Authors' Instructions
Preparation of Camera-Ready Contributions to SCITEPRESS Proceedings

First Author Name[1], Second Author Name[1] and Third Author Name[2]
[1]*Institute of Problem Solving, XYZ University, My Street, MyTown, MyCountry*
[2]*Department of Computing, Main University, MySecondTown, MyCountry*
{f_author, s_author}@ips.xyz.edu, t_author@dc.mu.edu

Keywords: The paper must have at least one keyword. The text must be set to 9-point font size and without the use of bold or italic font style. For more than one keyword, please use a comma as a separator. Keywords must be titlecased.

Abstract: Background: The several maintenance tasks a system is submitted during its life usually cause its architecture deviates from the original conceived design. Therefore software engineers need processes for recovering the knowledge embedded in legacy systems in order to get a better software comprehension. In this context, refactoring can be applied in a legacy system to do so and yet to clean up, to improve and to raise the level of reuse of the legacy system. Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the source-code yet improves its internal structure. Nowadays, with the advent of ADM (Architecture-driven modernization), an OMG (Object Management Group) standard for modernizing legacy software systems, the refactoring process follows a MDD (Model-Driven Development) approach using the KDM (Knowledge Discovery Metamodel) specification as the cornerstone of the standard. Problem: we should put the problem here. Objectives: This paper seeks to define ..... Method: .... Results: To provide some evidence of our approach....., we conducted a case study. .....

1 INTRODUCTION

Software systems are considered legacy when their maintenance costs are raised to undesirable levels but they are still valuable for organizations. However, they can not be discarded because they incorporate a lot of embodied knowledge due to years of maintenance and this constitutes a significant corporate asset. As these systems still provide significant business value, they must then be modernized/re-engineered so that their maintenance costs can be manageable and they can keep on assisting in the regular daily activities.

The first task that must be performed in order to carrying out a software modernization is understand the legacy system. It is not a trivial task, in fact studies estimate that between 50 percent and 90 percent of software maintenance involves developing an understanding of the software being maintained (**?**), thus several approaches have been developed to support software engineers in the comprehension of systems where reverse engineering (RE) is one of them (**?**). RE supports program comprehension by using techniques that explore the source code to find relevant information related to functional and non-functional features (**?**).

In this context, OMG (Object Management Group) has employed a lot of effort to define standards in the modernization process, creating the concept of ADM (Architecture-Driven Modernization). ADM follows the MDD (Model-Driven Development) (**?**) (**?**) guidelines and comprises two major steps. Firstly a reverse engineering is performed starting from the source code and a model instance (PSM) is created. Next successive refinements (transformations) are applied to this model up to reach a good abstraction level (PSM or CIM) in model called KDM (Knowledge Discovery Metamodel). Upon this model, several refactorings, optimizations and modifications can be performed in order to solve problems found in the legacy system. Secondly a forward engineering is carried out and the source code of the modernized target system is generated again. According to the OMG the most important artifact provided by ADM is the KDM metamodel, which is a multipurpose standard metamodel that represents all aspects of the existing IT (Information Technology) architectures. The idea behind the standard KDM is that the community starts to create parsers from different languages to KDM. As a result everything that takes KDM as input can be considered platform and language-independent. For example, a refactoring catalogue for KDM can be used for refactoring systems implemented in different languages.

2 MOTIVATION

Our motivation is fourfold. Firstly, is the present lack of a fully developed idea of "good" modernization by using ADM and its metamodels by using design patterns to assist the modernization of legacy systems. This is an important issue, for a clear notion of style is a fundamental prerequisite for the use of modernization by means of ADM, enabling programmers to see where they are heading when modernizing their legacy system.

Secondly, one problem with automated restructuring techniques that modify source-code is that they do not restructure in-line documentation (i.e., program documentations) along with the source-code. This means that manual labor to restructure documentation is nearly always needed after applying a restructuring approach. In addition, the source-code is the only available artifact of the legacy software. We argue that by applying model-base modernization, both the legacy software and the generated software will have either a restructure documentation or a new type of artifact as result improving their documentation.

Thirdly, we claim that unlike the code-based modernization, model-based ones are platform independent. Thus, models can be transformed and good designs can be produced regardless of programming language.

Finally, the absence of tool that supporting modernization by using the KDM specification in current integrated development environments. We argue that efficient tool can bring benefits to assist software engineer during the modernization process. Therefore, we also devised a proof-of-concept Modernization-Integrated Environment (MIE), which is an environment that modernizing a legacy system to services by using ADM and its metamodels.

