TECHNICAL TRANSACTIONS | **CZASOPISMO TECHNICZNE**
FUNDAMENTAL SCIENCES | NAUKI PODSTAWOWE

1-NP/2016

PIOTR ZABAWA*

THE SCOPE MANAGEMENT PROBLEM IN JAVA ENTERPRISE EDITION FRAMEWORKS

PROBLEM ZARZĄDZANIA ZAKRESEM WE FRAMEWORKACH JAVA ENTERPRISE EDITION

Abstract

The paper focuses on the problem of managing the scope understood as managing the multiplicity of elements that constitute the application context for Java Enterprise Edition (Java EE) frameworks. The subject of constructing graph modeling languages is the basis for scope management considerations. The problem can be demonstrated while the frameworks are superposed, which is necessary for meta-modeling compliant to the Context-Driven Meta-Modeling (CDMM) approach. The realization of the approach is based on Spring and AspectJ frameworks, which offer incompatible concepts of scope management. As part of the analysis the scope management problem in Java EE frameworks application context was identified, formulated, its area was defined and the sketch of the generalized concept of scope management elaborated and implemented by the author in relation to Java EE frameworks was presented..

Keywords: modeling language, meta-model, graph, application context, java bean, java enterprise framework, Spring, aspect-oriented programming, AspectJ

Streszczenie

Artykuł ten koncentruje się na problemie zarządzania zakresem rozumianym jako zarządzanie krotnościami elementów składających się na kontekst aplikacji we frameworkach Java Enterprise Edition (Java EE). Punktem odniesienia dla rozważań dotyczących zarządzania zakresem jest zagadnienie konstruowania grafowych języków modelowania. Problem ten ujawnia się przy składaniu ze sobą tych frameworków niezbędnym w meta-modelowaniu zgodnym z podejściem Context-Driven Meta-Modeling (CDMM). Jego realizacja oparta jest na frameworkach Spring i AspectJ, w których koncepcje zarządzania zakresem nie są zgodne. W ramach analizy zidentyfikowano problem zarządzania zakresem w kontekście aplikacji Java EE, sformułowano ten problem, określono jego zakres oraz zaprezentowano zarys opracowanej i zrealizowanej przez autora uogólnionej koncepcji zarządzania zakresem w odniesieniu do frameworków Java EE.

Słowa kluczowe: język modelowania, metamodel, graf, kontekst aplikacji, ziarno java, framework java enterprise, Spring, programowanie aspektowe, AspectJ

DOI: 10.4467/2353737XCT.16.137.5716

* Piotr Zabawa (pzabawa@pk.edu.pl), Department of Physics, Mathematics and Computer Science, Cracow University of Technology.

1. Introduction

Scope management in a broad sense comprises managing the number of instances during the process of constructing them, that is, at run-time. The conventional approach of programmers associates the responsibility of multiplicity determining mentioned above with a class. This is apparent, among other things, in singleton (anti)pattern. However, in Java EE frameworks this responsibility is moved to the framework. A bean multiplicity in the framework can be specified by the application context. The bean, on the other hand, reflects the way the framework (and in the consequence the software system implemented in the framework) perceives classes. The bean contains more information than the class, among other things just information about multiplicity of the bean. This additional information stored in the bean is specified in the application context file based on which the framework creates bean instances (and thus class instances). A particular class may occur once as the instance of one bean the scope of which is specified as singleton and, at the same time in the same application, the same class may occur multiple times as instances of other bean (associated to the same class) the scope of which is defined as prototype. In contrast to meta-models (modeling languages) constructing this solution turns out not to be sufficient due to the high complexity of graph meta-models. Also relating the scope to the bean only turns out not to be sufficient while applying it to graph modeling languages. That is why the need to enrich the current mechanism occurred.

In scientific papers [12] as well as in the IT industry literature [5] and in industry standards [6, 13, 19] meta-models are created statically – modeling languages are defined at compile time. However, as research results achieved by the author show [20], it is possible to define modeling languages at run-time. The application context mentioned above can be used to specify graph-like interrelations between language elements.

Further in the paper it is shown that when the scope notion is addressed to modeling languages constructed at run-time, this notion should be addressed both to Java EE beans and to classes. Moreover, bean sets as well as sets of classes involved with relationships interrelating particular bean sets play an important role in meta-modeling.

