First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software

A one-day workshop held in conjunction with the

First International Conference on

Aspect-Oriented Software Development (AOSD 2002)

April 22-26, 2002,

Enschede, The Netherlands

The Department of Computer Science

UNIVERSITY OF BRITISH COLUMBIA

201-2366 Main Mall
Vancouver, B.C.
V6T 1Z4
Fax: 1-604-822-5485

TR-2002-12

Editted by Yvonne Coady

Orthogonal Persistence using Aspect Oriented Programming

Koenraad Vandenborre

Muna Matar

Ghislain Hoffman

Inno.com cva

Belgium

INTEC

Ghent University

Belgium

koenraad.vandenborre@inno.com , muna.matar@intec.rug.ac.be , ghislain.hoffman@rug.ac.be

Abstract

This paper describes a novel approach towards the decoupling of persistence issues from a class library. It first describes the need for persistence and then how persistence issues get fully orthogonalised from the class library by using the Aspect Oriented Paradigm. It is completed by an example, using Java and AspectJ, to illustrate the sketched methodology.

Introduction

In software engineering applied to business systems, there clearly is an evolution from writing proprietary middleware code towards using middleware services. We for instance mention the efforts taken in the J2EE environment concerning persistence, transaction management... This evolution enables developers to separate the writing of business logic from the writing of middleware services.

This evolution in software engineering enforces and gets enforced by an evolution in the business paradigm many organisations nowadays are confronted with. This evolution, as well encountered inside as outside the walls of the organisation, drives organisations to migrate from a silo based towards a service based business.

In silo based environments there often is a culture of data replication resulting in many data sources, containing inconsistent and redundant data, a situation that becomes intolerable in a service based environment.

So, from a software engineering point of view we don't want to bother the developer with persistence issues and from a business point of view, unambiguous persistence is a major requirement. These considerations lead to the conclusion that from both the engineering and the business point of view there's a rationale to look for an effective persistence model.

Scope

In software engineering persisting an entity means extending its lifetime beyond the lifetime of the application that created it, so that the entity can be used later on in the same application, or in other applications. In achieving this goal software engineers are confronted with a myriad of challenges:

- The entity can be saved in a relational database, stored in an XML repository, put in a spreadsheet or it can be decided not to store the entity by itself but instead recalculate it from other persisted entities.
- Furthermore, the functionality to deal with the persisted entities - select, update, create, delete – can be written using 4GL, stored procedures, JDBC, entity EJB's, dedicated data access objects behind a Session

Façade, through a proprietary API of an EIS....

- How the entity should be stored and retrieved from its persistence medium can be influenced by the fact whether it's used in batch or on-line processing
- It could be necessary to replicate a data source if using the original data source would impact the performance of the systems already running on the original data source...
- On top of that, the entities we use in applications and store in data sources are merely models of real life entities. It could turn out that these entities must be remodelled after a certain period of time resulting in different versions
- Many classes in an OO-model need persistence, which results in scattering of the persistence code through the class library. This decreases the maintainability of the code and the reusability of the classes as they contain persistence related code that is not necessarily needed or wanted in another system or business domain.

This paper doesn't pretend to solve all persistence issues but sketches a methodology to decouple persistence related issues from Java classes.

Much work has been done to address the persistence problem, we mention for instance [2], [3], [8]. As far as we know however, all those approaches were restricted to one programming language and/or lost much of their intrinsic value due to the restrictions of the used programming language and the adhered paradigm. Up to the emergence of the aspect-oriented paradigm, there hasn't been a clean, language-independent meta-description of the problem.

The aspect-oriented paradigm on the other hand finds its reason of existence in capturing issues that crosscut a certain class library. This leads to our statement that persistence is an issue that can be captured and described using the aspect-oriented paradigm. Furthermore we state that using the aspect-oriented paradigm, persistence can be fully orthogonalised from a class system or business model. In doing so we introduce a novel approach towards the problem.

