Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 991–996

DOI: 10.15439/2015F247
ACSIS, Vol. 5



# Generating Database Access Code From Domain Models

Nassima Yamouni Khelifi*†, Michał Śmiałek*, Rachida Mekki†
*Warsaw University of Technology, Poland
Email:{nassima, smialek}@iem.pw.edu.pl
†University of Sciences and Technologies of Oran-Mohamed Boudiaf-, Algeria
Email:{nassima.yamounikhelifi, rachida.mekki}@univ-usto.dz

***Abstract*—Automatic processing of requirements (e.g. to generate code) remains a challenge in contemporary software development. Requirements are still treated as secondary artifacts by software developers, as they are written in natural languages which causes ambiguity. In this paper, we present an approach to generate working code from requirements through applying precisely formulated domain models. As the source, we use the Requirements Specification Language (RSL) which is a precise constrained language, based on a central domain model composed of domain notions. These notions are linked from use case scenarios and create a form of a 'wiki'. Notions are graphically visualized in RSL, and resemble UML classes with attributes. Notions can be used in phrases that can represent various operations used within use case scenarios. In our approach we introduce model transformation algorithms that allow to generate database access code associated with operations to persist (store, retrieve) data in a database system. To focus our work, we present code generated for Hibernate which is an object relational mapping framework.**

***Index Terms*—model-driven requirements engineering, model transformations, database access, metamodelling**

## I. Introduction and Background

TYPICAL requirements specifications in contemporary software projects use natural language, possibly with some elements of modelling. This poses a significant challenge for approaches to automate the process of turning requirements into working code. A prominent field of research that aims at changing this situation is Model-Driven Requirements Engineering (MDRE) [1]. In MDRE, requirements are expressed as models, often by using the Unified Modelling Language (UML) [2]. Such models are intended to be comprehensible to both software developers and end-users. This comprehension is most often assured by introducing a comprehensive vocabulary of the problem domain in the form of a domain model. This allows to use the techniques of domain engineering [3]. Most often, UML class diagrams are used. Classes represent domain objects (noun phrases) with associated atomic attributes (also nouns) and operations (verb phrases). What is more, class models can define relationships between domain objects (notions), thus allowing to build a certain semantic network of related notions.

Still, UML does not offer any means to associate domain models with the remaining models and it has no precise syntax for textual elements like scenarios. Thus, in this paper we will use a language dedicated to formulating precise requirements models, called the the Requirements Specification Language (RSL) [4], [5]. The syntax of RSL is defined through a metamodel using the Meta-Object Facility (MOF) meta-language [6]. An important feature of RSL is that it introduces precise (hyper-)linking of domain models within textually expressed requirements units.

To link textual requirements with domain models, we need to represent requirements by using notions from the domain model in a consistent way [7]. This means – for instance – that use case [8] scenarios should be composed of links to domain model elements – noun phrases and verb phrases, contained in a central domain model. In RSL, this is done very consistently: all the scenario sentences are in fact links to phrases contained in the domain model. This makes the whole RSL-based specification resemble a 'wiki' system (see Fig. 8 for an illustration) with consistent use of hyperlinks to specific vocabulary notions.

This consistency of RSL allowed Śmiałek et al. [9], [10], [11] to formulate formal translational rules for generating code from requirements models (use cases and their scenarios) down to UML design models and Java code. The resulting code follows the Model-View-Presenter (MVP) [12] architectural pattern. The rules focus on generating full code for the View and the Presenter layers, and method stubs for the Model layer. They also permit to generate Data Transfer Objects, that facilitate control flow between the three layers. Unlike for other approaches in MDRE, these rules allow for a fully automatic translation from high-level requirements models (use cases, scenarios, domain vocabularies) down to fully operational code.

The above approach with RSL still lacks rules for generating code associated with persistence operations at the Model layer. Thus, in the current work, we concentrate on defining and implementing rules for generating database access code that is responsible for processing and persisting data. We want this code to be consistent with that for the View and Presenter layers as introduced in the previous paragraph. For this, we will use the Hibernate framework [13] which is an Object Relational Mapping (ORM) that allows for mapping Java Classes (DTOs) to database tables, and for managing data with the Data Access Object (DAO) design pattern [14]. We give rules to generate general Create/Read/Update/Delete (CRUD) operations for persisting Java objects in database tables.