A characteristic feature of the CDMM approach [21] is constructing meta-model graph from elements consisting of meta-model entity classes and meta-model relation classes. The graph is constructed from Java EE beans defined for these classes, thus from entity beans and relation beans. Entity beans are placed in graph nodes while relation beans are placed in graph edges. In such an approach the application context XML file constitutes the definition of the meta-model graph. However, in such approach the correct management of relation instances quantity during relation beans injections into entity beans is an important problem. It is especially evident with reference to N-ary relationships [4, 8, 9, 14, 15, 17], relations that join more than two graph nodes at the same time. In the case of such relations the mechanism of injecting the same relation object (relationship bean instance or relation class instance) to all nodes involved with this relation must be provisioned. It appears, however that the possibilities offered by Java EE frameworks are not sufficient in the area of multiplicity management, as they are focused on management of multiplicities of singular beans only. It is worth noticing that the implementation of N-ary relations and the so called "arity problem" is difficult while constructing graph modeling languages. It is shown by documented problems visible in Object Management Group (OMG) standards, like Meta--Object Facility (MOF) – the definition of N-ary association was omitted here because of too much difficulty [1, 13], then the implementation of this relationship as a separate notion in Unified Modeling Language (UML) standard was retired [19] (it is represented on the diagramming and not on the modeling level, so the code cannot be generated from UML modeling tools) [7]. The root cause for these problems and limits is the lack of representation for relationships in all sources known from scientific literature, IT industry publications and software modeling tools documentation [3, 10, 11, 18] However, these problems can be solved in CDMM technology as the relations have their representation in it.

It should be pointed out that the scope management problem with reference to the CDMM technology concerns such meta-model elements only which are involved in representing relations, so they play the role of edges of the graph being the representation of a modeling language. Edge (meta-model relation) classes play the role of static responsibilities for node (meta-model entity) classes. These responsibilities are injected to entity classes as default implementations of interfaces of these relation classes with the help of dependency injections (Spring) and with the help of aspect-oriented inter-type declarations (AspectJ).

2. Scope Management in Spring Framework

The Spring framework offers scope management limited to the Spring beans. The bean is the way Spring as well as the Spring-based application (more generally – a software system), sees Java POJO classes. The object model in Spring is enriched in comparison to Java object model by many attributes that can be associated to beans. One of them is the "scope" attribute of a bean. According to the documentation of Spring framework [16] the scope attribute can have one of the following values: "singleton", "prototype". The "singleton" attribute informs Spring that the bean with this attribute value can have exactly one instance – the bean and not the POJO class behind the bean. The "prototype" attribute informs that the bean with this attribute can be multiplied as needed.

Static information about beans is defined in Spring application context XML file. As long as bean instances are created from the application through the Spring Application Programming Interface (API) the solution offered by the framework is sufficient.

When the instance of a particular bean is created from the Spring-based application through the API of Spring the constructor of the class which is behind the bean is called by default. However, Spring offers also another mechanism, which is applied in the approach presented in the paper. The bean instances may be created through factories. This approach is much more flexible and was originally added to Spring to simplify application of creational design patterns.

3. Scope Management Problems in AspectJ with Spring

The situation described in section 2, when Spring is used as the only framework and when bean instances are created from the Spring-based application is simple and does not trigger any problems. However, when the Spring is superposed with other framework and this additional framework influences or even takes control over bean instances creation process, some problems appear. They result from the fact that the additional framework may take responsibility for bean instance creation from the Spring-based application to the additional framework. Moreover, the additional framework may delegate this responsibility back to Spring and to application context. And this is the case when Spring is superposed with AspectJ [2].

The Spring framework is integrated to Aspect Oriented Programming (AOP) via two Spring sub-projects: SpringAOP [16] and Spring+AspectJ [16]. The first one constitutes a limited implementation of AOP concepts and is not sufficient for the CDMM-F implementation. However, the second project offers full AspectJ functionality and is sufficient for the application of the CDMM concept. The rest of the paper is limited to the full integration of Spring with AspectJ.

