Nr.: FIN-007-2010

Hurdles in Refactoring Multi-Language Programs

Hagen Schink and Martin Kuhlemann

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical report

Nr.: FIN-007-2010

Hurdles in Refactoring Multi-Language Programs

Hagen Schink and Martin Kuhlemann

Arbeitsgruppe Datenbanken

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Hagen Schink
Postfach 4120
39016 Magdeburg
E-Mail: hagen.schink@gmail.com

http://www.cs.uni-magdeburg.de/Technical_reports.html
Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 26.11.2010

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Hurdles in Refactoring Multi-Language Programs

Hagen Schink and Martin Kuhlemann

University of Magdeburg,
Germany
`hagen.schink@gmail.com`, `martin.kuhlemann@ovgu.de`

Abstract. Today, documents of different programming languages can be involved in the implementation of a single software application. These applications are called multi-language software applications. Source code of one programming language may interact with code of a different programming language. By refactoring a document of one programming language the interaction of this document with documents of another programming language may break. We present a study on refactoring multi-language software applications. After that, we automated object-oriented refactorings on a multi-language software application. We evaluate our findings with different case studies and report our results.

1 Introduction

Today, general-purpose and domain-specific languages are used in concert to implement software applications [20, 17, 26, 4, 33, 12][9, p. 169]. The usage of different programming languages allows us to accomplish complex tasks with less effort. However, pieces of code written in different languages may interact. For instance, we can use Java together with SQL [2]. Java allows us to implement complex algorithms, whereas SQL is efficient to describe database queries. In the end, we can use the algorithms defined in Java to process the data queried with SQL. There are other examples of interaction between code of different languages [25, 13][38, p. 143].

A *refactoring* is a code transformation which alters the structure but not the semantics of code [31][10, p. 53]. Refactorings exists for several programming languages and programming paradigms, e.g. for different object-oriented programming languages, UML diagrams, and database schemas [10, 24, 32, 37, 34, 1]. However, a refactoring described for code of one language does not describe the effects on interacting code written in different languages.

We present a study on how to refactor multi-language software applications. We apply refactorings on the different documents of a sample application. We implemented two of the described object-oriented refactorings for the multi-language sample application. We apply the implemented refactorings on a number of different software applications and evaluate our findings. As a result, we describe different effects of refactoring multi-language software applications. In summary, we conclude that a general approach to refactoring multi-language software applications is hard, if not impossible, to implement.

2 Background

In this section we introduce the term multi-language software application and describe challenges of refactoring multi-language software applications.

2.1 Multi-Language Software Application

A software application is a *Multi-Language Software Application* when it is implemented using different general-purpose and domain-specific languages [26]. The usage of different general-purpose and domain-specific languages is referred to as *polyglot programming* [8][9, p. 169].

Polyglot programming is common in modern software development [9]. But the specific usage of polyglot programming differs between programs.

- SQL is a standardized query language for databases and, therefore, was not intended itself as a general-purpose programming language [30]. It is possible to reference SQL statements in general-purpose programming languages like C++ or Java [16, 23, 2].
- XML is used in different application areas mainly for data exchange purposes [15]. XML is also used for describing configuration files or structured text data that can be referenced in general-purpose programming languages like C++ or Java [4, 15].
- C++ and different scripting languages, e.g. Java Script, can be called from or embedded in Java [25, 13]. The interfaces to Java are described by the *Java Native Interface* for C++, and the *Java Specification Request* 223 for scripting languages [25, 13].

Not using polyglot programming would make common tasks in software development more difficult, e.g. database access and data exchange [8, p. 9-10].

2.2 Multi-Language Refactoring

A refactoring is the semantic preserving modification of a program [31][10, p. 53]. A common refactoring is the RENAME FIELD REFACTORING. The RENAME FIELD REFACTORING is used, if the name of a field does not describe the purpose of the field. For instance, we want to refactor a field in the class `Employee`. `Employee` encapsulates a field which stores data of an employee and, therefore, encapsulates the fields `name` and `surname`. Figure 1 shows the application of a RENAME FIELD REFACTORING on the field `name`. By renaming the field `name` to `firstname` we explicitly describe the purpose of the field.

Besides source code a software application may contain documentation, design documents, specifications, and unit tests et cetera [29]. A document type describes a set of documents that share a common paradigm, e.g. source code of object-oriented programming languages, SQL statements, or specifications. For instance, Java and C++ have a common document type, because both are object-oriented languages. Refactorings for different document types exist, e.g.

