TECHNICAL TRANSACTIONS | **CZASOPISMO TECHNICZNE**
FUNDAMENTAL SCIENCES | NAUKI PODSTAWOWE
1-NP/2015

PIOTR ZABAWA*

CONTEXT-DRIVEN META-MODELING FRAMEWORK (CDMM-F) – CONTEXT ROLE

FRAMEWORK METAMODELOWANIA STEROWANEGO KONTEKSTEM (CDMM-F) - ROLA KONTEKSTU

A b s t r a c t

This paper introduces an implementation of the Context-Driven Meta-Modeling Paradigm (CDMM-P). This implementation is the proof of concept for the CDMM-P as it shows that the idea of the CDMM-P is feasible. The software system presented here takes the form of the Context-Driven Meta-Modeling Framework (CDMM-F). This framework plays the key role in the Context-Driven Meta-Modeling Technology (CDMM-T) dedicated to generating software system artifacts in a model-driven manner. In contrast to all contemporary approaches to meta-modeling, the meta-model is dynamically loaded from the application context file. In result, the framework has a self-organizing structure and the modeling language does not have a fixed hardcoded predefined structure. This structure, as well as the set of meta-model elements, plays the role of the parameter for the general modeling tool, a part of which is presented in this paper.

Keywords: *modeling language*, *meta-model*, *application context*, *Spring*, *aspect oriented programming*, *AspectJ*

S t r e s z c z e n i e

W artykule wprowadzono implementację Context-Driven Meta-Modeling Paradigm (CDMM-P). Implementacja ta stanowi potwierdzenie poprawności koncepcji CDMM-P, pokazując, że pomysł CDMM-P jest realizowalny. Przedstawiony tu system softwarowy to Context-Driven Meta-Modeling Framework (CDMM-F). Pełni on kluczową rolę w Context-Driven Meta-Modeling Technology (CDMM-T) przeznaczonej do generowania artefaktów systemów softwarowych w podejściu sterowanym modelem. W odróżnieniu od wszystkich współczesnych sposobów metamodelowania, metamodel jest ładowany dynamicznie z pliku kontekstu aplikacji. W efekcie framework ma samoorganizującą się structure, a język modelowania nie ma ustalonej i zapisanej w kodzie źródłowym predefiniowanej struktury. Struktura ta i zbiór elementów meta-modelu pełnią rolę parametru dla ogólnego narzędzia modelowania, którego część przedstawiono w niniejszym artykule.

Słowa kluczowe: *język modelowania*, *metamodel*, *kontekst aplikacji*, *Spring*, *programowanie aspektowe*, *AspectJ*

DOI: 10.4467/2353737XCT.15.119.4156

* Department of Physics, Mathematics and Computer Science, Cracow University of Technology; pzabawa@pk.edu.pl.

1. Introduction

There are several approaches to meta-modeling (defining modeling languages) known from scientific and industrial literature. The first approach assumes a fixed structure to the modeling language. In such a case, a particular modeling tool is dedicated to the application of one or several predefined modeling languages. This group of tools is represented by commercial solutions offered by IBM/Rational, Visual Paradigm, No Magic and many others. Some of these have free versions; however, these have limitations. There are also open-source solutions like, for example, Eclipse MDT. The second approach is focused just on meta-modeling [11]. This is represented by such commercial products as MagicDraw [10] or such open-source products as Eclipse EMF [1, 2, 5]. However, these tools are based on predefined and, as a consequence, limited elements that can be applied for the construction of the modeling language. Moreover, the result of the modeling process is a tool (appropriate plug-in) that is able to support the defined modeling language only.

It is worth noticing that there are also several modeling standards supported by the Object Management Group (OMG) [8]. These standards result from the long-term analyses of reasons for project failures which was initiated in the USA in the 1970s. This process led to the elaboration of dozens of software development methodologies from the 1980s and early 1990s. On the basis of these methodologies the Unified Modeling Language (UML) standard [8, 9] was born in Rational, and then was acquired by OMG for standardization in 1997. After that, such standards as the Meta-Object Facility (MOF) [7] and Model-Driven Architecture (MDA) [4] built on top of them were defined at the beginning of the twenty-first century. All of these standards and appropriate tools evolve very dynamically, but they introduced important limits to the possible approaches to constructing modeling languages from the very beginning. The most important factor is that the modeling language customization possibilities are very limited. The user of the modeling language can customize the UML language through UML Profiles but he/she is not able to introduce changes to the language structure. Another important limit introduced to the MOF+UML concept is the nature of the standardization process itself. This process is very long lasting, so when new technology is born, its potential user must wait many years for the acceptance of the new version of the UML standard enriched in the notions introduced by the new technology. This problem is known from such old technologies as Aspect Oriented Programming (AOP) [3] which was born in 2002 and was not yet accommodated by UML. It is also known, from such new technologies as Scala programming language [6], which introduced new relationship – traits. This feature of standards and the unacceptably long standardization process are in opposite to the main assumption of MDA – the ability of fast accommodating new technologies for the purpose of generating software project artifacts implemented in these new technologies.

