EnviroInfo 2008 (Lüneburg)
Environmental Informatics and Industrial Ecology
Copyright © Shaker Verlag, Aachen 2008. ISBN: 978-3-8322-7313-2

A Plugin-Based Framework for Domain Models and Persistence in Environmental Management Information Systems

Thorsten Busse[1], Nicolas Denz[2] and Bernd Page[2]

Abstract

In this paper, we present an open source framework that supports the implementation of persistable domain models in dynamic plugin-based architectures on Microsoft .NET. The framework includes a domain model service that allows plugins to provide new domain types to other plugins as well as to use, extend, and observe domain objects contributed by other plugins. Furthermore, a persistence service is provided as an abstraction from concrete repository implementations. The paper discusses our solutions to the challenges of persistence in dynamic architectures, presents an exemplary domain-specific language for material flow analysis, and compares our work to related approaches from the common software platforms .NET and Eclipse. Though the framework is in principle application-independent, we apply it primarily in the context of environmental management information systems.

1. Introduction

Plugin-based software development can enhance the modularity, flexibility, and extensibility of software [1]. The importance of plugin architectures is reflected in the success of the Eclipse Rich Client Platform (RCP) [2], an open source development framework for interactive Java applications [1].

Plugin-based approaches also suit the domain of environmental management information systems (EMIS) where a demand for flexible and customizable solutions exists [3]. Tools in this field are often based on Microsoft Windows with its development platform .NET [4]. Examples include the commercial material flow analysis system Umberto [5] and the scientific material flow simulator Milan [6].

In this paper we present results of the project PLUGIN.NET, a cooperation of the University of Hamburg and the Hamburg-based company ifu [7], which is also related to the project EMPORER at the FHTW, University of Applied Sciences Berlin. The objective is to develop parts of an open source plugin framework for Microsoft .NET. The effort is based on a core framework that provides a runtime platform for plugins as well as an Eclipse-like workbench metaphor for user interfaces (UIs).

Within this context, our project contributes a domain-specific language for material flow analysis utilizing code generation for different aspects of domain models (e.g. programmatic interfaces and UI parts) and XML specifications. Furthermore, we provide an abstract persistence mechanism for dynamic architectures. The contributions are evaluated by the example of an editor for material flow networks.

The remaining paper is structured as follows: Section 2 presents the underlying plugin framework. In Section 3 we elaborate on the challenge of domain model persistence in dynamic architectures and show an architecture of a persistence service for domain objects provided by plugins that abstracts from concreterepositories. We furthermore present solutions to cope with modifications and extensions of domain models in a dynamic environment. In Section 4 an example from the material flow analysis domain is given to illustrate how our approach can be used and extended by third-party developers. In Section 5 we

[1] ifu Hamburg GmbH, Grosse Bergstraße 219, 22767 Hamburg, Germany
[2] University of Hamburg, Department of Informatics, Vogt-Kölln-Straße 30, 22527 Hamburg, Germany, [busse, denz, page]@ informatik.uni-hamburg.de

compare our work to related efforts like the Eclipse Modelling Framework [8] and Eclipse Link [9]. We conclude the paper in Section 6 and provide an outlook to future topics including support for transactions.

2. A Plugin Framework for the .NET Platform

For the topics discussed in this paper it is helpful to have an idea of their technical fundament, a framework that allows to design and develop dynamic architectures built of multiple distinct software components. The core services of this framework were developed by the project EMPORER [10] at the FHTW, University of Applied Sciences Berlin and ifu [7]. The framework's design was derived from the architecture of the Eclipse RCP, and the target platform is Microsoft .NET [4].

The main intention of the plugin framework (also called platform here) is to provide a technical basis for applications with graphical user interfaces following the workbench metaphor known from Eclipse. The framework provides a set of basic services that allow building an application designed as a set of so-called plugins. Plugins differ from software components [5, p. 45] in that they are supposed to be exchangeable at application runtime and not only during development. In the framework a central plugin registry allows to acquire all necessary runtime information about the currently installed plugins. Since each plugin is defined by its interfaces and the dependencies between interfaces are restricted as far as possible, implementations of plugins are easily exchangeable.