The implementation of CDMM-F is based on extensive usage of AOP concept applicable to influencing class hierarchies, thus inter-type declarations, and more specifically, declare-parents construct. This way the relationship classes of a meta-model can be injected as default interface implementations to particular meta-model entity classes as their structural responsibilities (in contrast to dynamic responsibilities, which are more typical). The method for such injections is specified in Spring+AspectJ application context file according to the sample code presented in Listing 1.

```
<!– Meta-Model Entity Beans (Spring) -->
<bean
  class=”com.componentcreator.metamodel.coremetamodel.domains.DEntity”
  id=”entity”
  scope=”prototype”>
</bean>

<!– Meta-Model Relation Beans (Spring) -->
<bean
  class=”com.componentcreator.metamodel.coremetamodel.relations.RRelation”
  id=”relImplForDEntity”>
</bean>

<!– Meta-Model Graph Creation (Spring+AspectJ) -->
<!– Meta-Model Relation Injections to Meta-Model Entities -->
<aop:declare-parents
  types-matching
    =”com.componentcreator.metamodel.coremetamodel.domainsimpl.DEntity”
  implement-interface
    =”com.componentcreator.metamodel.coremetamodel.relations.IRRelation”
  delegate-ref=” relImplForDEntity”/>
```

Listing 1. Meta-model elements defined in Spring and their injections defined in Spring+AspectJ application context file (extract only)

It is clear from the Listing 1 that meta-model entity beans have their scope defined as “prototype” while the attribute is ignored for relationship beans. It is not specified in application context file to underline the fact that AspectJ ignores this attribute for beans it injects.

When the Spring integrated to AspectJ loads an application context file that contains such injections, the default implementations of interfaces are created as Spring beans. This behavior influences and destroys the original Spring concept of scope management. It is even impossible to change the Spring+AspectJ behavior from the bean "scope" parameter – from its predefined as well as from its user-defined version. The Spring interpretation of the "scope" bean parameter is completely overlapped by AspectJ. But, fortunately, Spring+AspectJ create injected beans of default implementation classes for each such injection. Moreover, the AspectJ mechanism does not overwrite the option of calling factories in place of constructors when a bean is instantiated. It is shown further in the paper that combining both mentioned features helps to take full control over the instantiating process when meta--model is created.

4. Scope Management Problem

This section is focused on two goals – showing how the control over scope management (introduced intuitively before) can be regained in case of overlapping incompatible solutions offered by different Java EE frameworks and presenting the skeleton of the concept of advanced scope management for meta-modeling purposes.

In order to address the two goals mentioned above, the scope management problem should be clearly stated and then its solution can be presented. At the end the correctness of the solution should be verified. All these stages are presented below.

4.1. Problem statement

The scope management problem is the problem of controlling the multiplicity of application elements while their construction process driven by Java EE application context under the assumption that the application context file is interpreted by more than one Java EE framework.

As the consequence we have the following situation – the superposition of frameworks:

$$F = F1^\circ F2^\circ \ldots ^\circ FN$$

where:

F	–	the framework created as the result of superposition of other frameworks,
F1 – FN	–	superposed frameworks.

The problem is at least two-dimensional as it concerns both classes and their beans. The problem of the actual dimension is discussed in section 4.2. The size of the problem does not depend of the number of frameworks *F1 – FN*.

The problem is limited to meta-model relation beans and classes.

The problem can be solved if the following conditions are fulfilled:

- *FN* framework tries to construct application elements whenever needed
- *FN* framework does not eliminate the ability to access factories for application elements construction purpose

Topological aspects only are taken into account in the paper. This means that such problems like cardinalities of meta-model relationships (meta-cardinalities) as well as the problem resulting from the above – the problem of existence of some nodes at the meta-model relation ends are ignored in the paper. The problem of meta-model relationship cardinalities which is new and separate from the scope management problem is intended for future publications.

4.2. Problem solution

It was mentioned before that scope may be addressed to beans and/or classes specified in the application context. Another observation related to Spring scope management is that the concept of scope management is related to the whole application. This means that the particular scope associated to a particular bean defines the multiplicity of the bean instances in the whole application. However, in the meta-modeling problem the range of the scope should be differentiated to such areas like meta-model, context file, constructor.

As the result, in the meta-modeling problem, the following dimensions of the scope management problem should be assumed:

Subject (relationship class, relationship bean)

Scope (meta-model, context file, constructor)

Thus, the name of scope fits better to the true meaning of this notion.

For each combination of the above elements, for each pair (Subject, Scope) the element of the framework F which is responsible for scope management should be identified. So, the divagations should be enriched by the following mapping:

$$(\text{Subject} \times \text{Scope}) \rightarrow \text{ScopeManager}$$

where:

$$\text{ScopeManager} = \{\text{class, bean, context, framework}\} \subset F$$

The communication between framework F and the right ScopeManager is controlled by factories that are called while constructing application elements. The special case is when the factory does not delegate the scope management responsibility to dedicated ScopeManager but takes this responsibility. This assumption was assumed in the rest of the paper for simplification. As the result, the naming convention for factories, which in consequence of this assumption can be predefined in F, can be introduced. The naming convention may be as follows:

`Responsibility<Subject><Scope><Manager>ScopeFactory,`

where, under the above assumption <Manager>=Factory $\subset F$

In consequence, the names of such factories are as the ones contained in Table 1.

