Computing and Informatics, Vol. 30, 2011, 621–637

DESIGNING AN ASPECT-ORIENTED PERSISTENCE LAYER SUPPORTING OBJECT-ORIENTED QUERY USING THE .NET FRAMEWORK 3.5

Mohammadreza Jooyandeh, S. Mehdi Hashemi

Computer Science Department
Amirkabir University of Technology
424 Hafez Ave.
15914, Tehran, Iran
e-mail: mohammadreza@jooyandeh.info, hashemi@aut.ac.ir

Communicated by Alok Miskra

Abstract. In this article, we discuss aspect persistence, how it can be implemented in the .NET framework, and how to use the .NET framework to provide object-oriented queries for aspect-oriented persistence layers. The manner in which aspect-orientation can be available in the .NET framework is investigated in the first part of this article. Then the procedure through which adding persistence concepts to the .NET framework as aspects will be explained. In the next step, providing object-oriented querying is discussed, which is the main part of this article. Having object-oriented querying ability helps processes query in the same object-oriented domain in which objects are defined (not in the relation entities' domain). Language Integrated Query (LINQ) is used to provide the ability of querying in an object-oriented manner. Then, the translation of queries from the real objects' domain to the storage-objects' domain is explained. After such translation, the queries can be run by using the existing LINQ providers (for example LINQ to SQL). Finally, translating the result of queries back into the real objects' domain is discussed.

Keywords: Persistence layer, aspect-orientation, aspect persistence, object-oriented query, language integrated query

Mathematics Subject Classification 2000: 68N15, 68N19, 68N20

1 INTRODUCTION

Persistence layers and services are bridges between applications and their data. They aim to have a set of common interfaces for communication with mechanisms which are used for storage and retrieval of data, and applications which use the data [12]. One of the important goals of designing a persistence layer is to release the software from its codes which deal with persisting data.

Aspect-orientation [1] is a technology which allows for modular thinking in the world of software development. It provides the ability to intercept code execution for any purpose. Using this technology, one can add a process before or after, handle or log thrown exceptions, check access rights or perform other useful processing.

Separation of concerns in computer science is dividing software processes into some features in such a way that functionality overlap is minimized. This is similar to the Java which can separate concerns into some objects or to the C programming language which can divide them into functions. Aspect-oriented languages can separate concerns into *aspects* and *objects*. Aspects are abstractions which serve to localize any cross-cutting concerns like logging and tracing which cannot be encapsulated within a class, but it is tangled over many classes. In aspect-oriented programming, classes are separated from aspects. They join each other at some point of program execution which is called a *join-point*. A set of join-points, which is cut by an aspect, is called a *point-cut*. Finally an *aspect-weaver* is used to combine each aspect with its related point-cut. Code 1 shows how access control can be implemented as an aspect. An access check is performed on all methods of the class Document which is defined as the CheckPoint point-cut. Aspect-Oriented Programming (AOP) provides a way to encapsulate cross-cutting concerns which can be helpful for system maintenance and changing each concern with minimum cost.

```
public aspect AccessCheck
{
    pointcut CheckPoint void Document.*();

    before CheckPoint
    {
        if (!AccessDecider.AccessAllowed(
            AuthenticationManager.CurrentUser,
            CurrentJoinPoint))
        {
            throw new UnauthorizedAccess();
        }
    }
}
```

Code 1: An access check aspect for verifying permission to run use-cases of documents

Querying is one of the most important features which any real-world persistence layer should support. Almost all databases provide the ability of querying to some extent. A persistence layer should make a bridge between this and the target programming languages. This problem is especially important when the target language is object-oriented because having relational queries in object-oriented languages is obviously a mismatch and is not suitable. Many studies have been conducted to solve this problem. In this work, we will discuss the .NET framework because of the querying ability that it provides. LINQ is introduced in the .NET framework 3.5 as one of its components. *"LINQ is a methodology that simplifies and unifies the implementation of any kind of data access"* [2]. LINQ is used in this work to provide object-oriented querying for a custom persistence layer.

So far, persistence is considered as a cross-cutting concern in many studies like [3, 5]; but having object-oriented queries is not considered in many of these studies. Having both persistence as an aspect and providing object-oriented querying mechanisms is discussed in this article. The rest of the paper is organized as follows. Section 2 presents some related works and the differences between this study and previous ones. Section 3 explains a method for adding the ability of programming in an aspect-oriented way with the .NET framework. Section 4 shows how some persistence concepts can be defined as aspects. Section 5 describes a way in which one can query on objects in an aspect-oriented persistence layer. Finally, Section 6 introduces the future works which can be done in this way.

