Model Reuse in MDA[1]

Salim Bouzitouna[1], Marie-Pierre Gervais[1,2] and Xavier Blanc[1]
1 Laboratoire d'Informatique de Paris 6 (LIP6)
LIP6 - 8 rue du Capitaine Scott - F75015 PARIS,
2 University Paris X
{Salim.Bouzitouna, Marie-Pierre.Gervais, Xavier.Blanc}@lip6.fr

Abstract: *The Model Driven Architecture (MDA) approach defined by OMG advocates a clear separation between platform-independent models (PIMs) and platform-specific models (PSMs) in distributed applications. As such, MDA would ease the migration of applications to new platforms. However, it deals only with the evolution of the platform underlying the applications not the evolution of the business part of applications, i.e., the PIM itself. A way for dealing with this later evolution is to reuse as much as possible the existing PIMs and their corresponding PSMs. We propose a new initiative based on two principal points: the expressions of the reuse of PIM without change them and from these expressions, the automatic generation of glue binding their corresponding PSM. We focus in this paper on the concept of GLUE, and more precisely on Glue for Enterprise Java Beans PSM.*

Keywords: Model-Driven Architecture, Enterprise Java Bean

1. Introduction

To manage the difficulties of developing distributed applications, solutions such as middleware technologies have been proposed and proved to be efficient. Nevertheless, the evolution and increasing number of such technologies impact the upgrading and maintenance of these applications. Since these depend on the middleware platform that supports them, their migration towards a new platform is increasingly complex and costly. Aiming to address this complexity, the OMG has proposed the Model Driven Architecture (MDA). Based on modelling techniques [1], it distinguishes business aspects from technical aspects. Its principles advocate that the designer must first capture the business concerns of the system to be built in a model called the PIM (Platform-Independent Model) while abstracting away technical details. Then, the PIM is transformed into a Platform-Specific Model (PSM) by introducing technical aspects of the target platform. This PSM constitutes the entry point toward code generation. The transformation of the PIM to a PSM is specified by a set of well mapping platform-specific rules.

Thus the MDA allows designers to focus on the business aspects of their applications. Furthermore, applying MDA would ease the migration of applications to new platforms. Since PIMs focus on business with no technical concerns, the migration of an application from one platform to another requires only the (re)mapping of the PIM onto the new target middleware. However, the MDA approach deals only with the evolution of the technical part of applications (i.e., the platforms) and how to easily obtain a new PSM from a PIM. But it does not really address the evolution of the business part of applications, i.e., the PIM itself. In fact, due to changing business logics, companies often need to adapt their applications. For instance, the merger of two companies often requires the merger of their applications. Other cases of evolution are the extension or modification of existing applications. A way for dealing with evolution in a cost-effective manner is to

[1] This work is supported in part by the "MODELWARE" project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006)"

investigate how to reuse as much as possible the existing applications. In the MDA approach, this reuse relates to the PIM and PSM. MDA does not bring a solution to this.

Our work aims at finding a way to cope with the gaps of the MDA approach to support PIM and PSM reuse. On one hand, it will enable the evolution of MDA applications and on the other hand the construction of new MDA applications by the reusing models (PIM as well as PSM) of existing applications. Our proposal is inspired by the study of existing approaches dealing with reuse [2, 3, 4]. Having studied these approaches compared to MDA levels namely PIM, PSM and code, we observe that none of them fits our requirements, i.e., they do not address the three MDA levels. In addition, they frequently impact the existing models.

Thus we present in this paper our approach that stands as follows. As preliminary requirement, we consider that the reuse of PIMs must not modify the existing PIMs. Consequently we provide a designer with means to express the PIMs reuse in terms of composition, extension, and modification. This expression of composition (*EC)* comes in addition to the existing PIMs to be composed (or extended or modified). Once this expression *EC* is available, we provide a means for its automatic translation into its corresponding pieces of model at the PSM level. We call the result of this translation the "*Glue*" as it binds the existing PSMs according to *EC.* Our work focus on PSMs dealing with component platforms such as EJB [5] or CCM [6]. Thus these PSMs model the application to be built in terms of interacting components. In this context, we consider the *Glue* as the component that will be used for the assembly of the existing PSMs.

