Storage Framework Based Information Modeling

Ghislain Hoffman, Muna Matar, Herman Tromp, Member ACM and IEEE Computer Society, and Koenraad Vandenborre

Abstract— **A strategy and a storage framework are presented, which allow to isolate business modelling issues from the underlying persistency infrastructure The main objective is to support a clean and maintainable application architecture.**

The more general problem of mapping an information model onto e.g. a relational database is addressed. It is shown that the expressive power of existing object-oriented languages is insufficient to bridge the semantic gap between the representation of information in a business model and the way it is persisted.

Two possible approaches are presented. One is based on the techniques of aspect oriented software development, while the other uses a more lightweight approach. In the latter, a declarative tagging language is used, which allows specifying the persistency characteristics of business object in a transparent way. That description is extracted into an XML format, and then used in run-time storage framework, which adds persistent behaviour to the business objects.

.

I. Introduction

A storage framework must provide more than just mechanically persisting the apparent state of objects. An information model consists of classes, representing domain relevant abstractions, as well as a description of the inter-object collaborations, in terms of composition, associations, etc. When mapping an information model to e.g. a relational database, essential information is lost. This makes it difficult to recreate, rather than reinstate, objects from persistent storage.

There is also a semantic gap between information models and what can be expressed with typical OO languages, such as Java, especially when the persistent characteristics of objects must be included. Mechanisms such as introspection or serialisation do not offer sufficient expressive power.

Manuscript received June 30, 2003.

G. Hoffman is with the Department of Information Technology, Ghent University, (e-mail : Ghislain.Hoffman@Ugent.be).

M. Matar was with the Institute for Permanent Education, Ghent University, Belgium, and is now with Bethlehem University, Palestine Territories (e-mail : mmatar@bethlehem.edu).

H. Tromp is with the Department of Information Technology, Ghent University (corresponding author : phone +32 9 264 3322, fax +32 9 264 3593, e-mail : herman.tromp@Ugent.be).

Koenraad Vandenborre is with Inno.com cva, Belgium (e-mail : koenraad.vandenborre@inno.com)

II. Application Architecure and Persistency Strategy

A strategy for isolating business modelling issues from the persistency layer must be defined, allowing optimal reuse of domain abstractions and supporting a clean application architecture.

A multi-tier architecture separates the business logic, which

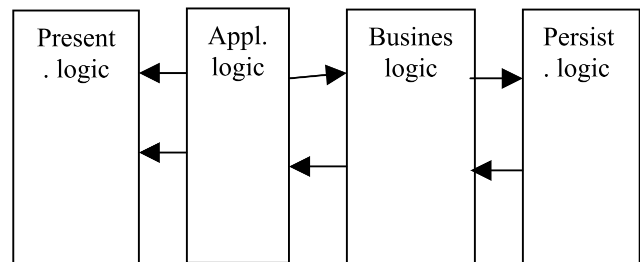

Fig.1. A multi-tier architecture

is common to all applications in an organization, from individual application logic (see fig.1). The latter is typically implemented in the user interface tier, although a separate presentation tier is preferably used.

Anyhow, the business logic is encapsulated in separate classes that represent the business objects. This form of encapsulation ensures that all applications use the same business logic in a coherent way. More details are given by Vandenborre et al. [8].

In order to cleanly design business objects, while safeguarding the separation of business logic from implementation issues, they should not contain anything that indicates how they are stored or retrieved from the underlying persistent storage mechanism. There is thus a clear need for a well-defined separation of concerns between the business logic tier and the persistency tier. This evolution is clearly present in more recent software methodologies. It enables software developers to separate the writing of business logic from the use of middleware

services. The J2EE environment, in view of transaction management e.g., offers a prominent example of the evolution from the development of proprietary middleware code towards using standard middleware services.

The above implies that

- From a software engineering point of view, the developer should not be bothered with persistency issues
- From a business point of view, unambiguous persistency is a major requirement

For these reasons, there is a rationale to look for an effective persistency model.