In [8], a methodology is developed to build a framework that has the ability, through combination of introspection and the use of JavaDoc tags – to overcome Java's lack of declarativity for persistence -, to build and maintain knowledge how to make objects of certain classes persistent. Therefore, it would be a realistic approach to restrict our selves to describe how to prepare classes for persistence. However, for the sake of proof and simplicity, the example implementation in this paper doesn't use the framework but instead gives a very rough, per class persistence implementation.

The rest of the paper focuses on how to achieve the orthogonalisation between persistence and a class as an abstraction of a real life entity. To prove our statements a programming language has to be chosen. The object-oriented language used is Java, the aspect oriented one aspectJ, developed at Parc Xerox.

First the general methodology, starting from a business model, is sketched and thereafter it is applied to a simple example.

Designing the business model

Suppose we have to model a simple invoicing system. As a first step the problem domain must be analysed and a business model must be designed. In this business model some classes must be made persistent. At this phase however, we don't want to be bothered with persistence related issues, we just want to design our classes as abstractions of real life entities. At a later stage it will be decided what classes must be made persistent and how this must happen. The business model for the simple invoicing system is the one depicted in Figure 1.

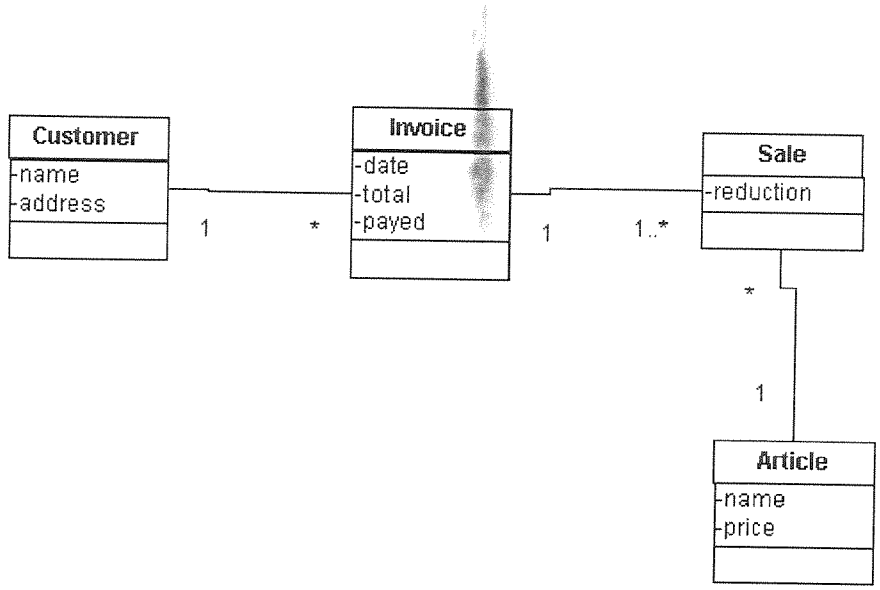

Figure 1: Simple Invoicing System

It is possible to implement this model at this stage and to test it, by providing it with dummy data.

Persistence and the business model

When considering object persistence, two main issues arise. First, every object to persist must have the appropriate functionality to be stored in and retrieved from a data source. Secondly, query functionality is needed. If not, we can save our selves a lot of trouble by just serialising the objects. This query functionality could for instance be: find all customers that have invoices that should have been payed last month. The question arises if this functionality should be part of one of the classes involved in the association. We believe not. This functionality results from the relation between Customer and Invoice for a specific business domain. Another business domain might impose the same relation between Customer and Invoice – a one to many relation - but demand other functionality from it. We believe this functionality should be put in a dedicated object that models the relation between Customer and Invoice. By using this approach, the Customer and Invoice classes become more reusable over different business domains. The needed query functionality for the relation class then becomes just a persistence issue for this class.

During implementation, if only pure Java is used, there exist two major approaches to indicate on the class level that a class must be made persistent. The first is to extend the involved classes from a base class, say PObject, which introduces the needed attributes and methods. Drawbacks in using this method are:

1. Java's only "extend" relationship is consumed for implementation purposes, not for design purposes.
2. Some attributes must be declared protected in order to manipulate them in derived classes, which introduces weaker encapsulation.