This paper is organized as follows. In section 2, we give a short analysis of some existing approaches dealing with reuse according to our requirements. Section 3 details our approach. The concept of *Glue* is explained in section 4, by restricting the presentation to Enterprise Java Beans platforms. Section 5 presents the conclusion and some future work.

2. Analysis of Existing Approaches in Reuse

There are several existing approaches which offer means for reuse such as Subject-Oriented Design [2], View-Oriented Design [3], or Subject-Oriented Programming [4]. We have studied them with an eye on their capacity to address our requirements, i.e., to be applied at the MDA levels and to preserve the existing models to be reused. For lack of space, we only present here our observations as follows:

- Most of the existing approaches offer techniques which apply at a single MDA level with no consideration of the others. For instance, Subject-Oriented Design or View-Oriented Design only deal with the PIM level. Others (e.g., Subject-Oriented Programming) only deal with code. Few approaches explicitly deal with the reuse of PSMs [7].
- These approaches propose means for the reuse that carry out direct changes on the reused models and do not guarantee a good traceability of their evolution.

These drawbacks led us to propose our own approach as described below.

3. A Proposal for the Integration of Reuse in the MDA

In order to introduce our approach to integrating reuse in MDA, let us consider the following example (Figure 1). Let A1 and A2 be two existing applications, developed according to an MDA approach. This means that for each of them, the PIM, the PSM, the translation from PIM to PSM, and the code are available. When developing a new application offering a functionality that could be obtained by composing the functionalities of A1 and A2, without altering their operation, the reuse of PIMs must not impact the existing ones, as well as the PSMs and code. Thus, to keep PIMs unchanged while composing A1 and A2, a designer starts to define the "expression of composition" *EC* between the PIMs. *EC* describes how the elements of PIM_1 and PIM_2 will be integrated, without changing them, in terms of merge, extension or update. For this, we have defined a set of composition rules presented below. Then, from this expression, a *Glue* is generated, which binds PSM_1 and PSM_2. This *Glue* models at the PSM level the integration expressed by *EC* at the PIM level. As such, it depends not only on the expression *EC* but also on the type of PSMs that it binds. We focus here on EJB PSMs. We detail hereafter our proposal.

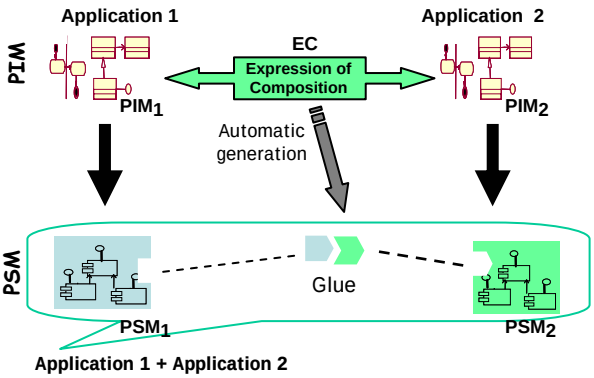


Figure 1. Our approach for the integration of MDA applications

3.1. Reuse of PIMs: Expression of Composition EC

In order that a designer can define an expression of composition *EC*, we have identified a set of composition rules [8]. We base them on the two following considerations: (1) PIMs are described in UML and (2) the reuse of applications can be motivated by other intentions than composition.

To describe PIM, OMG recommends the use of UML [9]. Although UML defines many diagrams, we are particularly interested in class and sequence diagrams, which respectively specify the structural and behavioural aspects of an application. In addition, they describe them independently of any platform, which is well suited for specifying PIM.

According to these aspects and based on these diagrams, the composition of PIMs can be structural or behavioural. The structural composition consists of either joining elements of class diagrams, or merging elements that correspond to the same entity (*classes*) or the same property (*attributes*) into a single element. The behavioural composition consists of identifying the interactions that must take place between operations of the sequence diagrams. This can be for example combining the processing of operations belonging to different PIMs.

The reuse of applications does not deal only with composition. Other types of reuse can be identified, such as extensibility or modification of an existing application. However, we consider in our work that extensibility and modification can be achieved by using mechanisms similar as those defined for composition, i.e., extensibility or replacement. Since extensibility consists of adding new functionalities to a PIM, these can be defined in a separate model, which is then composed with the original PIM. This enables detailed tracing of a PIM's evolution. In the same way, modification can consist in defining all the changes to be brought on a PIM in a separate model called the *substitute model*. Then, the PIM elements are composed with these of the substitute model.

