuss them shortly.

A. The Programming Language Side

This approach requires that a business logic and a data source are implemented on the programming language side. It involves a dedicated data source, which is not only compatible with the programming language but fully native to it. The compatibility condition is quite common and means ability to work with a particular platform. However, it does not mean common models. The most obvious examples are relational databases and ORMs. Undoubtedly, the latter are more convenient for programmers but still require at least manual mappings.

The nativity condition is fulfilled when plain objects of a programming language are persisted using an additional tool. Usually the tool has to be an object-oriented database management system (ODBMS), i.e., db4o [6], [7] or Objectivity [8]. Both of them are mature solutions existing on the market for at least 10 years. However in some cases, using them could be too complicated. Thus, a more lightweight solution would be a better choice. Our proposal follows this idea. More information, comparing the db4o to our prototype could be found in Section 3.

The reference [9] provides a list of open source persistence frameworks for the MS .NET platform. Unfortunately, most of them are implemented as ORMs, which of course introduces some level of the impedance mismatch. We have found only two tools, which do not utilize a relational database: Bamboo.Prevalence [10] and Sisyphus [11]. However they usually require some special approaches, e.g., the command pattern utilized for data manipulation for the Bamboo and necessity of inheritance from a special class for the Sisyphus.

B. The Database Side

This solution utilizes the database model both for business logic and data. Thus it requires that the entire application is implemented in a database programming language. There are various DBMS and dedicated languages on the market, i.e., T-SQL, PL/SQL. Both of them have imperative functionality and PL/SQL has some object-oriented constructs. There are also fully object-oriented solutions like SBQL for the ODRA platform [12]. These seem more appropriate thanks to the more powerful and flexible model.

The ODRA (Object Database for Rapid Application development) is a prototype object-oriented database management system based on SBA (Stack-Based Architecture). The main motivation for the ODRA project is to develop new paradigms of database application development. This goal is going to be reached mainly by increasing the level of abstraction at which the programmer works. ODRA introduces a new universal declarative query and programming language SBQL (Stack-Based Query Language), together with a distributed, database-oriented and object-oriented execution environment. Such an approach provides functionality common to the variety of popular technologies (such as relational/object databases, several types of middleware, general purpose programming languages and their execution environments) in a single universal, easy to learn, interoperable and effective to use application programming environment.

III. The Smart Persistence Layer

Programmers use databases for many reasons. One of the more important are persistence and a query language. A few years ago Microsoft introduced a query language called LINQ [13] to ordinary programming languages (e.g., C# and Visual Basic). The LINQ works with native collections of the programming language allowing querying them as regular databases. It is also supported by various ORM mappers including their own solution called Entity Framework [14]. Generally speaking, the mapper uses a relational database for storing data which, of course, causes some impedance mismatch (especially concerning inheritance).

Our approach is based on an observation: if we have a query language (LINQ) natively supported by the programming language, then we should use native data structures of the language as well. Such an approach guarantees that every bit of impedance mismatch simply disappears. Of course, in real case scenarios a persistency for the native data is required. At first glance it looks that such a mechanism already exists for modern programming languages and is called serialization. Unfortunately, it is not applicable as a replacement for databases. The main reason is the fact that the serialization every time stores the entire graph of objects. This behavior is caused by the way the serialization works: every saved object is valid, which means storing all connected objects, objects of connected objects and so on.

Our proposal focuses on delivering a persistency layer designed in a totally transparent way for the programmers. We do not want to make programmers use any kind of super classes or implementing special interfaces. The prototype is called The Smart Persistence Layer (SPL) and implemented for the MS .NET platform. However, it is possible to implement it for other platforms with the reflection capabilities, i.e., Java. In this case it would be possible to reuse significant parts of the source code and data files as well.

A. The Basic Functionality

The most basic functionality for a mapper is delivering an extent of objects belonging to a particular class. This could be achieved using many ways. For instance the db4o [8] uses the following code:

 ISBN: 978-1-61208-165-6

```
IList <Pilot> pilots =
db.Query<Pilot>(typeof(Pilot));
```

However in our prototype we have simplified that to:

```
IQueryable<Pilot> pilots =
db.GetExtent<Pilot>();
```

Please note that our method does not require the parameter, but the result is still strongly typed.

