UDC:007.52
Original scientific paper

DATA ACCESS ARCHITECTURE IN OBJECT-ORIENTED APPLICATIONS USING DESIGN PATTERNS

Danijel Matić
EUROCOMPUTER SYSTEM, Zagreb
danijel.matic@ecs.hr

Hrvoje Kegalj
Infodom, Zagreb
hrvoje.kegalj@infodom.hr

Dino Butorac
Epsilon, Zagreb
dino.butorac@epsilon.hr

Abstract: *This paper is describing data access architecture in a modern object-oriented application. Complex application solutions have multiple, parallel data sources. Each data source has specific properties and ways to access data. This architecture, by using already tried solutions, ensures a simple and flexible way to access different data sources. It's also describing singleton, data access object and abstract factory patterns and their interaction in achieving flexible and scalable data access architecture.*

Keywords: *design patterns, architecture, object-oriented, data access.*

1. INTRODUCTION

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder, but just keeping the focus on reusability is not enough. In order to deliver the desired quality to the end user, an application must also exhibit a wide variety of architectural requirements. Robert Grady categorized these necessary quality requirements of a software system, referred to as "*FURPS* " in his work [5]. The nonfunctional (the *URPS* part) requirements he described deal mostly with the architecture of an application, and he categorizes them in the following way:

- *Usability*, which is concerned with characteristics such as aesthetics and consistency in the user interface.
- *Reliability*, which is concerned with characteristics such as availability, accuracy of system calculations, and the system's ability to recover from failure.
- *Performance*, which is concerned with characteristics such as throughput, response time, recovery time, start-up time, and shutdown time.

- *Supportability*, which is concerned with characteristics such as testability, adaptability, maintainability, compatibility, configurability, installability, scalability, and localizability.

In order to meet all these demands the design of the architecture of an application becomes an important and vital task in a software development cycle.

This paper describes data access architecture in a modern object-oriented application, it shows, in a problem -solving oriented manner, the transition from the problem, which is stated as: "*to define a flexible, reusable, maintainable and platform independent data access architecture* ", to the realized solution. In designing the solution we remained at a level of abstraction, which is independent of the final implementation of the architecture. So, the implementation of the same architectural solution has been successfully implemented both in the J2EE and .NET environment.

2. PATTERNS

During the design of the data access architecture presented in this paper we used several Design Patterns to accomplish the stated problem. Although Design Patterns are not a new term in modern software development, at this point we will give a short overview about their history, their structure and what we think is the most important part, the pattern catalog, which represents a source of knowledge about previously solved design problems.

2.1. HISTORY

Patterns were first introduces by Christopher Alexander [1] in the 1970s. He realized that there where certain solutions that could be applied over and over again to the same or similar problem. He also combined these existing solutions to create new solutions to new problems. His definition of patterns is:

> "*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same twice.*"

Even though Christopher Alexander was talking about patterns in buildings and towns, his definition remains relevant also in object-oriented software development. This relevance was discovered by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, popularly called the Gang of Four (GoF), in their book [4], where they described 23 Patterns. Their work started a small revolution, maybe the second after the introduction of the object-oriented paradigm, and today the usage of patterns has become a widely accepted industry standard. The patterns described in their book are categorized in three main categories:

- *Creational*, concern the process of object creation.
- *Structural*, deal with composition of classes and objec ts.
- *Behavioral*, characterize the ways in which classes or objects interact and distribute responsibility.

Today there is a significant amount of literature dealing with Design Patterns, for different programming languages and for different problem domains , the majority of them are built upon the Design Patterns introduced by GoF. The expression "are built upon" shows us that Design Patterns can exist on different levels of abstraction, which when properly used can lead to platform independent architectural solution like the one presented in this paper.

Design Patterns have proved themselves in such a manner, that they were

implemented in some programming languages (e.g. Collections Framework in Java consist of several GoF patterns).