Persisting an entity means extending its lifetime beyond the lifetime of the application that created it. In achieving this goal the developer is confronted with a number of challenges:

- The entity can be saved in a relational database (the case that will be considered in the remainder of this paper), but might as well be stored in an XML repository, put in a spreadsheet, or not be saved at all but recalculated from other persisted entities;
- The functionality to deal with persisted entities (typically *select*, *create*, *update*, *delete*) can be expressed using a variety of mechanisms, such as a 4GL, stored procedures, JDBS, entity beans, dedicated data access objects, a proprietary API to an EIS, etc.;
- The issue whether an entity is used in a batch or on-line processing mode may well influence its persistency characteristics;
- Duplication of a data source may be needed for performance reasons;
- The objects as implemented in a business model are merely models of real life objects, and may have to evolve. This raises a *versioning* problem;
- Since many classes in an object oriented domain model need persistence, at different levels of granularity, the persistency-related code gets scattered throughout the code base. This is clearly a crosscutting concern, as explained in section V, and if not dealt with properly, severely decreases the maintainability and reusability of the code.

As an example, consider the model of a simple invoicing system, as shown in fig.2 [6]. In this business model, some classes will have to be persisted. At this phase in the design and development process, however, we only want to design classes and abstractions that represent reaf-life entities. It is only at a later stage that it will be decided which classes should be made persistent and how this is done. The important issue is that separation of concerns requires that persistency issues and domain abstractions are designed and implemented in a way that is as orthogonal and independent as possible.

While the architectural principles as outlined in the previous paragraphs are crucial for the development of new applications, they are even more essential in an effort to revitalise and migrate legacy systems. This is discussed in more detail in e.g. [1] [2]. The architectural problems encountered in legacy migration are quite similar to those in new system development, but the obstacles to their deployment are even more disturbing. In legacy systems, user interface code, business logic and the persistency layer are mostly always intermingled, which leads to a maintenance nightmare.

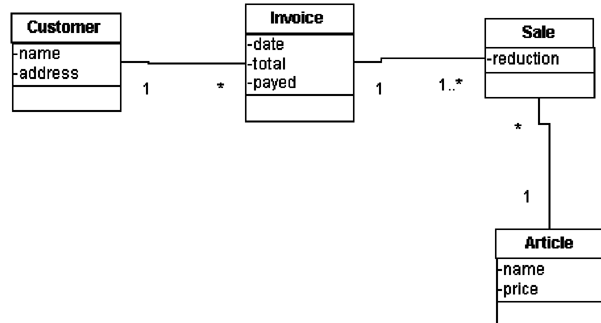

Fig. 2. A business model of a simple invoicing system.

III. Object-Relational mapping

The problem of mapping an object oriented domain model to the tabular structure of a relational database was already addressed by Keller et al. [11]. Essentially, this mapping requires the reconciliation between two widely different information modeling paradigms.

Heinckiens [4] provided a more advanced approach and Fowler [3] summarised the state-of-the-art in terms of patterns, which stand for best practices in current software engineering.

All those approaches, however, tend to have a rather technological and implementation oriented view. Fowler ([3], p. 37) states that 1/3 of overall application code, and hence maintenance effort, goes into object-relational mapping. This statement implies a serious problem, in view of maintainability and reusability.

If we really want to map an object oriented business model onto a relational structure, more is needed than a pure object to relational table mapping.

Links between objects are mostly an instance of associations between classes, which often lead to expensive join operations between tables, through foreign keys, operations which are not always needed.

The major problem with the more established object oriented programming languages, such as C++ and Java, is that they allow to express associations between entities at

the structural level, but do not allow to express the semantics of those associations, even less the behavioural aspects.