3 BACKGROUND

In this section we provide a brief background to Architecture-Driven Modernization (ADM) presenting the core ideas. Furthermore, this section describes the an of the ADM standards, e.i., Knowledge Discovery Metamodel (KDM).

3.1 Model-Driven Development

Software systems are becoming increasingly complex as customers demand richer functionality delivered in ever shorter timescales (**?**). In this context, Model-Driven Development (MDD) can be used to speed up software development and manage complexity is to shift from programming in solution-space in order to model problem-space.

MDD is an approach to software development that give particular emphasis upon making models the primary development artifact and subjecting such models to a refinement process, by using automatic transformations, until a running system is obtained. In this context, MDD aims to provide higher level of abstraction in development of systems which further results in an improved understanding of complex systems (**?**).

Furthermore, MDD handles problems in software systems development that originate from existence of heterogeneous platforms. It achieves this through keeping different levels of model abstractions; and by transforming models from Platform Independent Models (PIMs) to Platform Specific Models (PSMs). Therefore, automatic generation of application code (i.e. automatic model-driven code generation) offers many advantages such as the rapid development of high quality code, reduced number of accidental programming errors, enhanced consistency between design and code (**?**). Changes in implementation strategies can be achieved through modifying the transformations. Shifting engineering concerns to platform-independent levels allows developers to focus on designing applications without involvement in platform-specific concepts.

It is worth highlight that models in MDD are usually represented by domain-specific languages (**?**), i.e., a language that adequately represents the information of a given domain. Instead of representing elements using a general purpose language, producing source code (e.g., Java) and class diagrams (e.g., plain UML models), the knowledge is described in a language which the domain experts understand. Besides, as the expert uses a suitable language to describe the system at hand, the accidental complexity that one inherent from the restrictions of the language to properly describe a given domain is reduced, leaving only the essential complexity of the problem.

3.2 Architecture-Driven Modernization

Nowadays, researchers have been shifted from the typical refactoring process to the so-called Architecture-Driven Modernization (ADM). ADM is the concept of modernizing existing systems with a focus on all aspects of the current systems architecture and the ability to transform current architectures to target architectures by using all principles of MDD (**?**, p. 60). Figure 1 shows the ADM modernization domain model where the left side of the horseshoe is the current state of a business architecture "as-is" and the right side is what we want to get after the modernization "to-be".

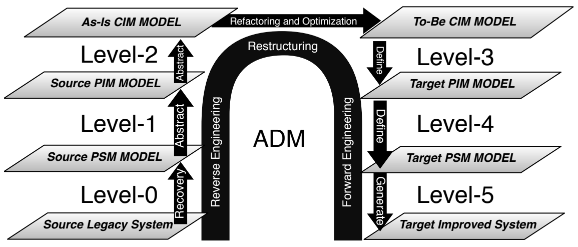

Figure 1: Modernization domain model (Adapted from Ulrich and Newcomb (**?**))

As can be seen in Figure 1 the horseshoe reengineering model has been adapted to ADM and it is nowadays known as horseshoe modernization model. As ADM uses the principles of MDD three kinds of models in the horseshoe are used, they are: (*i*) PIM - **P**lataform **I**ndependent **M**odel which represents a view of the system from the platform independent viewpoint at an intermediate abstraction level, (*ii*) PSM - **P**lataform **S**pecific **M**odel which constitutes a view of the system from the platform specific viewpoint at a low abstraction level, and (*iii*) CIM - **C**omputational **I**ndependent **M**odel that represents a view of the system from the computational independent viewpoint at a high abstraction level. These models are used in the steps of the ADM process, i.e., Reverse Engineering, Restructuring, and Forward Engineering. In the first step, a reverse engineering is performed starting from the artifacts of the legacy system (source code, database, configuration files, etc) and a set of PSM are created. Next, refactoring and restructuring techniques can be applied on these models in order to solve problems found in the legacy system. Therefore, this step consist of a set of transformation from the input model ("as-is") to obtain a target model ("to-be"). Finally, a forward engineering is carried out and the source code of the modernized target system is generated again.