In this paper, the new software framework is introduced. It is based on the Context-Driven Meta-Modeling Paradigm [12] – the new approach to meta-modeling. This approach can break the limits introduced into MOF&UML concepts. It results in the implementation of the general modeling tool that is able to support all current modeler requirements. In comparison to existing modeling tools, the new software framework provides extra opportunity to model in any user-defined modeling language. As a consequence, the modeler may reuse his/her previous modeling languages (both meta-model classes and relationships) and introduce any

structural changes to them according to the current demands. Another new option for the modeler is to customize the modeling language in the same manner as the one required when vertical OMG standards are applied for a particular application domain. The vertical standards being the superset of system of notions required by a modeler should be customized by the modeler to the needs of a particular software system. Moving this approach to the meta-modeling field, the modeler may customize an existing modeling language standard (such as UML) to any substructure of it. There are no tools on the market that offer such a possibility except from the tool which is introduced here.

2. Problem solution concept

In order to construct the implementation of dynamic loading of any required meta-model to the modeling tool, the right system of notions must be introduced. This problem is addressed in this section.. The concept of the solution to the problem is also presented here.

In contrast to the approaches presented above, the new approach introduced here results in the implementation of one general modeling tool for any modeling language. The user of the tool can define his/her own modeling language in the form of a graph composed of any kind of nodes and any kind of edges. This potentially unlimited language can be loaded to the tool from the meta-model file.

The solution concept is based on the observation that all contemporary known approaches to meta-modeling have a common disadvantage – namely, that the relationships between meta-model classes are represented in the form of hardcoded references. However, it is possible to represent them in the form of dynamically injected responsibilities. Thus, each meta-model can be decomposed into classes (pure entity classes) and relationship classes (entity relationship classes). They can remain unrelated on the level of the compiled source code. The meta-model can be created at run time from dynamically loaded meta-model structure representation. This meta-model structure takes the role of the modeling application context. As a result, the modeling framework structure can reflect the application context. Moreover, the relationships injected into meta-model classes can take the role of one of many possible responsibilities the meta-model classes can dynamically take – this is why the 'responsibility' term is frequently used for relationships in the rest of the paper.

3. Tool implementation

In this section, the implementation of the context-driven meta-modeling tool is presented. The tool is dedicated to construction and usage of a model which is compatible to the particular meta-model.

The implemented tool has the form of the framework. It is composed of fixed elements belonging to the framework, such as configuration, API classes and the distinguished meta-model root element. However, the framework is able to construct itself from the meta-model specification. The most natural meta-model specification is the XML file containing this model. This file takes the role of the application context.

In order to implement the framework, some research was performed. The most important programming language for the framework is Java, as all or almost all contemporary modeling tools are implemented in Java or at least expose their APIs for this programming language. The most attractive technology for this programming language is the Spring framework which supports the required dynamic injections and application context. But the Spring alone was recognized as insufficient. The lacking feature – the ability to introduce dynamic structural changes into classes was offered by AspectJ. Samples illustrating the method of combination of these technologies are presented in Section 4.

All these technologies made it possible to implement the framework briefly characterized in Section 2.

4. Testing

Model-driven tools are difficult to test. The reason for this is that they are able to load a potentially infinite number of different data structures, so they cannot be tested against each of these data structure. However, in the context of the problem stated in the paper, the purpose of testing is not to perform commercial verification of the tool, but to focus on verification of the correctness of the tool implementation against the ability of loading the meta-model dynamically. Therefore, this section is dedicated to verifying whether the tool makes possible to:
- load several different meta-models,
- create models compatible to each meta-model through the tool's API,
- navigate through the model structure and acquire model elements.

In order to check each of the functional features presented above, the following steps were performed:
- testing meta-models were created in the form of UML diagrams,
- testing simple XML graphs of these meta-models were created,
- testing Spring application contexts XML files for the framework for the above graphs were created,
- test scenarios for:
  - creating models,
  - navigating through each model structure,
  - getting model elements and checking if they are the same as created before.

In order to illustrate the testing approach, which is done in Subsection 4.1, an example meta-model was chosen. The test scenario for this meta-model is also presented below to display the nature of the client's code. Moreover, the different meta-model structures that were the subject of testing activities are shown below in Subsection 4.1.