2 RELATED WORKS AND COMPARISON

The idea of separating persistence in applications is not new and it is introduced in [3] which talks about how persistence can be aspectized in a highly reusable fashion. This study shows how persistence can be used as an aspect for Java applications with AspectJ, which is one of the most powerful aspect-oriented languages [6]. It covers database actions such as insert, delete and update. Although [3] shows that persistence is a cross-cutting concern, there is much to do to have a real aspect-oriented persistence layer. For example, a good persistence layer must be aware of different kinds of mappings and should support them in a modular way [7]. Also [5] presents an aspect-oriented implementation of persistence for Enterprise Java Beans (EJB) which manages the life-cycle of objects.

As pointed out in Section 1, one of the challenges when working on persistence is querying. Many works have been done recently to resolve the mismatch between object-oriented persistence layers and relational queries, e.g. [11, 10], but the most important step is the introduction of LINQ by Microsoft [15]. Many works have started since then [8] which uses LINQ to make the databases transparent or [9] which shows how LINQ can bridge object-oriented programs to relational databases. Maybe the most powerful feature of LINQ is the ability of querying in an object-oriented manner in the same domain where objects are defined. This property is very important in this scope because the object-oriented programming languages

need object-oriented ways to access their data while most of the persistence services and layers provide relational ones. The object-oriented features available in LINQ can be used to provide an appropriate solution for this problem. In this article, we discuss how LINQ can provide the ability of querying a custom persistence layer.

The .NET framework, unlike Java, is not frequently considered in works related to this scope. AspectJ is mostly used in works that are trying to implement persistence as an aspect because of its strong ability for aspect-orientation [3, 5]. This study discusses how to provide aspect-orientation and object-oriented queries for a .NET persistence layer, two concepts which normally are not studied together or linked together. In addition, as the .NET framework does not have built-in features for aspect-orientation, some parts of this article deal with a method of adding AOP to the .NET languages. Also, we tried to make the discussion independent of any persistence layer, so the details of using persistence layers and some implementation issues are omitted, and we tried to focus on the ability of aspect-orientation and querying.

3 ASPECT-ORIENTATION IN THE .NET FRAMEWORK

The .NET framework does not have any built-in features for using aspect-orientation; but there are some ways to add this ability into the .NET framework. There are some examples on the Web [16, 17, 18]. To be able to use aspect-orientation features, one may choose a new compiler, but some infrastructure ways of cutting the method calls is available in the .NET framework for context-bound and remote objects. Any method call on a remote object is translated to an instance of an *IMessage* object. The translated message is passed to the server sink which later processes the message. One can intercept the scenario on *SyncProcessMessage* method of the server-side sink, before or after the message is processed, and do everything which is needed there (See Code 2). This idea is borrowed from [18] which used *ContextBoundObject* infrastructure properties for intercepting method calls.

According to Micorsoft Developer Network's (MSDN) description for the class *ContextBoundObject*, a context-bound object resides in a context and is bound to its context rules. A context is a set of properties or usage rules that defines an environment where a collection of objects reside. The rules are enforced when the objects are entering or leaving a context [19]. One can add context properties by adding a *ContextAttribute* inherited class. When an instance of a context-bound object is being created, its calling context is checked to see if it is consistent for the object[1]. If it fails, a new context is created for the object and is initialized according to the properties which its context-attributes provide.

When a cross-context call occurs, the .NET framework infrastructure treats the call like a remote call. It assumes the caller is a client and the object whose method is called is a remote object. Any method call on a remote object is translated to an instance of an *IMessage* object. The translated message is passed throughout

[1] Its context-attributes specify if the current context is consistent.

the chain of client and server sinks to the remote object.

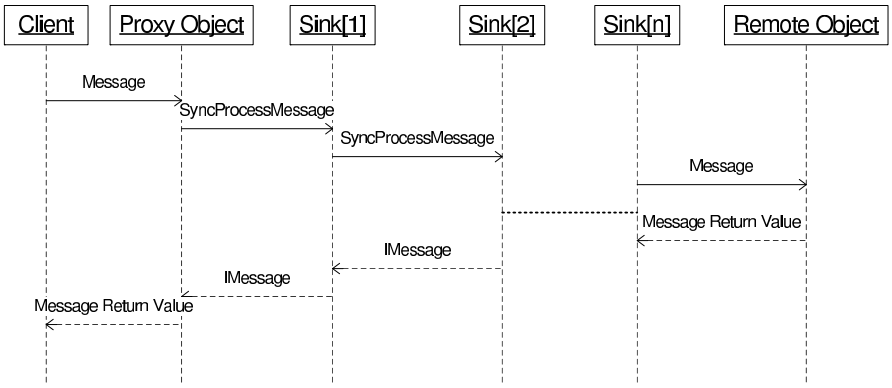

Fig. 1. A remote call passing through the chain of sinks

The *SyncProcessMessage* of the first client sink is called by the remoting infrastructure[2]. Each sink calls the *SyncProcessMessage* of its next sink until the message reaches to the boundary sink. The boundary sink transfers the message to the server boundary sink. Then, the message passes through the chain of server sinks to the remote object. Finally, the remote object method is called and the result of the call returns through the sinks back to the client (see Figure 1). Each context-property can add a sink to the chain of object message sinks whose attributes add the property to the context. In the *SyncProcessMessage* method of the added sinks, the scenario of the call can be modified (See Code 2). One can use this property to add advised methods using a custom sink.

