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 both better software comprehension and software modernization. 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 (Tilley and Smith, 1995), thus several approaches have been developed to support software engineers in the comprehension of systems where reverse engineering (RE) is one of them (Canfora et al., 2011). RE supports program comprehension by using techniques that explore the source code to find relevant information related to functional and non-functional features (Chikofsky and Cross II, 1990).

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) (Ulrich and Newcomb, 2010) (Izquierdo and Molina, 2010) 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 pro-

vided 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. We claim that by using design patterns (Gamma et al., 1994) during the modernization by means of ADM and its metamodels it is an important issue, once they provide a clear notion of style, as result 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. Generally, the source-code is the only available artifact of the legacy system. We argue that by applying model-base modernization, either the non up-to-date documentation of a legacy system or in case of missing documentation, updated/new documentation (such UML) can be obtained improving its understanding.

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 Architecture-Driven Modernization

Nowadays, researchers have been shifted from the typical refactoring process to the so-called Architecture-Driven Modernization (ADM) (Frey et al., 2012; Baresi and Miraz, 2011; Guzman et al., 2007; Bruneliere et al., 2010; del Castillo et al., 2009). 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 Model-Driven Development (MDD) (Guzman et al., 2007). 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".

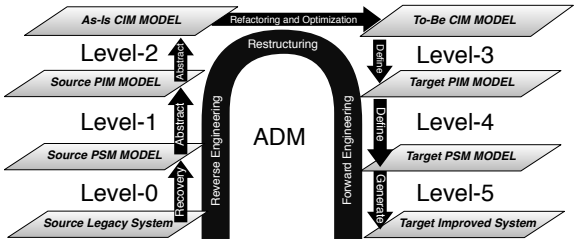

Figure 1: Modernization domain model.

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. As KDM is the metamodel more important (OMG, 2012) and our approach is based on it, herein we just present information about the KDM. Thus, next subsection present more information about it.

3.1.1 Knowledge Discovery Metamodel - KDM

Knowledge Discovery Metamodel (KDM) is the key within set of standards (Perez-Castillo et al., 2009a). 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 (Ulrich and Newcomb, 2010, 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.

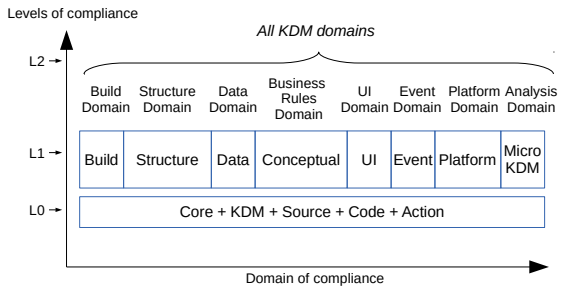

Figure 2: KDM domains of artifact representation (Adapted from Ulrich and Newcomb (Ulrich and Newcomb, 2010))

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.

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.