Table 1

The names of factory classes which are responsible for managing meta-model relationclasses

Subject / Scope	Class	Bean
Metamodel	ResponsibilityClassMetamodel ScopeFactory	ResponsibilityBeanMetamodel ScopeFactory
Context file	ResponsibilityClassFile ScopeFactory	ResponsibilityBeanFile ScopeFactory
Constructor	ResponsibilityClassConstructor ScopeFactory	ResponsibilityBeanConstructor ScopeFactory

The ResponsibilityBeanConstructorScopeFactory class is sufficient to solve the arity problem. That is why the nature, implementation and verification of just this class is discussed further in this section as the illustration of the factories implementation concept.

Two variants are taken into account below to characterize the nature of ResponsibilityBeanConstructorScopeFactory class. The simple case is presented first (one relation for a particular set of entities). Then the complex case (many relations for a particular set of entities) is shown. The problem of number of relations in meta-model has not been identified and has not been investigated before. The name suggested by the author for this problem is meta-cardinality and it is related to the CDMM system of notions. However, this problem is discussed in a separate paper. The two cases mentioned above are defined for:

– a particular relation (for a particular bean of a relation class) joining a set of entity classes – one bean instance is created by the factory
– many relations of the same kind (represented by the same bean of a relation class) joining a set of entity classes – the number of bean instances created is equal to the number of relations.

More generally speaking, for a particular set of constructors of any number of a relation beans (for the same relation class) the number of instances of this bean is equal to the number of beans and not to the number of the bean class injections to the set of entity classes.

The characteristics of ResponsibilityBeanConstructorScopeFactory class can be referenced to Figure 1 and Listing 3 in section 4.2.

In the next research stage all possible combinations of relation construction cases were identified for the meta-modeling application domain. These observations have theoretical nature (all cases were identified for consideration completness).

The following notational system was designed to specify scope in the application context file:

– CDMMFsubject (applied in each bean to determine if the scope is related to the bean or to its class)
– CDMMFscope (applied in each bean to define the scope for CDMMFsubject)
– CDMMFmanager (applied in each bean to define the element responsible for the scope management for this bean)
– The following comments are related to the system of tags introduced above:
– CDMMFmanager may be optional (if we assume that the scope management is default)
– CDMMFmanager may not be required if the right class will be determined by the pair (CDMMFsubject, CDMMFscope)
– as long as any Java EE framework *F* has its notation related to scope management the CDMM prefix is required

The implementation of the ResponsibilityBeanConstructorScopeFactory scope manager is presented in Listing 1.

```java
public class ResponsibilityBeanConstructorScopeFactory implements
  IResponsibilityBeanScopesFactory {

  private static Map<String, IResponsibility> relationshipMinimal
    = new HashMap<String, IResponsibility>();

  public IResponsibility getInstanceMinimal(String beanId, String cls,
    List<String> str) throws NoSuchMethodException, SecurityException,
    ClassNotFoundException, InstantiationException, IllegalAccessException,
    IllegalArgumentException, InvocationTargetException {
      // the instance of the beanId was already created
      if (relationshipMinimal.containsKey(beanId)) return
        relationshipMinimal.get(beanId);
      // the beanId has not been created yet
      else {
        // create the instance of cls object passing it str parameters
        // - Java reflection needed here
        relationshipMinimal.put(beanId, (IResponsibility)
          ResponsibilityBeansRegister.get(cls).getConstructor(new Class[]
          {List.class}).newInstance(new Object[] { str }));
        return relationshipMinimal.get(beanId);
      }
  }
}
```

Listing 2. Scope management factory class dedicated to N-ary relationship instance multiplicity handling

The factory implemented in the form presented on the Listing 1 works as follows. The meta-model relation bean (represented by beanId in the source code) is defined in the application context file for Spring Java EE. Then the relation bean is injected by AspectJ framework when the default interface implementation of a relation class is associated to a meta-model entity class. In place of constructor the method getInstanceMinimal() is called with the following parameters: beanId equal to the Id of relation bean, cls equal to the pathname of the relation class, str equal to the list of pathnames of entity classes the relation bean is injected to. The method determines if the object was already constructed for the set of parameters (beanId, cls, str) and creates it or returns the reference to already existing bean instance.