Plugins can both provide extensions to other plugins and be extended at well-defined extension points themselves without detailed knowledge about the extended or extending plugins [3]. This approach limits the extensibility of a plugin to a set of well-defined extension points foreseen by the plugin developer, which helps to make the concept technically more feasible. Extension points are defined by software interfaces [11, p. 37] that an extension must implement. Similar to Eclipse, extensions and extension points are defined, configured, and packaged in so called bundles [11, p. 36]. Dependencies between different bundles are resolved by the platform core during the launch of an application.

Along several other services[1] like error reporting, graphical navigators, interactive selection of objects in the user interface, etc. the plugin framework provides the possibility of code generation at runtime, a feature that is especially helpful in the context of domain models and persistence.

3. Domain Models and Persistence in Dynamic Architectures

Many – if not most – applications need to persist data in databases or other types of repositories. Obviously, this also holds true for applications in the area of EMIS. In applications assembled from (possibly large) sets of plugins it certainly seems promising to keep the different plugins as free from persistence-related code as possible. This leads to the demand for a central persistence service as discussed in this paper.

The technical requirements for a persistence service depend on how the domain model of the application is built. Since the plugin framework follows the OOP paradigm, the domain model is “an object model of the domain that incorporates both behaviour and data” [12, p. 116] for a given application domain. The behaviour of objects in the model can be described as a set of actions that objects can process or be part of and a set of rules that are required to keep the model in a valid state. This domain logic is often referred to as business logic [12, p. 20]. Plugin developers must be provided with means to describe the domain model in terms of data, logic, rules, and persistence.

[1] In this paper we use the term service in the general sense of [12] to denote a service providing software component.

Copyright © Shaker Verlag, Aachen 2008. ISBN: 978-3-8322-7313-2

3.1 Requirements and Challenges for the Persistence Service

The following requirements have been identified for the persistence service to be useful in the dynamic environment of the plugin framework and supportive for plugin developers:

- ease of use
- easy availability for other plugins
- provision and management of persistent objects for other plugins
- abstraction from specific persistence backends like a certain relational database management system (RDBMS)
- encapsulation of existing object-relational mappers (ORM) such as *NHibernate* [13]
- storage of data in single or multiple data repositories depending on the application's requirements
- support for different persistence strategies like continuous persistence of changes or a transaction-based approach
- extensibility and changeability in itself to be able to meet future requirements

During the design of the persistence service we were confronted with several challenges related to the dynamic environment. Most problems result from the fact that the flexible plugin-based approach requires to break up dependencies between different parts of the application and to make these parts reusable and exchangeable. These difficulties would not occur in more monolithic architectures where a simple ORM might suffice.

Changes in the domain model: One challenge is that the development of an application and its plugins over time will lead to changes in the domain logic or data. Support to handle such changes includes the upgrade of existing domain objects, the handling of changes in the data schema, and the enforcement of updated domain rules. The possibility to downgrade to previous versions of a plugin might be useful as well.

Modularization of the domain model: Another problem results from breaking up an application's architecture into distinct plugins. This leads to a split-up domain model, while the need for connections between its different parts remains. Incautious design might yield a model that is indeed split into several plugins but 'feels' monolithic due to numerous interconnections between its distributed parts. These internal dependencies complicate the partial re-use of a domain model, which means that important advantages of the plugin approach are denied.

Dynamic assembly of the domain model: The persistence service should also be able to deal with the dynamic assembly of the domain model during application launch that depends on the set of loaded plugins. As a result the domain model might change between two starts of the same application which must be detected and handled appropriately.

Extensions of the domain model: The persistence service should furthermore support the extension of a part of the domain model defined in one plugin by another plugin. Extension is a common use case in a plugin environment that must be considered on the persistence level as well. Observation of domain objects: A frequent requirement in application architectures is to support reactions to changes of the application's internal state that is to a large portion constituted by the domain model. Due to the persistence service's central position in the plugin architecture, it appears reasonable to augment it with a notification service for this purpose, which might be realized using the well-known Observer pattern [14, p. 293]. A typical usage scenario is the update of user interface elements triggered by changes in the domain model. The central notification service might also be an appropriate place to support 'undo/redo' functionality related to changes in the domain model as well as transactions. Locating transaction support in the proximity of the persistence service allows employing internal transaction handling facilities of the underlying repositories.