There is also a debate how objects belonging to different classes in the same inheritance's hierarchy should be treated. We believe that the extent of a super class must also contain all instances of subclasses. This approach guarantees that we can work on a higher level of abstraction (i.e., different subclasses of product processed just like products; see also Section 4). Of course, this relationship works only in one direction: extents of subclasses will not contain instances of super classes. Hence the above code returns a collection of objects belonging to the given class (as a type parameter) and all subclasses.

Another area related to an extent, which needs a clarification is how and when new objects will be incorporated into extent. Our proposal follows the following rules:

- an object could be added to an extent by executing by a programmer a dedicated method;
- every object, which is directly made persistent by a programmer is added to an appropriate extent.

If a programmer would like to achieve automatic adding to an extent, then the method could be executed in a constructor of a class. It is especially easy thanks to our designing decisions. We have utilized the C#'s extension method mechanism together with the default instance of the SPL. An extension method is a method adding a functionality to a class but defined outside the class. The listing 1 (due to readability all listing are located at the end of this paper) presents the mentioned method. Please note that the method's parameter is of type `object`, which means that any object could be added to an extent (and the extension method could be executed on any existing object). A dedicated logic adds a given object to appropriate extents (the current one and all super classes). This is performed based on the object's type. A similar extension method has been utilized for the `Save` operation, which persists a given object.

Another interesting concept is the default instance of our prototype layer. In case of many applications a persistence layer is available via a single object, i.e., a file stream or a DB instance/connection. Hence, we have introduced a concept of default instance, which is the first (and in many cases the only one) instance of the persistence object. The object has to be properly initialized at the very beginning. Otherwise, during accessing the default instance, appropriate exception would be thrown. This solution allows accessing the data without passing a reference to the object. This is also the case of the previously mentioned method adding an object to its extent.

Such an approach does not put any restraints on programmers i.e., implementing an interface or inheriting from a super class.

B. Bi-directional Associations

One of the key functionality of every data store is the ability for creating and persisting connections among objects. In our opinion, it is especially useful if the connections are bidirectional allowing navigation in both directions (i.e., from a product to its company and vice versa). Unfortunately, databases usually do not support the feature. According to [7] the db4o does not have it either. This is also the case of native references existing in popular programming languages (e.g., MS C#).

The implementation of the mentioned functionality is complicated especially if we would like to work with the POCO (Plain Old CLR Object) objects. This approach means that we cannot expect implementing a specified interface or functionality inherited from a super class. Another disadvantage of putting links into a super class would be problems with navigation using the LINQ.

Thus our goal was to design it as convenient as possible but still remembering that it would be extremely hard to find a perfect (totally transparent to a programmer) solution.

One of the approaches is generating classes based on same templates. This is the case of one of the options in the Microsoft Entity Framework [14]. However, this functionality requires some kind of support from a tool and in our opinion may not be useful for all programmers.

It seems that creating a bidirectional link requires defining the following data:

- role name,
- reverse role name,
- target object,
- reverse object.

We had to choose how and when to put them to minimize the amount of work required from a programmer. At the beginning we tried creating special annotations for classes. But it turned out that some data still has to be passed as string. After some research we came up with another solution, which spreads on two different levels (see Fig. 1).