2.2. PATTERN STRUCTURE

To understand and use patterns, knowing their structure is fundamental. Although there are a wide variety of different pattern description templates, all of them consist of the same basic structure. This structure is divided into the following 4 parts:

- **Pattern name**

  The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. This is a very important part of the pattern structure, because it improves our design vocabulary. It also allows us to talk about specific design problems and solutions on a higher level of abstraction. Finding an appropriate and «good» name is one of the hardest parts in describing a pattern.

- **Problem**

  The problem describes when to apply the pattern. It describes the problem and the context. In same cases it also describes some specific design problems or it gives a description of some class or object structures, which are symptomatic for an inflexible design. Sometimes the problem includes a list of precondition, which have to be met in order to apply the pattern.

- **Solution**

  The solution describes the elements, which make up the design, their relationships, responsibilities and collaborations. It doesn't describe a concrete solution, like some particular implementation; rather it gives a more abstract solution in order to be reapplied again in some other appropriate situation.

- **Consequences**

  The consequences are the results and trade-offs of applying a pattern. A pattern is not a "perfect" solution, in order to solve one problem; the pattern can introduce other aspects, which could impact the design negatively. So, when applying some pattern, there has to be a deep understanding on how it will impact the system, especially its flexibility, extensibility or portability.

2.3. PATTERN CATALOG

Using patterns is an effective way to capture and organize knowledge about solutions to recurring problems in every domain of life, not only in software engineering. Knowing patterns not only enriches the vocabulary of a software architect or designer, it also allows a company to capture knowledge in a formal and descriptive way, which can be used as a reference or source of solutions for their projects in the future.

Organizing patters in a catalog (*Pattern Language* is often used as a synonym for a pattern catalog) helps to effectively organize knowledge in such a manner, that the solution for recurring problem can be easily found and applied. Also the usage of patterns in different projects/products can be tracked and managed, which helps to write technical documentation, because the key architectural problems could have been solved by some patterns from the catalog, which are well documented.

Introducing a new employee in a software company can be a difficult and a long-lasting task. He has to learn the way the company does its business, the standards the company uses and the solutions for problems already solved. Solving the problem twice is a typical scenario with new employees. By using a pattern catalog, the new employee can quickly get familiar with company standards, concepts and the usual problem domain the company deals with.

3. DATA ACCESS ARCHITECTURE

3.1. INTRODUCTION

A typical application is composed of many logical packages; such as a user interface package, a database access package and so forth. Each package groups a set of cohesive responsibilities (e.g., database access). This is the basic practice of modularization to support a separation of responsibilities. One way to accomplish this separation of concerns is to organize the logical structure of the application into discrete layers of distinct, related responsibilities, with a clean separation of concerns such that the "lower" layers are low-level and general services, and the higher layers are more application specific.

A layer is a large-scale element, often composed of several packages or subsystem. Our proposed architecture uses the well known 3-tier approach. There is a lot of misunderstanding regarding the difference between layers and tiers, in this work we won't discuss the difference, rather we state that in this work a layer and a tier are logically the same, whereby a tier represent a distributed layer implemented at another node.

The three-tier architecture imposed itself as a good and proven solution for the majority of applications. This approach divides the architecture of an application into three distinct tiers: *presentation, business and persistence*. The presentation tier is responsible for the presentation of data, receiving user events and controlling the user interface. The business logic tier encapsulates the business functionality of the application. The persistence tier is responsible for data storage. Besides the widespread relational database systems, existing legacy system databases are often reused here.

The data access architecture, presented in this paper, is the link between the business tier and the persistence tier, in other words the business tier uses this data access architecture to both populate its business objects with the needed data and to store this data as needed.

3.2. DATA ACCESS OBJECT

Many real world applications consist of business objects, which collaborate in order to perform some needed business functionality. Some of this business objects need to persist their data at some point.

A typical implementation of such a business object is shown on Figure 1. As the figure shows the data access logic is embedded into the business object, so the business object, not only knows its data, it also has the knowledge of how and where to access it.