Copyright © Shaker Verlag, Aachen 2008. ISBN: 978-3-8322-7313-2

3.2 Architecture

Our architecture to support domain models and persistence is divided into multiple plugin-based services. Figure 1 shows a layered overview of the different components.

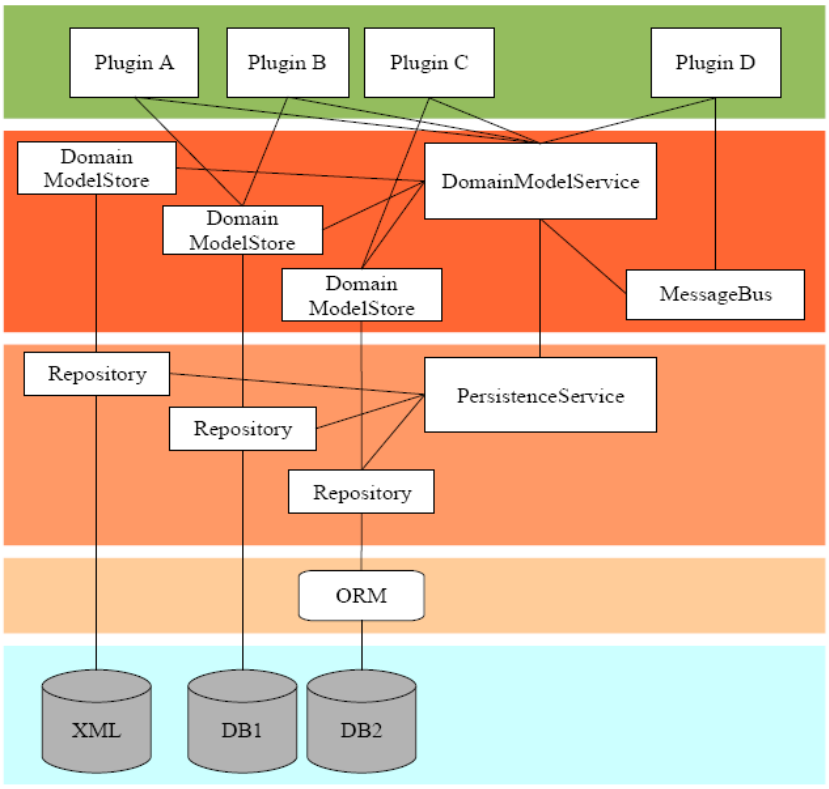

Fig. 1: Layered schema of persistence and domain model service (simplified)

The *DomainModelService* acts as the central access point for all plugins that need to persist data. Its main purpose is to provide access to the list of available *DomainModelStores*, i.e. abstract and generic representations of data repositories like XML files or relational databases. An application may use an arbitrary number of domain model stores.

As each client plugin is bound to an abstract *DomainModelStore*, the underlying repository can be replaced easily with another repository type. The configuration of the stores used by an application can either be fixed or specified via user-defined preferences managed by the plugin platform. The interface of the *DomainModelStore* provides three different types of members:

4. methods to create new instances of domain objects or to provide object factories that may have advanced instance builder methods to initialise properties of domain objects;
5. query methods that provide access to persistent data in the store;
6. mehods that help to persist transient objects and delete persistent objects from the store.

The implementation of the functionality specified in the *DomainModelService* interface is provided by classes that implement the *Repository* interface. This serves as an adapter to provide consistent access to different persistence backends. A repository can of course employ additional layers of indirection, e.g. use an object relational mapper like *NHibernate* to utilize existing solutions and keep the implementation small.

Copyright © Shaker Verlag, Aachen 2008. ISBN: 978-3-8322-7313-2

The set of available Repositories is managed by the *PersistenceService*, which is not intended to be used by external clients. The *PersistenceService* and the different Repositories are merely a persistence - specific layer mirroring the abstract layer of *DomainModelService* and *DomainModelStores*.

The *MessageBus* is assigned to the *DomainModelService* in order to publish changes in the domain model to listening plugins. Its main purpose is to act as a broker between components that raise notifications about changes and components consuming these notifications.