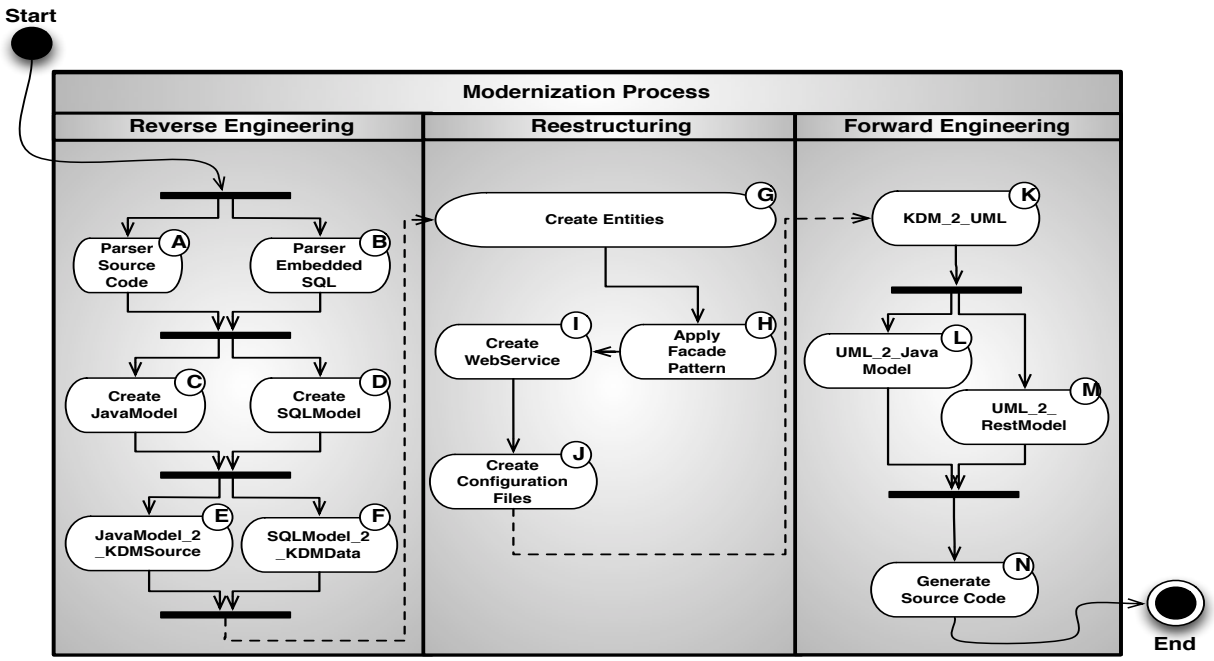

Figure 3: Modernization Activity Diagram.

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 discovery their knowledge, i.e., their components and 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 a 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 embedded 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 3, Update in line 6, Insert in line 8 and Delete in line 10 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 3) 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.

Nevertheless, 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, in order to address this issue the approach need to connect to the legacy system's database to discovery these information. As for getting this information the approach uses database metadata (data about database data).

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 a 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 an 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. The complete Java meta-model contains 126 meta-classes, thus, due space limitation in Figure 4 is depicted just a chunk of the Java meta-model. 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" found are transformed to instances of the meta-classe `ClassDeclaration`, "Fors" statements are transformed to instances of the meta-classe `ForStatement`, etc;

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[1]. 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. To realize the transformation rules (R1 through R6) need to be applied. Next, we describe each of these rules.

- **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

[1] http://savage.net.au/SQL/sql-92.bnf.html

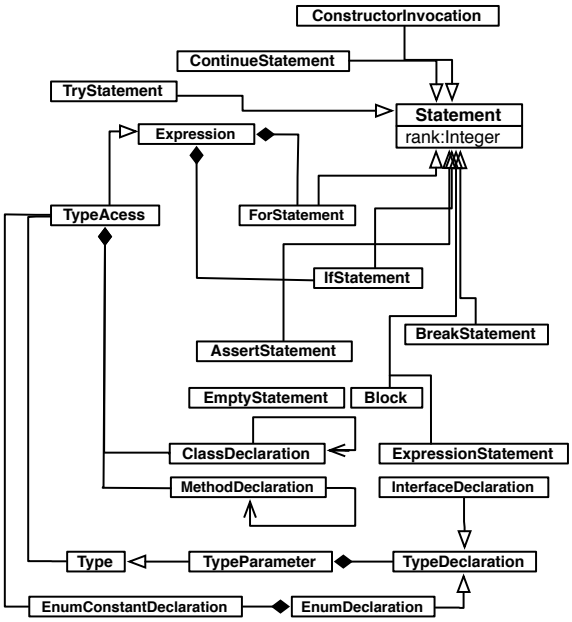

Figure 4: Java meta-model (simplified excerpt).

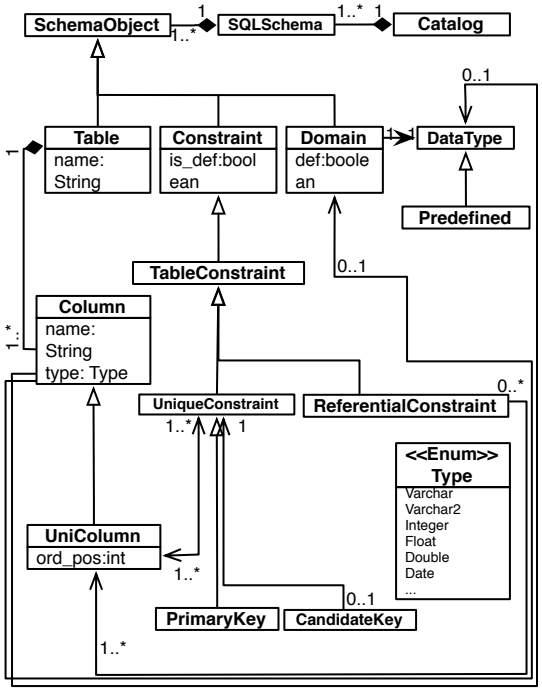

Figure 5: SQL meta-model (simplified excerpt).