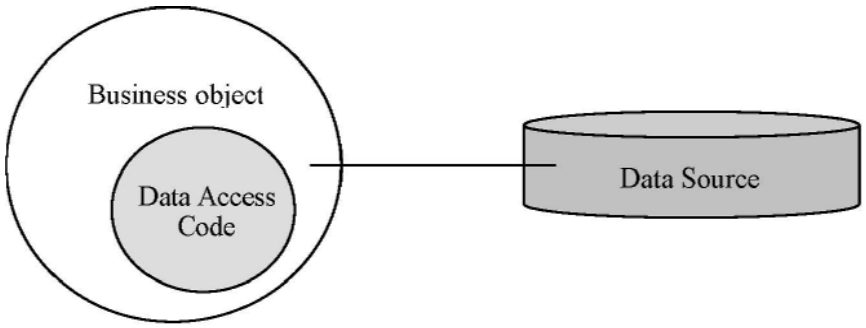

Figure 1: Business object, which encapsulates data access logic

Having knowledge usually means having power and is considered as something "good", but not in this particular case. The bad thing about this approach is the fact that whenever the data source is changed, the business object has to be changed as well, in order to access this new data source. In the worst case, it should be rewritten all over again, but mostly only one segment must be changed, the data access code. The change of th e business object does not only demand the change of the actual implementation code, but as well affects the production environment in which the application is deployed. Usually this means that the application has to be re-deployed, which in some cases dem and the application to be stopped for some time. This is not quite flexible, especially if the application represents a 24/7/365 service, where such a downtime could have negative business impacts.

A known principle says: "*Every software problem can be sol ved with another layer of indirection.*" The principle is also true in this case. In order to make the business object unaware of data source changes, we separated the access logic into a special object, the data access object (See Figure 2). This data access object implements the necessary program logic to access some data source.

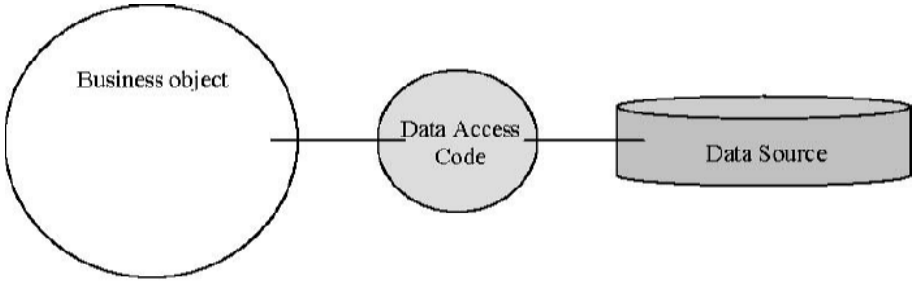

Figure 2: The separation of the data access logic into a separate object

Now the change of data sources does not affect the business object. Another advantage of this approach is that the development of the application can be divided into more teams, which will, according to their expert knowledge, work on the data access objects or on the implementation of business processes in the business logic tier.

This particular solution is called "*Data Access Object*" and is a well know design pattern. The class diagram of the DAO pattern is shown on Figure 3.

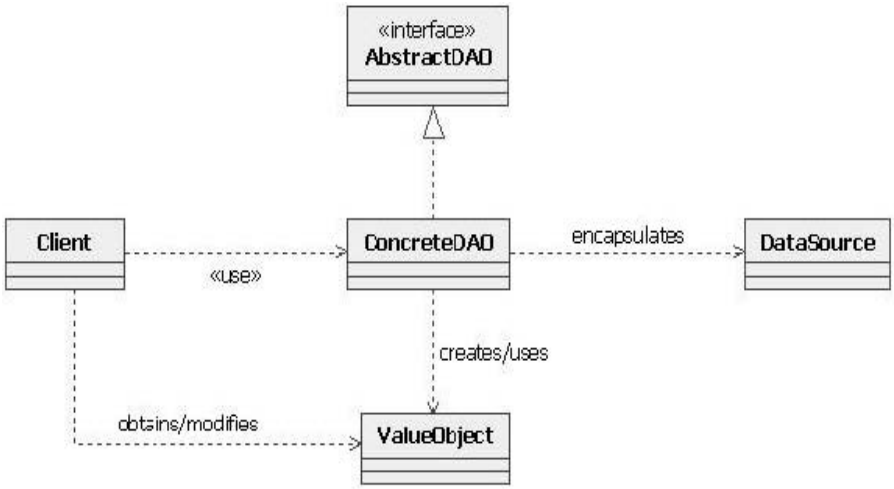

Figure 3: The DAO pattern defines the following classes:

- **Client –** represents the data client, which requires access to the data source to obtain, modify and store data, in our case this is the business object
- **AbstractDAO** - represents the interface which the ConcreteDAO implements, providing such an interface assures that the Client, by programming to the interface, remains intact when the ConcreteDAO is replaced with another implementation which implements this interface
- **ConcreteDAO** – represents the key object of this pattern, it abstracts the underlying data access implementation for the Client, providing transparent access to the data source.
- **DataSource -**represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system or another system (legacy/mainframe).
- **ValueObject (TransferObject) -** represents a data carrier object, which encapsulates all data read or transferred to the data source, rather then populating the client with data directly from the ConcreteDAO, we use this object to achieve low coupling between the client and the data source.

3.3. ABSTRACT FACTORY

Every business object will use its own data access object in order to access data. The access to one particular data source will generate a certain number of data access objects; we refer to it as "a family" of objects. This leads to the following conclusion: "*Every data source in the application will create one family of objects*". The business object will use the same type of object from another family to access data from different data sources. The business object has to be aware of the different families even though, in one moment, it uses just one object to access a data source.

This approach is not good because, when introducing a new data source, a new family of objects has to be created. Also the business object has to be changed in order to be aware of the new data source object family, which is a similar problem like the awareness of the data access logic.

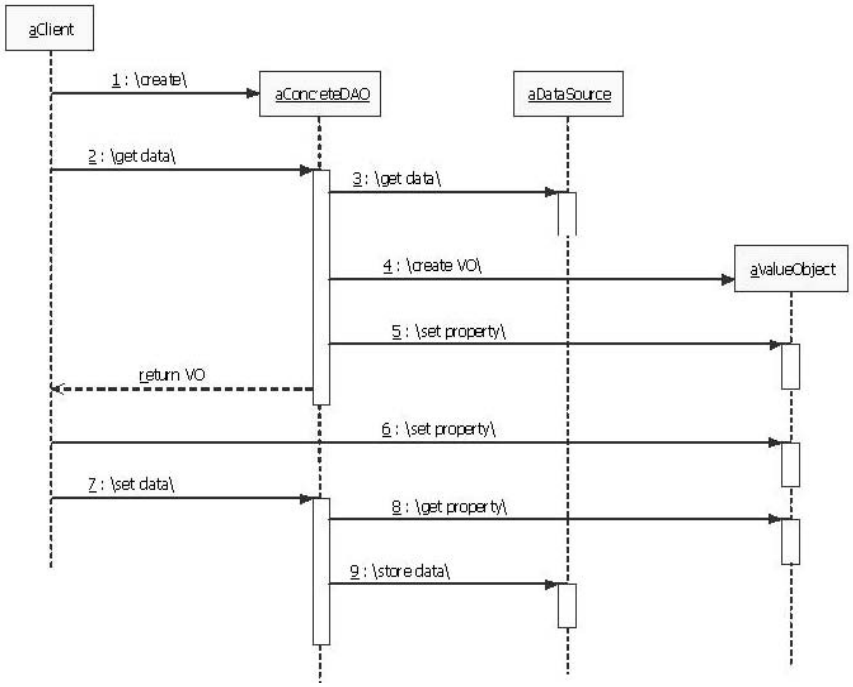

Figure 4: DAO Pattern – Sequence Diagram

We solved this problem by creating a factory object to which we delegated the process of object instantiation. This approach is called the Abstract Factory Design Pattern. By using this pattern we are able to isolate the process of object creation from the business object (this is another example of the "*indirection*" principle stated before). The class diagram of the Abstract Factory Pattern is shown on Figure 5.

