Multi-Language Refactoring with Dimensions of Semantics-Preservation

Hagen Schink
Institute of Technical and Business Information Systems
Otto-von-Guericke-University
Magdeburg, Germany
hagen.schink@gmail.com

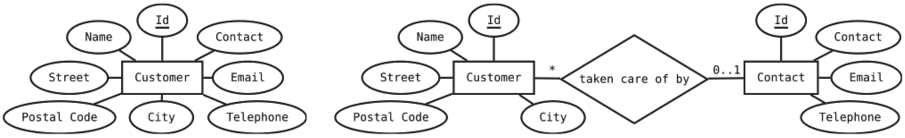
Abstract: Today, software developers utilize different *general-purpose* (GPL) and *domain-specific languages* (DSL) to implement *multi-language software applications* (MLSA). MLSAs, thus, contain artifacts of different GPLs and DSLs, e.g., source-code files and configurations. In a recent study we found that refactoring an artifact can break artifact interaction and that interaction cannot be re-established by additional refactorings. In this paper we propose an approach that supports developers in understanding and adapting changes to artifact interaction due to refactoring.

1 Introduction

Refactoring is a technique to modify a source-code's structure while preserving the source-code's semantics [Fow99]. Originally, refactorings are defined for single programming languages or paradigms. Thus, today refactorings exist for object-oriented and functional programming languages, and relational databases [Fow99, LT08, Amb03].

Today developers use multiple *general-purpose* (GPL) and domain-specific languages (DSL) in concert to implement software applications [SKL06, LLMM06, CJ08, Vis08, For08]. We call a software application implemented by means of different GPLs and DSLs *multi-language software application* (MLSA). An MLSA includes artifacts of different types, i.e., artifacts of different GPLs and DSLs. Developers access (interact with) different artifact types by means of *application programming interfaces* (API). We call a refactoring considering different artifact types of an MLSA and their interaction a *multi-language refactoring* (MLR). Current MLR implementations share the same idea: If a single-language refactoring breaks interaction with other artifact types, apply single-language refactorings to the interacting artifact types and, eventually, re-establish artifact interaction [SKL06, MS12]. But, in general, we cannot assume that suitable refactorings for all affected artifact types exist to re-establish the MLSAs semantics [SKSL11].

But what if no suitable single-language refactorings for interacting artifacts exist? Two options may be considered: (1) revert the initial single-language refactoring or (2) manually apply the necessary modifications to the interacting artifacts. These two options are dissatisfying because (1) a refactoring cannot be successfully applied or (2) manual modifications relying on a developer's intuition are required to complete the refactoring.



(a) Original table layout

(b) Table layout after Split Table refactoring

Figure 1: The database schema (1a) before and (1b) after splitting the table `Customer`.

So, assuming that no suitable refactoring is available and a developer wants to complete a refactoring, how can we support the developer's refactoring effort in an MLSA?

In Section 2 we describe a use case for MLR. Section 3 describes a possible solution to improve the situation for developers applying MLRs. The current state of our work and our methodology is part of Section 4 before we present related work in Section 5 and conclude our work.

2 Motivating Example - Database Interaction

In this section we discuss the effect of the Split Table Refactoring [AS06, p. 145] upon C# code that accesses query results row by row using .Net's DataReader object. Therefor, we apply the Split Table Refactoring on the table `Customer`. Figure 1a shows table `Customer` which, originally, contains two different information: the customer's address and contact person. But certain information may be misleading, e.g. it is not obvious whether the attribute *`email`* belongs to the customer or the contact person. Thus, we split the table to separate the different information from each other. Figure 1b shows the resulting schema of the Split Table Refactoring. After refactoring the schema consists of the tables `Customer` and `Contact`. Table `Contact` holds all attributes related to a customer's contact person. A customer may only have at most one contact person and a contact person may takes care of zero or more customers. The C# application queries all customers and prints the customer's name and contact person details.

With a DataReader object we can access results of an SQL query row by row. Listing 1 outlines the basic usage of a DataReader. In Line 1 we create the DataReader object by executing an SQL statement. In the following Lines 3 to 6, we access the columns of each row of the result set by index.