The *AspectReceiverAttribute* class adds a custom sink to objects (see Code 3). Any *ContextBoundObject* inherited class which has this attribute (*AspectReceiverAttribute*) adds an instance of the *AOProperty* to its context. The *AOProperty*, in turn, adds an instance of the *AOSink* class to the chain of the object sinks. The *AOSink* implements the *IMessageSink* interface in a way that its *SyncProcessMessage* method can intercept method invocations from outside of the context. Also, the *IsContextOK* method of *AspectReceiverAttribute* which returns *false* causes each object to have its own context. So, all cross-object calls are passed through the sinks.

The *AOSink* class is defined as shown in Code sample 2 and provides the ability of adding pre-process or post-process calls to the main function. Now what is needed to intercept the method invocations is almost available. Any *ContextBoundObject* inherited class, directly or indirectly, with *AspectReceiverAttribute* as an attribute, has the ability to have aspects.

The *AspectManager* class manages all aspects. It has a collection of instances of the *IAspect* interface which aspects should be implemented. The *IAspect* interface

[2] Synchronous calls are considered.

```
class AOSink: IMessageSink
{
    ...
    public IMessage SyncProcessMessage(IMessage msg)
    {
        // BEFORE METHOD CALL ACTIONS

        IMessage retMsg = NextSink.SyncProcessMessage(msg);

        // AFTER METHOD CALL ACTIONS
        return retMsg;
    }
    ...
}
```

Code 2: *SynchProcessMessage* method of a server sink

has a collection of *IAdvise* objects which introduces the advises of the aspect. The *AspectManager* class uses advises in all the aspects to find the point-cuts and their advises. Point-cuts can be introduced using regular expressions. When a method is called, the *AOSink* asks the *AspectManager* if the calling method is a join-point. If it is a join-point, the *AOSink* asks for its advises and invokes them in the advised place. Code sample 4 shows a sample class *Document* with the access check advised to it before viewing the document. Also, action of viewing the document is reported after the completion of the view use-case.

4 ASPECT PERSISTENCE

Persistence, i.e., the storage and retrieval of application data from a storage medium is often assumed to be a cross-cutting concern. Persistence has different aspects which are discussed in [3]. Reference [5] discusses its implementation for the EJB persistence framework JSR220 [14]. In this study the creation, modification and deletion of persistent-objects are examined. Also, some points about the life cycle of persistent-objects and using different mappings are mentioned.

The state of the object can be considered as being composed of two parts: the *dynamic-state*, which is typically in memory and is not likely to exist for the whole lifetime of the object (for example, it would not be preserved in the event of a system failure), and the *persistent-state*, which the object could use to reconstruct the dynamic-state [13].

The persistent state of objects can be introduced either by using some attributes or by using XML files. Code samples 5 and 6 are samples which make use of attributes and XML files, respectively.

For any persistent-type, a *storage-type* is defined whose instances called the *storage-object* which handles the persistent-state of the persistent-object. Storage-

```
[AttributeUsage(AttributeTargets.Class)]
public class AspectReceiverAttribute: ContextAttribute
{
    ...
    public override bool IsContextOK(
        Context ctx,
        IConstructionCallMessage ctorMsg)
    {
        return false;
    }

    public override void GetPropertiesForNewContext(
        IConstructionCallMessage ctor)
    {
        ctor.ContextProperties.Add(new AOProperty(Name));
    }
    ...
}
```

Code 3: Definition of the *AspectReceiverAttribute* class

```
[AspectReceiver(typeof(Document))]
public class Document: ContextBoundObject
{
    ...
    [AdviseBefore(typeof(AccessCheck))]
    [AdviseAfter(typeof(Auditing))]
    public void View()
    {
        ...
    }
    ...
}
```