In the following sections we introduce our approach which

978-83-60810-66-8/$25.00

is consistent with typical model transformation approaches of Model-Driven Software Development [15], [16]. In Section II we briefly present the domain vocabulary part of RSL which is the source language for our transformations. In Section III we present the transformation itself: selected rules and algorithms that implement them. The rules define the translation from RSL to UML with inserted operational Java code. The algorithms are expressed in a graphical transformation language called MOLA (MOdel Transformation LAnguage) [17], [18]. The last section presents conclusions stemming from implementing the presented algorithms within the RSL environment called ReDSeeDS [19] and using a UML tool (Enterprise Architect - EA, from Sparx Systems, sparxsystems.eu) to visualise the generated UML models and to generate the final Java code.

## II. Requirements Specification Language: Overview of the Domain Notions Part

Requirements Specification Language (RSL) is a semi-formal language for specifying precise requirements [5], [20]. The fundamental idea behind RSL is separation of concerns: separating description of the system's application behavior and the description of the system's problem domain. The behavior of the system is described with ***use cases*** and their ***textual scenarios*** written in constrained natural language. The domain specification is defined using ***notions*** (words). Phrases in scenario sentences constitute the **application logic**, and are linked to notions that constitute the **domain logic**. Notions can be composed of both "nouns" and "verbs".

The RSL's grammar is defined formally through a metamodel written in MOF which is standardized by the Object Management Group (OMG, www.omg.org) [6]. The primary goal of MOF is to allow metamodels to be defined using basic class model syntax (classes with attributes and relationships). The full description of RSL consists of the abstract syntax (i.e. the metamodel), the concrete syntax (definitions of visual language elements), and informally specified semantics (natural language descriptions similar to those in the UML specification [21]). It can be found in a comprehensive report from the ReDSeeDS project [4] which uses the 'Complete' (CMOF) dialect of MOF. A variant that uses the 'Essential' dialect of MOF (EMOF) is presented in the book by Śmiałek and Nowakowski [11]. The book also presents a comprehensive approach to defining RSL's semantics in a formal, translational way.

Figure 1 presents a small excerpt from the RSL metamodel pertaining to notions and their relationships. As we can notice, in contrast to UML, **Notions** contain *notionAttribute*s that are also **Notions**. Attributes can be distinguished from regular notions by their possession of an **AttributeDataType**. The possible data types include:

- "text": string-like textual description;
- "number": an integer number;
- "floating number": a number with a possible decimal;
- "truefalse": a boolean value;
- "date": a value containing date or time;

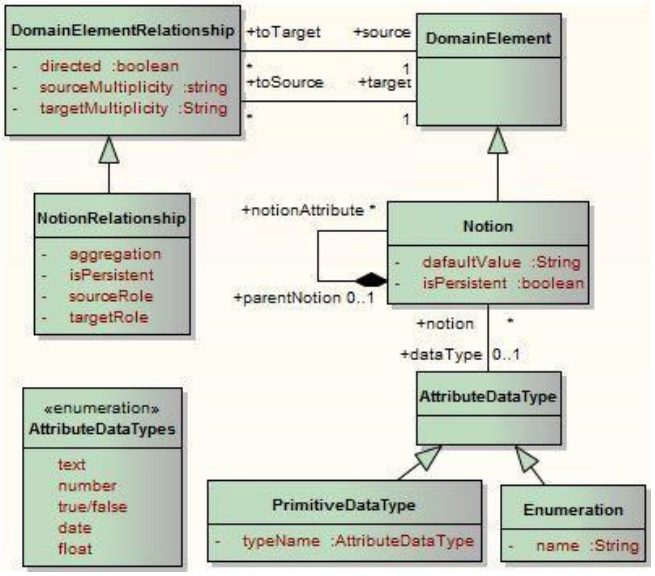



Fig. 1. Part of the RSL's metamodel for notions and their relationships

It can be noted that to represent requirements for software we need to define domain elements in two areas: 1) the problem (business) domain, and 2) the application domain. The problem domain is the actual reality that the software supports. It is stable and quite independent from the application to be built and changes when the reality changes. The application domain changes when the application changes and contains various parameters and user interface elements.