The interface of the *MessageBus* is divided into two parts that reflect these different roles: Domain model-related components like stores, domain objects, and object factories contact the *MessageBus* to announce changes in the domain model. This includes events like the creation and deletion of domain objects as well as changes to an existing domain object's properties.

The *MessageBus* collects the notifications and forwards them to each of the registered listening components. Any plugin can become a listener by registering with the *MessageBus* as an observer [14, p. 293]. Since the number of messages in any non-trivial application can become rather large, the observer can define a set of flexible message filters to reduce the notifications it receives to a subset of the possible event types.

3.3 Extension and Modification of Domain Models

As pointed out in chapter 3.1 one of the crucial points to realize an easily extensible domain model is to avoid too close relations between the different bundles that conjointly define the classes of the model. The key to tackle this problem is a mechanism of the plugin platform called extending properties. Extending properties allow dynamically extending an object with new properties, similar to the Decorator design pattern [14, p. 175]. In doing so, the extended object does not require any knowledge of the additional properties that get attached to it.

Extending properties are realized by registering a helper class for a given type that is able to retrieve the value of the extending property for a given domain object. To break the dependencies between bundles with the help of extending properties it is necessary to organize the bundles in two different layers. This organization is based on the idea that a “higher layer uses services defined by the lower layer, but the lower layer is unaware of the higher layer” [12, p. 17]. The different parts of the domain model are thus partitioned into at least two layers, where the lower layer contains simplified domain types without interconnections. This way all bundles providing domain types located in the lower layer can be separated, reused, and exchanged without consequences for the remaining bundles.

In the higher layer the low-level domain types are 'glued' together in an application-specific fashion with the aid of extending properties. The goal is to provide the main functionality at the lower layer and limit the higher layer to mere interrelations of elements from the lower layer. Depending on the complexity of the domain model the higher layer can be placed in one bundle or separated into several bundles. Even though this layer can be rather specific to a certain application, it is still valuable to make it reusable for related types of applications.

Dividing the domain model into two layers also influences the business rules that are part of the model. The lower layer will typically contain basic constraints on single objects whereas more complex rules describing relations between multiple objects are likely to be part of the higher layer.

Extending properties provide a convenient means to link additions to an existing domain model. Nevertheless it might on occasion become necessary to modify the original domain model and provide a new 'version', e.g. due to changed requirements or the identification of design flaws. With regard to the persistence layer, such changes lead to the typical problem of ensuring 'backward compatibility' in the face of a changed data schema.

One solution to tackle this problem is to gather data about the necessary changes by comparing the old and the new state. However, this solution might lead to overly complex code and unsatisfying results for

Copyright © Shaker Verlag, Aachen 2008. ISBN: 978-3-8322-7313-2

non-trivial changes. We therefore decided to let the plugin developer himself provide details about the changes as a part of the domain model description using the abstract domain-specific language. This abstract description can then be interpreted by the repositories in order to transform the existing data with respect to the underlying backend technology. Concerning domain rules the minimal requirement is to revalidate existing data with the updated set of rules to detect and expose inconsistencies. The rules may be augmented with instructions to fix these inconsistencies on the programming level.