In order to perform such steps, ADM introduces several modernization standards: Abstract Syntax Tree Metamodel (ASTM), Knowledge Discovery Metamodel (KDM), Structured Metrics Metamodel (SMM), etc. The next subsections present more information about these standards:

3.2.1 Knowledge Discovery Metamodel - KDM

Knowledge Discovery Metamodel (KDM) is the key within set of standards (**?**). KDM allows standardized representation of knowledge extracted from legacy systems by means of reverse engineering. KDM provides a common repository structure that makes possible the exchange of information about existing software assets in legacy systems. This information is currently represented and stored independently by heterogeneous tools focused on different software assets (**?**, p. 32). Figure 2 shows each of the varying views of the existing IT architecture represented by the KDM. For example, the build view, depicts system artifacts from a source, executable, and library viewpoint. Other perspectives include design, conceptual, data, and scenario views.

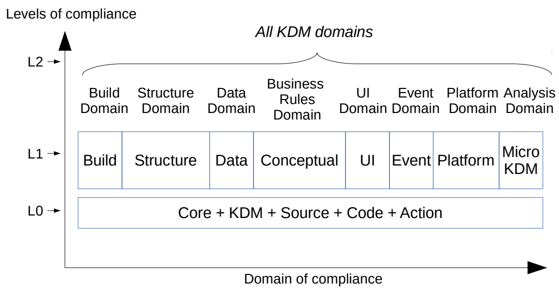

Figure 2: KDM domains of artifact representation (Adapted from Ulrich and Newcomb (**?**))

The Level 0 (L0) encompasses the Infrastructure and Program Elements Layer. Infrastructure Layer consists of the Core, kdm, and Source packages which provide a small common core for all other packages. Program Elements Layer consists of the Code and Action packages providing programming elements such as data types, data items, classes, procedures, macros, prototypes, templates and captures the low level behavior elements of applications, including detailed control and data flow between statements. The Level 1 (L1) cover the Resource Layer which represents the operational environment of the existing software system. For example, the knowledge related to events and state-transition, the knowledge related to the user interfaces of the existing software system and the knowledge related to persistent data, such as indexed files, relational databases, and other kinds of data storage. The Level 2 (L2) cover the Abstraction Layer which represents domain and application abstractions.

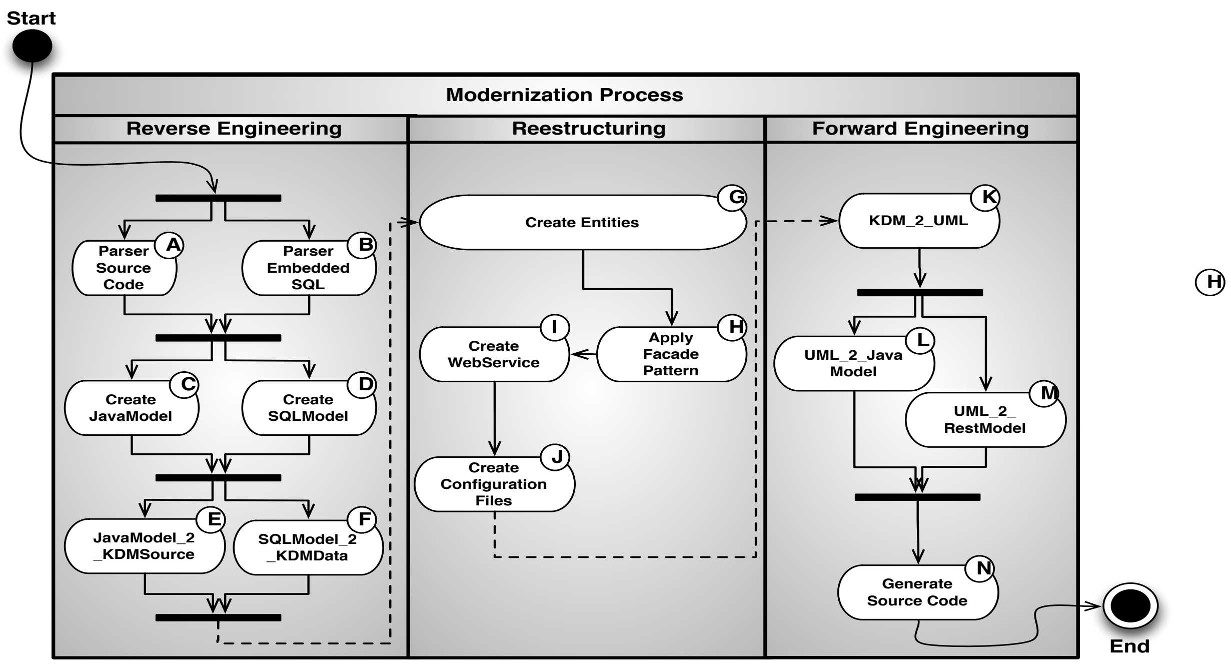

Figure 3: Modernization Activity Diagram.