Two main issues arise when implementing object persistency. First, every object to be persisted must have the appropriate functionality to be stored in and retrieved from a data source. Moreover, query functionality is needed. If this were not the case, simple object serialisation might suffice. Take for example a query (referring to fig.2) that must find all *Customers* who have *Invoices* that should have been paid last month. It is best not to have this query functionality embedded in one of the classes, involved in the query. The functionality results from the relationship between *Customer* and *Invoice* as defined in a specific business domain. Another business domain might impose the same relationship, but implying different semantics, and thus different query functionality. This problem could be resolved by putting the query functionality in a dedicated class that models the relationship between *Customer* and *Invoice*, but even that solution has an ad-hoc flavour.

To conclude this section, it is clear that more pervasive mechanisms are required to express persistency issues in a way that lets the business objects and the entities they model undisturbed, in other words to bridge the semantic gap between the domain model and its persistent representation.

IV. Expressing persistency in object-Oriented Languages

To bridge the semantic gap that exists in describing the persistent properties of information and to do this while achieving optimal separation of concerns, any OO language such as Java must be extended. Using aspect oriented programming is one way to do this. In this paper we also present an alternative, lightweight approach, using standard techniques: Javadoc style tagging expressed as a formalised declarative language, an XML description and run-time framework interpreting it. This approach offers a lightweight, declarative, extensible and non-intrusive mechanism to introduce persistency in existing Java classes. Both approaches will be described in subsequent sections (see section V).

Some object-oriented languages, such as C++, do not offer any support for object persistence, except through the use of dedicated and proprietary libraries. In the remainder of this paper we will concentrate on the use of Java.

Even in Java, support for persistency is limited. This was explored by Matar, Vandenborre et al. [5] [6] [7]. Inheritance may be used to introduce persistency in the business classes, and some forms of introspection are available to explore the attributes of classes. This does not solve however, the general problem, in view of class associations, as it was explained in section III. A more powerful mechanism, that is non-intrusive in view of the business class design, is definitely needed.

V. Approaches to implement Persistency

A. *Aspect Oriented Approach*

1) *Overview*

Aspect Oriented Software Design (AOSD) is a new software development paradigm which allows to design and/or implement crosscutting concerns on an object model. A general discussion of AOSD is given by Kiczales et al. [12], and also in [18] [21].

While AOSD is mainly a software design paradigm, its implementations are usually based upon a base object-oriented language, such as Java. The most prominent example is AspectJ [19] [20] [22].

The AspectJ language offers two techniques to capture crosscutting concerns:

- *Introduction*, which provides the ability to introduce attributes, methods and constructors in existing classes
- *Advice*, which provides a way to insert code at certain execution points (typically method calls).

Important to note is that both are done in a non-intrusive way, i.e. without disrupting the original class design, nor its implementation.

As was explained above, persistency is clearly a crosscutting concern since it pervades a large number of the classes (but not all off them) in the business model. The use of AOSD can alleviate this pervasive behaviour. In AspectJ terms, the use of introduction can introduce in existing classes the extra features to obtain a persistent class. This is achieved by having every class, to be made persistent, to implement the empty interface *Persistent*, which results in adopting that class the type *Persistent.*

In persistent classes, we have to introduce:

- A unique object identifier and methods to retrieve it. These methods should be private, in order to restrict external knowledge about the persistency characteristics of the class.
- Generic methods to write, update and delete persistent objects. These methods return a *Boolean* to indicate success or failure.
- A generic method to read multiple objects, returning typically a *Vector*.

The methods introduced are empty. The *advice* mechanism of AspectJ is then used to have extra code executed whenever e.g. a *write()* method is called on an object that implements the interface *Persistent*. AspectJ provides the possibility to know when the method is executed and from which object it originates, allowing reacting appropriately.

2) Example

As an example, take the UML diagram shown in fig.2. Assume that the business classes have been designed without any persistency requirements in mind and that it is now decided to make the classes *Customer* and *Invoice* persistent. To this effect the aspect *PersistentIntroductor* is introduced in both classes. Its purpose is to introduce persistence related attributes and methods needed in both classes. The AspectJ code is shown below.

```
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 point only generic methods are introduced, that do not execute actual persistency code. This is to be defined in other aspects. This process is illustrated by the *Pinvoice* aspect for the *Invoice* class. This aspect

- Is privileged to access the object identifier attribute
- Defines the pointcuts (the “interaction points”)
- Is responsible for the database connection
- Defines an *after advice* after the methods defined in the pointcuts, which has access to the return value of the original method.

The skeleton AspectJ code is shown below.