4. Example: A Domain-Specific Language for Material Flow Analysis

As indicated above, the second main objective of the project PLUGIN.NET is the provision of a domainspecific language (DSL) for material flow analysis that is intended to support a standardized communication between plugins from this domain in EMIS. The design and implementation of this DSL integrates aspects from previous object models of the EMIS applications Umberto [5] and Milan [6] and utilizes the domain modelling facilities of the plugin framework. In the following, we will not present the DSL in its entirety, but only show example fragments to illustrate the use of the outlined domain model and persistence services. This might help to understand what is necessary to create a plugin that benefits from these facilities.

```
<!-- Definition of 'Material' bundle -->
<bundle id="platform.material" description="components related to materials"
version="0.1">
   <extension id="platform.material.persistenceconsumer"
      point="platform.persistence.repositoryconsumerpoint">
      <repositoryConsumer>
         <requiredMapping>Platform.Material.dll</requiredMapping>
      </repositoryConsumer>
   </extension>
</bundle>

<!-- Domain object definition of class 'Material' -->
<businessObject>
   <classDefinition classNamespace="Platform.Material" name="Material">
      <property name="Name" domain="MaterialName" translatable="true">
         <interfaceAttribute name="PropertyVisibleAttribute" parameters="true"/>
      </property>
      <property name="BasicUnit" domain="Interface" customType="IUnit" notNull="true">
         <interfaceAttribute name="TypeConverter"
            parameters="typeof(StringConverter)"/>
      </property>
      <property name="Properties" domain="InterfaceCollection" columnName="MATERIALID"
      customItemType="IMaterialProperty"/>
   </classDefinition>
</businessObject>

<!-- Domain object definition for extension of material by project attribute -->
<businessObject>
   <using>Platform.Material</using>
   <classDefinition classNamespace="Platform.Project" name="MaterialToProject">
      <property name="Project" domain="Interface" customType="IProject" />
      <property name="Material" domain="InterfaceCollection" customItemType="IMaterial"
      nameSpace="Platform.Material" columnName="MATERIALID" />
   </classDefinition>
</businessObject>
```

Fig. 2: Simplified excerpts from XML-based declaration of material flow DSL

Copyright © Shaker Verlag, Aachen 2008. ISBN: 978-3-8322-7313-2

Figure 2 shows excerpts from declarations related to the domain type Material of the material flow DSL. In the example all material-related parts of the DSL (e.g. types representing materials, units, etc.) are aggregated into a bundle called *Platform.Material*. This bundle holds a reference to the domain model service (abbreviated DMS in the following) and provides an abstract description of the data it needs to persist. The data is described in an XML-based script (shown in the middle section of Figure 2) which is used by the DMS to create an appropriate implementation with the aid of the platform's code generator. The XML-based bundle declaration (shown in the first section of Figure 2) includes an extension to the extension point *repositoryConsumerPoint* of the DMS plugin. This declaration informs the DMS about the physical .NET assembly that contains the classes of the domain model and their persistence-related mappings.

The following domain object definition of the type *Material* shows that multiple aspects of a domain model can be described in the XML dialect that serves as input to the code generator. The example includes declarations of three properties of *Material* objects: The first property *Name* is a simple string representing the name of the material. Nevertheless a custom domain *MaterialName* is used in order to enable the code generator and the runtime environment to perform a specific validation of values bound to this property. The embedded XML element *interfaceAttribute* is used for UI-related declarations. The example indicates that the material name should be displayed by the property viewer of the platform's workbench.

The following two properties are slightly more complex. The property *BasicUnit* refers to a user-defined type represented by a programmatic interface. The embedded *interfaceAttribute* element indicates that a class *StringConverter* is to be used to display the value of this property in the UI.

The next property Properties describes a list that holds objects of the programmatic type *IMaterialProperty*. The XML attribute *columnName* of this declaration provides persistence-related information. It indicates that an ORM should establish a n:m relation between materials and material properties where the column MATERIALID of the mediating database table refers to the material. During the start-up of the application, the repositories automatically create a mapping between the domain model and a database schema from the information contained in the domain object descriptions. If the application is tailored towards a specific type of repository, it is, however, also possible to circumvent this automated process but include a repository-specific description of the data model in the bundle instead.

The last section of Figure 2 shows how the part of the material flow DSL provided by the bundle *Platform. Material* can be extended by other plugins. The excerpt is taken from a plugin called *Platform.Project* in the example, which purpose is to provide support to manage material flow analysis projects. When both bundles are loaded in common, materials incorporate an additional reference to their parent project. This property is not available if the bundle *Platform.Material* is used in isolation. The declaration indicates that the persistence of this extending property in a relational database is again realized by means of a mediator table mapping materials to projects.

The missing part that prevents the domain model presented so far from being regarded as a full-blown DSL are the domain rules. We are currently developing support to specify and enforce constraints on domain objects. Due to the high development and runtime effort caused by a business rule language and engine, our approach will instead be based on a typical combination of programming by contract and aspect oriented programming (see e.g. [15]) as supported by the .NET open source framework LinFu [16].