4 PROPOSED APPROACH

By combining the concepts of software modernization, ADM and MDD, this paper proposes an approach to furnish software reengineering of legacy systems to web services. In others words, the central goal of the approach is to supply an automate way to support the modernization of legacy systems, aiming to provide reduction in time and effort spent by using code generation and to assist the migration of these systems to web services.

We assume that a legacy systems which uses databases can decomposed into services. More specifically, our approach relies on seeking for embedded Structured Query Language (SQL) queries into the source-code of the legacy system to restructure and re-organize the system by using design patterns, such as Facade. Then CRUDs (Create, Retrieve, Update and Delete) by using KDM are devised, i.e., services are created by means of model transformation. For example, suppose that an embedded SQL *F* is found into the source-code of the legacy systems. In this case, a set of both transformations and rules are applied into this SQL, then a model *F'* which contains information (table(s), column(s), relationship, etc) related to the embedded SQL *F* is created. Thus, *F'* is said to be equivalent to $F \Leftrightarrow F'$, but now *F'* is represented by model. Then, this *F'* is transformed into an instance of KDM and rules of modernization are applied until it reach the intended behavior, i.e., services. Finally, a forward engineering is carried out and the source code of the modernized target system is generated.

In order to explain our approach, there is an activity diagram on Figure 3 with all steps illustrated by a capital letter inside a circle. Each step must be carry out in order to modernize the legacy systems. Moreover, this diagram consist of three main phases: (*i*) Reverse Engineering (RE), (*ii*) Reestructuring, and (*iii*) Forward Engineering (FE). These phases are explained as follows:

4.1 Reverse Engineering - RE

The aim of this phase is to analyze the legacy systems in order to identify the components of the system and their interrelationships. Furthermore, it also intends to build a set of representation of the legacy system's artifacts at a higher level of abstraction, i.e., both PSM and PIM are created in this phase.

In our approach the RE starts by parsing two artifacts: (*i*) the legacy system's source-code, and (*ii*) the embedded SQL queries, Figure 3 steps Ⓐ and Ⓑ. Therefore, the approach need to use parser to obtain information related to these artifacts.

The former parser (see Figure 3 step Ⓐ) is responsible to take as input the source-code of the legacy system and then to build a data structure as output, e.g., herein the output is represented by an Abstract Syntax Tree - (AST) which is a tree representation of the abstract syntactic structure of Java source code, by using it is possible to carry out either analysis or transformation on documents that contain programming language text.

The latter parser (see Figure 3 step Ⓑ) exhaustively scans the source-code. As the parser finds an SQL statement embedded into the source-code of the legacy system, such as *Select*, *Delete*, *Update* and *Insert*, it translates those statement into an AST. As a common programming technique, many SQL statements (or partial of them) are declared as string variables, therefore variable declarations and assignments are also of our focus. In Listing 1 depicts a chunk of code which contains four SQL statement.

```
 1 public class Entity {
 2
 3   private String sql1 = "SELECT * FROM TABLE_1";
 4
 5   public void meth() {
 6     String sql3 = "UPDATE TABLE3 SET column1=value1 , WHERE
           some_column=some_value;"
 7
 8     String sql4 = "INSERT INTO TABLE_4 VALUES (value1,value2,
           value3);"
 9
10     String sql5 = "DELETE FROM table_name WHERE some_column=
           some_value;"
11   }
12 }
```

Listing 1: Example of Embedded SQL

For instance, for each SQL statement, i.e., Select in line 5, Update in line 8, Insert in line 10 and Delete in line 12 the parser recognizes, analyzes and creates an AST. This AST contains information related to the name of the tables and some columns that are involved in such statements. Nevertheless, as can be seen some statements hide important informations (see Listing 1 line 5) such as the columns of a table. Therefore, initially this AST is not complete once it just owns the names of the tables and some columns. However, name of the tables and some columns are not sufficient as the approach need to identify all columns of a table, to recognize if the column is either primary key or foreign key, to pinpoint the relationships among the tables, and also to identify the type of the columns. Therefore, to address this issue the approach need to connect to the database of the legacy system in order to get these information. As for getting this information the approach uses database metadata (data about database data). Metadata provides a structured description of database information resource and services.

After parsing the artifacts two steps must be carry out: (*i*) to create a PSM to represent the source-code of the legacy systems - the AST obtained by the first parser described aforementioned will be transformed to an instantiation of the Java meta-model, and (*ii*) to create a PSM which represents the SQL identified in the source-code - the second AST obtained and described earlier will be transformed to an instantiation of an SQL meta-model, these steps are depicted in Figure 3 steps Ⓒ and Ⓓ, respectively.