Code 4: Sample document class with two aspects

objects are managed by *storage-homes* which are themselves managed by the *catalog* [12]. The type that manages the storage-objects is defined by the provider of the data-domain used. For example, if MS SQL database is used as data-domain, one may use *Table<TEntity>* or *DataTable* classes to manage storage-objects. For communicating with the persistence layer, each provider implements an interface called *ICatalog* which provides the functionality of the provider which is needed for the persistence layer. Each database or any other data-domain can be introduced to this persistence layer by implementing the *ICatalog* interface. For example, *DataContext* can be considered as a catalog providing MS SQL if it implements the *ICatalog*

```
[AspectReceiver(typeof(School)), TablePerClass]
class School: ContextBoundObject
{
    [PersistentState(IsPrimaryKey = true)]
    public Guid Id { get; private set; }

    [PersistentState(Length = 100, Nullable = false)]
    public string Name { get; set; }

    [PersistentState(Nullable = false)]
    public DateTime EstablishDate { get; set; }

    [PersistentState(Cardinality = Cardinality.OneToOne)]
    public Person Principal { get; set; }

    [PersistentState(Cardinality = Cardinality.OneToMany)]
    public List<Room> Rooms { get; set; }
}
```

Code 5: Defining persistent-state with attributes

```
<?xml version="1.0" encoding="utf=8"?>
<persistentMapping xmlns="urn:persistentMapping">
  <persistentState type="School" mapping="TablePerClass">
    <key name="Id" type="Guid" />
    <property name="Name" length="100" nullable="false" />
    <property name="EsablishDate" nullable="false" />
    <one2one name="Principal" type="Person" />
    <one2many name="Rooms" type="Room" />
  </persistentState>
</persistentMapping>
```

Code 6: Defining Persistent-State with XML Tags

interface. Storage-homes should implement the *IQueryable*$<T>$ interface in which T is the type of the storage-objects which are managed by the storage-home. Note that the storage-homes can only implement the *IEnumerable*$<T>$ interface, but it is not preferable because in this case running queries may take much more time.

Three kinds of actions on a persistent-object should be handled by a persistence layer. These actions are creation, modification and deletion.

Creation of a persistent-object has an advised method, which is called after the creation. After the construction of the persistent object, an instance of the related storage-object is created. Then, the persistent-object is introduced to the created storage-object and both are reported to the provider for querying. Figure 2 shows the scenario for the creation of a persistent-object.

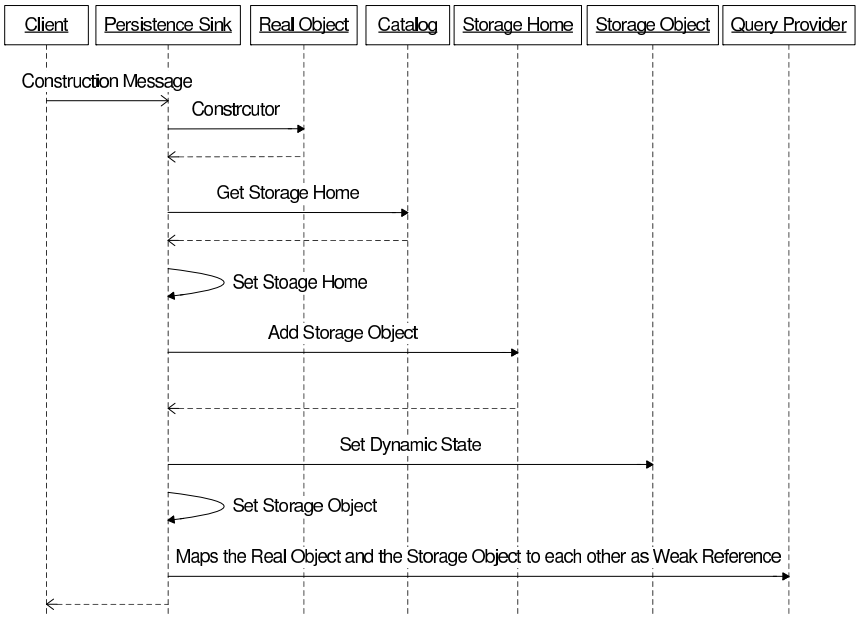

Fig. 2. Creation of a persistent-object

The modification of a persistent-object occurs in its method call if all of its fields are defined as *private* or *protected*. When a method on a persistent-object is called, the object is checked to see if its persistent-state is modified. If there is any modification, the storage-object holding its persistent-state is updated according to the new state. Figure 3 shows the scenario for the modification of a persistent-object.

Deletion of a persistent-object cannot be handled in an aspect-oriented way because destructors of objects are called when garbage collector decides to call them, and there is no *delete* command in the .NET framework as in the *C* or *C++* languages. One is obliged to use a direct way for deleting persistent objects. A persistent-object should be removed with the *Remove* method of the using catalog.

According to Section 3, advised methods of creation and modification of persistent-objects can be added to them by using aspects. The persistent-types should be inherited from *ContextBoundObject* and have an *AspectReceiverAttribute* as a part of their meta-data. So, an aspect should be defined for them which has two advised methods, one for the point-cut of object creation and the other for the point-cut of object modification. With such advised methods, the creation and modification of persistent-objects can be handled.