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 3.1.1 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.

Our approach uses ATL (ATLAS Transformation Language) (Jouault et al., 2008) 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.

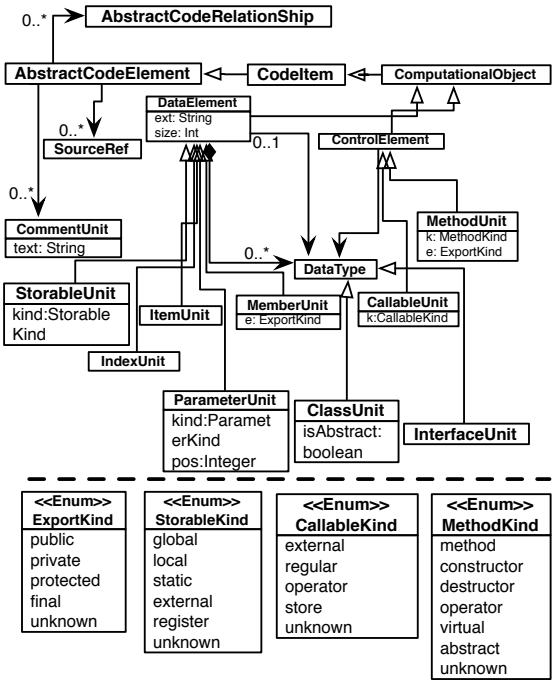

Figure 6: KDM code meta-model (simplified excerpt).

```
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

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 trans-

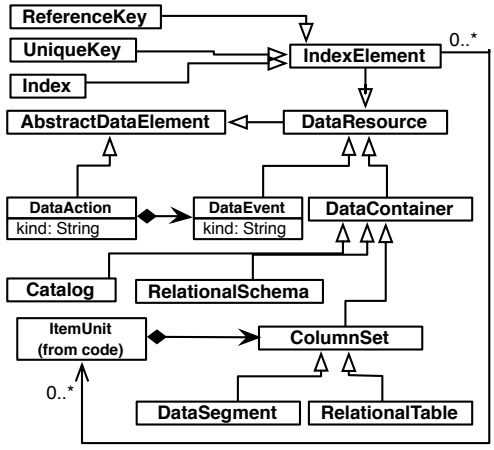

Figure 7: KDM data layer (simplified excerpt).

form 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 illustrates how the services are create, more information related to the steps are as follows.

Services, in Java 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, 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 meta-classes of the `KDMData` to the correct meta-classe 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 meta-model `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 (Perez-Castillo et al., 2009b). 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 function 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 (Gamma et al., 1994), 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 create an instance of the `ClassUnit` which now represents the service. Thirdly, in line 9 the function `createCRUD` depicts that for each entity four instance of meta-classe `MethodUnit` are create which

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
```

represent the operations of the services, 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`, and `@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 (Demeyer et al., 2002). 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) (OMG, 2012).