In the steps Ⓒ a set of rules must be applied to transform the AST into a instance of the Java meta-model. Java meta-model is the reflection of the Java language, as defined in version 3 of "Java Language Specification" from Sun Microsystems. Due space limitation in Figure 4 is depicted just a chunk of the Java meta-model; the complete Java meta-model contains 126 meta-classes. As can be seen, in Figure 4 each meta-classes is intended to represent an element of the Java language. For instance, each "package" found in the source-code is transformed to instances of the meta-classe "Package", "Classes" are transformed to instances of the meta-classe "ClassDeclaration", "Fors" are transformed to instances of the meta-classe "ForStatement", etc;

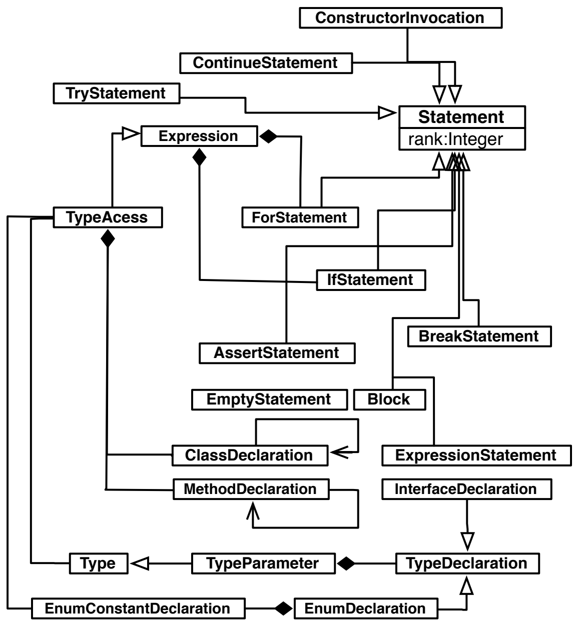

Figure 4: Java meta-model (simplified excerpt).

On the fourth step (see Figure 3 Ⓓ) an instance of the SQL model must be obtained. The SQL model used herein is represented in a PSM model according to the SQL-92 standard (ref). Figure 5 presents the SQL meta-model. This meta-model contains meta-classes to represent each element identified by the second parser. More specifically, the tables identified earlier (rule R1), its columns (rule R2), the data type associated with each column (rule R3), the primary key associated with each table (rule R4), and the foreign key associated with each table (rule R5) is represented by this meta-model. Next, we describe each of these rules.

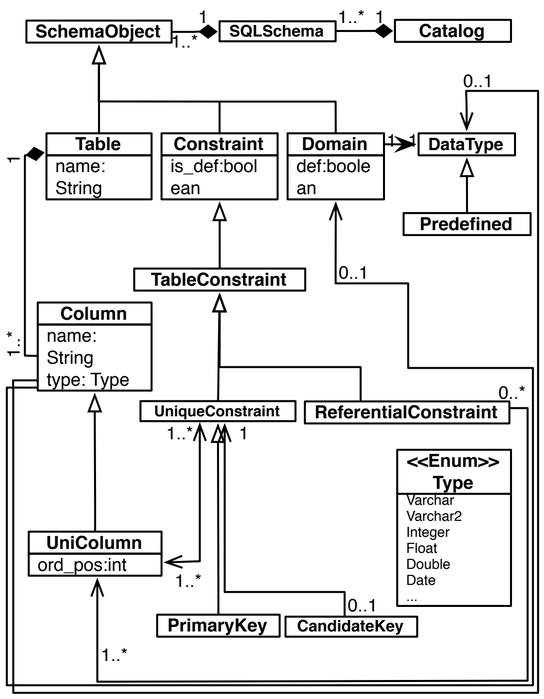

Figure 5: SQL meta-model (simplified excerpt).

- **Rule R1:** Tables that were found in any SQL statement (Insert, Select, Update or Delete) as either source or target clauses (From, Set, Into, and so on) are represented by an instance of the meta-classe Table, see Figure 5;
- **Rule R2:** The columns that are identified by means of the parser or by the database metadata in the SQL statements are created in the corresponding tables. These tables have previously been created through the application of **R1**. Those columns are represented by the meta-classe Column, see Figure 5.
- **Rule R3:** The data type associated with each column was deduced through the database metadata. Therefore, for the columns that have previously been created through the application of **R2**, are now updated with their specific type, see the meta-Enumeration named Type in Figure 5.
- **Rule R4:** The constraint of columns was also deduced through the database metadata. As result, for the columns that have previously been created through the application of **R2**, at least one should be flagged as primary key. Thus, if a column is primary key then an instance of the meta-classe PrimaryKey is instantiated.
- **Rule R5:** Foreign key that were found in the database metadata are represented by an instance of the meta-classe ReferentialConstraint, see Figure 5.