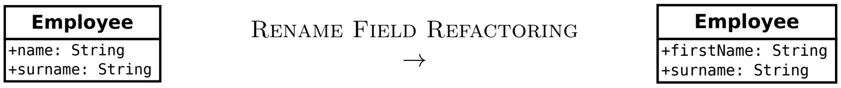


Fig. 1. The figure shows the UML model of the class `Employee` before (left) and after (right) the application of the Rename Field Refactoring.

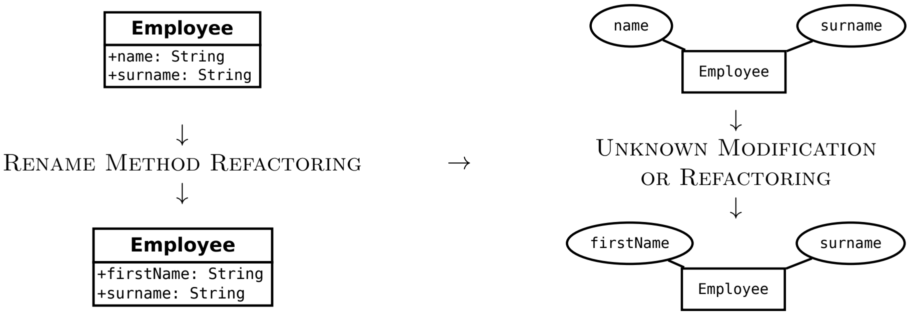


Fig. 2. The figure shows the initial Rename Method refactoring (left) and the respective modification of the database schema (right).

for object-oriented, functional, and logical programming languages, UML diagrams, and database schemes [10, 24, 32, 37, 34, 1]. These refactorings do not describe at all or not in detail how they influence different document types. For instance, consider a class `Employee` and a table `Employee` as shown in Fig. 2. We assume that the class `Employee` relates to the table `Employee` by name. Based on the relation, a software tool is able to retrieve a dataset from the table `Employee` and to provide that dataset as an instance of class `Employee`. The class `Employee` and the fields `name` and `surname` are connected to the table `Employee` and the respective columns `name` and `surname` defined in that table. We apply a Rename Field Refactoring on the field `name`. To preserve the relation between field `name` and column `name` we have to modify the database schema, too, though, the modification of the database schema is not part of the refactoring.

3 Refactoring a Multi-Language Software Application

HRManager is a rudimentary software application implemented by the authors to manage employee data. HRManager founds the basis upon we show effects of refactoring multi-language software applications. HRManager has been implemented using two programming languages, an object-relational mapper, and a database. We applied a number of refactorings on HRManager. Figure 3 shows the document types used in HRManager and how respective documents interact. We use HRManager as our running example throughout the paper, so we will present the different document types and their relations in detail.

Java is used to declare classes, e.g. `Employee`, `Manager`, and `Department`. Figure 4 shows the class hierarchy of `Employee`. All classes of HRManager have

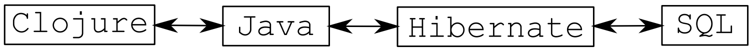
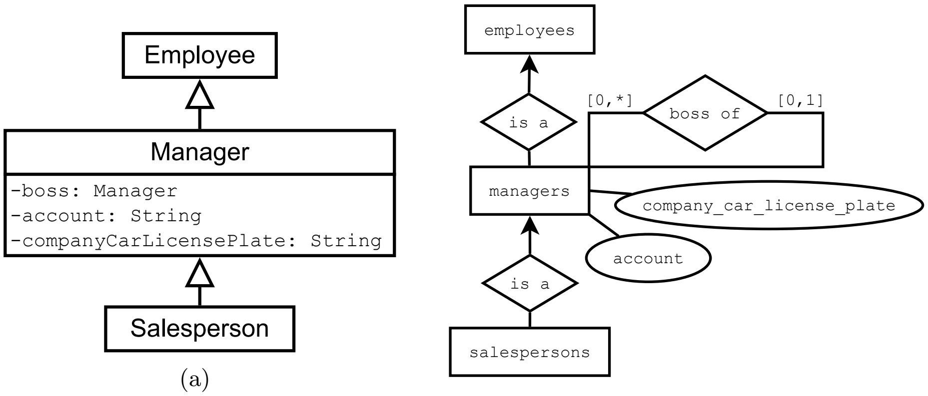


Fig. 3. The different document types in HRManager and their relation.



