HURDLES IN MULTI-LANGUAGE REFACTORING OF HIBERNATE APPLICATIONS*

Hagen Schink, Martin Kuhlemann, Gunter Saake
Otto-von-Guericke University, Germany
hagen.schink@gmail.com, martin.kuhlemann@ovgu.de, saake@iti.cs.uni-magdeburg.de

Ralf Lämmel
University of Koblenz-Landau, Germany
rlaemmel@gmail.com

Keywords: Refactoring, Multi-language Software Application, Object-relational Mapping, Hibernate, Database

Abstract: Different programming languages can be involved in the implementation of a single software application. In these software applications, source code of one programming language interacts with code of a different language. By refactoring an artifact of one programming language, the interaction of this artifact with an artifact written in another programming language may break. We present a study on refactoring an software application that contains artifacts of different languages.

1 INTRODUCTION

Today, different programming languages are used in concert to implement software applications (Strein et al., 2006; Linos et al., 2007; Chen and Johnson, 2008; Visser, 2008; Ford, 2008). The use of different programming languages allows developers to accomplish specific tasks with less effort. However, artifacts of different languages may interact. For instance, we can use Java together with SQL (Andersen, 2006) and artifacts of both languages must be kept consistent.

A *refactoring* is a transformation which alters the structure but not the semantics of an artifact (Opdyke, 1992; Fowler, 1999). Refactorings exist for several artifact types, e.g. source code of object-oriented programming languages, UML diagrams, and database schemas (Fowler, 1999; Li, 2006; Schrijvers et al., 2004; Van Gorp et al., 2003; Sunyé et al., 2001; Ambler, 2003). However, a refactoring for an artifact of one type does commonly not describe changes that must be done in artifacts of other types.

We applied different refactorings on a software application that contains artifacts of object-oriented and functional programming languages, and a database (Schink, 2010; Schink and Kuhlemann, 2010). In the following we focus on object-oriented and database refactorings for brevity. Based on our observations, we implemented two refactorings and performed experiments. We conclude that a general approach to refactoring multi-language software applications is hard, if not impossible, to automate.

A coupled transformation describes the consistent transformation of software applications after the (possibly semantic-preserving) transformation of a database schema or XML schema (Cunha and Visser, 2007; Cleve, 2009). Our study differs to work on coupled transformations in that (1) we focus on refactorings only, (2) our software application uses an object-oriented mapper to communicate with a relational database, (3) we discuss problems of refactoring a software application on the one hand and a database on the other hand.

*An extended version of this paper has been published as Master thesis and technical report before (Schink, 2010; Schink and Kuhlemann, 2010).

2 BACKGROUND

In this section we define the term describing applications that are implemented by means of different programming languages and describe the idea of refactoring these applications.

2.1 Multi-Language Software Application

A software application is a *Multi-Language Software Application* (MLSA) if it is implemented using different general-purpose and domain-specific languages (Linos et al., 2007). Different languages are combined frequently, e.g. SQL is a standardized query language for databases and was not intended to be a general-purpose programming language (Michels et al., 2003). But it is possible to embed SQL statements in general-purpose programming languages like C++ or Java (ISO/IEC, 2008; Leyderman, 2005; Andersen, 2006). Other examples of language interaction exist (Chen and Johnson, 2008; Harold and Means, 2002; Grogan, 2006). The combination of different programming languages reduces the effort to implement common tasks in software development, e.g. data storage (Fjeldberg, 2008).

2.2 Multi-Language Refactoring

A refactoring is a semantic-preserving modification of a software application (Fowler, 1999). A common refactoring is Rename Field that is used, if the name of a field does not describe the purpose of that field.

Besides source code, a software application may contain documentation, design documents, specifications, and unit tests et cetera (Mens and Tourwé, 2004). An *artifact type* describes a set of artifacts that share a common paradigm. For example we consider artifacts of the object-oriented languages Java and C++ to have a common artifact type. Refactorings for artifacts of individual types exist, e.g. for artifacts of object-oriented (Fowler, 1999), functional (Li, 2006), and logical programming languages (Schrijvers et al., 2004), and others (Van Gorp et al., 2003; Sunyé et al., 2001; Ambler, 2003).

Refactorings for individual artifact types do not describe at all or not in detail how they influence artifacts of other artifact types, so developers are forced to adapt interacting artifact types manually. For instance, consider a class `Employee` and a table `Employee`. Let us assume that the class `Employee` relates to the table `Employee` by name (common with object-relational mappers). Based on the relation, an object-relational mapper retrieves datasets from the table `Employee` and provides these datasets as instances of class `Employee`. If we apply a Rename Class refactoring on class `Employee`, we have to modify the database schema to preserve the relation between class `Employee` and table `Employee`, though, this modification of the database schema is not part of the standard refactoring definition. We call a refactoring that describes a semantic-preserving modification on artifacts of at least two different artifact types a *multi-language refactoring* (MLR).