After splitting table `Customer` the query in Listing 2 is no longer valid because the attributes *`contact`*, *`telephone`*, and *`email`* are, then, attributes of table `Contact`. At first sight this change is not obvious because the broken query will not become visible until runtime. At runtime an error is thrown indicating that certain columns referenced in the SQL query do not exist. When the developer becomes aware of the error, the developer is forced to understand the implications of the relational schema refactoring for the SQL statement in Listing 2 to be able to adapt the SQL statement accordingly. An option to adapt the SQL statement 2 is by defining an appropriate join as shown in Listing 3.

Listing 1: Reading results of query shown in Listing 2.

```
1 using (DbDataReader reader = cmd.ExecuteReader()) {
2  while (reader.Read()) {
3   string custName = reader.GetString(0);
4   string contName = reader.GetString(1);
5   string contTel = reader.GetString(2);
6   string conEmail = reader.GetString(3);
7  // data processing...
8 }
```

Listing 2: Query customer and contact information from the original table `Customer`.

```
SELECT name, contact, telephone, email FROM Customer
```

Listing 3: Query customer and contact information from the refactored table `Customer`.

```
SELECT name, contact, telephone, email FROM Customer
 JOIN Contact ON (Customer.id_contact = Contact.id)
```

3 Improving MLR with Dimensions of Semantics-Preservation

Different APIs exist to interact with artifacts of different types. For instance, Section 2 shows one way of accessing a relational database from an object-oriented language. A different approach is to utilize an object-relational mapper [SKSL11]. So, the complexity of MLR involving relational database artifacts may range from rather simple SQL query modifications in Section 2 to complex modifications involving different artifact types. In general, we question the feasibility of tools for MLR as they exist for single-language refactoring because of the diversity of programming languages and APIs. But the question remains: How can we support developers in their MLR efforts?

Artifact interaction is realized through APIs, which in turn provide types and identifiers, e.g., data-structures and functions. Hence, a modification to types and identifiers may break artifact interaction. Our idea is to make these modifications to types and identifiers visible to developers by extracting and comparing type and identifier states. Type and identifier information are embedded in structures (e.g., tables, classes, procedures, and methods) of different artifact types (e.g., relational databases and object-oriented source-code). We call the different artifact types *dimensions of semantics-preservation* because different artifact types may consist of different syntactic structures supporting different refactorings and, thus, different kinds of semantics-preservaton. Then, it is a developers decision to choose refactorings and modifications to adapt interacting artifacts to the new type and identifier state.

Source code includes type and identifier expectations. In Listing 2, the query expects the columns *`name`*, *`contact`*, *`telephone`*, and *`email`* to be part of table `Customer`. In Listing 1, the code expects strings on the first four positions of the query result (Lines 3

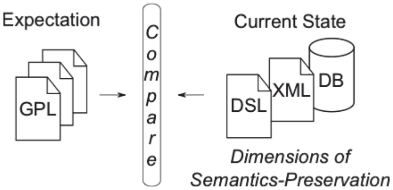


Figure 2: Comparison of types and identifiers extracted from interacting artifacts.

to 6). So, additionally, we may compare type and identifier expectations with the current type and identifier state, as Figure 2 shows, to determine if artifact interaction is broken.

Our idea is to visualize type and identifier states of artifact types in an IDE. Modern IDEs like Eclipse and Visual Studio already augment source code with a variety of information. In particular, developers get detailed information when types mismatch or identifiers cannot be resolved. But these information are only available for languages supported by the IDE at hand. Artifacts not supported by the IDE do not benefit from detailed type and identifier information. We argue that especially type and identifier information can help developers to identify issues after refactoring an MLSA.

4 Methodology and Current State

Basically, our work is based on three main hypotheses: (1) In general, it is not feasible to apply MLRs (semi-)automatically. (2) Only type and identifier information are necessary to understand artifact interaction. (3) Visible information about type and identifier modifications are sufficient to improve the usefulness of single-language refactoring in an MLSA. Based on these assumptions, we first search the literature for an appropriate model to describe and compare the type and identifier information of different artifact types. The next step is to implement front-ends to fill the model with type and identifier information from different artifacts. We plan to implement front-ends for Java and SQLite. Furthermore, we plan to integrate the model and front-ends into the Eclipse IDE as a prototypical plug-in. With the prototype we, then, conduct experiments to investigate the practicability and usefulness of our approach in regard to different use cases.

Currently, we investigate appropriate models to describe the type and identifier information. Furthermore, we started to get into the details of Eclipse Plug-In development.