4.1. Different meta-model examples

Several different meta-model UML diagrams that are the subject of testing are presented below. They are very simple as they were intended to test meta-model relationship class and meta-model entity class features which are especially difficult for implementation.

The set of meta-models presented in Figures 1–6 is limited to chains and trees only. All the meta-models shown in these figures are elaborated to test the possibility of building any meta-model in the form of a graph and should be interpreted as the proof of the concept. Therefore, the ability of constructing meta-models from meta-model building blocks is tested through the meta-models presented below. This approach solves the typical problem also known from modeling tools – how to test the correctness of the potentially infinite number of models. However, the problem known from modeling discipline is moved here to meta-modeling field.

Some notational conventions should be explained here:

- green rectangles – class under test,
- red associations – responsibility under test,
- stereotypes in red – type of a responsibility (the relation class),
- stereotypes in black – logical exclusions of a particular types of responsibilities, which are necessary to test all cases,
- j, k, l postfixes in class names – particular domain classes that fulfill the following logical condition: $i \neq j$ and $i \neq k$ and $j \neq k$,
- m, n postfixes in stereotypes – particular responsibilities that fulfill the following logical condition: $m \neq n$,

The particular classes: *R* for root class of the meta-model and *Di* for domain classes are always tested in combination with their responsibilities – this is why associations going out from the green rectangles are always red according to the convention rules.

The first group of Figures 1–4 shows meta-models that share the same responsibilities between classes involved in testing process. Figure 1 presents a meta-model for testing the correctness of enrichment of root meta-model class (R) by the same responsibilities (Rm).

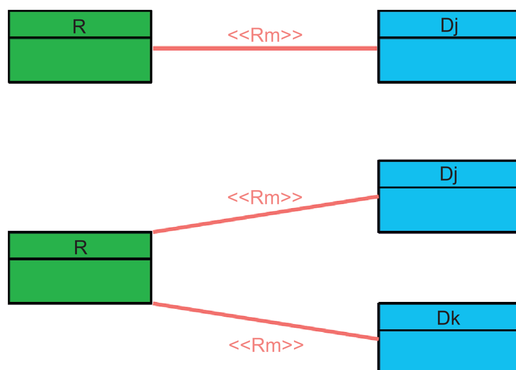

Fig. 1. The same responsibility (Rm) originated in the Root (R) meta-model element

Enrichment of meta-model domain classes (Dj) in the same responsibilities (Rm) is presented in Fig. 2.

Figure 3 is focused on the meta-model which is dedicated to testing the transitiveness of the enrichment of both the root meta-model class (R) and the domain meta-model class (Dj) in the same responsibilities (Rm).

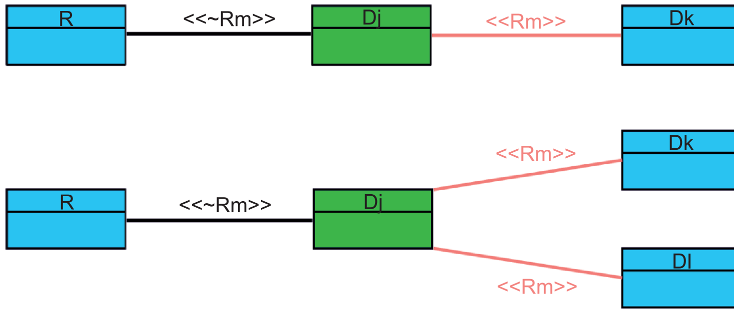

Fig. 2. The same responsibility (Rm) originated in the Domain (Dj) meta-model element

Fig. 3. The same responsibility (Rm) originated in both Root (R) and Domain (Dj) meta-model elements

Similarly, Fig. 4 presents a meta-model used to test the correctness of the transitive enrichment of the two different meta-model domain classes (Dj, Dk) in the same responsibilities (Rm).

Fig. 4. The same responsibility (Rm) originated in different Domain meta-model elements (Dj, Dk)

The Figs. 5–6, on the other hand, contain meta-models that test the correctness of combining different responsibilities injected into same classes. The meta-model root class (R) is enriched in different responsibilities (Rm, Rn) in Fig. 5.

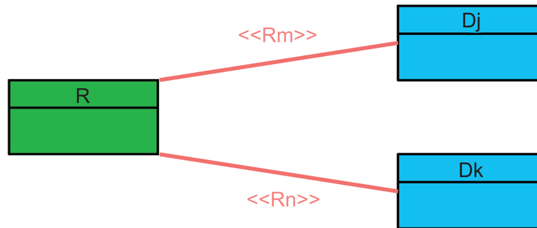

Fig. 5. Different responsibilities (Rm, Rn) originated in the Root (R) meta-model element

Figure 6 presents enrichment of the meta-model domain (Dj) class in different responsibilities (Rm, Rn).