(a)

(b)

Fig. 4. The class hierarchy of the superclass `Employee` (a) and the ER model of the respective database layout (b).

a counterpart in a relational database, i.e. the database schema defines the tables `employees`, `managers` and a column for every field in the classes. We have two options to map the class hierarchy to the database schema: in one table or multiple tables. We map the class hierarchy to multiple tables. In that approach, class hierarchies are emulated by foreign key references between the tables in the database, e.g., a tuple of the table `managers` has a foreign key reference to the key of the table `employees` because class `Manager` is a subclass of class `Employee` (cf. Fig. 4).

Hibernate[1] maps classes and their fields onto their counterparts in the relational schema. This connection is called *object-relational mapping* (ORM). To connect Java classes with the respective tables in the database schema, we use Java annotations.[2] Listing 1 shows an excerpt of the ORM of class `Employee`. HRManager uses the `@Entity` annotation (Line 1) to make Hibernate map the class `Employee` onto the database. In Line 2, we specify the table name `employees` for the class `Employee` with the `@Table` annotation. Without using the `@Table` annotation, Hibernate maps the class `Employee` to an equally named table `Employee` (case-insensitive).

Like classes on tables, Hibernate maps class attributes to the respective table columns. By default Hibernate uses the setter and getter methods `setName` and `getName` to map the respective class attribute onto the column `name` (cf. Listing

---

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

[2] Another option is to define the mapping in an XML file.

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

```java
@Entity
@Table(name="employees")
public class Employee implements Serializable {
    /* snip further attributes */
    private String name;

    /* snip further methods */
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Listing 2. Application of the `@Column` annotation.

```java
@Column(name = "employee_name")
public String getName() {
    return name;
}
```

1) [18, p. 73]. With the `@Column` annotation we can override the default behavior. Listing 2 shows how we map the getter and setter methods of `name` onto the column `employee_name`.

In HRManager, we use the scripting-facilities of the functional programming language Clojure[3] to compute the overall salary of employees and to find employees with certain attributes. Using Clojure, we can modify parts of the application logic without recompiling the application. Clojure allows us to access methods defined in Java from Clojure and vice versa. In Java, we build references to Clojure functions by method `var` of class `RT` [38, p. 149-150]. Listing 3 shows in Line 1 how the Clojure function `sumSalary` defined in the namespace `scripting` is referenced from Java code.

3.1 Applying Refactorings on *HRManager*

In the following, we report on effects we observed when we applied a number of refactorings on HRManager. We applied all refactorings manually and evaluate whether the refactoring can be automated. We call a refactoring on HRManager *successful*, if a set of refactorings exists, that preserves the semantics of HRMan-

---

[3] http://clojure.org

Listing 3. Referencing the Clojure function `sumSalary` from the Java source code.

```
Var sumSalary = RT.var("scripting", "sumSalary");
float sum = (Float)sumSalary.invoke(managers);
```

ager. By semantic we refer to the specification of HRManager[4]. The specification describes the desired behavior of HRManager regardless of document types. That is, we are able to evaluate the correct behavior of HRManager after applying an MLR for all document types existing in HRManager.

For the database, we distinguish two terms of semantic preservation that describe if a database refactoring can be undone: *reversible* and *symmetrically reversible* [14]. A transformation of a database schema and the related data instances is semantic-preserving, if the transformation is reversible [14]. That is, for transformation $T1$ a transformation $T2$ exists, that undoes $T1$. A transformation of a database schema and the related data instances is symmetrically reversible, if for $T1$ a transformation $T2$ exists, so that $T2$ is the inverse transformation of $T1$ and vice versa [14]. Hence, we can undo symmetrically reversible transformation without loosing any data.

Rename Method Refactoring is used when the name of a method does not describe the purpose of the method correctly [10, p. 273]. In HRManager, class `Employee` and its method `getSalary` are defined but the method's name *getSalary* does not describe the purpose of the method. The method `getSalary` returns the monthly salary, so we rename the method to `getMonthlySalary`. We must perform the following actions to preserve the semantics of HRManager:

1. rename `getSalary` to `getMonthlySalary`
2. rename `setSalary` to `setMonthlySalary`
3. restore the ORM by choosing one of the following alternatives
   (a) rename column `salary` in table `employees` to `monthlysalary`
   (b) add `@Column` annotation to the method `getSalary`
      i. set the `name` attribute of the `@Column` annotation to the column name `salary`.

By default, Hibernate maps getter/setter pairs defined in the Java class on columns defined in the database schema, so we need to apply the 2nd step to restore the Hibernate mapping.

We made two interesting observations when we performed this refactoring. First, we had to refactor a document twice, that is we rename the methods `getSalary` of the class `Employee` and `setSalary` of the same class (see Steps 1 and 2). Second, in Step 3 we have the choice between two actions for restoring the ORM. If we choose the first action (Step 3a) we have to rename the column

---

[4] As HRManager is a simple software application, we refer to the unmodified HRManager source code as specification.

Listing 4. Utilizing the `@Column` annotation for restoring the ORM.