Another way around would be to let every class that has to be made persistent, implement an interface, say Persistent. Drawbacks here are:

1. We cannot introduce non-static, non-final attributes in the classes implementing the interface.
2. There is no way of introducing some standard behaviour for the methods of the interface.

Whatever approach is chosen, extending from a base class or implementing an interface, the programmers implementing the class involved are concerned with the class as an abstraction and with persistence aspects, be it overriding methods from the ancestor class PObject or implementing methods from the interface Persistent. Furthermore, persistence issues are spread over all the classes that are to be made persistent, which causes them to ripple throughout the entire code, a maintenance nightmare.

The key problem is – as stated above - that persistence crosscuts the entire object model in a way that cannot be cleanly expressed by just using the Java implementation of the OO-paradigm.

Introducing persistence aspects

The aspectJ language provides us with two techniques to capture crosscutting concerns. The first one - known as introduction – provides the ability to introduce attributes, methods and constructors in existing classes. The second one - known as advice - provides the ability to execute extra code at certain points in time defined by what in aspectJ is called pointcuts. It is the combination of both which will allow full orthogonalisation.

Using introduction one can introduce in existing classes the extra features needed to obtain a persistent class. The two items to decide on are: what and where to introduce.

To solve the question where to introduce, we can make every class that has to be made persistent implement an empty interface Persistent, very much like the standard Java interface Serializable. This makes that the class can be considered being of the type Persistent.

What to introduce can for instance be an attribute objectIdentifier and the methods read(), write(), update() and delete() - to read, write, update or delete an object from or to a persistence medium. The introduced methods can contain implementation code. This however is not such a good idea because for every class whose objects must be made persistent, one must clearly define how to make them persistent. The writing of a generic write() and read() would, if not impossible, at least cause lots of trouble. We can, however, introduce the methods read() and write() as being (nearly) empty, solving the "what" question, and consider the second technique provided by aspectJ.

Taking into account the second technique - advice - it's possible to execute extra code at certain points in time. A set of points in time could for instance be the invocation of a method write(). Whenever a write() is called on an object which instantiates a class which implements the interface Persistent, extra code is executed. The aspectJ language provides us with the possibility to know, when a write() is executed and from which object it originates, allowing us to react appropriately.

The application of these novel techniques will be illustrated in the following paragraphs.

An example

Let's reconsider the UML diagram of Figure 1. The business model has been constructed, the classes have been developed using pure Java, and at this point it is decided that the classes Customer and Invoice must be made persistent. To that extent we construct the aspect PersistentIntroductor that will introduce the necessary persistence related attributes and methods that are common to both classes. Therefore, to be able to treat the classes as being of the same type Persistent, the aspect declares that each class should implement the empty interface Persistent, which is also written at that point, or reused. The aspect further introduces

- An object identifier
- A method remaining private to the aspect to retrieve the object identifier. This restricts calls to this method, and thus knowledge about persistence, to the aspect. It's not a part of the interface of the objects that will be used in the applications
- Generic methods to write, update and delete persistent objects, each returning Booleans to indicate if the operation succeeded
- A generic method read() returning a Vector, because a read operation can return multiple objects, customers not necessarily have unique names

```
public aspect PersistentIntroductor
{
  declare parents : Invoice implements Persistent;
  declare parents : Customer implements Persistent;

  private Long Persistent.oID = new Long(Math.round(Math.random() * 1000000));

  private Long Persistent.getOID()
  {return oID;}
  public Boolean Persistent.write(Persistent p)
```

```
    {return new Boolean(false);}
    public Vector Persistent.read(Long i)
    {return new Vector();}

    public Boolean Persistent.update(Long i)
    {return new Boolean(false);}
    public Boolean Persistent.delete(Long i)
    {return new Boolean(false);}
}
```

At this stage, however we have only introduced rather generic methods that don't execute persistence code. To address this problem, we construct other aspects. For brevity, this is illustrated for the aspect PInvoice working on the Invoice class. This aspect,