Upon finishing the instantiation of both PSMs (Java model and SQL model), the next two steps consist of transforming them into KDM, which represent the same information but in a platform-independent manner. To do so, two steps must be carried out: (*i*) *JavaModel2KDMCode*, and (*ii*) *SQLModel2KDMData*, Figure 3 steps Ⓔ and Ⓕ, respectively. As stated in Section 3.2.1 KDM contains severals meta-models, but herein we are only interested in the Code meta-model, which represents the code elements of a program and their associations and in the Data meta-model which defines a set of meta-model elements whose purpose is to represent organization of data in the existing software system. A briefly overview of both meta-models KDMCode and KDMData are depicted in Figure 6 and 7.

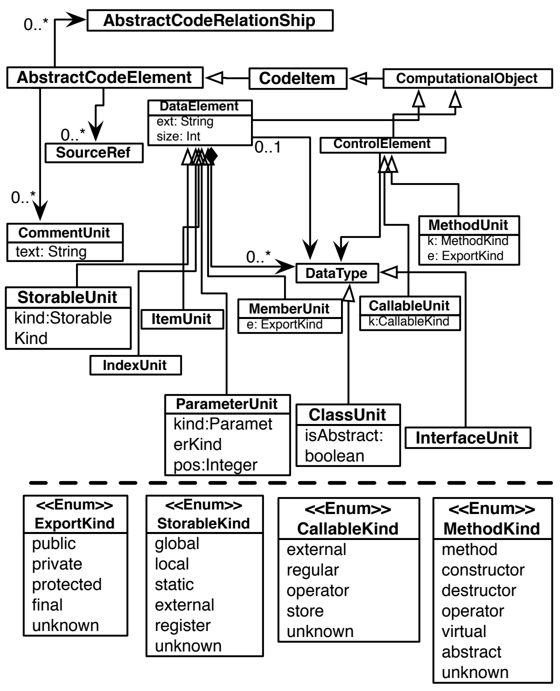

Figure 6: KDM code meta-model (simplified excerpt).

Our approach uses ATL (ATLAS Transformation Language) (**?**) to realize model-to-model transformations. In Listing 2 is depicted a chunk of code related to the transformation *JavaModel2KDMCode*, i.e., step Ⓔ. Notice that the *JavaModel2KDMCode* is carried out on instances of Java meta-mode (Figure 4), and produces a corresponding model based on the KDMSource meta-model, see Listing 2 in lines 1 and 2. ATL is based in *rules*. Therefore, we have defined a set of rules to transform each meta-classe of the Java meta-model (Figure 4) to a instance of KDMCode.

Lines 3-8 show a rule to transform instance of packages from the source (Java meta-model) to packages of the target meta-model (KDMCode), by keeping the same name. Lines 10-20 illustrate a transformation rule responsible to transform from the Java meta-model instance of class (*ClassDeclaration*) and its methods (*MethodDeclaration*) to *ClassUnit* and *MethodUnit* of the target KDMCode meta-model.

```
 1  module JavaModel2KDMSource;
 2  create OUT:KDMSource from IN:JavaModel;
 3  rule Package{
 4    from
 5        ps:JavaModel!Package
 6    to
 7        pt:KDMSource!Package(
 8        name<-ps.name)
 9  }
10  rule Class{
11    from
12      cs:JavaModel!ClassDeclaration(
13            cs.ownedOperation->notEmpty())
14    to
15         ct:KDMSource!ClassUnit(
16          name<-cs.name,
17          package<-cs.package,
18          methodUnit<-opeLst
19      ),
20      opeLst:distinct JavaModel!MethodDeclaration foreach
21  (oper in cs.methodUnit.asSequence()) (name<-oper.name)
22  }
23  [...]
24  }
```

Listing 2: Chunk of JavaModel2KDMCode

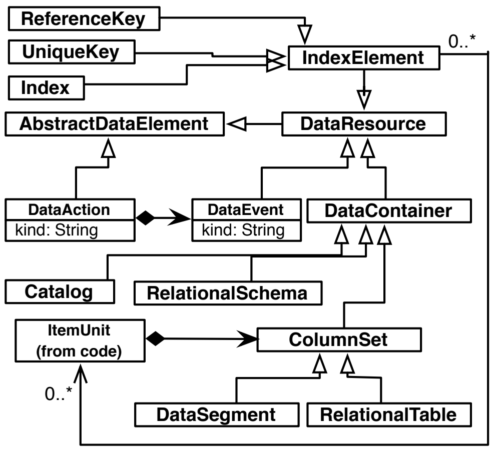

Figure 7: KDM data layer (simplified excerpt).