The first one is a dedicated class parameterized with two types: target objects (`TTargetType`) and reverse object (`TReverseType`). Utilizing a parameterized class makes possible detecting some errors during a compilation time. The next level uses information passed to the constructor of the class. It takes a reverse attribute name, which will store the reverse link and an instance of the class, which should be the reverse target. The following listing presents the code, which should be placed inside a business class (see also Section 4).

```
ICollection<Tag> Tags = new
SplLinks<Tag, Product>("Products",
this);
```

 ISBN: 978-1-61208-165-6

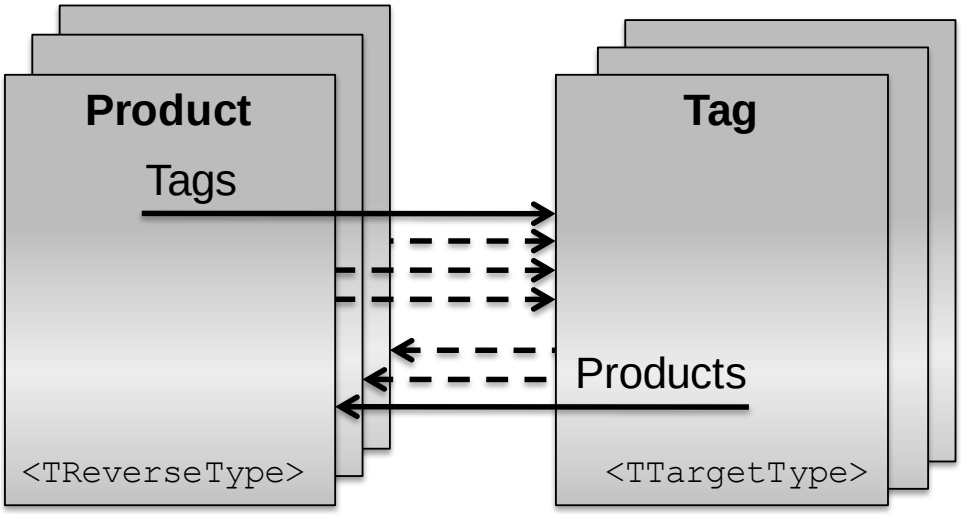

Figure 1. Explanation of the implemented bidirectional links mechanism.

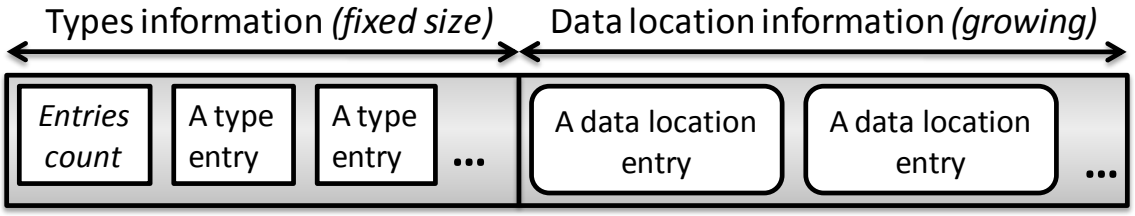

Figure 2. Structure of the file storing types and location information

It may look a bit complicated but it is created only once for each link. The `SplLinks` class implements ordinary .NET interface for accessing collections thus using it is exactly the same as any other .NET collection. Creating a bidirectional link requires only executing a single Add method with the target object. The reverse connection will be created automatically based on previously defined data. Of course, all LINQ queries work as well.

C. The Transparent Persistence

The goal of the persistence process is to store data on some non-volatile media, usually in a disk file. In case of our solution we need to persist three types of data, namely:

- business content of the objects,
- location of the above,
- information about types (classes).

All of them can change and grow during the run-time. After some research we have decided to use two files: the first one will hold business information whereas the second the rest. Initially we thought about three files but the types information is usually quite small and repeatable thus can be stored at the beginning of the second file (Fig. 2). A programmer can define amount of the allocated space for the purpose. A default value is 1MB, which makes possible storing about 3000 entries. It is possible to use just one file but at cost of more complicated design and possibly worse performance.

The single entry regarding the location of data (the *type entry* from Fig. 2) consists of:

- object identifier;
- identifier of its type;
- location in the data file where the object's content starts. This entry is updated every time when an object is saved;
- location in the index file where the location data starts.

The above information also exists in the memory to boost performance. It is saved to disk only as a backup and for reading objects purposes.

As mentioned previously we do not persist classes (types) in the file. Thus during an object initialization those classes have to be accessible by the .NET run-time (e.g., as standard DLL libraries).