5 Related Work

In the following, we introduce existing approaches to MLR and discuss their strengths and weaknesses.

The source-code meta-models FAMIX [Tic01], MOOSE [DLT00], and UML [VSMD03] model object-oriented languages. The idea is to use FAMIX, MOOSE, as well as UML to generalize refactorings over a common meta-model. These approaches focus on object-oriented languages, and, thus, may not able to abstract artifacts of MLSAs in general. Another meta-model based approach is implemented in the IDE *X-Develop* upon a *Common Meta-Model* [SKL06]. The authors evaluate a refactoring in X-Develop on an MLSA. But the languages used in the MLSA 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 [CJ08, KKKS08]. The authors show that MLR is possible for certain interactions (e.g. frameworks and the corresponding configuration files). In a recent study, we analyzed and implemented refactorings beyond renaming and showed that under certain conditions MLR is not easy to automate [SKSL11].

Coupled Software Transformations or *Co-transformations* are modifications of different interacting artifact types [Läm04]. Some argue that a semantics-preserving transformation of a database schema leads to transformations that do not modify the functionality of related applications [Cle09]. In a recent study, we applied both object-oriented and database refactorings [SKSL11]. 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 *model-driven architecture* (MDA), platform-specific models (PSM), e.g., classes in programming languages and database schemas in database instances, are generated from a high-level platform-independent model (PIM) [Ste08]. Our approach is different: First, we consider interaction between PSMs. Our approach does not depend on a PIM. Second, we consider refactorings in contrast to arbitrary transformations. Third, because we do not want to support automatic MLR, we may consider artifact-specific concepts if they influence interaction of artifact types.

6 Conclusion

The current idea of *multi-language refactoring* (MLR) requires that for each refactoring that affects artifact interaction a corresponding refactoring exists to re-establish a broken interaction. We argue that due to the plurality of available APIs this approach is hard to realize. We propose to extract type and identifier information of interacting artifact types and to make these information visible to developers, so developers are enabled to apply modifications themselves after a single-language refactoring broke artifact interaction.

References

[Amb03] Scott Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer.* John Wiley & Sons, Inc., New York, NY, USA, 2003.

[AS06] Scott Ambler and Pramodkumar Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[CJ08] N. Chen and R. Johnson. Toward Refactoring in a Polyglot World: Extending Automated Refactoring Support across Java and XML. *Workshop on Refactoring Tools*, pages 1–4, 2008.

[Cle09] Anthony Cleve. *Program Analysis and Transformation for Data-Intensive System Evolution*. PhD thesis, University of Namur, 2009.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. MOOSE: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. *CoSET*, 2000.

[For08] N. Ford. *The Productive Programmer*. O'Reilly, 2008.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[KKKS08] Martin Kempf, Reto Kleeb, Michael Klenk, and Peter Sommerlad. Cross Language Refactoring for Eclipse plug-ins. *OOPSLA*, 2008.

[LLMM06] Panos K. Linos, Whitney Lucas, Sig Myers, and Ezekiel Maier. A Metrics Tool for Multi-Language Software. *Undergraduate Research Conference*, 2006.

[Läm04] R. Lämmel. Coupled Software Transformations. *Workshop on Software Evolution Transformations*, 2004.

[LT08] Huiqing Li and Simon Thompson. Tool Support for Refactoring Functional Programs. *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*, pages 199–203, 2008.

[MS12] Philip Mayer and Andreas Schroeder. Cross-Language Code Analysis and Refactoring. *SCAM*, pages 94–103, 2012.

[SKL06] Dennis Strein, Hans Kratz, and Welf Lowe. Cross-Language Program Analysis and Refactoring. *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 207–216, 2006.

[SKSL11] Hagen Schink, Martin Kuhlemann, Gunter Saake, and Ralf Lämmel. Hurdles in Multi-Language Refactoring of Hibernate Applications. In *Proceedings of the 6th International Conference on Software and Database Technologies*, pages 129–134. SciTePress - Science and and Technology Publications, 2011.

[Ste08] Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software & Systems Modeling*, 9(1):7–20, December 2008.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, Switzerland, 2001.

[Vis08] J Visser. Coupled Transformation of Schemas, Documents, Queries, and Constraints. *Electronic Notes in Theoretical Computer Science*, 200(3), 2008.

[VSMD03] P. Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards Automating Source-Consistent UML Refactorings. *UML*, 2003.