Thus, the set of composition rules for PIM reuse can be classified in three categories of rules as follows:

Correspondence rules *express correspondences between elements (packages, classes, operations...) of separate models, which represent the same concept. In case of merging models, these rules specify model elements which will be merged into a single one. In the case of replacing models, these rules establish a relation between elements of a model and those to be replaced in the target model. In both cases, these rules do not specify how these elements will be merged or replaced.*

Combination rules *express the way in which a merge is carried out between a set of elements. These should be put in correspondence beforehand. The merge of a set of elements consists in unifying their sub-elements. If there is a correspondence between these sub-elements, only one among them will be kept in this unification. The combination rules enable one to express which elements will be kept into a merge. For instance, JoinClasses [packageA.ClassA, packageB.ClassB] expresses that classA and classB should be merged and if there are correspondences between their sub-elements, those of classA should be kept. For the combinations of operations, we have defined special rules, which enable to express the execution process of operations in correspondences.*

Replacement rules *are used to express updates of elements in a given PIM. As updates are described in the substitute model, these rules enable one to identify the elements of the substitute model that replace those of the source model. For instance OverrideClasses[packageA.ClassA, packageB.ClassB] expresses that the properties (attributes & operations) of ClassA replace those of classB which correspond to them.*

By these rules, the designer expresses the composition between existing PIMs. This expression *EC* together with the original PIMs constitute the new PIM of the application to be built. As such, this PIM is the set {PIM_1, PIM_2, EC} rather than a new self-contained model. Thus, contrary to the major trend, our rules do not lead to the construction of a new model by modifying the existing ones. In our approach, the source PIMs are not modified at all. This guarantees a good traceability of the evolution of the reused models in conformance with the basic principle of MDA approach, which supposes that the PIM of a given application remains stable.

3.2. Reuse of PSMs: Generation of the Glue

According to the MDA approach, once the new PIM is available, it is transformed into a PSM by introducing technical aspects of the target platform. This mapping consists of applying *EC*, i.e., the *combination rules* or *replacement rules*, on the existing PSMs. There are two ways for doing this:

- Modify the existing PSMs: This implies the regeneration of code and data represented in these PSMs whereas our goal is to reuse them.
- Reuse the existing PSMs without changing them: This enables to maintain the existing applications while they are evolving.

According to our idea that the reuse must not affect the existing entities (PIMs, PSMs, code), we adopt the second solution. For this, *EC* constitutes the entry point for the generation of one or several entities, i.e., the *Glue*.

Since this generation depends on the type of the target platform, we chose in our work to consider component-based platforms and more particularly EJB platforms. Actually, these are widely used for the construction of distributed applications. Furthermore, they offer a good flexibility in terms of reuse, extensibility and assembly. Thus the rest of this paper focuses on the concept of *Glue* dedicated to EJB platforms.

4. The Concept of Glue in the EJB Platforms

4.1. What is the "Glue"?

In order to precisely specify our idea of *Glue* for EJB platforms, we initially examined the transformation from PIMs to EJB PSMs. To this end, we studied some EJB applications developed according to MDA approach such as PetsStore [10]. Our purpose was to capture the mapping rules of the UML class and sequence diagrams onto EJB platforms. Actually, let us recall that the PIMs we consider are defined with a UML class diagram for the business and process entities and a UML sequence diagram for the behaviour of operations, i.e. actions and interaction, of these entities. Although the developers of such applications do not specify explicitly the transformation they implement, we can extract their rules through the study of the application.

For the sake of space, we simply enumerate these rules (Table 1).

1) A *business* class maps onto an Entity bean. Its attributes, depending on whether they are persistent or not, map onto persistent or simple fields in the bean.
2) A *process* class maps onto a Session bean. Its attributes map onto fields in the bean.
3) An association or a dependency between classes, depending on the nature of the class (business or process), maps onto *EJB Relationships* or *EJB references* between the corresponding beans.

Table 1: Mapping Rules from PIM to EJB PSM

These rules enable us to identify the three following statements :

I) These rules illustrate how to map PIM elements onto EJB PSM elements. So they are useful for our purpose, which is the translation of *EC* defined at PIM level by a designer. Actually, let us recall that *EC* is a set of combination and/or replacement rules applying on PIM elements.