3 ANALYSIS

HRManager is a software application we implemented to manage employee data (Schink and Kuhlemann, 2010). HRManager is the basis upon which we show effects of refactoring MLSAs. HRManager has been implemented using the object-oriented programming language Java and the functional programming language Clojure. Application data is stored in a relational database. Datasets in the database are accessed through the object-relational mapper Hibernate. As we use HRManager as our running example throughout the paper, we will present the different artifact types and their relations in detail. In this paper we focus on the description of the Java/Hibernate artifacts and the database.

All classes representing domain entities in HRManager, e.g. employees or managers, are implemented in Java. For each class that represents a domain entity a table exists as counterpart in the relational database, whereas table columns represent the states of the respective class. We map the class hierarchy to multiple tables, i.e., a table for each class (other options are possible). Class hierarchies are described by foreign key references between the tables that represent the classes of the respective hierarchy.

Java classes are mapped onto tables of the relational database by an *object-relational mapping* (ORM). We use Hibernate[1] to realize the ORM in HRManager. To describe the ORM between Java classes and the respective tables in the database schema, we use Java annotations defined by the Java Persistence API (DeMichiel, 2009) that is implemented by Hibernate. Listing 1 shows an excerpt of the ORM of class `Employee`. We use the `@Entity` annotation (Line 1) to mark the class `Employee` as persistent class of which instances are stored in the database. With the `@Table` annotation we specify that Hibernate shall store instances of `Employee` in table `employees` (Line 1). Without using the `@Table` annotation, Hibernate would map the class `Employee` to an equally named table `Employee`.

Hibernate maps object states to the respective table columns. We use Hibernate in property-access mode, i.e., we use getter and setter methods to define the properties to store in the database (Keith and Schincariol, 2009).

[1]http://www.hibernate.org

Listing 1: Excerpt of the ORM of the class `Employee`.

```
@Entity @Table(name="employees")
public class Employee implements
      Serializable { ... }
```

3.1 Refactoring an MLSA

We applied a number of refactorings manually to our sample MLSA and evaluated whether the refactorings can be automated. Our aim was to preserve the semantics of HRManager. First, we applied a refactoring to one artifact. If the refactoring broke the interaction with other artifacts, e.g. by introducing compiler or runtime errors, we tried to re-establish the interaction by refactoring the interacting artifacts of possibly different artifact types. We call an MLR on HRManager *successful*, if the MLR describes a set of refactorings for different artifact types, that together preserves the semantics of HRManager. By semantics we refer to the specification of HRManager.[2]

We apply a special definition for semantics on the database to take the existence of persistent data in the database into account. A transformation of a database schema and the related data instances is semantic-preserving, if the transformation is reversible (Hainaut, 1996). For databases, we distinguish two terms of semantic-preservation that describe if a database refactoring can be undone: *reversible* and *symmetrically reversible* (Hainaut, 1996). That is, a transformation $T1$ is reversible, if for $T1$ a transformation $T2$ exists, that undoes $T1$. A transformation of a database is symmetrically reversible, if for $T1$ a transformation $T2$ exists, so that $T2$ is the inverse transformation of $T1$ and vice versa (Hainaut, 1996). Hence, we can undo symmetrically reversible transformation without loosing any data already stored in the database.

In the following, we present the necessary modifications to realize an object-oriented (Pull Up Method) and a database (Introduce Default Value) refactoring on the MLSA HRManager.

3.1.1 Pull Up Method Refactoring

In HRManager, only the class `Manager` provides the methods `getBoss` and `setBoss` to manage the supervisor of a manager. But also employees have a supervisor, though, the class `Employee` (superclass of `Manager`) does not provide any method to manage supervisors. Hence, we want to pull up the methods `getBoss` and `setBoss` from `Manager` to `Employee`. The following modifications are necessary to perform the Pull Up Method refactoring (Fowler, 1999) on HRManager: (1) pull-up method `getBoss` from `Manager` to `Employee`, (2) pull-up field `boss` from `Manager` to `Employee`, (3) pull-up method `setBoss` from `Manager` to `Employee`, (4) move column `boss` and data stored in this column from table `managers` to table `employees`, and (5) update all references to column `boss` of table `managers` to reference column `boss` in table `employees`. Step 2 is necessary, because after being pulled up `getBoss` in `Employee` can no longer access the private field `boss` defined in `Manager`. We use Hibernate in property-access mode, so Hibernate maps getter/setter *pairs* defined in a Java class to columns defined in the database schema; so we need to apply step 3 to restore the getter/setter *pair* `getBoss`/`setBoss` in `Employee`.