Due space limitation in Listing 4 is depicted just a 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 metamodel (UML). Lines 13-23 depict a rule to transform all `ClassUnit` from the source (KDM) to `Class` of UML. This rule copies the name of the `ClassUnit`, all `memberUnit`, all `methodUnit` and also the relationships among the classes. After transform `ClassUnit` to `Class`, all `memberUnit` and `methodUnit` must be transformed. The former, is depicted in rule `MemberUnit2Property`, lines 24-33. In lines 28-31 the `Property`'s name and type are assigned. The latter, is shown in rule `MethodUnit2Operation` in which lines 38-47 depict how all `MethodUnits` are properly transformed to `Operation`.

```
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

and etc are set are shown a rule that transforms all *memberUnit*, which represents in KDM both association and attributes,

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). In Figure 8 is depicted the architecture of MIE. As shown in this figure, we devised it on the top of the Eclipse Platform and used both Java and

Groovy as programming language. Moreover, we used Eclipse Modeling Framework (EMF)[2] to create the SQL model, the SOA model and to reutilize the UML model. MoDisco is used by the infrastructure since it provides an **A**pplication **P**rogramming **I**nterface - (API) to easily access the KDM model.

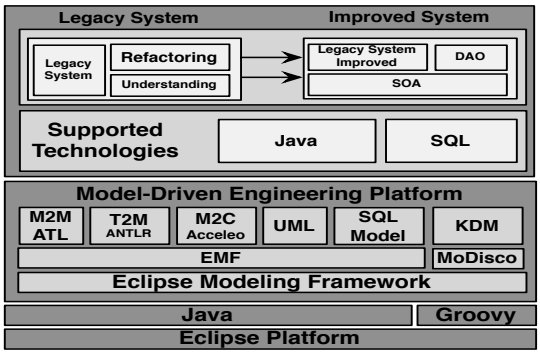

Figure 8: Architecture of Modernization-Integrated Environment.

ANother **T**ool for **L**anguage **R**ecognition - (ANTLR)[3] is used herein to create parsers to obtain information related to the legacy system's artifacts. Therefore, two parser were developed: (*i*) the first takes as input a Java grammar and generates as output an AST and (*ii*) the second parser is an extension of the first one to identify SQL embedded in the legacy system's source code, i.e., it takes as input a Java source-code and generates as output an AST which contains informations such as, tables, columns, primary keys, etc. Then, to transform these ASTs in PSMs we used an API provided by EMF. Afterwards, all the transformations M2M are done by **A**tlas **T**ransformation **L**anguage - ATL, which provides ways to produce a set of target models from a set of source models. Therefore, ATL is used to transform the PSMs to conform the KDM specification and to transform the improved KDM to an UML that represents the target systems.

Finally, in order to transform this last model in a set of physical artifacts (source code), i.e., model-to-code transformations, Acceleo[4] was used, which is based on textual template approach. A template can be thought of as the target text with holes for variable parts. The holes contain metacode which is run at template instantiation time to compute the variable parts. Furthermore, we have used Java Persistence API (JPA) 2.0 to deal with the way relational data is mapped to Java objects. Similarly, RESTful API have been used to implements SOA artifacts.

[2]http://www.eclipse.org/modeling/emf/

[3]http://www.antlr.org/

[4]http://www.eclipse.org/acceleo/

6 RELATED WORK

Several research works have been proposed by the academic community are related to the concepts discussed in this paper.

Modernization of Legacy Web Applications into Rich Internet Applications (Rodríguez-Echeverría et al., 2012) proposes a approach for systematic and semiautomatic modernization of legacy Web applications in rich interfaces applications. In this process, the MDD principles are applied and the use of ADM specifications for the generation of rich interfaces from the source code of navigation and presentation layers of legacy web applications. The proposed approach differs from this work by having a more general purpose and are not intended to only a single type of applications, such as Web applications. In addition, the proposed approach offers the possibility to use the database in the process of reengineering.

Model-Driven Reengineering of Database (Wang et al., 2009) presents a process to perform relational databases reengineering. The process is conducted through of repeated model transformations and divided into the stages of extraction and contextualization. In the extraction stage, a PSM is obtained from the database structure. In the contextualization stage, a PIM is generated from the PSM. As a result of the process, there are entity-relationship models and class diagrams. The proposed approach differs from this work by using object-relational mapping frameworks and metaprogramming techniques to obtain a model of the database structure, rather than using several model transformations, resulting in a more simple and faster way to extract knowledge from the database.

7 CASE STUDY

This section presents a case study to validate the proposed approach by applying it to a real-life legacy information system. As stated in Section 5 we devised a proof of concept tool which implements our approach. Notice that the case study was carried out following the protocol for planning, conducting and reporting case studies proposed by Brereton *et al.* in (Brereton et al., 2008) improving the rigor and validity of the study. The next subsections show more details about the main phase defined in this protocol, such as: background, design, case selection, case study procedure, data collection, analysis and interpretation and validity evaluation.