```
1 @Column(name = "salary")
2 public float getMonthlySalary() {
3     return salary;
4 }
```

and we must change an unknown amount of SQL statements referring to that column. The second action (Step 3b) includes two modifications. Listing 4 shows the `@Column` annotation in Line 1. We use the attribute `name` of the annotation to restore the ORM to the column `salary` of the database table `employees`.

In comparison, the modifications described in Step 3a and Step 3b differ in their complexity. Step 3b saves us the modification of SQL statements at all. Furthermore, by saving the modification of SQL we also prevent the clash with keywords. For instance, if we rename a method to `getTable`, we have to rename the database column to `table` too. But in SQL `TABLE` is a reserved keyword, hence, we cannot rename the database column to `table` without provoking database errors and thus we would have to abort the MLR.

Pull Up Method Refactoring unifies one or more methods in a superclass, whereas the method is or can be used in the same manner in different subclasses [10, p. 322]. In HRManager, only the class `Manager` provides methods `getBoss` and `setBoss` to manage the supervisor of a manager. But also employees have a supervisor, though, the class `Employee` does not provide any methods to manage supervisors. Hence, we want to pull up the methods `getBoss` and `setBoss` from `Manager` to `Employee`. The following modifications are necessary to preserve the semantics of 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` from table `managers` and all related data instances to table `employees`
5. update all references to column `boss` of table `managers` to reference column `boss` in table `employees`

Step 2 is necessary, because `getBoss` in `Employee` cannot access the field of its subclass `Manager`. By default, Hibernate maps pairs of getter/setter methods defined in a Java class on columns defined in the database schema, so we need to apply the 3rd step to restore the getter/setter pair `getBoss`/`setBoss` inside class `Employee`.

The transformation of the database schema informally described by the Steps 4 and 5 is reversible, because we can move the column `boss` from `employee` back to `managers` without loosing any of the original information in column

Listing 5. Establishing a supervisor relationship between the manager *Greenspan* and the supervisor *Gartner*.

```
1  UPDATE managers
2     SET boss = (SELECT id FROM employees WHERE surname = 'Gartner')
3     WHERE (SELECT id FROM employees
                 WHERE employees.surname = 'Greenspan'
                 AND employees.id = managers.id);
```

Listing 6. Establishing a supervisor relationship between the manager *Greenspan* and the supervisor *Gartner* after the Pull Up Method refactoring is applied.