Listing 2: Establishing a supervisor relationship between subordinate *S* and boss *B*.

```
UPDATE managers
  SET boss = (SELECT id FROM employees
              WHERE surname = 'B')
  WHERE (SELECT id FROM employees
         WHERE employees.surname = 'S'
         AND employees.id = managers.id);
```

Listing 3: Establishing a supervisor relationship between subordinate *S* and boss *B* after refactoring.

```
UPDATE employees
  SET boss = (SELECT id FROM employees
              WHERE surname = 'B')
  WHERE employees.surname = 'S';
```

The transformation of the database schema described in the Steps 4 and 5 is reversible, because we can move the column `boss` from `employee` back to `managers` without losing any of the original information in column `boss`. However, the transformation is not symmetrically reversible, because with removing the column `boss` from table `employees` (required when inverting the refactoring) tuples of employees that are not managers lose their relation to an employee tuple representing their boss. That is, we cannot guarantee the informational integrity of each tuple in `employees` when undoing the Pull Up Method refactoring. Hence, we may not be able to revert the Pull Up Method refactoring by a reverse refactoring.

In the refactored database schema the column

[2]There exist different semantic definitions for the different languages discussed. As HRManager is a simple program, we refer to the unmodified HRManager source code as specification of HRManager's semantics. An exhaustive discussion of semantics is not in the scope of this paper.

boss is part of table employees and removed from table managers. Listings 2 and 3 show that the modification of SQL statements referencing the column boss can be challenging.[3] In Listing 2, the UPDATE statement introduces a subordinate-boss-relation between the datasets of *B* (boss) and *S* (subordinate). One way to adapt the UPDATE statement in Listing 2 to the new database schema is to swap the table referenced in Line 1 (managers) and the table referenced in the FROM clause in Line 3 (employees). Listing 3 shows an additional modification. We can simplify the WHERE clause in Listing 2, Line 3, by replacing the SELECT statement with a comparison (Listing 3, Line 3). We conclude, there exist at least 2 possible modifications of the UPDATE statement of Listing 2 that differ in the amount of changes to apply. We further conclude that the described transformations can possibly only be realized by means of semantic analysis of the SQL statement at hand (e.g. Listing 2).

3.1.2 Introduce Default Value Refactoring

We use the Introduce Default Value refactoring (Ambler, 2003) to unify already existing default values (in the database itself and in HRManager) by introducing a single default value.

In HRManager, we want to set the default value for the column account in table managers to *acquisition*, because we define that a manager has to report to the account *acquisition*, by default. We have to modify HRManager in the following way to introduce the default value *acquisition*: (1) define the default value *acquisition* for the column account of the table managers by using the keyword DEFAULT, (2) initialize the field account of the class Manager with the value *acquisition*. Step 2 is necessary to preserve the semantics of the default value defined in the database for classes implemented in Java. Consider, we would not have applied Step 2; when we create a new instance of class Manager the field account is initialized with null. When we store this instance in the database, null is written to the column account. The default value of the column account would never be applied to the instances of class Manager.

There can be methods assuming the field account being initialized with null. Those methods would behave differently after refactoring, so the modification described in Step 2 can be semantics-changing.

If property access for Hibernate is set, the modifications in Step 2 may require semantic analysis because from Hibernate's point of view the representation of the state account is hidden in the implementation of its getter/setter methods (Keith and Schincariol, 2009). The analysis of the implementation of getter/setter methods may be achieved statically for trivial implementations, but needs advanced treatment for non-trivial getter/setter methods. In Listing 4, we defined a non-trivial example for the method setAccount. In this method, we parse a parameter of type String and store the parsed values in two different fields accountName (Line 4) and accountID (Line 5). Without semantic analysis of setAccount we would not know how to implement the new default value in the fields accountName and accountID. But static source code analysis may not reveal all needed information (Laski et al., 1998), and, hence, does not offer a general solution to the problem of finding the fields accessed by getter/setter pairs.

Listing 4: Method definition setAccount.

```
1 public void setAccount(String acc) {
2   int l = acc.length();
3   String accID=account.substring(l-3, l);
4   accountName=acc.substring(0,l-3);
5   accountID=Integer.parse(accIDString); }
```

4 EVALUATION

We implemented an MLR version of the Rename Method and the Push Down Method refactoring for applications written in Java, Hibernate, and SQL. The Push Down Method refactoring removes a method definition from a superclass and copies the method definition to all subclasses (Fowler, 1999).