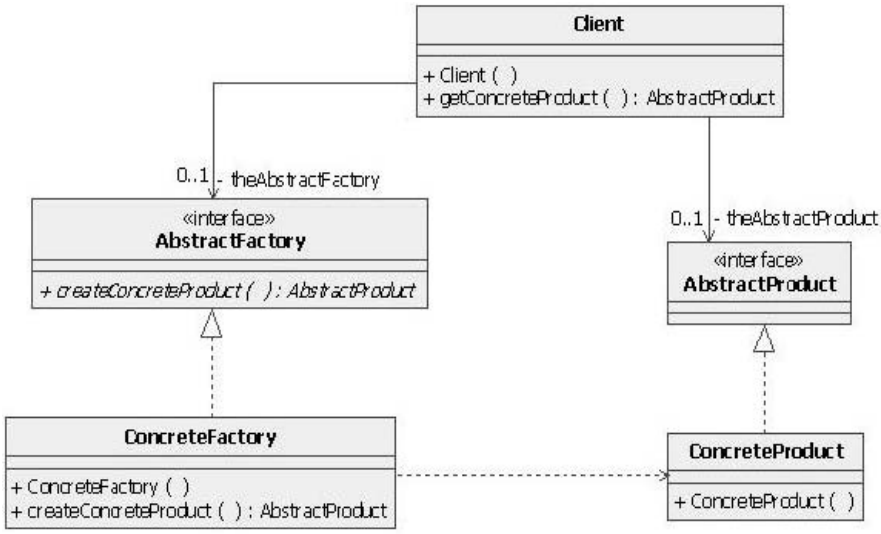

Figure 5: Abstract Factory Pattern – Class Diagram

The Abstract Factory Pattern defines the following classes:

- **AbstractFactory** – declares an interface for operations that create abstract product object.
- **ConcreteFactory** – implements the operations to create concrete product objects.
- **AbstractProduct** – declares interface for type of product objects.
- **ConcreteProduct** – implements AbstractProduct interface. Also defines a product to be created by the corresponding factory.
- **Client** – uses interfaces declared by AbstractFactory and AbstractProduct creating and using concrete products, in our case this is the business object

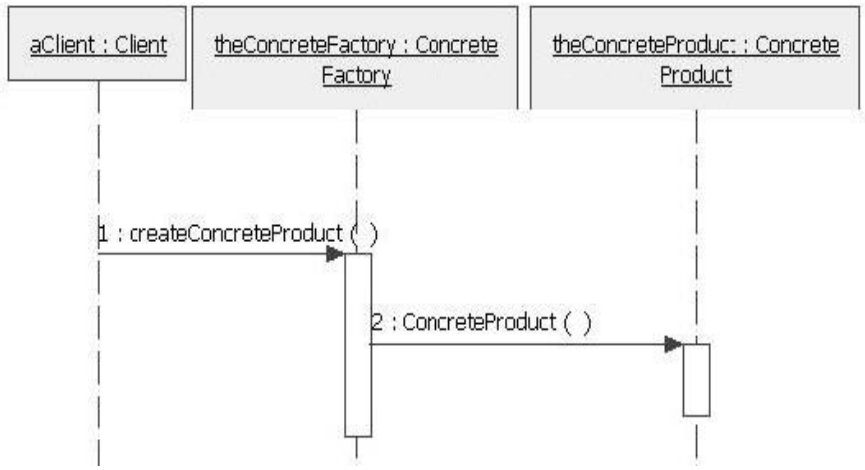

Figure 6: Abstract Factory Pattern – Sequence Diagram

The factory completely abstracts the creation and initialization of the product from the client. This indirection enables the client to focus on its discrete role in the application without concerning itself with the details of how the product is created. Thus, as the product implementation changes over time, the client remains unchanged.

The most important aspect of this pattern is the fact that the client is abstracted from both the type of product and the type of factory used to create the product. Furthermore, the entire factory along with the associated products it creates can be replaced in a wholesale fashion. Modifications can occur without any changes to the client.

By throwing the responsibility of creating certain data access objects over from the client class to the factory class, the change of the source is made possible. The change of data source claims the change of the factory class that is used to create DAOs which is possible to implement into the configuration files.

The negative aspect of this approach is complicated introducing of the new product. Abstract factory interfac e fixes the set of products that can be created. Adding new kind of product requires extending the factory interface. Once factory interface is changed all subclasses of that factory also must be changed.