All meta-models presented above were tested by unit tests of each class. The meta-models were also tested by scenario tests. Each scenario test has two parts: one for the

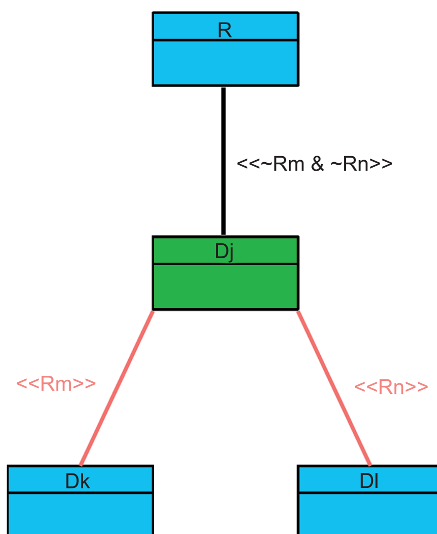

Fig. 6. Different responsibilities (Rm, Rn) originated in the Domain (Dj) meta-model element

model construction and one for the model traversal. Several models (meta-model instances) were created to execute such a particular scenario test.

The framework passed all tests explained above.

4.2. Sample meta-model testing

In this subsection, the sample test scenario is presented for the meta-model shown in Figure 5. However, first the Spring and AspectJ application-context file which constitutes the definition of the modeling language (meta-model) is shown in Fig. 7.

The test scenario for constructing a model which is an instance of the sample meta-model from Fig. 7 and then for traversing the model are presented in Fig. 8.

Such tests were created for all meta-models which were taken into account during the activity of tests strategy planning. They confirmed the correctness of the approach and feasibility of the framework implementation based on the concept presented in the paper.

```xml
<beans xmlns:aop="http://www.springframework.org/schema/aop"
xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:int-security="http://www.springframework.org/schema/integration/security"
xmlns:sec="http://www.springframework.org/schema/security"
xsi:schemaLocation="http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.0.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <import resource="../metamodel-core-context.xml"/>
  <import resource="../appContextNamedElement.xml"/>

  <!--  API  -->
  <bean class="com.componentcreator.metamodel.coremetamodel.apidynamic.APIdynamic"
   id="coreApiDynamic" scope="singleton"/>

  <!--  Root  -->
  <bean class="com.componentcreator.metamodel.coremetamodel.root.R" id="root"
   scope="singleton"/>

  <!--  Domain  -->
  <bean class="com.componentcreator.metamodel.coremetamodel.domainsimpl.Dj" id="class"
   scope="prototype"/>
  <bean class="com.componentcreator.metamodel.coremetamodel.domainsimpl.Dk" id="role"
   scope="prototype"/>

  <!--  Responsibility implementations  -->
  <bean class="com.componentcreator.metamodel.coremetamodel.responsibilitiesimpl.Rm"
   id="rmImplForR">
    <constructor-arg>
      <value>
        com.componentcreator.metamodel.coremetamodel.domainsimpl.Dj
      </value>
    </constructor-arg>
  </bean>

  <bean class="com.componentcreator.metamodel.coremetamodel.responsibilitiesimpl.Rn"
   id="rnImplForR">
    <constructor-arg>
      <value>
        com.componentcreator.metamodel.coremetamodel.domainsimpl.Dk
      </value>
    </constructor-arg>
  </bean>

  <!--  Responsibility injections  -->
  <aop:config>
    <aop:aspect id="relationship" ref="relationshipAspect">
      <aop:declare-parents
       types-matching="com.componentcreator.metamodel.coremetamodel.root.R"
       implement-interface="com.componentcreator.metamodel.coremetamodel.responsibilities.IRm"
       delegate-ref="rmImplForR"/>

      <aop:declare-parents
       types-matching="com.componentcreator.metamodel.coremetamodel.root.R"
       implement-interface="com.componentcreator.metamodel.coremetamodel.responsibilities.IRn"
       delegate-ref="rnImplForR"/>
    </aop:aspect>
  </aop:config>
</beans>
```