The other file, with business data of persisted objects, can be read only using the location and types information. It is read at the very beginning. The current prototype reads all data to the memory. This could be a problem in some cases but modern computers are usually equipped with a lots of RAM. However, in the future versions we will probably introduce some kind of programmer's policy for defining this kind of behavior.

 ISBN: 978-1-61208-165-6

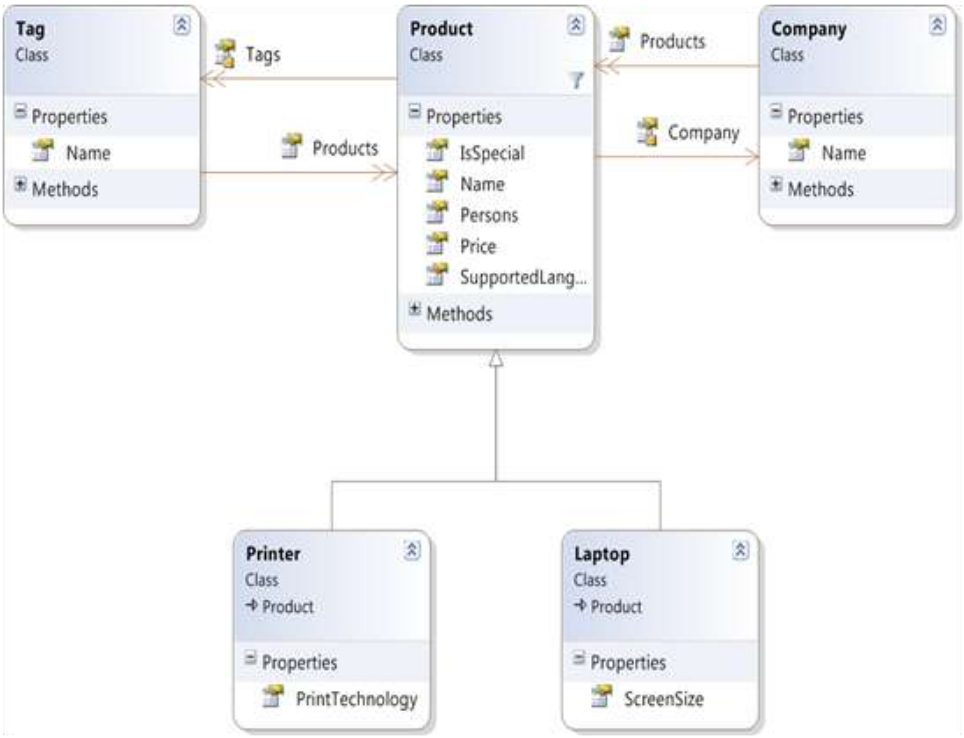

Figure 3. A class diagram of the sample implemented using the SPL.

The process of saving and reading objects intensively uses the reflection mechanism. Currently it is able to deal with atomic types, lists (classes implementing the `IList` interface), `ICollection` (see Section 3.B) and other types built using these invariants (see also the sample utilization in Section 4).

One of the problems related with links, which should be addressed, is persisting connected objects. When we would like to persist an object, how should we act with all referenced objects? There are different approaches, i.e., db4o [7] uses a concept called *update depth*. This is simply a number telling how many levels of connections should be saved. We have decided to follow another approach. When we save an object, all referenced unknown (not saved previously) objects are saved, no matter how deep they are. Thus the first execution could be costly, but the objects have to be saved anyway. All next updates will not save known objects. If a programmer wants to save them, then it has to be done directly by executing the `Save` method. The method should also be utilized every time a single object is modified (its content will be persisted in the file). This policy guarantees that persisting an object will not be costly.

IV. THE USE CASE AND SOME BENCHMARKS

Fig. 3 presents a class diagram of the sample created using our prototype implementation. It describes the following business case:

- Products have various properties including: a name, a price and a list of supported languages;
- Every product can be described using various tags;
- A company manufactures many products, but a product is related to a single company;
- There are various kinds of products with different properties. Printers contain information about utilized print technology and laptops store a screen size.