```
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 advising 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*/
}
```

3) Conclusion on the AOSD approach

All persistency issues are removed from the business classes to separate aspects, which makes the business classes far more suitable for reuse : they represent only design abstractions, uncluttered by persistency issues and independent from the type of data source being used. Adjusting the business model only takes a review of the aspect code and a recompilation. This implies also that the business model can be tested before the persistency features are introduced.

It could be argued that a technique like introduction breaks the principle of encapsulation: New methods and attributes are inserted in existing classes. This is done, however, in a clean and controllable way and is only visible at the level of the implementation of the business model, and not at the level of the application tier built on top of it.

B. Lightweight Approach

Instead of augmenting an existing programming language such as Java, it could be useful to explore how standard language constructs, API’s and tools can be used to express persistency concerns. This topic was explored by Matar et al. [5] [7] and led to a lightweight approach for the problems explained in previous sections. The following subsections will summarise those findings.

1) *A declarative Language*

Exploring the capabilities of Java to introduce persistency, it appears at first sight that *introspection* offers the ability to obtain information about classes and objects at run-time. Unfortunately, it allows only gathering information about class structure and attribute values. Meta-information about classes, which is related to persistency, cannot be extracted from the mere class code. Therefore a declarative tagging language was developed to identify and declare persistency related information, the Persistency Definition Language (PDL). It is based on Javadoc tags, and together with introspection it provides a complete description of persistent classes. It should be noted, however, that an extra requirement is that all persistent classes inherit from a predefined base class, *Pobject,* which defines some functionality required to have persistency function in the context of the framework discussed below. This consumes

the single inheritance relationship available in Java and might be a hindrance to class design. Ways to overcome that problem are discussed in [5].

The combination of the techniques mentioned above makes it possible to store and retrieve objects defined in Java to and from a relational database. Except for the required inheritance from a single base class *Pobject*, the original Java code of the business classes remains unmodified from a non-persistent to a persistent version of the class.

As mentioned before, PDL is a Javadoc tag based declarative language. The tags, which are needed, were identified by looking for persistency aspects not covered by standard Java features. The tags introduced can be classified as follows:

- Versioning tags, defining class versions. They are *@major* and *@minor*, to identify major and minor versions of a class, respectively. A versioning system is inherent to PDL and the encompassing persistency framework.
- Mapping tags, which help with the mapping between classes and their attributes to/from database tables. They help to span the object-relational mapping gap. They are *@persistent* (to tag attributes that are to be mapped to the database), *@database* and *@table* to identify the database and table, respectively, to be mapped to.
- Retrieval tags used in queries. There is one tag in this category, *@accessor*. It identifies attributes which can be used to access objects in the database.
- Internal state tags : *@state* (to tag attributes which are essential to define the state of the object), *@size* (to specify the size of a string-like attribute, especially of attributes of class *ByteField*, which can be mapped directly to columns in a database), *@contained* (to tag attributes which have a composition relationship with the containing class, rather than a being a reference which is by default the case in Java), *@compType* (to identify a Java homogeneous collection component. They define the internal state of objects as far as this is relevant for its persistent characteristics
- Table design tags, helping in designing relational table columns : *@unique* and *@index*, which assist in defining the database index structure.

A simple example of the use of PDL tags is shown below.