Here we will concentrate on the the problem domain that consists of domain notions (e.g. "book", "publisher", "author"...etc), and their attributes (e.g. "title", "name", "address" ..etc.). The concrete notation for domain notions in RSL is similar to that found in UML class models. An example is shown in Figure 2. The basic type of **Notion** in RSL is the *"Concept"*. Graphically, it resembles a UML class. The second type of **Notion** is the *"Attribute"*. We can notice that unlike in UML, attributes are not contained graphically in the *"Concepts"*. This is a feature of RSL that facilitates sharing attributes between various notions, especially of the 'view' type (see below).

*Concepts* and *attributes* are presented as rectangles adorned with appropriate tags. *Attribute* elements additionally contain information about the data type included in brackets. We should note that the data types are not limited and can be easily extended in the metamodel, depending on the problem domain (e.g. with sound, graphics, etc.), but should be defined in advance prior to developing a transformation.

Other notion types in RSL are called "**Data views**" and are divided into two kinds: "Simple data view", and "List data view". Data views point to sets of attributes. "Simple data views" serve to present single instances of combined attributes. "List data views", are used to present lists, containing many instances. "Data views" and other RSL elements are not treated in detail in this paper, for more details please refer to the book by Śmiałek an Nowakowski [11].

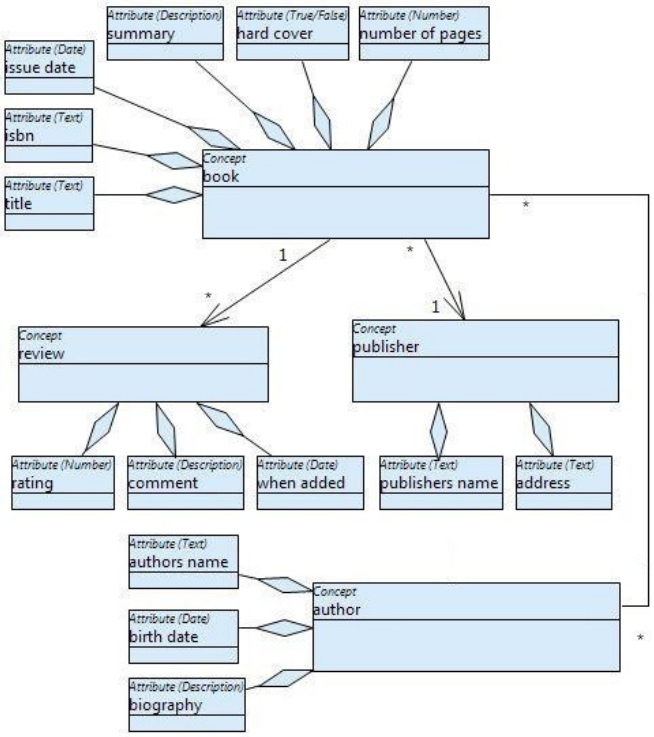



Fig. 2. Example of notions and their relationships in RSL's concrete notation

The different types of domain elements are connected by "relationships". The first kind of relationship is denoted similarly to associations in UML. It relates two concepts, and can have multiplicities. The second type of relationship is containment of attributes within concepts, where the diamond is placed on the concept side. The notation of containment relationship is also taken from UML and resembles aggregations.

In RSL, we can define all kinds of problem domains (e.g. Physics, Aeronautic, Finance, etc.). Figure 2 illustrates an example of RSL model for the "Library Management" problem domain. In this domain we have four elementary **concepts**: book, author, publisher and reviewers. Each concept can hold a number of attributes (shown via the containment relationship). For example, the "book" concept has attributes like: ISBN, title, number of pages, issue date, etc. The four concepts are connected via associations, that have appropriate multiplicities (one-to-many, many-to-one, many-to-many, ... etc.).

## III. Generating Database Access Code From RSL

This section consists of two parts. In the first part we introduce selected transformation rules that constitute translational semantics for RSL's domain vocabulary constructs, where the target languages are UML and Java. In the second part we present algorithms expressed in MOLA that implement these transformation rules.