In order to increase application performance, methods which do not modify the persistent-state can be specified with the *PreservePersistentState* attribute or the *preserveState* XML tag. This is because many methods of persistent-objects do not change their persistent-state. Such methods can be ignored and should not be

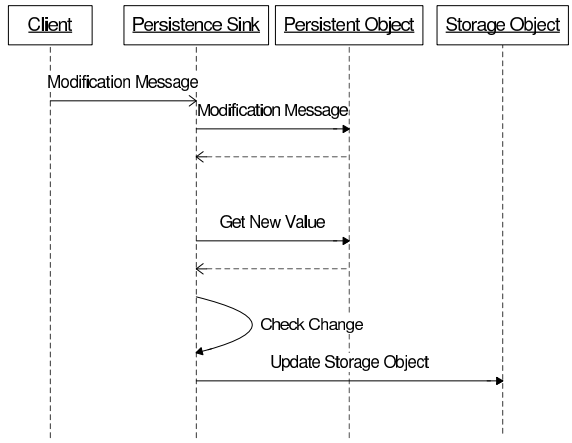

Fig. 3. Modification of a persistent-object

added to the modification point-cut. Also, each method may change only a small part of the persistent-state. So, one can use the *ProbableChange* attribute or the *probableChange* XML tag for some methods to specify the parts of persistence-state which may be modified by the methods. These meta-data help the persistence layer know which parts of the persistent-state should be checked for changes after callings on those methods.

Mapping strategies can be controlled by an XML file, although classes may override the default settings using attributes (See Code 5) or *persistentState* XML tags (see Code 6). Four common mapping strategies [4] are supported in the persistence layer:

Mapping hierarchy to a single table: In this method, all of the attributes of all the persistent-types are stored in a single table.

Mapping each concrete class to its own table: In this method, a table exists for any persistent-type which holds inherited attributes in addition to its own attributes.

Mapping each class to its own table: In this method, a table exists for any persistent-type which holds its own attributes and not inherited attributes.

Mapping classes to a generic table structure: In this method, the value of all properties are held in a single table which holds tuples in the form of (*ObjectId*, *AttributeId*, and *Value*).

The life-cycle of objects is controlled by their storage-objects. For example, if one uses a *DataSet* class as the catalog, *DataRow* instances, which hold the persistent-states, manage the objects' life cycle. When an instance of *DataRow* is added into a *DataTable*, its *RowState* property is set to *Added*. When it is modified, the property is set to *Modified*. After deletion, it changes to *Deleted*. When a transaction commits

it, the property changes into *Unchanged* if it is not deleted and otherwise becomes *Detached*. According to this state, the persistence layer can decide how to act with storage objects.

5 OBJECT-ORIENTED QUERIES

Having the ability of performing object-oriented queries is one of the key-features of a persistence layer. Having object-oriented querying ability helps the developers query in the same domain in which their objects are defined (not in the relational entities' domain). Also, a persistence layer which allows its users to query in an object-oriented manner provides more functionality which makes data retrieval much easier. Also, the code which is written with such queries is more clean and readable. If syntax-highlighting and compile-time error checking is provided for the query language that is used, it helps the developers solve many query problems while the project is under development, and avoid many run-time problems.

LINQ is a uniform programming model for any kind of data access. LINQ enables the users to query and manipulate data with a consistent model that is independent from the data sources. The model's consistency is one of the key-features of LINQ, because writing and understanding of such queries require nothing more than understanding the model used. Also, the codes can be maintained more easily. LINQ is used in this article to add the ability of object-oriented querying.

The *QueryTranslator* is a generic class which is designed for querying persistent-objects. One can access it with the *GetSource*<*T*> method of the *QuerySource-Provider* class in which *T* is the persistent type which the set of its instances is assumed as the source of the query.

Figure 4 represents the summarized class diagram of these types. *QueryTranslator*<*T*> implements *IQueryable*<*T*> in a way that it translates queries on the domain of persistent-objects into queries on the storage-objects' domain, and converts the result to the source domain. The *QueryTranslator* class uses *QueryTranslation-Provider* to provide data of queries which implements *IQueryProvider*.

The *CreateQuery* methods of *QueryTranslationProvider* construct instances of *QueryTranslator* with the given expression. *Execute* methods of *QueryTranslation-Provider* send the given expression to *ExpressionTranslator* class to be translated. If the translation succeeds, the resulting expression is sent to the related storage-home which is taken from the using catalog. The result of the query should be converted to the requested type. The result of queries can vary from simple types such as a persistent-type to very complex ones like any custom anonymous type. Converting all such results is very difficult; but some of the types which are used almost in every query are more important. In the present study, some types as the result are supported which are:

Simple types: Simple types need no translation because the results of such queries are automatically correct. For example, extracting the property name of a persistent-type customer needs no conversion because it is of type *string*.