In the LinFu framework, contracts (i.e. pre- and post-conditions for method invocations and state invariants of classes [16;17, p. 369]) are specified by code in the primary .NET programming language (e.g. C#) and 'woven' into the processing of method invocations at runtime. In the context of the plugin platform, this provides two advantages [16;18]: Firstly, a plugin developer does not need to learn an own language to specify domain rules. Secondly, the current version 3.0 of the C# programming language provides promising support for the declarative description of rules and constraints including closures, tuples, iteration, and navigation expressions. These are quite similar to the language scope of the well-known ob-

Copyright © Shaker Verlag, Aachen 2008. ISBN: 978-3-8322-7313-2

ject constraint language OCL [18]. For convenience reasons we are nevertheless going to include a code generation of common constraints from the XML-based domain object descriptions as well.

5. Related Work

Due to page restrictions we have to limit the discussion of related work to the most closely related projects from the Java and .NET world. To our knowledge, no plugin-based domain modelling and persistence framework with the whole functional scope of our approach exists for .NET. Spring.NET, an open source application framework [19] with a modular design, provides (among other things) support for transaction management and integration of NHibernate [13], an expression language for querying and manipulation of object graphs, and a validation framework to support business rules. Though these are the basic ingredients for persistent domain models, Spring.NET has no integrated support to describe the domain model of an application as a combination of the data model and the business rules. As a consequence, developers must be aware of the different aspects of persistence in Spring.NET and the chosen backend.

Related attempts to build an Eclipse-inspired plugin framework for .NET are reported in [20] and [1]. [20] mainly tackle the problem to dynamically load and unload plugins in .NET. [1] describe a plugin platform and basic workbench UI quite similar to the core framework presented in Section 2. However, the focus is a lightweight description of extensions and extension points (slots) by means of .NET attributes. The authors present no concepts to integrate domain modelling and persistence facilities needed for the devlopment of individualizable and extensible EMIS as well as a range of other application types.

In the Java world several projects aim to provide support for object persistence. Hibernate [21] and EclipseLink [9] are common implementations of the Java Persistence API [22] and focus on "solving the object-persistence impedance mismatch" [23]. In the context of dynamic plugin-based architectures they do not provide extra benefits themselves. However, in combination with the Eclipse Modeling Framework (EMF) [8] it becomes possible to define domain models using representations like UML class diagrams, XML schemas, or Java interfaces and persist these models in databases [24]. Teneo [25], a subproject of the EMF, also aims at using ORM like Hibernate as persistence backends for domain models. However, both approaches do not seem to address the further challenges discussed in Section 3.3. To our knowledge, EMF neither includes support for changes of the domain model in new bundle versions nor an extension mechanism comparable to extending properties.

6. Conclusions and Outlook

The main focus of the presented work is to provide an integrated approach to define domain models in a dynamic plugin environment including the data model, business logic, and persistence mapping. Instead of developing yet another ORM, we focused on a more abstract framework to integrate existing solutions into a larger context and to provide support for the challenges specific to persistence in a plugin environment. Extending properties are a powerful means to permit flexible and non-monolithic domain models. Defining the domain model with the help of a DSL allows to bring together all aspects of the model in one place. In the near future we will refine the DSL to be more expressive (especially with regard to domain rules) and support a more advanced transaction management for long term operations.

Acknowledgement

The presented work is funded by the Innovationsstiftung Hamburg. The authors would like to thank Stefan Simroth and Tobias Schnackenbeck for their support in the design and implementation of the presented concepts.

Copyright © Shaker Verlag, Aachen 2008. ISBN: 978-3-8322-7313-2

References

[1] R. Wolfinger, D. Ghungana, H. Prähofer, H.P. Mössenböck: A Component Plug-In Architecture for the .NET Platform. Proceedings of the Joint Modular Languages Conference 2006, Oxford, UK, September 2006, LNCS, Volume 4228, pp. 287-305. Springer-Verlag, 2006.

[2] Eclipse Rich Client Platform. http://www.eclipse.org, last visit 02-28-2008.