3.4. SINGLETON

The final step in defining the data access architecture is to ensure the existence of only one instance of a concrete factory.

All business objects use one concrete factory, depending on the type of data source they access, in order to access their concrete DAOs. If every business object would access its

own factory, it could easily result in an explosion of objects. Also by assuring that there is only one factory per family we have one single point of control. By changing this single factory we automatically change the behavior that factory provides for all business objects that access it.

The solution might be the usage of a global variable; however this approach doesn't guarantee that there will be only one instance at the time. A better solution is to transfer the responsibility of creating one instance to the class itself. The class, by intercepting requests to create new objects of it, can control the instantiation process in order to assure only one instance per time. This is the Singleton pattern (see Figure 7).

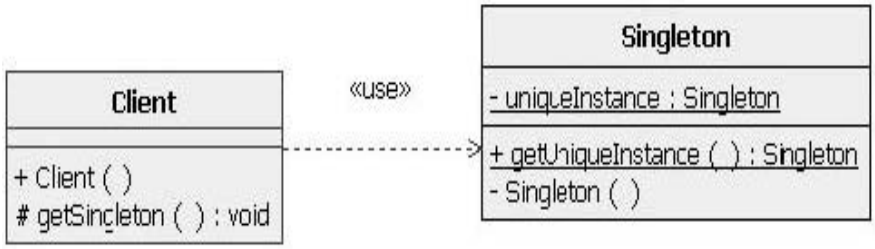

Figure 7: Singleton Pattern – Class Diagram

Clients access any instance of a singleton only through the getUniqueInstance()method. How the instance gets created is the responsibility of the singleton class itself. We also want to be able to control how and when an instance will get created. In OO development, special object creation behavior is generally best handled in the class constructor. This case is not different. We can define when and how we construct a class instance and then keep any client from calling the constructor directly. This is the approach always used for singleton construction.

3.5. DATA ACCESS ARCHITECTURE MODEL

In order to explain the data access architecture we consider the following hypothetical application:

The application will use data, which can be found in the local system as well as data available from other systems. This particular example will access both a RDBMS system and a XML file repository. So, the application must be able to work with both data sources in the same way, there should be no coding in the business logic tier in order to switch to another data source. So, the main problem here is to abstract the data access mechanism in such a way that the business logic tier is unaware of the actual switch of data sources. Also, there should be only one point in the application where the switch has to be performed; multiple points of variation could lead to inconsistent changes.

The solution to this problem is described in the previous parts of this paper and finally presented on Figure 8.

As can be seen on Figure 8 the central part of the data access architecture is the AbstractFactory, which is implemented as a Singleton. It knows which data source is "active" at a particular moment, it knows how to switch to another data source and it is responsible to create objects of a particular family of objects in order to serve the BusinessObject. There is only one AbstractFactory, so there is only one point of variation where the switch between data sources occurs.

The BusinessObject "uses" the AbstractFactory to get its DAO Object. By programming to the AbstractDAO interface, the BusinessObject is unaware of the underlying data access mechanism and is not able to tell which data source its been used.

SQLDAOand XMLDAO are concrete implementations, which implement the AbstractDAO interface and implement the data access code for their particular data sources.

SQLFactoryand XMLFactory are concrete factories responsible to create their DAO objects. They extend the AbstractFactory.