### A. Transformation Rules

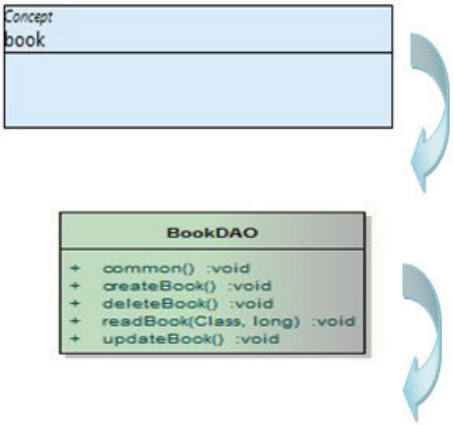
In order to generate database access code from RSL we need to explain its semantics concerning this aspect. For this, we will use a pragmatic translational approach [22], which is based on translating a "Source language" to a "Target language" which has already well-defined semantics [23], [24]. In our case, the source language is "RSL", and the target languages are "UML" and "Java". The reason behind choosing Java and UML as target languages, is that they are widely used and understood by a large community of software developers.

We will define a set of translation rules to generate data base access code from RSL domain models. For this, we will use the Data Access Object (DAO) design pattern [14], that implements the access mechanism required to work with the data source (e.g. Relational Database Management Systems like: MySQL, Oracle, PostgreSQL, etc.). We will apply this pattern in the context of the Hibernate framework, which is an open source Object-Relational-Mapping (ORM) that allows for persisting and storing data in a database [13], via CRUD (Create/Read/Update/Delete) operations within the DAO classes.

Hibernate maps Java persistent classes to database tables, and from Java data types to SQL data types, and provides data query and retrieval facilities. The Java persistent classes are Data Transfer Objects (DTOs) [25] or POJOs (Plain Old Java Objects). Hibernate uses XML files for mapping Java classes into tables, which are: Hibernate Configuration file, and Hibernate Mapping files. The Hibernate Configuration file contains all the required information related to the database, and other related parameters. The Hibernate Mapping files should be generated for each DTO class, and should contain information related to associations between database tables.

In the following, we will present two translation rules that defines semantics of RSL domain models in terms of DAO classes, and Hibernate classes. Other rules, that allow to generate Data Transfer objects and configuration files are out of scope of this work.

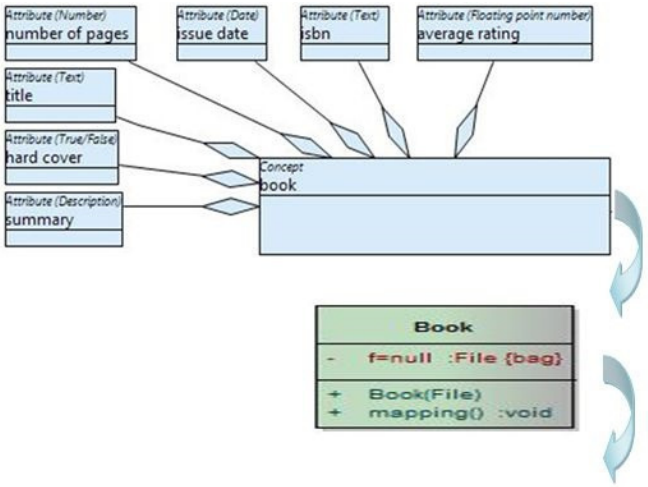
- **Rule R1:** Every "Concept" in the RSL domain model is translated into a DAO class. The name of the class is derived from the Concept's name and concatened with the "DAO" string. Each class contains four operations: "Create", "Read", "Update", and "Delete", plus the "common" operation. The operations' names are derived from the name of one the CRUD operations, concatenated with the name of the "Concept". Figure 3 provides a description of Rule R1. In this example we can see that the "book" concept in RSL, is first translated into a UML class named "BookDAO" with four CRUD operations (createBook, readBook, updateBook, and deleteBook), plus the "common" operation. The UML class is then translated into a "BookDAO" Java class. The "common" operation (lines 11-17) initialises various variables required by Hibernate. In lines 19-26, we show a fragment of code for the "readBook" operation, which takes two parameters as input: the Class (i.e. the DTO class) and the identifier (ID), and reads the proper object using the "load" method of the 'session' object.




```
1  public class BookDAO {
2
3     public BookDAO(){
4
5     }
6     private static Session session;
7     private static Transaction tx;
8
9     Book book=new Book();
10
11    public void common(){
12       Configuration cfg=new Configuration();
13       cfg.configure();
14       SessionFactory sf=cfg.buildSessionFactory();
15       session=sf.openSession();
16       tx=session.beginTransaction();
17    }
18
19    public void readBook(Class clazz, long id){
20       Object obj;
21       try{
22          common();
23          obj= session.load(clazz, id);
24          (...)
25       }
26    }
27    (...)   /*The other CRUD operations*/
28 }
```