4.3. Verification

The concept of scope management was tested for the superposition of Spring and AspectJ frameworks. This combination of frameworks is sufficient for obtaining the superposition with required features as defined in section 4.1. This superposition of just these frameworks is also good enough for defining sufficiently complex meta-models.

The correctness of the approach presented in the paper was verified in three following stages:
– all factory classes presented in Table 1 were implemented,
– appropriate meta-models were defined to generate all test cases (at least one test case was needed to test each factory class),
– appropriate unit tests were implemented to test each test case resulting from meta-models defined above.

All test case executions confirmed the correctness of both the approach and the implementation of all factories dedicated to support meta-modeling. It is worth noticing that the elaboration of all meta-model concepts required to implement test cases for each factory class was especially demanding and time consuming. This complexity resulted from the fact that in this case the special meta-modeling problems should be invented to check the correctness of the solutions which were foreseen before during theoretical research. This approach was abnormal as usually the problem appears first and the solution comes later.

As the illustration of the use of the factory for a sample meta-model is presented in Figure 1 and then the extract from the application context file is shown.

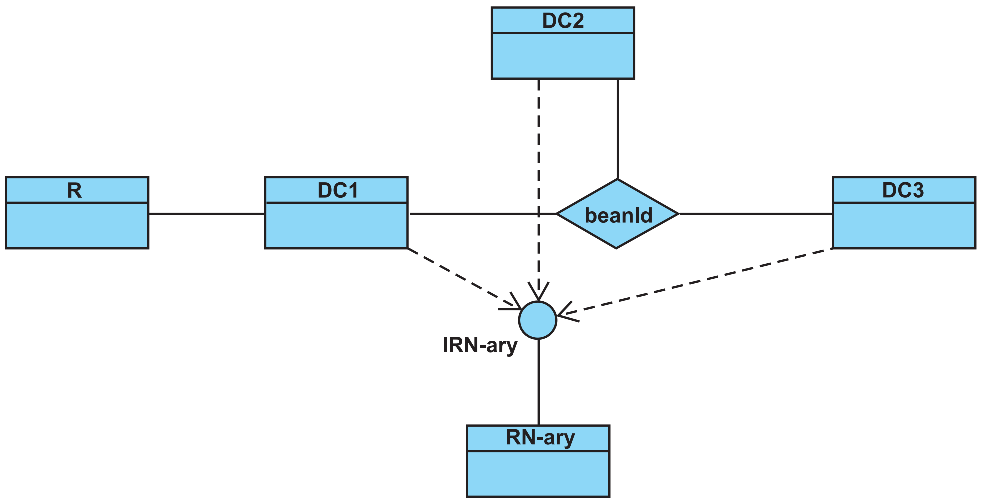

Fig. 1. Sample meta-model for the N-ary relationship

The way the factory is specified in the application context file and how it is associated to the RN-ary bean is clarified in Listing 3.

```
<bean
  class="com.componentcreator.metamodel.coremetamodel.scopefactories
  .ResponsibilityBeanConstructorScopeFactory"
  id=" responsibilityBeanConstructorScopeFactory " scope="singleton"></bean>

<bean class="com.componentcreator.metamodel.coremetamodel.relations.RNary"
  id="naryImpl"
  factory-bean="responsibilityBeanConstructorScopeFactory"
  factory-method="getInstanceMinimal">
    <constructor-arg>
      <value>"naryImpl"</value>
    </constructor-arg>
    <constructor-arg>
      <value>
        "com.componentcreator.metamodel.coremetamodel.relations.RNary"
      </value>
    </constructor-arg>
    <constructor-arg>
      <list>
        <value>
          com.componentcreator.metamodel.coremetamodel.domainsimpl.DC1
        </value>
        <value>
          com.componentcreator.metamodel.coremetamodel.domainsimpl.DC2
        </value>
        <value>
          com.componentcreator.metamodel.coremetamodel.domainsimpl.DC3
        </value>
      </list>
    </constructor-arg>
</bean>
```

Listing 3. Meta-model scope factory and relation beans specification in the application context file

5. Conclusions

The scope management problem was identified for meta-modeling purposes. The meta-modeling application domain as defined by CDMM approach is complex enough to study the problem. The concept of the scope management solution was also implemented in CDMM-F with the help of appropriate factories. Then the solution correctness was verified by appropriate test cases.

The paper initiates further research efforts in the field of scope management by creating solid fundamentals and presenting the skeleton of the solution for the next problems related to scope management. The mentioned problems are named and characterized briefly below.