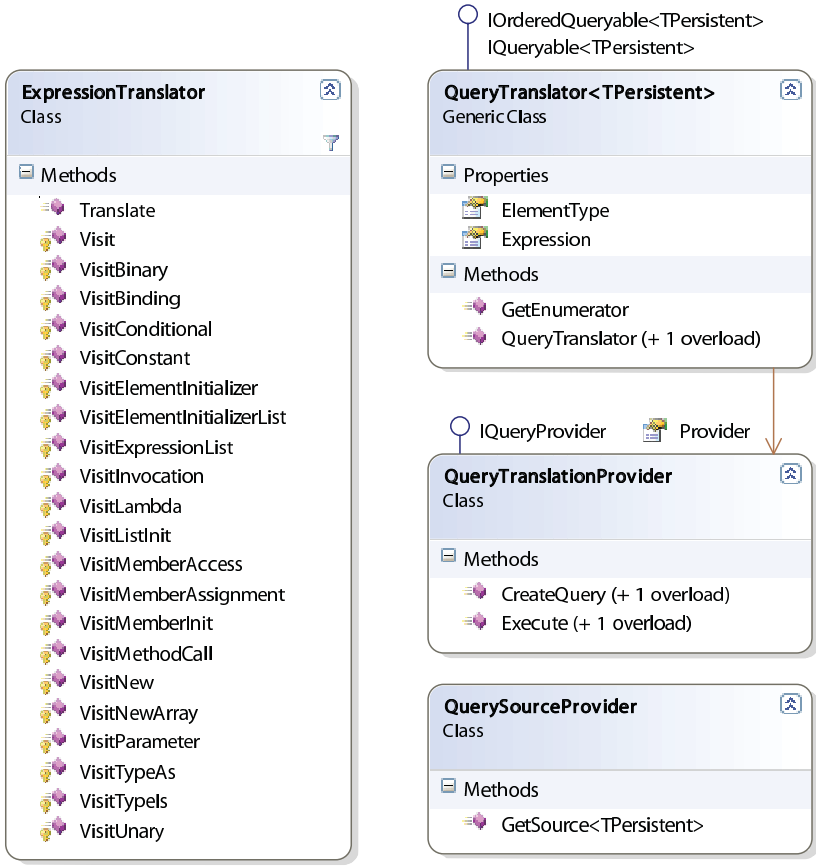

Fig. 4. Brief class diagram of the main querying types

Persistent-types: The output of query operators like *First*, *Last* and *Single* can be a persistent-object. A generic factory type is used for all persistent-types and assumes that all the persistent-types have a constructor with no parameter, although other factories can be introduced with the *Factory* attribute or the *factory* XML tag.

IQueryable<T>: Many of the query operators return *IQueryable*$<T>$ instances (for example, the result of *Distinct*, *Where*, *Intersect* and *Join*). Instances of *IQueryable*$<T>$ are converted when T itself can be converted.

IEnumerable<T>: Using *foreach* loops on the result of a query, calls the *Execute* method of *QueryTranslationProvider* with this type. *IEnumerable*$<T>$ is converted when T itself can be converted.

IGrouping<TKey, TElement>: Grouping is a very important command in querying. Some overloads of *GroupBy* commands return a collection of instances of *IGrouping*. *IGrouping<TKey, TElement>* is convertible whenever *TKey* and *TElement* can be converted.

Anonymous Types: Anonymous type instances can be the result of queries. The *Select* family operators can have *IQueryable<T>* as output in which *T* is an anonymous type. Anonymous type instances are converted when their properties can be converted.

Considering the above categories, one can realize that a recursive approach is used for the result translation process. When the result of a query is going to be translated, the target type is checked. If it needs some nested data for construction, the translation will be recalled for the nested related data, until the translating data has no nested data. When all the nested data are translated, the data itself is translated.

If the translation fails for any reason, *QueryTranslationProvider* itself runs the query over the set of all the storage-objects of the related type instead of using the querying abilities of the catalog. In more detail, the provider runs a query which retrieves all the storage-objects. Then using these objects, it converts the result from the set of storage-objects to a set of persistent-objects and runs the query over the resulting set.