7.1 Background

According to the protocol proposed by Brereton *et al.* in (Brereton et al., 2008) firstly it is needed to identify previous research on the topic. Hence, in Section 6 we stated some researches related to modernize legacy system by using both ADM and MDD. Particularly, the approach herein described aims to identify embedded SQL statement in a legacy system and then by means of the ADM approach model transformations are realized until obtain a new system - now reestructured to use services with RESTFULL. As result, the object of this study is that the proposed approach identifies the embedded SQL statements in a legacy system, and the purpose of this study is the evaluation of the approach herein described related to its effectiveness and efficiency.

Therefore, taking into account the objects and purpose of the study, it was defined two research questions, as follows:

- $\mathbf{RQ}_1$: Can the proposed approach obtain embedded SQL statement from legacy systems to effectively create services by using RESTFULL?
- $\mathbf{RQ}_2$: Is the proposed approach efficient as compared to the classic life cycle of the software?

The former question, $\mathbf{RQ}_1$, verifies if the approach can obtain embedded SQL statement in a legacy system. In addition, $\mathbf{RQ}_1$ also assesses whether the identified SQL statements can be effectively transformed to services. The latter question, $\mathbf{RQ}_2$ aims to verify the efficiency of the proposed approach when compared to the classic life cycle of the software.

7.2 Design

The described case study consist of a single case (Yin, 2002), i.e., it focuses on a single legacy system. To assess the effectiveness of the proposed approach through the $\mathbf{RQ}_1$, we chose to use two measures, they are: (*i*) precision and (*ii*) recall. These measures are used because precision can be seen as a measure of exactness or fidelity, whereas recall is a measure of completeness. In our context, precision illustrates the amount of relevant recovered embedded SQL statement within the set of recovered SQL statement in a legacy system. A SQL statement is considered relevant if this statement faithfully represents SQL operations of the legacy system in the real world. Recall represents the amount of relevant recovered embedded SQL statement of the total of relevant SQL statement (recovered and not recovered) that depict the whole SQL operations of the legacy system.

As for answering $\mathbf{RQ}_2$ we applied empirical estimation model according to Software Equation (Putnam and Myers, 1991), see Equation 1.

$$E = [LOC * B^{0.333}/P]^3 * (1/t^4) \quad (1)$$

where:

E = effort in person-months or person-years

t = project duration in months or years LOC = lines of code

B = special skill factor

P = productivity parameter

7.3 Case Selection

In this section is described the suitable case that was chosen to be studied. Some criteria were applied to select the suitable case, as follows: (*i*) it must be an enterprise system, (*ii*) it must be a Java-based system, (*iii*) it must use embedded SQL statements, (*iv*) it must be a legacy system and (*v*) it must be of a size not less than 10 KLOC. After applying these criteria we chose PrograWeb[5] of Federal University of São Carlos (UFSCar). PrograWeb is an academic system on the Web for managing information about teachers, students and undergraduate programs of UFSCar.

The original architecture of the system was developed using Java Servlets and web pages in the Java Server Pages (JSP) language. The system runs on a server configured with Apache Tomcat[6] and connects with a PostgreSQL database[7], using the API Java Database Connectivity (JDBC). However, over time and the appearance of new requirements, the system maintenance became expensive, requiring its reengineering.

7.4 Case Study Procedure

In this section is shown how the execution of the study was planned. Notice that the execution was aided by the tool developed to support the proposed approach, see Section 5. The case study was carried out in a machine with an Intel Core I5 CPU 2.5GHz, 4GB of physical memory running Mac OS X 10.8.4.

As stated in Section 4 the approach proposed herein starts with the RE. Firstly, discovering of knowledge must be carried out. As described earlier two parser are executed to obtain this knowledge: (*i*) the first parser takes as input the PrograWeb's source-code and generates as output an AST and (*ii*) the second one is an extension of the first one; and its

[5]https://pradweb.ufscar.br/pradweb/

[6]http://tomcat.apache.org/

[7]http://www.postgresql.org/

aims is to identify embedded SQL in the legacy system's source-code, i.e., it takes as input a Java source-code and generates as output an AST which contains informations such as, tables, columns, primary keys, etc. Secondly, the ASTs obtained are transformed in two PSMs, the Java PSM and the SQL PSM. Such transformations are executed by using an API provided by EMF. Thirdly, these PSMs are transformed to PIMs to be in conform to the KDM standardization, i.e., the Java model to `KDMCode` and SQL model to `KDMData`. They are transformed by means of rules written in ATL.