[3] T. Schnackenbeck, D. Panic, V. Wohlgemuth: Eine offene Anwendungsarchitektur als Fundament eines Methodenbaukastens für betriebliche Umweltinformationssysteme. In: Simulation in den Umwelt und Geowissenschaften, pp. 49-59, 2007, ISSN 1616-0886. [4] Microsoft .NET framework. http://msdn2.microsoft.com/en-us/netframework, last visit 02-29-2008.

[5] ifu Hamburg GmbH.: Umberto web site. http://www.umberto.de, last visit 02-29-2008.

[6] V. Wohlgemuth: Komponentenbasierte Unterstützung von Methoden der Modellbildung und Simulation im Einsatzkontext des betrieblichen Umweltschutzes: Konzeption und prototypische Entwicklung eines Stoffstromsimulators zur Integration einer stoffstromorientierten Perspektive in die auftragsbezogene Simulationssicht. Dissertation at the University of Hamburg, Department of Informatics, Berichte aus der Umweltinformatik, Shaker, Aachen, 2005.

[7] ifu Hamburg GmbH: Institut für Umweltinformatik. http://www.ifu.com, last visit 02-29-2008

[8] Eclipse Modelling Framework, http://www.eclipse.org/modeling/emf, last visit 02-29-2008.

[9] Eclipse Persistence Services Project (EclipseLink), http://www.eclipse.org/eclipselink, last visit 05-26-2008.

[10] EMPORER Project, http://www.emporer.net, last visit 05-22-2008.

[11] P. Jahr, S. Simroth, Entwicklung von Teilbereichen eines Software-Frameworks im Einsatzkontext betrieblicher Umweltinformationssysteme, Bachelor thesis at the FHTW, University of Applied Sciences Berlin, Berlin, 2007

[12] M. Fowler: Patterns of Enterprise Application Architecture, Addison-Wesley, Boston (MA), 2003.

[13] NHibernate, http://www.hibernate.org/343.html, last visit 05-22-2008.

[14] E. Gamma et al, Design Patterns, Addison-Wesley, Reading (MA), 1999.

[15] M. Lippert, C. Videira Lopes: A Study on Exception Detection and Handling Using Aspect-Oriented Programming. 22nd International Conference on Software Engineering (ICSE '00), pp. 418-427, Limerick (Ireland), 2000.

[16] P. Laureano: Introducing the LinFu, Part V: LinFu.DesignByContract2 – Adding Transparent Design by Contract Features to Any .NET Language. http://www.codeproject.com/KB/cs/LinFu_Part5.aspx, last visit 05-27-2008.

[17] B. Meyer: Object-Oriented Software Construction. 2nd edition, Prentice Hall, Upper Saddle River (NJ), 1997.

[18] D.H. Akehurst, W.G. Howells, M. Scheidgen, K.D. McDonald-Maier: C# 3.0 makes OCL redundant! Electronic Communications of the EASST-ECEASST, Volume 9, TU Berlin, http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/viewFile/103/99, last visit 05-27-2008.

[19] Spring.NET, http://www.springframework.net, last visit 05-27-2008.

[20] C. Escoffier, D. Donsez, R. Hall: Developing an OSGi-like Service Platform for .NET. Consumer Communications and Networking Conference, CCNC 2006. 3rd IEEE, Volume 1, pp. 213-217, 8.-10. Jan. 2006.

[21] Hibernate, http://www.hibernate.org, last visit 05-27-2008.

[22] Java Persistence API, http://java.sun.com/developer/technicalArticles/J2EE/jpa, last visit 05-27-2008.

[23] Oracle: TopLink Developer's Guide, http://www.oracle.com/technology/products/ias/toplink/doc/10131/main/_html/undtl002.htm, last visit 05-28-2008.

Copyright © Shaker Verlag, Aachen 2008. ISBN: 978-3-8322-7313-2

[24] S. Eberle, S. Smith, EclipseLink and EMF: Enterprise Data Access in Eclipse, http://www.eclipsenowyoucan.com/commun/docs/EclipseNowYouCan%20EMF-JPA-EclipseLink.pdf, 2007.

[25] Teneo, http://www.eclipse.org/modeling/emft/?project=teneo, last visit 05-27-2008.

Copyright © Shaker Verlag, Aachen 2008. ISBN: 978-3-8322-7313-2