Fig. 3. Generation of the "BookDAO" class with CRUD operations

- **Rule R2:** Every "Concept" with its attributes in the RSL domain model is translated into a mapping class. The class' name is derived from the concept's name. Each mapping class contains a constructor with a parameter, and the "mapping" operation. Rule R2 is illustrated in Figure 4. The example shows a fragment of the code for the "mapping" operation. This operation is responsible for mapping of the Book class into the book table which exists already in a database, this table of course has a unique identifier (ID). Each attribute is mapped into a column in a database table (see lines 27-31).




```
1  public class Book {
2
3     private File f=null;
4
5     public Book(File f){
6        this.f=f;
7     }
8
9     public void mapping(){
10       try{
11          DocumentBuilderFactory dbf;
12          dbf= DocumentBuilderFactory.newInstance();
13          DocumentBuilder db;
14          db= dbf.newDocumentBuilder();
15          Document doc = db.newDocument();
16          Element r, cl, id, col, pr;
17          /*r is the rootElement*/
18          r= doc.createElement("hibernate-mapping");
19          cl = doc.createElement("class");
20          cl.setAttribute("name", "book");
21          cl.setAttribute("table", "Book");
22          id = doc.createElement("id");
23          id.setAttribute("name", "bookID");
24          id.setAttribute("type","long");
25          cl.appendChild(id);
26          (...)
27          col= doc.createElement("column");
28          pr=doc.createElement("property");
29          pr.setAttribute("name", "title");
30          pr.setAttribute("type", "STRING");
31          col.setAttribute("name", "Title");
32          pr.appendChild(col);
33          cl.appendChild(pr);
34          (...)
35       }
36    }
```

Fig. 4. Generation of Hibernate mapping code for the "Book" concept

### B. Transformation Algorithm

The presented rules define the expected outcome of a transformation from RSL's domain models to database access code. To implement these rules, we have developed appropriate transformation algorithms. Here we introduce them by using MOLA, a language dedicated to model transformations, developed at the University of Latvia [17], [18]. The MOLA notation is graphical, as illustrated in Figures 5-7 and is based on graph-grammar rules that are defined in the context of UML activity diagrams.

MOLA is a procedural language, and Figure 5 shows a sequence of 6 procedure calls. This forms the main procedure of our algorithm. After cleaning-up the target model, the procedure creates a general package structure. Then, it creates appropriate Data Transfer Objects and Data Access Objects. Finally, it generates Hibernate mappings and configuration files.

In this current short introduction to the transformation algorithm we will concentrate on creating the DAOs (cf. Rule 1 in the previous section). The appropriate procedure for