```
package MyApplication;
    /**
     * @database "Company"
     * @table "Employee"
     * @major 01
     * @minor 00
     */
public class Employee extends Pobject {
      public  static ClassVersion classVersion
        = new ClassVersion("01","00");
    /**
     * @persistent
     * @accessor
     * @index
     * @unique
     */
    private Name empName;

    /**
     * @persistent
     * @contained
     */
    private Address address;

    /**
     * @persistent
     * @accessor
     * @index
     * @size 10
     */
    private ByteField jobTitle;

    //constructor
    public Employee(){
    }

//other constructors and methods go here
```

2) Generation of an XML description of persistent classes

The Java source code is parsed to extract the PDL tags and, by using a Java doclet, to produce an XML file to represent the persistent behaviour of the classes. This XML description will be used by the run-time framework discussed below. This PDL processor helps to generate SQL code that is used by the framework for purposes such as creating database tables, storing and retrieving objects to/from those tables, …

The XML description extracted from the code sniplet given in the previous subsection is shown below.

```
<?xml version='1.0'?>
<!DOCTYPE ClassLibrary >
 <classDescriptor Class="MyApplication.Employee">

   <classVersion major="01" minor="00">
   </classVersion>

   <db database="Company " table="Employee">
   </db>

 <pAttribute accessor="true" index="true"
      unique="true" contained="false">
      <attributeOfClass>
        MyApplication.Employee
      </attributeOfClass>
      <attributeName>
        empName
      </attributeName>
 </pAttribute>

 <pAttribute accessor="false" index="false"
      unique="false" contained="true">
      <attributeOfClass>
        MyApplication.Employee
      </attributeOfClass>
      <attributeName>
        address
      </attributeName>
 </pAttribute>
```

```
 <pAttribute accessor="true" index="true"
      unique="false" contained="false">
      <attributeOfClass>
        MyApplication.Employee
      </attributeOfClass>
      <attributeName>
        jobTitle
      </attributeName>
      <size size="10">
      </size>
 </pAttribute>