The second transformation *SQLModell2KDMData* is carried out on instances of SQL meta-mode (Figure 5), and produces a corresponding model based on the KDMData meta-model (Figure 7), see Listing 3 in lines 1 and 2. Lines 4-12 describe a rule to transform the *SQLSchema* from the source (SQL meta-model) to *RelationalSchema* of the target meta-model (KDMData), by copying the name and the references of all tables. After, in lines 13-21 are described a rule to transform the *Table* from the source (SQL meta-model) to *RelationalTable* of the target meta-model (KDMData), by copying the name and keeping all columns of such *Table*. In lines 22-28 show a rule to transform *Column* and its type from the source meta-model to *ColumnsSet* of the target meta-model. Finally, in lines 30-34 the types of the columns are identified.

```
 1  module SQLModel2KDMData;
 2  create OUT:KDMData from IN:SQLModel;
 3
 4  rule SQLSchema2RelationalSchema {
 5     from
 6         t: SQLModel!SQLSchema
 7     to
 8         r: KDMData!RelationalSchema (
 9             name <- t.name,
10             relationalTable <- t.table
11         )
12  }
13  rule Table2RelactionalTable {
14     from
15         t: SQLModel!Table
16     to
17         r: KDMData!RelationalTable (
18             name <- t.name,
19             ownedAttribute <- t.columns
20         )
21  }
22  rule Column2ColumnSet {
23     from
24         t: SQLModel!Column (t.oclIsTypeOf(SQLModel!Column))
25     to
26         r: KDMData!ColumnSet (
27             name <- t.name,
28             type <- typeDATA
29         ),
30         typeDATA : MM2!PrimitiveType(
31             name <- t.itemUnit.first().type.name,
32             type <- t.item.primitiveType
33         )
34  }
35  [...]
36  }
```

Listing 3: Chunk of SQLModel2KDMData

4.2 Reestructuing

The goal of this phase is to analysis the models created earlier to modernize automatically the legacy system into services, i.e., RESTFull operations. Therefore, in this phase is carried out an algorithm which takes as input both models KDMCode and KDMData - then four steps are conducted to create a set of services. The Algorithm 1 presents how the services are create, more information related to the steps are as follows:

Services are entities that are either annotated with @Path or have at least one method annotated with @Path or a request method designator, such as @GET, @PUT, @POST, or @DELETE. Thus, firstly,

Algorithm 1: Creating RESTFull

```
   Input: KDMSource source, KDMData data
 1 begin
 2     foreach data.RelationalTable do
 3         entity ← createClassUnit (data, source);
 4         entity ← createDataElement (data, source);
 5         entity ← createMethodUnit (data, source);
 6         if entity != null then
 7             absFac ← createAbstractFacade (entity)
 8             entiFac ← createEntityFacade (absFac)
 9             createCRUD (entiFac)
10             createGeneralization (absFac, entiFac)
11             createComposition (entity, entiFac)
12             createEntityRESTFUL (entity)
13         end
14     end
15 end
```

it is necessary to create entities with the information identified earlier. In other words, one must to transform all identified embedded SQL code (the tables, columns and even the relationship), which are now represent by the metaclasses of the KDMData to the correct metaclasse of the KDMCode. This is carried out by the step Ⓖ, see Figure 3. Lines 2 of the Algorithm 1 depicts a loop which is executed for each meta-classe *RelationalTable* belonging to the metamodel KDMData. After, in line 3, the function *createClassUnit* is called. This function gets the meta-attributes name of the *RelationalTable* and then create an instance of the meta-classe *ClassUnit*. *ClassUnit* is a meta-class that represents user-defined classes in object-oriented languages (?). In line 4, the function *createDataElement* is executed. This function obtains the all columns of the *RelationalTable* and then create an instance of the meta-classe *DataElement* which represent attributes in the KDM. In line 5 the function *createMethodUnit* is carried out. This fuction is similar to the last one, however, instead of attributes, *gets* and *sets* are created. These methods are represented by instances of the meta-classe *MethodUnit* which represents member functions owned by a ClassUnit.