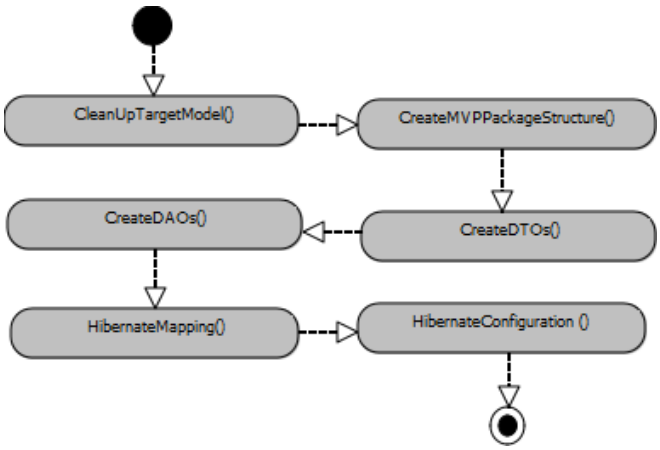



Fig. 5. Main steps of the algorithm expressed in MOLA

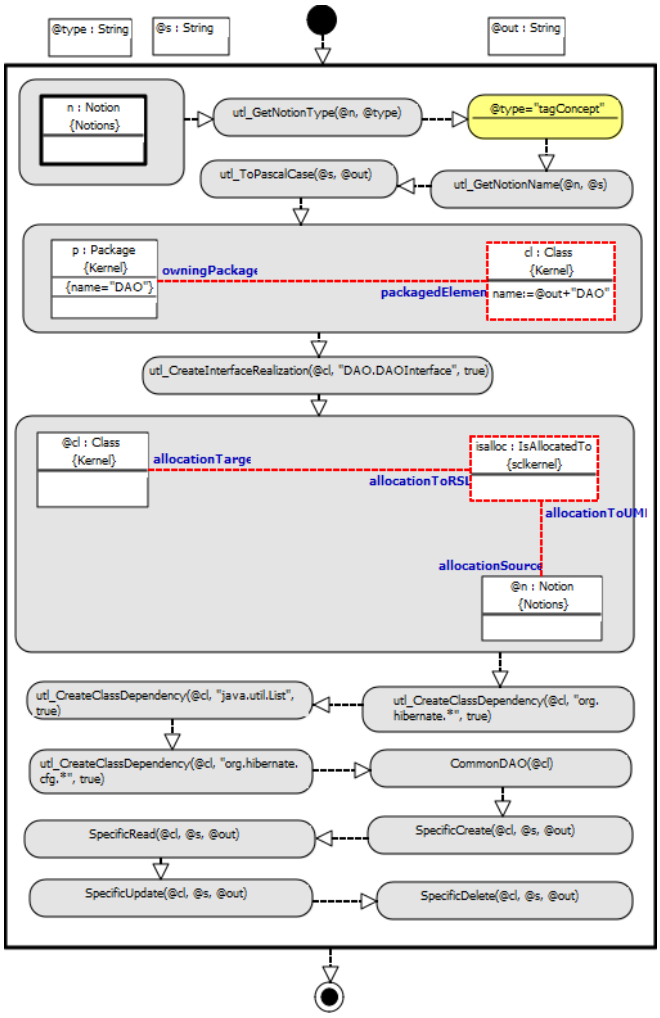



Fig. 6. Procedure for creating DAOs ("CreateDAO") expressed in MOLA

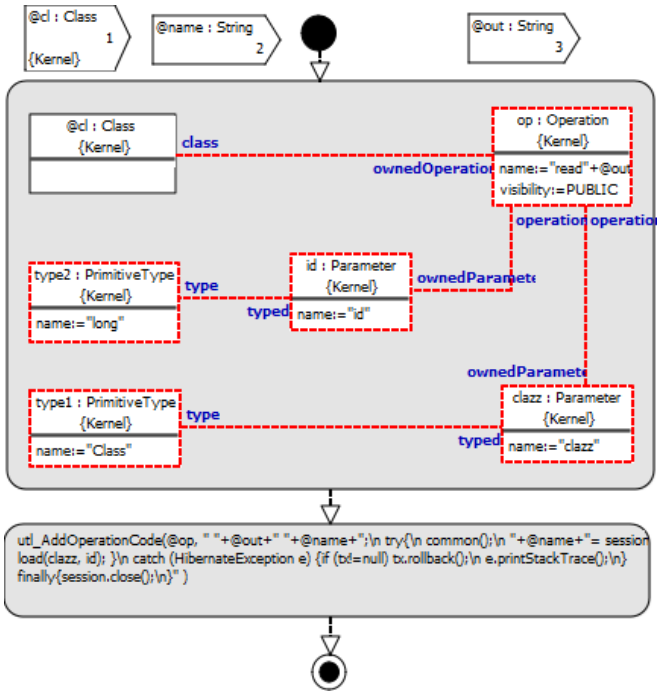



Fig. 7. MOLA rule for "SpecificRead" operation

implementing this is presented in Figure 6. The procedure iterates over all the **Notion** objects (refer to Figure 1) found in the source model (see the object 'n:Notion' in the top-left of the figure). Note that in MOLA, loops are represented by rectangles with thick borders which encompass all the actions to be performed within them. For each notion, the loop determines the notion's type, and if it is of type 'concept' it creates a new **Class** type object (see 'cl: Class'). This object is placed inside an existing **Package** (see 'p:Package') named 'DAO'. Additionally, the newly created class is related through a realisation relationship to a DAO interface object and through a mapping relationship (see 'isalloc:IsAllocatedTo') – to the original notion object. It is also completed by generating some relations to other elements that allow for importing certain elements specific to the Hibernate framework.