</classDescriptor>
```

The XML description is one of the two stages any persistent class needs to go through, in order to be able to be registered with the framework discussed in the next subsection.

3) Run-time persistency framework

The persistency definition language framework (PDLF) is an object-relational mapping that enables developers to easily persist Java objects to relational databases. Classes that have to be persisted must be registered with the framework. This implies that both the byte code (the .class file) and the XML file generated, as mentioned in the previous subsection, must be provided. Once registered, with the framework, instances of the class can use the methods available in it to perform persistency related operations. The versions of the class and their XML descriptor must be identical, however, or a run-time exception will be raised by the framework. It is the responsibility of a "database administrator to register a class to be persisted with the framework. This technique supports a clear separation of concerns.

4) Conclusion on the lightweight approach

The main features of the lightweight approach to persistency are :

- A purely declarative framework based on standard Java and PDL
- A total separation of concerns is implemented.
- It helps developers to concentrate on business logic, without having to be concerned with persistency issues.
- The meta-data mappings separate the details of the storage mechanisms from the business logic. Database schemes are automatically generated.
- The mapping strategy is saved to a central repository that is used by the persistence layer at run-time. The persistence layer provides an API to allow business objects to be persisted and queried.
- Developers do not have to write SQL statements to read and write Java objects; SQL is generated automatically.

The main point, however, is that it has been proven to be possible to express persistency using standard Java techniques and related tools.

C. *Comparison of the AOSD and the lightweight approach*

Both approaches (those of sections V.A and V.B) have been proven to be valuable and feasible. While the AOSD approach requires the extension of existing programming languages, implying a steeper learning curve, the lightweight approach exposes the limitations of those same languages. Especially the requirement to inherit from a common persistent base class imposes severe limitations to the design of business classes.

A major conclusion might be that the lightweight approach offers an interesting stepping stone to tackle the major problem of persistency issues from domain modelling, but that the AOSD approach offers a more durable solution, especially since these techniques are rapidly becoming mainstream in software development.

VI. Related Work

The application of the principles of AOSD to persistency was also discussed recently by Rashid et al. [14] and Soares et al. [15]. They come to similar solutions and/or conclusions as discussed in section V.A.

The importance of persistency issues in legacy migration was already stressed at the end of section II. Other authors have dealt with the subject before, such as Henrard et al. [9], Plakosh et al. [10] . Best practices in reengineering of object-oriented systems are described extensively by Ducasse et al. [13].

The examples in this paper are based on Java as a base language. Similar efforts have been reported, based on Ada by Crawley et al. [17], as well as on C++ and Modula-3 by Hosking et al. [16].

VII. Future Work

Currently, work is going on to merge the two approaches mentioned in section V (A and B). The simplicity of the declarative technique described in section V.B will be conserved, and the aspect code will be generated automatically from it. Whether this is best done directly from the Javadoc tagging or from the intermediate XML description has still to examined.

Persistency is also not the only service to worry about. Transaction management and security are also services, which could be expressed as aspects. The relationship with the services offered in the context of a J2EE container needs further investigation. This is also the case for the co-operation between (unrelated) aspects.

REFERENCES

[1] H. Tromp, G. Hoffman, "Evolution of legacy systems: strategic and technological issues, based on a case study". *International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*, Amsterdam, Sep. 2003.

[2] I. Michiels, D. Deridder, H. Tromp, A. Zaidman, "Identifying ICT problems in legacy software: preliminary findings of the ARRIBA project". *International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*, Amsterdam, Sep. 2003.

[3] M. Fowler, *Patterns of Enterprise Application Integration.* Addison-Wesley Signature Series, 2002.

[4] P. M. Heinckiens, *Building Scalable Database Applications.* Addison-Wesley Object Technology Series, 1998.

[5] M. Matar, *A methodology for object persistence in Java based on a declarative strategy*. PhD thesis, Ghent University, Department of Information Technology, 2001.

[6] K. Vandenborre, M. Matar, G. Hoffman, "Orthogonal persistence using aspect oriented programming", in *Workshop on Aspects, Component, and Patterns for Infrastructure Software, Enschede, NL, Apr. 2002*

[7] M. Matar, K. Vandenborre, G. Hoffman, H. Tromp, "A declarative persistency definition language", in *ASE 2002 Workshop on Declarative Meta Programming to support software development*. Edinburgh, UK, Sep. 2002.

[8] K. Vandenborre, P. Heinckiens, G. Hoffman, H. Tromp, "Coherent Enterprise Modelling in Practice". *13th European-Japanese Conference in Information Modelling and Knowledge Bases,* Kitakyushu, Japan, June 2003.