Although the presented case is quite simple, it contains different kinds of business information. Thus it allows verifying the usefulness of our approach.

Listing 2 contains the complete source code of the Product and Tag classes. The code, aside from normal C# functionality, together with the SPL provides full persistency, extents and query capabilities (thanks to the native LINQ). No additional configuration/mapping files, known from ORMs, nor special identifiers are required. Please note utilization of different types of data including the `SplLinks` class accessed using a standard C# interface (`ICollection`).

Similar simplicity can be observed on listing 3. A programmer creates instances of the `Product` and `Tag` classes, links them together (the `Tags` property) and persist (the `Save` method) using a few simple steps.

The important aspect of every data management system is its performance. We plan to perform detailed tests comparing our solutions to other approaches including ORMs and raw databases. Currently we have run some simple tests measuring speed of our prototype (the test computer configuration: Intel Core i7 2.93GHz, RAM: 8GB, Windows7 x64). The results are promising.

 ISBN: 978-1-61208-165-6

The test utilized two classes from the above business sample: the `Product` and `Company` (Fig. 3). They were connected using our bidirectional link. Table 1 presents times required by various operations.

TABLE I. RESULTS OF THE SIMPLE PERFORMANCE TESTS (ALL RESULTS ARE IN SECONDS; LESS IS BETTER)

Number of objects and the operation	Products: 50,000 Companies: 5,000 Total: 55,000 objects	Products: 100,000 Companies: 1,000 Total: 101,000 objects
Initializing the SPL	0.0180 s	0.0180 s
Generating and persisting data	17.3210 s	31.4018 s
Retrieving entire extent of Products	0.0100 s	0.0130 s
Retrieving entire extent of Companies	0.0040 s	0.0050 s
Opening file, reading all data and creating objects	21.9753 s	58.4323 s
Query Products for the price (LINQ)	0.0320 s	0.0550 s
Query Products with the specified Company's name (LINQ)	0.0120 s	0.0330 s

As it can be seen, the results are decent, especially for an early prototype. Please note short times for executing the LINQ queries, i.e., finding all products manufactured by a particular company took only 0.03s (for 100,000 products). It is probably caused by the fact that in the current prototype all data is kept in the RAM memory and a disk file is only a backup. That's why the time of opening the file and reading all data could be significant in case of bigger data sets (for 101,000 objects it is about 58 seconds). As we mentioned previously, we plan to add an option for loading data only when needed.

V. THE CONCLUSION AND FUTURE WORK

The impedance mismatch is a real problem experienced by many programmers for a very long time. In this paper, we have presented our approach to solve it. The idea is based on eliminating the causes rather than improving medicines (in this case various ORMs). We believe that the best method is to use the same coherent model both for programming and a data source. This could be achieved by providing a persistence layer and extent management for native objects created in a particular programming language.

The mentioned solution is even more useful if there is an existing query language natively supported by the programming platform. This is the case of the .NET and the LINQ query language. The implemented prototype follows our proposal by adding persistency and extent functionality to standard C# objects. Moreover the functionality has been achieved without imposing on a programmer any special requirements regarding a super class nor interfaces.

Furthermore, our prototype adds functionality for easy-to-use bidirectional associations. They are usable as standard C# collections implementing the `ICollection` interface.

As a future work we would like to extend our prototype with some other useful functionalities associated with databases like indexes or transactions. However, we would like to implement them (in a way) preserving the lightness and flexibility of our solution.

Another field, which could be researched is performance. We are going to conduct dedicated tests comparing our prototype to other similar solutions like object-oriented databases or ORMs.