The main loop concludes by generating four CRUD operations within the new class. One of these procedures is illustrated in Figure 7. This procedure takes two parameters as input – the class and its name. In the class it creates an operation (see 'op:Operation' with two parameters (see 'id:Parameter' and 'clazz:Parameter). As we can see, the main MOLA rule presented in Figure 7 shows a configuration of 5 objects to be created (one operation, two parameters and two primitive types). This configuration is consistent with the UML's metamodel which can be found in its official specification [21].

Note that the generated operation matches code illustrated in line 19 in Figure 3. In addition, by using a simple text processing statement (see 'utl_AddOperationCode'), the procedure appends the method of the operation with the code like in lines 20-24 in the same figure.

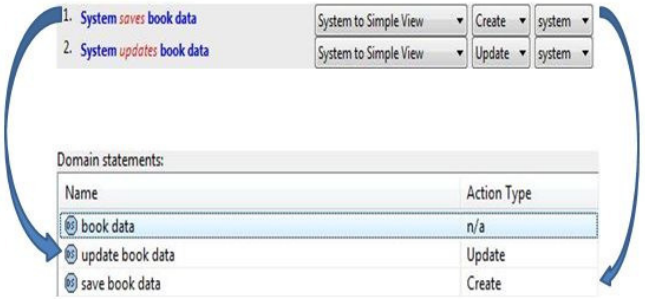



Fig. 8. Linking domain elements from scenarios

## IV. Conclusion and Future Work

To validate the presented approach we have implemented the above introduced algorithm within the framework of the the ReDSeeDS (Requirements-Driven Software Development System) tool suite [19] (see: www.redseeds.eu). The tool offers a full Model-Driven Software Development (MDSD) life cycle: i.e. from requirements to UML models down to Java code. In our application, the source RSL domain model with its notions were specified using the ReDseeDS editor (see Fig. 2). After writing and testing the transformation rules using the MOLA environment (see: mola.mii.lu.lv) we have integrated them into the ReDSeeDS tool. The MOLA transformation has been compiled and made available within the ReDSeeDS transformation menu. More details about this integration process can be found in the book by Śmiałek and Nowakowski [11]. The transformations generate UML classes with embedded method code. These UML classes are then handled by the UML tool (Enterprise Architect) and its standard code generator, to produce Java code.

The resulting transformation from RSL to Hibernate ORM produced good quality, consistent code that could be used directly to implement the data access layer. In current work we did not approach at generating the database tables, but it can be noted that our solution shows that a complete persistence layer could be generated automatically from domain models in RSL. This is an interesting research direction and we treat this as future work.

Moreover, our future work will also include integration of the persistence layer within the Model-View-Presenter (MVP) [12] architectural pattern. The upper layers (View and Presenter) of a complete software system can be fully generated from RSL models as shown by Śmiałek et al. [10], [11], [26]. This approach does not generate any contents of the Model layer. However, we can approach at generating meaningful code for the CRUD operations. Such operations are used frequently in RSL scenarios, as illustrated in Figure 8. Links between scenario sentences (e.g. 'System saves book data') and domain statements (e.g. 'save book data') can be transformed into calls from the Presenter layer to the Model layer. The RSL environment allows for determining the type of operation (e.g. Create or Update) and thus appropriate database access code can be provided in proper places.

## References


[1] B. Berenbach, "A 25 year retrospective on model-driven requirements engineering," in *IEEE Model-Driven Requirements Engineering Workshop (MoDRE'12)*, 2012, pp. 87–91, DOI: 10.1109/MoDRE.2012.6360078.

[2] B. A. Berenbach, "Comparison of UML and text based requirements engineering," in *Companion 19th OOPSLA Conference*, 2004, pp. 247–252, DOI: 10.1145/1028664.1028766.