By using the KDM models obtained the next phase starts where 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, see Algorithm 1. First, entities are created in the `KDMCode` (`ClassUnit`, `DataElement` and `MethodUnit`) by using the information of the `KDMData` (`RelationalTable`, `ColumnSet`). Second, services (CRUDs) are created by using the design pattern Facade (Gamma et al., 1994).

The FE is started with the new KDM metamodels. ATL is also used in this phase to transform the KDM metamodel into both UML and SOA models. Finally, in order to transform these last models into physical artifacts (source-code) model-to-code transformations was applied. This is performed by mapping context models using a template-based approach to a corresponding programming language and automatically generating the implementation.

After carrying the described approach, the new Prograd Web was obtained. Then all embedded SQL statement, the created entities and the created services are collected according to the data collection plan. Also, during the execution of the case study, we gathered the time that each step (see Figure 3) used up. After that, the data collected previously is analyzed and interpreted to draw conclusion to answer the research questions. The next sections present more information related to the results obtained from this case study.

7.5 Data Collection

According to the protocol proposed by Brereton *et al.* in (Brereton et al., 2008) the data to be collected and the data sources must be defined before starting the execution of the case study to ensure the future repeatability. As stated in Section 7.4 we gathered the time that each step during the execution of the case study. We also analyzed all identified embedded SQL statement, all created entities and the created services. By using these information gathered we can draw conclusion and answer the research questions. These information are arranged in both Table 1 and Table 2. The first one depicts (*i*) the phases of our approach, (*ii*) each steps of our approach, (*iii*) the time that each step took during the modernization of the legacy system, and (*iv*) the sum of the time of each step. The second table has nine columns, the first six ones are abbreviated. "SQL-S" stands for SQL Statements, "IS" typifies Identified Statements, "IT" means Identified Table, "CrRS" stands for Created Relevant Services, "CrNRS" means Created Non-Relevant Services, and "NCrRS" stands for Non-Created Relevant Service. The last three columns depict the precision, recall and F1-Score.

Table 1: Time gathered.

	Steps	Time
RE	Parser source-code	9'
	Parser Embedded SQL	12'
	Create JavaModel	7'
	Create SQLModel	16'
	JavaModel_2_KDMSource	11'
	SQLModel_2_KDMData	7' 35"
	TOTAL Phase	62' 35"
Reestructuring	Steps	Time
	Create Entities	15'
	Apply Facade Pattern	9'
	Create WebService	10'
	Create Configuration Files	2'
	TOTAL Phase	36'
FE	Steps	Time
	KDM_2_UML	13'
	UML_2_JavaModel	14' 50"
	UML_2_RestModel	6' 27"
	Generate Source-Code	5' 39"
	TOTAL Phase	40' 33"
TOTAL Modernization		2 hours and 30'

7.6 Analysis and Interpretation

This section presents the case study findings. After the data have been collected, it is analyzed to draw the conclusions. The analysis should obtain the evidence chains from the data to answer both research questions.

As aforementioned, Table 2 summarizes the results related to the precision, recall and F1-score of our approach. As can be seen the precision and recall related related to `Select` statements are 100% and 98.52%, respectively. Which means that in this case our approach automatically created 100% truly relevant services when dealing with the `Select` statements, i.e., no "false negatives" and 98.52% "rate of

Table 2: Precision and Recall.

SQL-S	IS	IT	CrRS	CrNRS	NCrRS	Precision $\frac{CrRS}{CrRS+CrNRS}$	Recall $\frac{CrRS}{CrRS+NCrRS}$	F1-Score $2\frac{precision*recall}{precision+recall}$
Select	382	67	67	0	1	100%	98.52%	
Update	83	40	38	2	3	95%	97.56%	
Delete	33	20	15	5	2	75%	88.23%	
Insert	37	28	27	1	1	96.42%	96.42%	
Total	535	155	147	8	7			
Mean	133.75	38.75	36.75	2	1.75	91.60%		

true positive". In other words, from all created services the approach created 67 services but missed 1, i.e., 98.52%. As for the `Update` statements the precision and recall gently decrease, 95% of all services created related to `Update` statement were relevant and 97.56% off these services are "true positive", respective. Related to the `Delete` statements the precision and recall slightly dropped 75% and 88.23%. Thus, our approach created more irrelevant services for each truly relevant services. As for the `Insert` statements the metrics suddenly rose to precision equals to 96.42% and recall to 96.42%, which means that our approach created 96.42% services truly relevant and also created 96.42% "true positive" ones. Thereby, the $\mathbf{RQ}_1$ can be answered as true, that is, the proposed approach can obtain embedded SQL statements from legacy systems to effectively create services.