REFERENCES

[1] Neward, T.: The Vietnam of Computer Science, http://blogs.tedneward.com/2006/06/26/The+Vietnam +Of+Computer+Science.aspx. Last accessed: 02-04-2011

[2] Atwood, J.: Object-Relational Mapping is the Vietnam of Computer Science, http://www.codinghorror.com/blog/2006/06/object-relational-mapping-is-the-vietnam-of-computer-science.html, Last accessed: 02-04-2011

[3] Kuliberda, K., Wiślicki, J., Adamus, R., and Subieta, K.: Object-Oriented Wrapper for Relational Databases in the Data Grid Architecture, w: On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops, Agia Napa, Cyprus, October 31 – November 4, 2005, Proceedings. LNCS 3762, Springer 2005, pp. 528-542

[4] Chalin, P., R. Kiniry, J., T. Leavens, G., and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, pages 342-363. Volume 4111 of Lecture Notes in Computer Science, Springer Verlag, 2006, pp. 342-363

[5] Barnett, M., Rustan K., Leino M., and Schulte W.: The Spec# programming system: An overview. In CASSIS 2004, LNCS vol. 3362, Springer, 2004, pp. 144 - 152

[6] Paterson, J., Edlich, S., and Rning, H.: The Definitive Guide to Db4o. Springer (August 2008), ISBN: 978-1430213772

[7] db4o tutorial, http://developer.db4o.com/Documentation/Reference/db4o-8.0/net35/tutorial. Last accessed: 2011-04-02

[8] The Objectivity Database Management System. http://www.objectivity.com. Last accessed: 2011-04-02

[9] Open Source Persistence Frameworks in C#. http://csharp-source.net/open-source/persistence. Last accessed: 2011-04-02

[10] Bamboo.Prevalence - a .NET object prevalence engine. http://bbooprevalence.sourceforge.net/. Last accessed: 2011-04-02

[11] Sisyphus Persistence Framework. http://sisyphuspf.sourceforge.net. Last accessed: 2011-04-02

[12] Adamus, R., Daczkowski, M., Habela, P., Kaczmarski K., Kowalski, T., Lentner, M., Pieciukiewicz, T., Stencel, K., Subieta, K., Trzaska, M., Wardziak, T., and Wiślicki, J.: Overview of the Project ODRA. Proceedings of the First International Conference on Object Databases, ICOODB

2008, Berlin 13-14 March 2008, ISBN 078-7399-412-9, pp. 179-197.

[13] Magennis, T.: LINQ to Objects Using C# 4.0: Using and Extending LINQ to Objects and Parallel LINQ (PLINQ). Addison-Wesley Professional, ISBN-13: 978-0321637000 (2010)

[14] Lerman, J.: Programming Entity Framework: Building Data Centric Apps with the ADO.NET Entity Framework. O'Reilly Media, Second Edition, ISBN: 978-0-596-80726-9 (2010)

LISTING 1. AN EXTENSION METHOD ALLOWING ADDING AN OBJECT TO ITS EXTENT

```
public static class Helpers
{
    // ...
    public static void AddToExtent (this object objectToAdd)
    {
          NmoDatabaseManager.DefaultInstance.AddToExtent(objectToAdd);
    }
}
```

LISTING 2. A CODE USED FOR THE PRODUCT AND TAG CLASSES

```
public class Product
{
    public string Name { get; set; }
    public decimal Price { get; set; }
    public bool IsSpecial { get; set; }
    public IList<string> SupportedLanguages { get; set; }
    internal ICollection<Tag> Tags  { get; set; }
    internal Company Company  { get; set; }

    public Product() {
          Tags = new SplLinks<Tag, Product>("Products", this);
    }
}

public class Tag
{
    public string Name { get; set; }
    public ICollection<Product> Products  { get; set; }

    public Tag() {
          Products = new SplLinks<Product, Tag>("Tags", this);
    }
}
```

LISTING 3. A CODE USED FOR PERSISTING INSTANCES OF THE PRODUCT AND TAG CLASSES

```
var tagSpecialOffer = new Tag(){Name="Special Offer"};
var product1 = new Product(){Name="Everyday Desktop VX5000", Price=799.0m,
                                SupportedLanguages = new List<string>(){"en",
                                                     "de", "pl"}};
product1.Tags.Add(tagSpecialOffer);
product1.Save();
```