[3] D. Bjôrner, "Rôle of domain engineering in software development. why current requirements engineering is flawed!" *Lecture Notes in Computer Science*, vol. 5947, pp. 2–34, 2010, DOI: 10.1007/978-3-642-11486-1_2.

[4] H. Kaindl, M. Smialek, P. Wagner *et al.*, "Requirements specification language definition," ReDSeeDS Project, Project Deliverable D2.4.2, 2009, www.redseeds.eu.

[5] M. Śmiałek, A. Ambroziewicz, J. Bojarski, W. Nowakowski, and T. Straszak, "Introducing a unified requirements specification language," in *Proc. CEE-SET'2007, Software Engineering in Progress*. Nakom, 2007, pp. 172–183.

[6] *OMG Meta Object Facility (MOF) Core Specification, version 2.4.1, formal/2013-06-01*, Object Management Group, 2013.

[7] M. Śmiałek, J. Bojarski, W. Nowakowski, A. Ambroziewicz, and T. Straszak, "Complementary use case scenario representations based on domain vocabularies," *Lecture Notes in Computer Science*, vol. 4735, pp. 544–558, 2007, MODELS'07, DOI: 10.1007/978-3-540-75209-7.

[8] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[9] M. Smialek, W. Nowakowski, N. Jarzebowski, and A. Ambroziewicz, "From use cases and their relationships to code," in *Second IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012*, 2012, pp. 9–18, DOI: 10.1109/MoDRE.2012.6360084.

[10] M. Smialek, N. Jarzebowski, and W. Nowakowski, "Translation of use case scenarios to Java code," *Computer Science*, vol. 13, no. 4, pp. 35–52, 2012, DOI: 10.7494/csci.2012.13.4.35.

[11] M. Śmiałek and W. Nowakowski, *From Requirements to Java in a Snap: Model-Driven Requirements Engineering in Practice*. Springer, 2015.

[12] M. Potel, "MVP: Model-View-Presenter the Taligent programming model for C++ and Java," Taligent Inc., Tech. Rep., 1996.

[13] C. Bauer and G. King, *Hibernate in Action (In Action Series)*. Greenwich, CT, USA: Manning Publications Co., 2004.

[14] H. Feddema, *DAO Object Model: The Definitive Reference*. O'Reilly Media, 2000.

[15] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[16] A. G. Kleppe, J. B. Warmer, and B. Wim, *MDA Explained, The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[17] A. Kalnins, J. Barzdins, and E. Celms, "Model transformation language MOLA," *Lecture Notes in Computer Science*, vol. 3599, pp. 62–76, 2005, MDAFA'04, DOI: 10.1007/11538097_5.

[18] *The MOLA Language, Reference Manual, Version 2.0 final*, University of Latvia, 2007, http://mola.mii.lu.lv/.

[19] M. Smialek and T. Straszak, "Facilitating transition from requirements to code with the ReDSeeDS tool," in *20th IEEE Requirements Engineering Conference (RE'12)*, 2012, pp. 321–322, DOI: 10.1109/RE.2012.6345825.

[20] W. Nowakowski, M. Śmiałek, A. Ambroziewicz, and T. Straszak, "Requirements-level language and tools for capturing software system essence," *Computer Science and Information Systems*, vol. 10, no. 4, pp. 1499–1524, 2013, DOI: 10.2298/CSIS121210062N.

[21] *OMG Unified Modeling Language, version 2.5, ptc/2013-09-05*, Object Management Group, 2013.

[22] J. van Wijngaarden and E. Visser, "Program transformation mechanics: A classification of mechanisms for program transformation with a survey of existing transformation systems," Utrecht University, Tech. Rep. UU-CS-2003-048, 2003.

[23] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley, 2008.

[24] J. Evermann and Y. Wand, "Toward formalizing domain modeling semantics in language syntax," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 21–37, 2005, DOI: 10.1109/TSE.2005.15.

[25] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002, p. 401.

[26] M. Śmiałek, N. Jarzebowski, and W. Nowakowski, "Runtime semantics of use case stories," in *IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC'12)*. IEEE, 2012, pp. 159–162, DOI: 10.1109/VLHCC.2012.6344506.