To answer the $\mathbf{RQ}_2$ we applied a software equation, see Equation 1. Counting the physical artifacts generated in this case study, we have that 75847 lines of code (SLOC), without considering blank lines and comments, were generated 721 files. By this equation, others can be derived, such as the estimated calculation of minimum development time (in months), see Equation 2.

$$t_{min} = 8.14 * (LOC/P)^{0.43} \text{ as } t_{min} > 6 \text{ months} \quad (2)$$

$$E = 180 * Bt^3 \text{ as } E > 20 \text{ person-month} \quad (3)$$

Setting the values $LOC = 75847$ and according to Pressman (Pressman, 2001) we can set $P = 28000$. According to Pressman this latter value is a typical parameter for commercial application such as the software herein used, PrograduWeb. As applying the Equation 2, we can check that the minimum time spent in development of a similar application is closer to 12.49 months. However, it worth to notice that we spent approximately 3 months to devise the tool used in this case study, thus, the gain of time is 9.49 months. According to Pressman (Pressman, 2001) we can also define another equation to measure the effort in person-month, see Equation 4.

$$E = 180 * Bt^3 \text{ as } E > 20 \text{ person-month} \quad (4)$$

In order to answer the $\mathbf{RQ}_2$ we gathered the time consumed by each phase, see Table 1. In the first phase, reverse engineering, the time spent on identifying all embedded SQL statements, creating the PSMs and the PIMs was a total of 62 minutes and 35 seconds. This phase took more time than the other phases due the fact that two parser need to be executed in order to discovery legacy system's knowledge. All the steps in the second phase, reestructuring took 36 minutes. The steps that spent more time was `Create Entities`. Maybe this came about because the algorithm that is carried out need to create a entity for each identify embedded SQL statement, so, the hard drive need to moving files, causing a gap. Finally, the last phase, forward engineering, spent 40 minutes and 33 seconds. In this phase, the steps that took more time were `KDM2UML` and `UML2JavaModel`. Maybe this came about once the ATL seem to be an easy language to realize model-to-model transformation but it presents efficiency issues when applied to medium and large models (Tisi et al., 2011). As a result, the final time spent by our approach was two hours and 30 minutes.

ACKNOWLEDGEMENTS

REFERENCES

Baresi, L. and Miraz, M. (2011). A component-oriented metamodel for the modernization of software applications. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 179–187.

Brereton, P., Kitchenham, B., and Budgen, D. (2008). Using a protocol template for case study planning. In *Proceedings of EASE 2008*.

Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. (2010). Modisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 173–174. ACM.

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.

del Castillo, R. P., García-Rodríguez, I., and Caballero, I. (2009). Preciso: a reengineering process and a tool for database modernisation through web services. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 2126–2133, New York, NY, USA. ACM.

Demeyer, S., Ducasse, S., and Nierstrasz, O. (2002). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, San Francisco, CA, USA.

Frey, S., Hasselbring, W., and Schnoor, B. (2012). Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. *Journal of Software: Evolution and Process*.

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

Guzman, I. G.-R. d., Polo, M., and Piattini, M. (2007). An adm approach to reengineer relational databases towards web services. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 90–99, Washington, DC, USA. IEEE Computer Society.

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., de Guzman, I. G.-R., Avila-Garcia, O., and Piattini, M. (2009a). On the use of adm to contextualize data on legacy source code for software modernization. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09.

Perez-Castillo, R., Garcia Rodriguez de Guzman, I., Piattini, M., and Piattini, M. (2009b). 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.

Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition.

Putnam, L. H. and Myers, W. (1991). *Measures for Excellence: Reliable Software on Time, within Budget*. Prentice Hall Professional Technical Reference.

Rodríguez-Echeverría, R., Conejero, J. M., Clemente, P. J., Preciado, J. C., and S$#225;nchez-Figueroa, F. (2012). Modernization of legacy web applications into rich internet applications. In *Proceedings of the 11th international conference on Current Trends in Web Engineering*, pages 236–250, Berlin, Heidelberg. Springer-Verlag.

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

Tisi, M., Martnez, S., Jouault, F., and Cabot, J. (2011). Lazy execution of model-to-model transformations. *Springer Berlin Heidelberg*.

Ulrich, W. M. and Newcomb, P. (2010). *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Wang, H., Shen, B., and Chen, C. (2009). Model-driven reengineering of database. In *Software Engineering, 2009. WCSE '09. WRI World Congress on*, volume 3, pages 113–117.

Yin, R. K. (2002). *Case Study Research: Design and Methods*. 3rd edition.