Several interesting subjects for research are connected to meta-cardinality (the problem of defining the number of relation instances). This problem is very complex and is not supported by currently available technologies.

Another interesting problem which is new for meta-modeling and modeling disciplines is the problem of navigability of meta-model relationships named by the author meta-navigability. This problem is connected to traversing the directed graph of modeling language and impacts CDMM-F API significantly.

Also a complex problem of combining scopes may appear when several application context files that are based on different scope management concepts are used (reused) to constitute the whole meta-model application context. In the paper a simple case is implemented (see relationshipMinimal), but the concept of relationshipRedundant was also designed (but not verified yet) to support future solution of the scope combining problem.

Other challenging problems are connected to the so-called arity problem. The N-ary relationships can be handled in CDMM-F but in order to gain the full solution of the problem the meta-cardinality and meta-navigability problems must be completely solved and published.

References

[1] Akehurst D., Howells G., McDonald-Maier K., *Implementing associations: UML 2.0 to Java 5*, Softw Syst Model, Springer-Verlag 2006, DOI 10.1007/s10270-006-0020-1

[2] AspectJ framework, https://eclipse.org/aspectj/.

[3] Bildhauer D., *On the relationship between subsetting, redefinition and association specialization*, [in:] Proc. of the 9th Baltic Conference on Databases and Information Systems 2010, Riga, Latvia (07 2010).

[4] Bildhauer D., *Associations as First-class Elements*, Proceedings of the 2011 conference on Databases and Information Systems VI: Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, p. 108-121, IOS Press Amsterdam, The Netherlands, The Netherlands 2011.

[5] Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language User Guide*, Addison--Wesley, 2005.

[6] Object Management Group (2011), Business Process Model and Notation 2.0. http://www.omg.org/spec/BPMN/2.0/.

[7] Diskin Z, Easterbrook S., Dingel J., *Engineering Associations: From Models to Code and Back through Semantics*, [in:] *Objects, Components, Models and Patterns*, Volume 11, Lecture Notes in Business Information Processing, Proceedings of 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30–July 4, 2008, p 336-355.

[8] Feinerer I., *A Formal Treatment of UML Class Diagrams as an Efficient Method for Configuration Management*, PhD. dissertation, Vienna, March 2007.

[9] Feinerer I., Salzer G., *Numeric semantics of class diagrams with multiplicity and uniqueness constraints*, Software & Systems Modeling, 13(3), 2014, p. 1167-1187.

[10] Génova, G., Lloréns, J., Martínez, P., *The meaning of multiplicity of N-ary associations in UML*, Software and System Modeling 1(2), 2002, 86-97.

[11] Génova G., Ruiz del Castillo C., Llorens J., *Mapping UML Associations into Java Code*, Journal of Object Technology, Vol. 2, No. 5, September–October 2003.

[12] Kleppe A. G., Warmer J., Bast W., *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] Object Management Group (2006), Meta Object Facility (MOF) core specification version 2.0. http://www.omg.org/spec/MOF/2.0/.

[14] Roques P., SysML vs. UML 2: A Detailed Comparison, MoDELS'11 Tutorial, October 16th, Wellington, New Zealand, 2011.

[15] Sergievskiy M., *N-ary Relations of Association in Class Diagrams: Design Patterns*, (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 7, No. 2, 2016.

[16] Spring framework, https://spring.io/.

[17] Szlenk M., *Formal Semantics and Reasoning about UML Class Diagram*, 2006 International Conference on Dependability of Computer Systems, IEEE, 25–27 May 2006, p. 51-59, DOI: 10.1109/DEPCOS-RELCOMEX.2006.27.

[18] Tan H.B.K., Yang Y., Bian L., *Improving the Use of Multiplicity in UML Association*, Journal of Object Technology, Vol. 5, No. 6, July–August 2006.

[19] Object Management Group (2009), Unified Modeling Language (UML) superstructure version 2.2, http://www.omg.org/spec/UML/2.2/.

[20] Zabawa P., *Context-Driven Meta-Modeling Framework (CDMM-F) – Context Role*, Technical Transactions, 1-NP/2015, p. 105-114, DOI: 10.4467/2353737XCT.15.119.4156.

[21] Zabawa P., Stanuszek M., *Characteristics of Context-Driven Meta-Modeling Paradigm (CDMM-P)*, Technical Transactions of Cracow University of Technology, Fundamental Sciences, 3-NP (111), 2014, p. 123-134.