Fig. 7. Definition of the sample meta-model in the form of Spring and AspectJ application context

```java
@Test
public void test() throws Throwable {

  // obtain initiated interface object
  ICoreMetamodelAPIdynamic ifc =
    Configuration.getCoreMetamodelAPIdynamic(APP_CFG_FNAME);

  // get access to meta-model root singleton
  RootMetamodelCore root = ifc.getRoot();

  // create meta-model domain classes and add them to the root
  Dj dj1 = (Dj) ifc.getDomain(Dj.class, "dj1");
  ((Name)cls1).setName("dj 1");
  Dj dj2 = (Dj) ifc.getDomain(Dj.class, "dj2");
  ((Name)cls2).setName("dj 2");
  Dj dj3 = (Dj) ifc.getDomain(Dj.class, "dj3");
  ((Name)cls3).setName("dj 3");

  BaseMetamodelCoreProxy rootRmAccessorProxy =
    ifc.getAccessor(RootMetamodelCore.class, IRm.class);

  rootRmAccessorProxy.add(root, dj1);
  rootRmAccessorProxy.add(root, dj2);
  rootRmAccessorProxy.add(root, dj3);

  Dk dk = (Dk) ifc.getDomain(Dk.class, "dk");
  ((Name)att).setName("dk 0");

  BaseMetamodelCoreProxy rootRnAccessorProxy =
    ifc.getAccessor(RootMetamodelCore.class, IRn.class);

  rootRnAccessorProxy.add(root, dk);

  System.out.println("Djs count = "+rootRmAccessorProxy.count(root));
  for(int i=0; i<rootRmAccessorProxy.count(root); i++){
    Dj dj = (Dj) rootRmAccessorProxy.get(root, i);
    System.out.println("Dj's name = "+((Name) dj).getName());
  }
  System.out.println();

  Dk dk = (Dk) rootRnAccessorProxy.get(root);
  System.out.println("Dk's name = "+((Name) dk).getName());

  ifc.close();
}
```

Fig. 8. Framework implementation scenario test for the sample model created for a particular meta-model

5. Conclusions

In this paper, the materialization of an idea based on the CDMM-P paradigm was presented. It was shown that the CDMM-F framework being the implementation of the CDMM-P can be realized in the form of a self-organizing software system the structure of which is defined by the application context of this system. The set of risks specific for such an approach was identified and addressed by test cases. The CDMM-F was implemented and tested against these risks. Thus, the proof of the concept for both CDMM-P and CDMM-F was made and presented in the paper.

There is a lot of possible investigative directions that are worthy of taking in future. However, the most important aim at the moment is to elaborate several API versions for the framework. Two such APIs have been defined and tested so far. These require information about the meta-model structure for the purpose of navigation through the model from the test

or framework's client source code. In the next step, the reflective API should be elaborated and tested. As the result of this step the client's source code should become independent of the meta-model structure.

References

[1] Kalnins A., Vilitis O., Celms E., Kalnina E., Sostaks A., Barzdins J., *Building Tools by Model Transformations in Eclipse*, Proceedings of DSM 2007 workshop of OOPSLA 2007, Montreal, Canada, Jyvaskyla University Printing House, p. 194-207.

[2] Kern H, Kühne S., *Model Interchange between ARIS and Eclipse EMF*, 7th ooPSLA Workshop on Domain-Specific Modeling, Momtreal, Canada, Oktober 2007.

[3] Kiczales G., Lamping J., Mehdhekar A., Maeda C., Lopes C.V., Loingtier J., Irwin J., *Aspect--Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241, June 1997.

[4] 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.

[5] [Krahn H., Rumpe B., Völkel S., *Efficient Editor Generation for Compositional DSLs in Eclipse*, Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling DSM' 07, Montreal, Quebec, Canada, Technical Report TR-38, Jyväskylä University, Finland 2007.

[6] Odersky M., Micheloud S., Mihaylov N., Schinz M., Stenman E., Zenger M., et al., *An overview of the Scala programming language*, EPFL, Lausanne, Switzerland 2004.

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

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

[9] Selic B., *A systematic approach to domain-specific language design using UML*, Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on, 0:2–9, 2007.

[10] Silingas D., Vitiutinas R., Armonas A., Nemuraite L., *Domain-specific Modeling Environment Based on UML Profiles*, [in:] *Information Technologies 2009*, Proceedings of the 15th Conference on Information and Software Technologies, IT 2009, Kaunas, Lituania, April 23–24, Kaunas University of Technology, Technologija, 2009, p. 167-177.

[11] Sprinkle J., Mernik M., Tolvanen J.-P., Spinellis D., *What Kinds of Nails Need a Domain-Specific Hammer?*, IEEE Software, Guest Editors' Introduction: Domain Specific Modelling, 26(4): 15–18, July/August 2009.

[12] Zabawa P., Stanuszek M., *Characteristics of Context-Driven Meta-Modeling Paradigm (CDMM-P)*, Technical Transactions, 3-NP/2014, p. 123-134.