Thus, we can state:

Translation of *EC*
=
Mapping the combination and the replacement rules expressed between PIM elements onto effective combination and replacement of their corresponding EJB PSM elements

II) At the PIM level, the designer can express some combination rules aimed at merging some PIM elements, such as:

1) The merge of features *(attributes/ operations)* of different classes into one feature *(attribute/ operation)*;
2) The merge of classes of different packages into one class;
3) The merge of sub-packages of different other packages into one package.

However, since the encapsulation unit of PIM is the class, all these merges consist in merging classes. Actually, the merge of the packages consists in merging their corresponding classes. In addition, the merge of attributes or operations must initially deal with the merge of their container, which are classes.

This also holds for the replacement rules, which consist in replacing elements of one PIM by those of another. Thus, we can state:

Unit of combination or replacement at PIM level = class

III) When considering the transformation rules mentioned in Table 1, we note that the classes at PIM level usually map onto beans. Thus, we can state:

Merge or the replacement of classes = Merge or the replacement of the corresponding beans

These three statements enable us to define the *Glue* as a PSM binding entity responsible for the merge or replacement of beans. We study in the following what these two operations are and how the *Glue* achieves them.

4.2. The merge of beans

The merge a set of source beans into a single target bean consists in merging an attribute existing in each source bean and representing the same property into a single attribute of the target bean. This must be done for all common attributes of the source beans. In the same way, the operations of the source beans must be merged. Thus the operations that correspond to the same behaviour are represented by a single operation, which combine their behaviours. The identification of the attributes and operations to be merged comes from the EC correspondence and combination rules defined at PIM level by the designer.

The *Glue* achieves the merge of beans according to the EJB design pattern CompositeEntity [11]. We explain it thanks to the example presented in Figure 2.

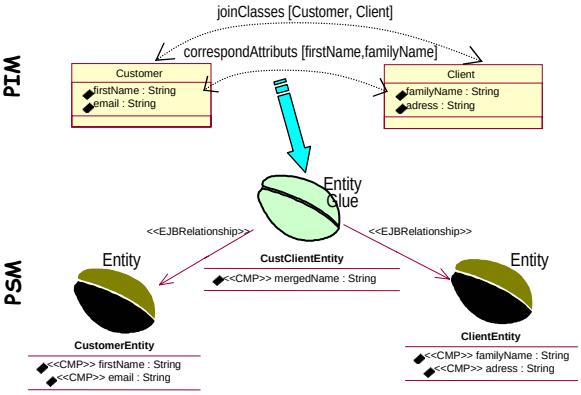
Let us focus on the first part of the figure, which represents two PIMs, each of them consisting of a business class, respectively *Customer* and *Client*. Each class maps onto an entity bean according to the mapping rules presented Table 1. Then let us consider a designer, having to compose these PIMs, who has expressed an *EC* between these classes This *EC* includes the rules *joinClasses[]* and *correspondAttributs[]*. This enables one to express that the *firstName* attribute of the *Customer* class corresponds to the *familyName* attribute of the *Client* class, i.e. both attributes represent the same property.
In such a context, the *Glue* is an entity bean *(CustClientEntity),* which plays the role of a *wrapper* described as follows:

All accesses to the features *(attributes and operations)* of source beans must be carried out through the *Glue* bean. Thus the *Glue* bean unifies the access to the corresponding features of source beans by defining a new feature, which represents them. This is illustrated in Figure 2 with the *mergedName* attribute in *CustClientEntity* bean, which respectively corresponds to the *firstName* attribute in the *CustomerEntity* bean and to the *familyName* attribute in the *ClientEntity* bean. However, the *Glue* bean delegates the access to features of source beans, which do not have correspondence, to their source beans.

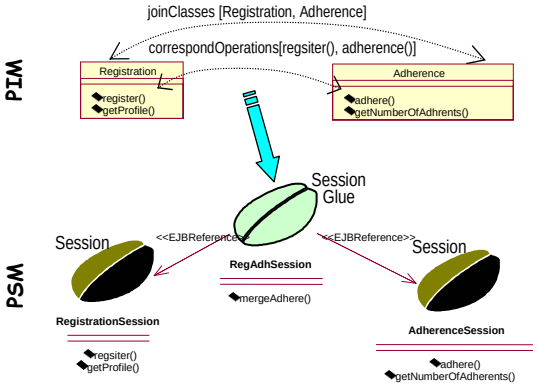
Consequently, the *Glue*, as a *wrapper*, must have an access to source beans it merges by using references. These references are defined by using the *EJB relationship* association in the case of the merge of Entity beans. In the first part of figure 2, two *«EJBRelationship»* associations are defined for the merge of *Customer* and *Client* entities.