- Is privileged to be able to access the getOID() method, and to have access to the private attributes of the object, not having getXXX() methods, in order to write them to the database.
- Defines the pointcuts, the object involved must be of type Invoice and are exposed in the context
- Is responsible for the database connection
- On each pointcut there's an advice after returning from the methods denoted in the pointcut. This advice has access to the return value of the original method

```
public privileged aspect PInvoice
{
  pointcut reader(Invoice p) : target(p) && call(public Vector read(..));
  pointcut writer(Invoice p) : target(p) && call(public Boolean write(..));
  pointcut updater(Invoice p) : target(p) && call(public Boolean update(..));
  pointcut deleter(Invoice p) : target(p) && call(public Boolean delete(..));
  private Connection con = null;
  private void setConnection()
  {/*connects to database*/}
  after(Invoice p) returning (Vector v) : reader(p)
  {
       /*retrieves the argument of the method the advice is advicing on, gets a
       connection to the database, builds a PreparedStatement to read the invoice from
       the invoice table and the associated customer from the customer table, builds an
       Invoice Object and puts this object in the vector v returned by the original
       method being the subject of this advice*/
  }
  after(Invoice p) returning (Boolean success) : writer(p)
  {
       /*gets a connection to the database, builds a PreparedStatement to write the
       invoice object to the appropriate tables and returns true on success. This return
       value becomes the return value of the original method being the subject of this
       advice*/
  }
  /*analogous after advices for updater and deleter*/
}
```

Conclusions

1. All persistence issues - attributes and methods - can be removed from the business classes to a separate place: an aspect. In the coding example this was done for the attribute oID and the methods getOID(), read(), write(), update() and delete().
2. Business classes are better suited for reuse, because they're closer to just being a design abstraction, uncluttered with persistence issues. Furthermore the business classes become independent of the data source used because all persistence related code resides in the aspects.
3. The business model can be tested to a certain extent before any persistence feature is introduced. Even if it's not yet decided what persistence mechanism will be used, the business model can be implemented.
4. Adjusting the business model to make extra classes persistent takes an aspect source code operation and a recompilation.
5. The persistence interface of persistent classes is very simple and very logical to application programmers making use of the business model. They just have to call the read(), write()... methods.
6. The Java Virtual Machine doesn't need to be changed, because the aspectJ compiler ajc generates intermediate java code that gets compiled with the regular java compiler.
7. It can be argued that a technique like introduction breaks encapsulation. We do insert new methods and attributes in existing classes. However this is done in a clean and controllable way and stays at the level of implementation of the business model, not at the level of applications built on top of the business model.

Future work

In this paper we sketched a novel approach to decouple persistence related issues from a business model. The coding effort is rough and ad hoc. We certainly want to investigate the possibility of merging this approach with the framework from [8]. This would allow separation of the preparing of classes to be made persistent through AOP, from the actual repository that contains and maintains knowledge of how to make classes persistent.

Persistence is not the only middleware service. Transaction management and security for instance are also candidates for "aspectisation". We guess there's a lot of work to do in not only writing those aspects but certainly in the co-operation of different aspects.

References

[1] AspectJ™ , Xerox Corporation, Palo Alto : http://aspectj.org

[2] Peter M. Heinckiens : Building Scalable Database Applications, Object Oriented Design, Architectures and Implementations, The Addison Wesley Object Technology Series, 1998

[3] Atkinson : The Pjama project, University of Glasgow, Department of Computing Science : http://www.dcs.gla.ac.uk/pjava

[4] Scott W Ambler : Mapping objects to relational databases http://www.ambysoft.com/mappingObjects.pdf

[5] H.Ossher and P.Tarr : Multidimensional separation of concerns and the Hyperspace Approach, IBM T.J. Watson Research Center : http://www.research.ibm.com/hyperspace/Papers/sac2000.pdf

[6] http://www.research.ibm.com/journals/sj/361/srinivasan.html

[7] JavaBlend 2.0 tutorial from Sun Microsystems

[8] Muna Matar : A methodology for object persistence in Java based on a declarative strategy, PhD thesis Ghent University, faculty of applied sciences, Department of Information Technology, 2001