We evaluated the refactorings on applications which use the Rich Internet Application Framework JBoss Seam.[4] Depending on the application, we are able to automate MLRs at least partially. The reason for a partial application of a refactoring is the missing support of certain artifact types by our prototype. We identified all manual modifications required to complete refactorings to be semantic-preserving.

The refactoring of the evaluated software applications is possible, because their implementation is slightly different to the implementation of HRManager (Schink and Kuhlemann, 2010). The different implementation allows the refactoring of an artifact of one type without applying semantics-changing modifications to artifacts of a different artifact type. Hence, a refactoring applied on two MLSAs containing the same artifact types may be successful on one MLSA

[3]The SQL statements are defined with the syntax of SQLite (http://www.sqlite.org).

[4]http://seamframework.org

while the refactoring fails on the other MLSA due to differences in the implementation of the MLSAs.

5 RELATED WORK

The following approaches share the idea of finding commonalities between artifact types. *Generic Refactorings* describe refactorings of concepts, programming languages have in common, for instance methods (Lämmel, 2002). Our study shows that artifact-specific concepts exist which we cannot treat in a generic way, because these concepts do not exist in all languages. So the use cases for Generic Refactorings in multi-language software applications may be limited to languages which share at least some common concepts.

An approach to describe a refactoring in an abstract way is to use meta models of source code. The meta models FAMIX (Tichelaar, 2001), MOOSE (Ducasse et al., 2000), and UML (Van Gorp et al., 2003) are used for describing refactorings of OOP languages independently from the OOP language at hand. Therefore, FAMIX, MOOSE, as well as UML cannot be used to abstract artifacts of MLSAs in general. Another meta model based approach is used in the IDE *X-Develop* (Strein et al., 2006). X-Develop realizes MLR on top of a *Common Meta-Model* which unifies the concepts of the supported languages in a single representation. The authors evaluate the Rename Method refactoring implemented in X-Develop on a project that utilizes different languages. But these languages can be compiled into a common base language, hence, the languages share common properties and, therefore, belong to the same artifact type in our understanding. Refactorings of other artifact types are not considered by the authors.

Some authors analyze and implement renaming for different artifact types (Chen and Johnson, 2008; Kempf et al., 2008). The authors show that the implementation of MLR is possible for certain relations (e.g. frameworks and the corresponding configuration files). We analyzed and implemented refactorings beyond renaming and showed that under certain conditions MLR is not easy to automate.

Coupled Software Transformations or *Co-transformations* are modifications of different interacting artifact types (Lämmel, 2004). Co-transformations describe semantics-preserving as well as semantics-changing modifications (Lämmel, 2004). We argue that a general application of semantics-changing modifications is irreconcilable with the term refactoring. But co-transformations exist for semantics-preserving database schema transformations and the associated program transformations (Cleve, 2009). In (Cleve, 2009) the authors argue that a semantic-preserving transformation of a database schema leads to transformations that do not modify the functionality of related applications. We applied both, object-oriented and database refactorings. Although we applied semantic-preserving transformations, i.e. refactorings, on Java source code and a relational database, we found cases where semantic-changing modifications are hardly avoidable.

In (Cunha and Visser, 2007) another approach to the consistent transformation of a data schema and related queries is presented. Refactorings are not discussed, and transformations of the application source code are not considered.

Some of the problems shown in this paper may be directly related to the *object-relational impedance mismatch* (Carey and DeWitt, 1996). But not all problems, like alternative refactoring realizations and the need for possibly semantics-changing modifications, can be explained by the object-relational impedance mismatch.

6 CONCLUSIONS

We applied an object-oriented and a database refactoring on a multi-language software application (MLSA). The MLSA uses the object-relational mapper Hibernate to communicate with a database. When we applied the refactorings, we observed that (1) a refactoring of artifacts of one artifact type can lead to semantic-changing modifications in artifacts of other artifact types, and that (2) there can be alternative ways to realize a refactoring for artifacts of different types. The alternative ways can differ substantially in the amount of modifications or may not even preserve program-semantics. The latter case leads to conflicts with the refactoring term. We argue that a general approach to automated multi-language refactorings (MLR) covering all possible MLSAs is not feasible due to the different concepts of existing artifact types.

We automated the Rename Method and the Push Down Method refactoring for software applications using the object-relational mapper Hibernate. We conclude that we are able to realize automated MLRs if the conditions described in our study are met.