The *ExpressionTranslator* class translates query expressions from the persistent-objects' domain into the storage-objects' domain. Each expression may consist of some subexpressions in addition to the data itself. For example, an instance of the *MethodCallExpression* class has a collection of expressions as its parameters, an expression for the object whose method is called, and an instance of *MethodInfo* which holds the information of the calling method. *ExpressionTranslator* uses a dynamic-programming approach to translate expressions. Some sample rules are presented in Code 7.

```
MethodCallExp(Method, Object, Params) ⟹(Trans)
  MethodCallExp(Trans(Method), Trans(Object), Trans(Params))

ConditionalExp(Test, IfTrue, IfFalse) ⟹(Trans)
  ConditionalExp(Trans(Test), Trans(IfTrue), Trans(IfFalse))

MemberAccessExp(Member, Expression) ⟹(Trans)
  MemberAccessExp(Trans(Member), Trans(Expression))

ConstantExp(Value) ⟹(Trans)
  ConstantExp(Trans(Value))
```

Code 7: Sample Rules of Translation

The rules which are presented in Code 7 are not the exact rules which are used. They show the main action of the convertors. Some rules are needed to handle some special cases. For example, *MethodCallExpression* should convert the method *QuerySource.GetSource<TPersistent>* into the method *Catalog.CurrentCatalog.GetStorageHome<TStorage>*.

Translation of non-expression objects also should be done. Although the following rules do not cover all the expressions, they may help run many kinds of queries:

ConstructorInfo: If the constructor of a persistent-type is used in an expression, it should be converted to the constructor of its related storage-type.

MethodInfo: If a method is a member of a persistent-type or is a generic method with some generic parameter of persistent-types, it should be converted to a related method of a storage-type or a generic method with generic parameters of storage-type respectively.

MemberInfo: If a member of a persistent-type is used in an expression, it should be converted to a related member of its storage-type. Please note that if an instance of *MethodInfo* is used as a *MemberInfo* instance, it can be converted using the previous rule.

Type: If a persistent-type, query source or a generic type with some generic parameter of a persistent-type is used, it should be converted to the type of its related storage-object, storage-home or generic type with the generic parameters of storage-types respectively.

MemberBinding: If an initialization of some members of a persistent-object occurs in a query, they should be converted to the initialization of members of their storage-object.

ElementInit: If a collection is initialized inside a query, its element should be converted. Each *ElementInit* has a method and a list of parameters.

Object: If an object is used in an expression which is a persistent-object or a query source, it should be converted to a storage-object or a storage-home respectively.

Using these rules, one can translate many kinds of queries on persistent-objects into queries on storage-objects. So the querying abilities of the using data-domain would be available for designers and developers. Code samples 8 and 9 present two samples of successful and unsuccessful translation, respectively.

6 CONCLUSION

The persistence layer is a part of many softwares. One of the goals of this article is to show how persistence can be an aspect. Despite other alternative methods, a remoting-based approach is used in this article to add aspects to objects.

Having object-oriented queries is a very important feature of a persistence layer. This ability helps the developers query in the domain in which their objects are

```
var query      = from Document d in QuerySource.GetSource<Document>()
                 where d.Name.StartsWith("Technical")
                 orderby d.CreateTime
                 select d;
// Is converted to ...
var innerQuery = from PDocument pd in
Catalog.CurrentCatalog.GetStorageHome<PDocument>()
                 where pd.Name.StartsWith("Technical")
                 orderby pd.CreateTime
                 select pd;
var query      = from PDocument pd in innerQuery
                 select
Catalog.CurrentCatalog.GetFactory<Document>.Create(pd);
```

Code 8: A sample of a successful query translation

```
var query      = from Document d in QuerySource.GetSource<Document>()
                 where SomeUnknownMethod(d)
                 select d;
// Is converted to ...
var allObjects = from PDocument pd in
Catalog.CurrentCatalog.GetStorageHome<PDocument>()
                 select pd;
var innerQuery = from PDocument pd in allObjects
                 select
Catalog.CurrentCatalog.GetFactory<Document>.Create(pd);
var query      = from Document d in innerQuery
                 where SomeUnknownMethod(d)
                 select d;
```

Code 9: A sample of an unsuccessful query translation

defined. This article showed how object-oriented queries can be available for developers through LINQ. How queries on persistent-objects can be translated into queries on storage-objects is also explained. After that, the translated query was submitted to the using data provider. Finally, the result of queries was translated back into persistent-objects' domain.

What has been presented in this article can be developed through future research. It would be a proper research project to try to use a more powerful method of translating the result of queries back to the using model since the method which is used in this article cannot be applicable on all kinds of queries although it can translate many kinds of queries. For example, if a query uses some non-persistent properties of a persistent type, our approach cannot translate it successfully. Maybe this problem can be solved if one uses MS SQL Server 2005 or higher

version because of the ability to run .NET code inside new versions of MS SQL Server.

Another point which can be a subject for future research can be introducing an approach of having aspects, because the using method forces the persistent-objects to inherit from *ContextBoundObject*, directly or indirectly, which is not good. Also, translating calling messages and translating them back adds some overhead to the time of the method calls which may be considerable and could be decreased by the use of more optimal techniques.