[9] J. Henrard, J-M. Hick, P. Thiran, J-L. Hainaut, "Strategies for data reengineering". *Working Conference on Reengineering,* IEEE Computer Society Press, pp. 211-220.

[10] D. Plakosh, S. Commela-Dorda, G.A. Lewis, P.R.H. Place, R.C. Seacord, *Maintaining transactional context: a model problem*. Report CMU/SEI-2001-TR-012, Carnegie Mellon Software Engineering Institute, Aug. 2001

[11] A. Keller, R. Jensen, S. Agarwal, "Persistence software: bridging object-oriented programming and relational databases". ACM *SIGMOD Record,* May 1993. *Proc. 1993 ACM SIGMOD Int. Conf. On Management of Data,* Volume 22 issue 2, pp. 523-528, June 1993.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, "Aspect-Oriented Programming". *Proc. of the European Conference on Object-Oriented Programming, June 1997.*

[13] S. Ducasse, S. Demeyer, O. Nierstrasz, *Object-Oriented Reengineering Patterns,* Morgan Kaufmann and Dpunkt, 2002.

[14] A. Rashid, R. Chitchyan, "Persistence as an aspect", *Proc. 2nd Int. Conf. on Aspect-oriented Software Development*, Boston, MA, 2003, ACM Press, pp. 120-129.

[15] S. Soares, E. Laureano, P. Borba, "Implementing distribution and persistence aspects with AspectJ", *Proc. 17th ACM Conf. on Object-oriented programming Systems, Languages and Applications*, Seattle, 2002, pp. 174 - 190, also in *ACM SIGPLAN Notices*, Vol. 37, No. 11 (November 2002)

[16] A. L. Hosking, J. Chen, "Mostly-copying reachability-based orthogonal persistence", *Proc. 1999 ACM SIGPLAN Conf. on Object-Oriented Programming Systems*, Languages and Applications, Denver, 1999, pp. 382 - 398

[17] S. Crawley, M. Oudshoorn, "Orthogonal Persistence in Ada", *Proc. Conf. on TRI-ADA '94*, Baltimore, 1994, pp. 298 – 308

[18] T. Elrad, R. Filman, A. Bader (eds.), "Theme section on Aspect-Oriented Programming", *CACM*, 44(10), 2001

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. A. Kersten, J. Palm, W. G. Griswold, "An Overview of AspectJ", *ECOOP*, 2001, Springer-Verlag, LNCS 2072, pp. 327 – 353.

[20] I. Kiselev, *Aspect-Oriented Programming with AspectJ,* SAMS, 2002

[21] K. Mens, C. Lopes, B. Tekinerdogan, G. Kiczales, "Aspect-Oriented Programming Workshop Report", *ECOOP Workshop Reader*, 1997, Springer-Verlag, LNCS 1357.

[22] J. D. Gradecki, N. Lesiecki, *Mastering AspectJ,* Wiley, Indianapolis, 2003.

Ghislain Hoffman holds the degrees of civil engineering and doctor in applied sciences from the University of Gent in Belgium. He is full professor, affiliated to the Department of Information Technology in the Faculty of Applied Sciences of the University of Gent, where he is teaching and researching software engineering.

His primary concern is the application of the principles of object orientation in analysis and design of large administrative systems, including client/server systems and network and mobile computing, where strategic decisions are an absolute necessity. Business modeling, ICT architecture, legacy recovery, enterprise application integration and reuse of software components have been his primary academic research topics. His current focus is on Aspect Oriented Software Design. He is consulting on ICT strategy for several public administrations and industrial corporations (the Belgian Senate is one of the examples in the public sector). Recent projects involve advanced network technology in the fields of e-commerce en messaging, and major projects in migrating legacy environments to distributed environments.

Prof. Hoffman is a member of the scientific advisory board of Inno.com and of its strategic committee.

Muna Matar holds a PhD degree in Information Technology from the Department of Information Technology at Ghent University, Belgium, obtained in November 2001. Currently she is working at the Institute for Continuing Education (IVPV) at Ghent University after which, in September 2003, she will be joining the Faculty of Science at Bethlehem University in the Palestinian Territories. In 1985 she obtained a M.Sc. degree in Software Engineering from Oregon State University, Oregon, USA, after which she started working at Bethlehem University.

Dr. Matar's interests are in the areas of object orientation, analysis, storage frameworks and persistency. For the past year and a half Dr. Matar was working on setting up an e-learning platform.

Herman Tromp was born in Antwerp, Belgium, in 1949. He holds an Electrical Engineering degree (1972), a degree in Telecommunications Engineering (1973) and a Ph.D. in Engineering (1978), all from Ghent University, Belgium. Since 1972 he is with the Department of Information Technology, Ghent University, Belgium, on leave with McMaster University, Canada in 1974-1975 and the Belgian Military Academy (Telecom Department) in 1975-1976. In 1998-2002 he also was an independent senior IT-consultant on software architecture and EAI (Enterprise Application Integration).

The main interests of Prof. Tromp are currently in research on architectural resources for the revitalization and integration of legacy IT systems, and on software development methodology in general

Koenraad Vandenborre holds an Electrical Engineering degree (1992) obtained at Ghent University. From 1998 until 2001 he was affiliated with the Department of Information Technology, Ghent University, Belgium as a Ph.D. student. In 2001 he became an IT-consultant within Inno.com, a Belgian consulting firm delivering IT architecture and IT strategy services.

His main interests are in software development methodologies with a current focus on researching the applicability of the Aspect Oriented Paradigm in large-scale organizations.