The same applies for the merge of session beans, as represented in the second part of the figure 2. In this case, let us notice that the *Glue* bean is a session bean and the references to source beans are carried out using the EJB Reference *(«EJBReference»).* Actually, the type of *Glue* bean depends on the nature of source beans, i.e. entity or session beans. This is due to the fact that the nature and the lifecycles of these two types of

beans are different, and it seems appropriate to choose a *Glue* bean of the same type of the source beans. We can also notice that usually, the merge applies on beans of the same nature as it comes from the merge of classes at PIM level. Now, it seems irrelevant to merge business and process classes.



Merge of entity beans



Merge of session beans

Figure 2. The Glue for the merge of beans

4.3. The replacement of beans

The replacement of a beanA by a beanB is in fact an overload between B and A. Actually, the replacement rule that can be defined at PIM level between two classes *classA* and *classB* means that attributes and operations of *classB* replace the corresponding ones in the *classA*, while maintaining other attributes and operations defined in *classA* which have no correspondence. Consequently, this overload must be carried out between the beans corresponding to these classes, i.e. the bean corresponding to *classB* **overloads** the bean corresponding to classA. A replacement bean must be created which will have the replaced features according to the replacement rules, together with the other features not involved in these rules. This bean is the *Glue* for the overloading.

The *Glue* achieves the replacement of beans according to the decorator pattern [12]. Using this pattern, the *Glue* bean plays the role of the decorator while the source bean plays the role of the decorated. As previously, we explain this idea thanks to an example presented in Figure 3.

Let us focus on the first part of the figure, which represents a PIM consisting of a business class, *Client* that has to be replaced by a new class *Customer*. Both classes are mapped to an entity bean according to the mapping rules presented in Table 1. An *EC* has been defined between these classes. This *EC* includes the rules *overrideClasses[]* and *correspondAttributs[]*. This enables to express that the *Name* attribute in *Client* class, and with *String* type, corresponds to the *Name* attribute, *FullNameClass* type, in *Customer* class.

In such a context, the *Glue* is an entity bean (*CustomerEntity*), which plays the role of *wrapper* described as follows:

All accesses to the features *(attributes and operations)* of source bean must be carried out through the *Glue* bean. Then the *Glue* bean redefines the features of source bean, which are overloaded. In the first part of figure 3, this is applied for the access to *Name* attribute. However, the Glue bean delegates the access of features, which are not overloaded to the source bean. In the first part of figure 3, this is applied for the access of *address* attributes in the overload of *ClientEntity* bean. The glue bean could define new features for the source bean, such as an email attribute in the overload of *ClientEntity* bean.

Consequently, the Glue bean, as a *wrapper*, must have an access to the beans, which it overloads, by using references. These references are defined, using *EJB relationship* association in the case of the overload of an Entity bean. In the first of figure 3, an *«EJBRelationship»* association is defined for the overload of *ClientEntity* bean.

The same applies for the replacement of session beans, except that the reference to the source bean is carried out using the EJB reference *(«EJBReference»)* as represented in the second part of the figure 3. Let us notice that, like the Glue for the merge of beans, the Glue for the replacement is the same type as the type of the bean to be replaced.