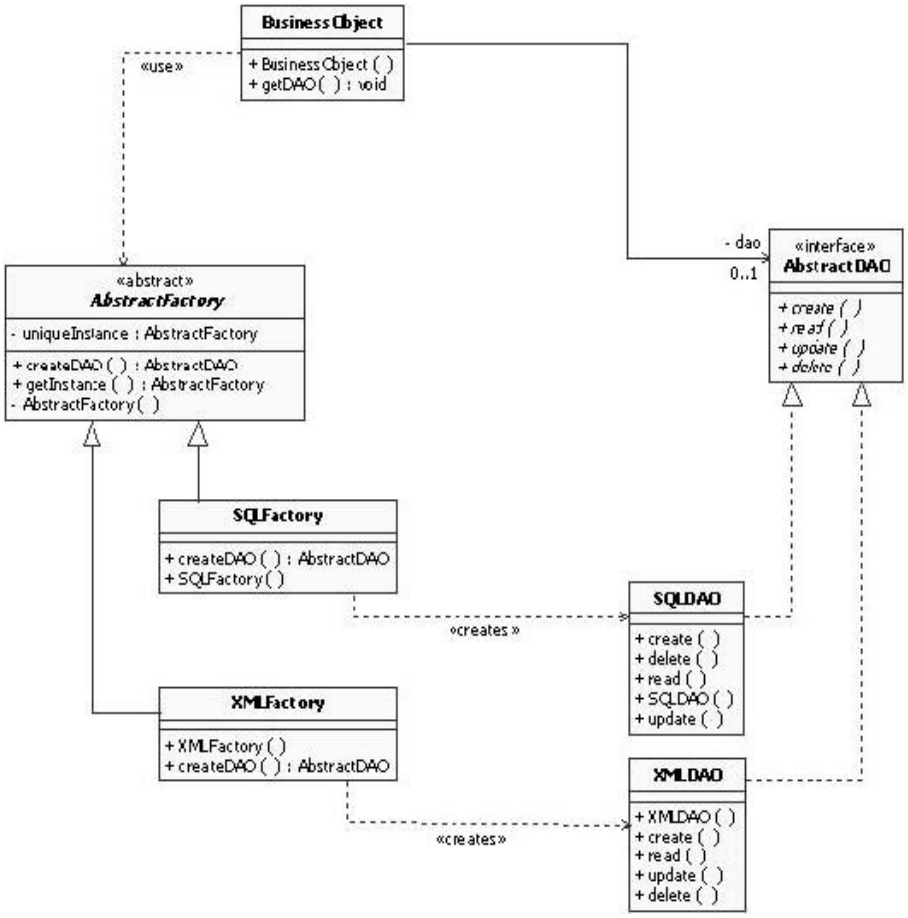

Figure 8: Data Access Architecture

4. CONCLUSION

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder, but just keeping the focus on reusability is not enough. In order to deliver the desired quality to the end user, an application must also exhibit a wide variety of architectural requirements.

Design Patterns are used to define, document and develop reusable components for recurring types of problems. Knowledge of Design Patterns enriches the language and makes communication more efficient during design discussions.

Technology evolves at daily rate, so familiarity with concepts such as Design Patterns at different abstraction levels gives any architect, designer and developer a powerful tool for their jobs. Already there are tools that make it possible to document Design Patterns and to use them in a development environment. Rational XDE is one of them and apart from saving and documenting Design Patterns (GoF Patterns included) it supports model-to-

model and model-to-code transformations.

All of the GoF Design Patterns are independent of programming languages and technology environments. Not only that they do not age, but also a whole set of new Patterns can be built based upon them, for some actual technology, programming language or platform.

This paper described how to build effective and flexible data access architecture with design patterns.

REFERENCES

[1] Alexander, C.; Ishikawa, S.; Silverstein, M.; Jacobson, M.; Fiksdahl-King, I.; Angel, S. (1977): A Pattern Language, Oxford University Press.

[2] Alur, D.; Crupi, J.; Malks, D. (2001): Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall / Sun Microsystems Press.

[3] Fowler, M.; Scott, K. (2000): UML Distilled – A Brief Guide to the Standard Object Modeling Language - Second Edition, Addison-Wesley.

[4] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995): Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley.

[5] Grady, R. (1992): Practical Software Metrics for Project Management and Process Improvement, Prentice-Hall.

[6] Kegalj, H. (2003): The Role of Design Patterns in Building Information Systems, CASE 15, Opatija.

[7] Larman C. (2002): Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process – Second Edition, Prentice-Hal.

[8] Trowbridge, D.; Mancini, D.; Quic, D.; Hohpe, G.; Newkirk, J.; Lavigne, D. (2003): Solution Patterns Using Microsoft .NET, Microsoft Corporation.

Received: 17 December 2003
Accepted: 3 July 2004