Afterwards, the services must be indeed created. As for we used the Facade Pattern (?), see Figure 3 step Ⓗ. The Algorithm 1, lines 7-12 depicts how the services are created. Firstly, in line 7 the function *createAbstractFacade* is called to instantiate a meta-classe *ClassUnit* which represents the *AbstractFacade* of each entity. Secondly, in line 8 shows the function *createEntityFacade* that is also an instance of the *ClassUnit* which now represents the service. Thirdly, in line 9 the function *createCRUD* depicts that for each entity four methods are create, i.e., create, retrieve, update and delete. Fourthly, the functions *createGeneralization* and *createComposition* in line 10 and 11 depict that relationships of generalization and composition are created. Finally, in line 12 the function *createEntityRESTFUL* is carried out. It is responsible to inject request method designator (@GET, @PUT, @POST, or @DELETE), see Figure 3 step Ⓘ.

In the last step a set of configuration files are created. These files store project configuration data or settings, see Figure 3 step Ⓙ.

4.3 Forward Engineering - FE

Forward Engineering (FE) is the process of bringing high-level abstractions to physical implementation of a system[ref]. This phase starts with the step Ⓚ, see Figure 3. In this phase the restructured KDM model obtained in the *Reestructuring* phase now is transformed to an instance of Unified Modeling Language (UML) (?).

```
 1 module KDM2UML;
 2 create OUT:UML from IN:KDM;
 3
 4 rule Package2Package {
 5     from
 6         t: KDM!Package
 7     to
 8         r: UML!Package (
 9             name <- t.name,
10             [...]
11         )
12 }
13 rule ClassUnit2Class {
14     from
15         t: KDM!ClassUnit
16     to
17         r: UML!Class (
18             name <- t.name,
19             ownedAttribute <- t.memberUnit,
20             ownedOperation <- t.methodUnit,
21             [...]
22         )
23 }
24 rule MemberUnit2Property {
25     from
26         t: KDM!MemberUnit
27     to
28         r: UML!Property (
29             name <- t.name,
30             type <- t.ownedType,
31             [...]
32         )
33 }
34 rule MethodUnit2Operation {
35     from
36         t: KDM!MethodUnit (t.oclIsTypeOf(SQLModel!MethodUnit)
               )
37     to
38         r: UML!Operation (
39             name <- t.name,
40             return <- returnDATA
41             [...]
42         ),
43         returnDATA : MM2!ReturnType(
44             name <- t.itemUnit.first().type.name,
45             type <- t.item.primitiveType
46         )
47 }
48 [...]
49 }
```

Listing 4: Chunk of KDM2UML

In Listing 4 shows the chunk of the code written in ATL which is responsible to realize such transformation. In lines 4-11 are described a rule to transform the *Package* from the source (KDM) to *Package* of the target meta-model (UML). Lines 13-23 depict a rule to transform

activity Refine Model, which in- cludes the Analysis and Design disciplines of software devel- opment process. This activity is essential for the development of the new application, since the OO model generated in the RE needs to be refined and complemented according with new requirements and specifications not performed by code generation.

5 PROOF-OF-CONCEPT IMPLEMENTATION

We devised a proof-of-concept implementation named Modernization-Integrated Environment (MIE). Our proof-of-concept implementation automates three key steps of mutation: execution of the original program, mutant execution, and mutant results comparison.

ACKNOWLEDGEMENTS

REFERENCES

Canfora, G., Di Penta, M., and Cerulo, L. (2011). Achievements and challenges in software reverse engineering. *Commun. ACM*, 54:142–151.

Chikofsky, E. J. and Cross II, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition.

Izquierdo, J. and Molina, J. (2010). An architecture-driven modernization tool for calculating metrics. *Software, IEEE*, 27(4):37–43.

Jouault, F., Allilaire, F., Bezivin, J., and Kurtev, I. (2008). Atl: A model transformation tool. *Science of Computer Programming*, 72.

OMG (2012). Object Management Group (OMG) Unified Modeling Language (UML), Infrastructure, V2.1.2 - OMG Available Specification without Change Bars.

Perez-Castillo, R., Garcia Rodriguez de Guzman, I., Piattini, M., and Piattini, M. (2009). On the use of adm to contextualize data on legacy source code for software modernization. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 128–132.

Tilley, S. and Smith, D. (1995). Perspectives on legacy system reengineering.