In an MLSA artifacts of different types are involved. When we refactor an artifact of one type, artifacts of different types may be subject to semantic-changes. Thus, we think that a modified definition of semantics-preservation for MLSA is required. We think that we will not be able to treat a range of potentially useful semantics-preserving modifications of single artifact types on MLSAs as MLRs, otherwise. Therefore, the issue of semantics- preservation in MLSAs is subject of our ongoing work.

REFERENCES

Ambler, S. (2003). *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, Inc., New York, NY, USA.

Andersen, L. (2006). *JDBC ^{TM} 4.0 Specification*. Sun Microsystems, Inc., Santa Clara, USA, final edition.

Carey, M. and DeWitt, D. (1996). Of objects and databases: A decade of turmoil. In *VLDB*. Citeseer.

Chen, N. and Johnson, R. (2008). Toward Refactoring in a Polyglot World: Extending Automated Refactoring Support across Java and XML. *Workshop on Refactoring Tools*.

Cleve, A. (2009). *Program Analysis and Transformation for Data-Intensive System Evolution*. PhD thesis, University of Namur.

Cunha, A. and Visser, J. (2007). Strongly Typed Rewriting For Coupled Software Transformation. *Electronic Notes in Theoretical Computer Science*, 174(1).

DeMichiel, L. (2009). *JSR 317: JavaTM Persistence API, Version 2.0*. Sun Microsystems, Inc., Santa Clara, USA, final edition.

Ducasse, S., Lanza, M., and Tichelaar, S. (2000). MOOSE: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. *CoSET*.

Fjeldberg, H.-C. (2008). *Polyglot Programming*. Master thesis, Norwegian University of Science and Technology, Trondheim/Norway.

Ford, N. (2008). *The Productive Programmer*. O'Reilly.

Fowler, M. (1999). *Refactoring: Improving the Design of existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Grogan, M. (2006). *JSR-223 Scripting for the Java ^{TM} Platform*. Sun Microsystems, Inc., Santa Clara, USA, final edition.

Hainaut, J.-L. (1996). Specification Preservation in Schema Transformations – Application to Semantics and Statistics. *Data & Knowledge Engineering*, 19.

Harold, E. R. and Means, W. S. (2002). *XML in a nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

ISO/IEC (2008). *International Standard ISO/IEC 9075-1 Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*. ISO/IEC, third edition.

Keith, M. and Schincariol, M. (2009). *Pro JPA 2: Mastering the Java Persistence API*. Apress, Berkely, CA, USA.

Kempf, M., Kleeb, R., Klenk, M., and Sommerlad, P. (2008). Cross Language Refactoring for Eclipse plugins. *OOPSLA*.

Lämmel, R. (2002). Towards Generic Refactoring. *ACM SIGPLAN Workshop on Rule-based Programming*.

Lämmel, R. (2004). Coupled Software Transformations. *Workshop on Software Evolution Transformations*.

Laski, J., Stanley, W., and Hurst, J. (1998). Dependency analysis of Ada programs. *ACM SIGAda Ada Letters*, XVIII(6).

Leyderman, R. (2005). *Oracle ® C ++ Call Interface*. Oracle Corporation.

Li, H. (2006). *Refactoring Haskell Programs*. PhD thesis, University of Kent, Canterbury, Kent, UK.

Linos, P. K., Lucas, W., Myers, S., and Maier, E. (2007). A Metrics Tool for Multi-Language Software. *SEA*.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2).

Michels, J.-E., Kulkarni, K., Farrar, M. C., Eisenberg, A., Mattos, N., and Darwen, H. (2003). The SQL Standard. *it – Information Technology*, 45(1).

Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

Schink, H. (2010). *Sprachübergreifende Refactoring Feature Module*. Master thesis, Otto-von-Guericke-University, Magdeburg. Available online: `http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/thesisSchink.pdf`.

Schink, H. and Kuhlemann, M. (2010). Hurdles in refactoring multi-language programs. Technical Report FIN-007-2010, University of Magdeburg, Germany.

Schrijvers, T., Serebrenik, A., and Demoen, B. (2004). Refactoring Prolog Code. *Workshop on (Constraint) Logic Programming*.

Strein, D., Kratz, H., and Lowe, W. (2006). Cross-Language Program Analysis and Refactoring. *Workshop on Source Code Analysis and Manipulation*.

Sunyé, G., Pollet, D., Traon, Y. L., and Jézéquel, J. (2001). Refactoring UML Models. *UML*.

Tichelaar, S. (2001). *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, Switzerland.

Van Gorp, P., Stenten, H., Mens, T., and Demeyer, S. (2003). Towards Automating Source-Consistent UML Refactorings. *UML*.

Visser, J. (2008). Coupled Transformation of Schemas, Documents, Queries, and Constraints. *Electronic Notes in Theoretical Computer Science*, 200(3).