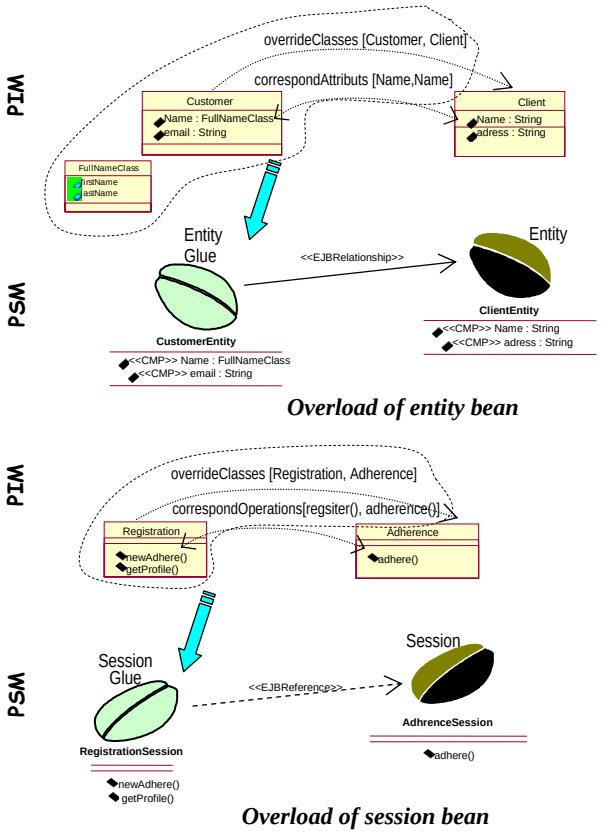


Figure3. The Glue for the overload of bean

5. Conclusion

In this paper, we have presented a new initiative for the reuse of applications in the MDA approach. This initiative will enable to support the evolution of MDA applications and to construct new MDA applications by reusing models of existing applications. Our idea is that the reuse must not affect the existing entities (PIMs, PSMs, code). It is based on two main ideas: the reuse expressions of PIMs and the generation, from these expressions, of Glue which binds the corresponding PSMs. To express the reuse of PIM we have defined a set of Composition rules in [8]. These rules allow dealing with the reuse of applications in terms of composition, extension and modification.

We have focused in this paper on the identification of Glue in the context of component-based platforms, namely EJB platforms. For this, we started by the identification of mapping rules onto this kind of platform in order to study the structure of PSMs in EJB platform. Then we have studied the mapping of the reuse expressions at PIM level onto this kind of platform. We have noted that the mapping of these reuse expressions consists in merging and overloading beans. Consequently, the Glue should be able to merge or overload beans. For this, we based our approach on patterns in order to propose the structure of Glue beans allowing the merge and the overload of entity as well as session beans.

Further work would be initially to develop the glue generation tool. This tool consists of two parts: an analysis part which consists to identify the type of glue to be generated. For this, it examines on one hand the set of input composition rules to identify whether the Glue is for merge or for overload. On other hand it examines the types of beans to merge or to overload, i.e. entity beans or session beans. The second part is the generator which effectively generates the glue according to the type of the glue identified by the analysis part. The choice of this architecture of glue generation tool allows the generation of glues for different platforms. In addition, we think of refining our composition rules in order to consider other type of reuse of PIMs.

6. References

[1] OMG, *"Model Driven Architecture Guide version 1.0"*, Document Number: omg/2003-05-01. May 2003.

[2] S. Clarke, *"Composition of Object-Oriented Software Design Models"*, PhD thesis, School of Computer Applications, Dublin City University, January 2001.

[3] G. Vanwormhoudt, *"CROME : un cadre de programmation par objets structurés en contextes"*, PhD thesis, Laboratoire d'Informatique Fondamentale de Lille I, Lille, 1999.

[4] W. Harrison, H. Ossher, *"Subject-Oriented Programming (A Critique of Pure Objects)"*, Proceedings of OOPSLA, October 1993.

[5] Sun: EJB, *"Entreprise JavaBeans"*, http://java.sun.com

[6] OMG, *"CORBA Component Model Volume 1"*.TC Document ad/99-01-01 OMG 1999. http://www.omg.org

[7] A. Muller, O. Caron, B. Carrée, G. Vanwormhoudt, *"Réutilisation d'aspects fonctionnels : des vues aux composants"*, Proceeding of LMO'03, February 2003.

[8] S. Bouzitouna and M. P. Gervais, *"Composition rules for PIM"*, Proceedings of EWMDA'04, September 2004.

[9] OMG, *"Unified Modeling Language Specification v1.3"*. OMG Document: formal/2003-03-01. March 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[10] Sun Microsystems, Java pet store 1.3.2 blueprint application, "http:/java.sun.com/developer/releases/petstore", 2003.

[11] D. Alur, J. Crupi and D. Malks *"Core J2EE Patterns"*, Prentice Hall PTR 2003.

[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides *"Design Pattern"*, Addison-Wesley Professional Computing Series, 1995.