REFERENCES

[1] Kiczales, G.—Lamping, J.—Mendhekar, A.—Maeda, C.—Lopes, C.—Loingtier, J. M.—Irwin, J.: Aspect-Oriented Programming. In: Lecture Notes in Computer Science, Proceedings of the Eleventh European Conference, ECCOP '97, Finland, June 1997, pp. 220–242.

[2] Pialorsi, P.—Russon, M.: Introducing Microsoft LINQ. Microsoft Press, Redmond, Washington 2007.

[3] Rashid, A.—Chitchyan, R.: Persistence as an Aspect. In: M. Aksit (Ed.): Aspect-Oriented Software Development, Proceedings of the Second International Conference (AOSD '03), Boston 2003, pp. 120–129.

[4] Amber, S.: Agile Database Techniques: Effective Strategies for the Agile Software Developer. John Wiley & Sons, Inc., New York, NY, USA 2003.

[5] Hohenstein, U.—Meunier, R.—Schwanninger, C: An Aspect-Oriented Implementation of the EJB3.0 Persistence Concept. In: Aspects, Components, and Patterns for Infrastructure Software, Proceedings of the Sixth workshop ACP4IS '07, Vancouver 2007, p. 4.

[6] Kiczales, G.—Hilsdale, E.—Hugunin, J.—Kersten, M.—Palm, J.—Griswold, W.: An Overview of AspectJ. In: Lecture Notes in Computer Science, Proceedings of the Fifteenth European Conference ECOOPS '01, Budapest June 2001, pp. 327–353.

[7] Hohenstein, U.: Using Aspect-Orientation to Add Persistency to Applications. In: Proceedings of Database System in Business, Technology andWeb (BTW), Karlsruhe 2005.

[8] Wiedermann, B.—Ibrahim, A.—Cook, W. R.: Interprocedural Query Extraction for Transparent Persistence. ACM SIGPLAN Notices, Vol. 43, 2008, No. 10, pp. 19–36.

[9] Meijer, E.: There is no Impedance Mismatch: Language Integrated Query in Visual Basic 9. In: Dynamic Languages Symposium, Companion to the Twenty First ACM SIGPLAN symposium OOPSLA '06, Portland 2006, pp. 710–711.

[10] Alashqur, A. M.—Su, S. Y. W.—Lam, H. L: OQL: A Query Language for Manipulating Object-Oriented Databases. In: Very Large Data Bases, Proceedings of the Fifteenth International Conference VLDB '89, Amsterdam, August 1989, pp. 433–442.

[11] Willis, D: The Java Query Language. M. Sc. Thesis, Victoria University of Willington 2008.

[12] OMG, Persistent State Service Specification, September 2002.

[13] OMG, Persistent Object Service Specification, April 2000.

[14] JSR 220 Specification: Proposed Final Draft 19. 12. 2005 (EJB Simplified API, Java Persistence API, EJB Core Contracts and Requirements). Available on: `http://jcp.org/aboutJava/communityprocess/pr/jsr220/index.html`, 2008.

[15] Microsoft Corporation: The LINQ Project. Availaible on: `http://msdn.microsoft.com/netframework/future/linq`.

[16] NKalore Compiler. Availaible on: `http://aspectsharpcomp.sourceforge.net/about.htm`.

[17] Castle Project. Availaible on: `http://www.castleproject.org/AspectSharp/index.html`.

[18] Interceptin Method Calls in C#, an Approach to AOSD. Availaible on: `http://www.codeproject.com/KB/cs/aspectintercept.aspx`.

[19] Microsoft Corporation: ContextBoundObject Class, The MSDN Library. Available on: `http://msdn.microsoft.com/en-us/library/system.contextboundobject.aspx`, 2008.

Mohammadreza Jooyandeh has graduated from Amirkabir University of Technology in computer science and pure mathematics for B. Sc. in 2006 and 2008, respectively. In 2009 he has received his M. Sc. in computer science from the same university. His research interest is isomorph-free exhaustive generation of graphs; he started his Ph. D. in 2010 at the Australian National University under supervision of Prof. Brendan McKay. He has published seven scientific papers so far.

S. Mehdi Hashemi has graduated in the field of dynamical systems from Ferdowsi University of Mashad in 1988. In 1996 received his Ph. D. degree in applied mathematics in computer sciences from Ottawa University, Ottawa, CA. He is currently a Full Professor in the Department of Computer Science, Amirkabir University of Technology, Tehran, Iran. His research interests span the areas of graph drawing, network flow optimization and information technology. He has published more than 40 scientific papers and has translated a textbook on applied and algorithmic graph theory. He has also managed several research projects in network transportation and information technology.