```
1  UPDATE employees
2     SET boss = (SELECT id FROM employees
                    WHERE surname = 'Gartner')
3     WHERE employees.surname = 'Greenspan';
```

`boss`. Therefore, we call the Pull Up Method refactoring an MLR in HRManager. However, the transformation is not symmetrically reversible, because with removing the column `boss` from table `employees` (required when inverting the refactoring) tuples of pure employees loose the relation to a 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 without loosing information.

The modification of other SQL statements referencing the column `boss` can be challenging as Listings 5 and 6 show.[5] In Listing 5, the `UPDATE` statement introduces a subordinate-boss-relation between the datasets of *Gartner* (boss) and *Greenspan* (subordinate). One way to adapt the `UPDATE` statement in Listing 5 to the new database schema is to swap the table referenced in Line 1 (`managers`) and the table referenced in the `FROM` clause of the `SELECT` statement in Line 3 (`employees`). Listing 6 shows an additional modification. We can simplify the `WHERE` statement in Listing 5, Line 3 by changing the `SELECT` statement to a comparison (Listing 6, Line 3). Therefore, there exist at least 2 possible modifications of the `UPDATE` statement in Listing 5 that differ in the amount of changes to apply and may also differ in their performance (assuming that the comparison provides a better performance than the nested `SELECT` statement). Furthermore, we argue that the transformations described can only be accomplished by semantic analysis of the source statement (e.g. Listing 5). In our opinion, only by the structure of SQL we cannot fathom how to change `UPDATE` statements like the one in Listing 5 in general.

---

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

Move Class Refactoring changes the superclass of a class to allow reuse of the class's functionality [31]. The new superclass can be part of the current class hierarchy or be part of a different one. We examine moving a class within a class hierarchy.

In HRManager, the class `Salesperson` extends the class `Manager`, because managers and salespersons share the attribute *company car* (see Figure 4). But in reality, salespersons are no managers, hence, we want to change the superclass of `Salesperson` to `Employee`. Therefore, we apply the Move Class refactoring as follows:

1. copy the fields `account`, `companyCarLicensePlate` and their respective getter and setter methods from class `Manager` to class `Salesperson`
2. change the superclass of `Salesperson` to `Employee`
3. copy the columns `account` and `company_car_license_plate` from the table `managers` to the table `salespersons`
4. in the table definition of `salespersons` change every foreign key relation from table `managers` to table `employees`
5. change SQL statements accessing datasets in the table `managers`, if the datasets belong to `salespersons`

The database transformation described by the Steps 3 to 5 are reversible, because we can undo the changes described without loosing any data of the original table `salespersons`. Furthermore, the transformation is symmetrically reversible, because datasets in the tables `salespersons` and `managers` are unambiguously identifiable by the id in table `employees`. That is, we can undo the changes of the Move Class refactoring without violating the data integrity. Thus, these steps can be considered a database refactoring. However, because `Salesperson` is not a `Manager` anymore, code that assumes semantically all instances of `Salesperson` being part of the set of `Manager` instances is broken. Therefore, we can only call the Move Class refactoring an MLR when there is no code assuming salespersons to be a subset of managers. We cannot detect this automatically.

Introduce Default Value Refactoring introduces a default value for a table column [1]. We use the Introduce Default Value refactoring to unify already existing default values (in the database itself or in applications using the database) by introducing a single default value for a column in a database table [1].

In HRManager, we want to set the default value to *Akquise* for the Column `account` defined in the table `managers`, because a manager has to report to the account *Akquise*, by default. We have to modify HRManager in the following way to introduce the default value *Akquise*:

1. define the default value *Akquise* for the column `account` in the table `managers` by using the keyword `DEFAULT`
2. initialize the field `account` of the class `Manager` with the value *Akquise*

Step 2 is necessary to preserve the semantics of the default value defined in the database for classes defined in Java. Consider, we would not have applied

Listing 7. Method definition `setAccount`.

```
1 public void setAccount(String account) {
2     int len = account.length();
3 
4     this.accountName = account.substring(0, len - 3));
5     this.accountID = Integer.parse(account.substring(len - 3, len));
6 }
```

Step 2. When we create a new instance of the class `Manager` the field `account` is initialized with `null`. When we store the instance of the class `Manager` in the database, `null` is written to the column `account`. The default value of the column `account` is never applied to the instances of class `Manager`.

The modification described in Step 2 can be semantic-changing, because there can be methods assuming the field `account` being initialized with `null` instead of being set to null after initialization. Those methods would behave differently after the refactoring. In Step 2, setting the initial value requires semantic analysis of the getter/setter methods of the field `account`. The semantic analysis can hardly be automated. By default, Hibernate maps the methods `getAccount` and `setAccount` onto the column `account`. But Hibernate does not know the fields modified by the methods `getAccount` and `setAccount`. Thus, only the implementation of the methods `getAccount` and `setAccount` can provide the information which field we have to initialize. The analysis of the implementation of getter/setter methods is not hard for trivial implementations, but needs advanced treatment for non-trivial getter/setter methods. In Figure 7, we defined a non-trivial example for the setter method `setAccount`. In the method `setAccount`, 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 apply a default value defined in the database on the fields `accountName` and `accountID`. Hence, by the semantic analysis the refactoring becomes more complex.

During the application of the Introduce Default Value refactoring we have identified two problems of refactorings in a multi-language software application. First, we cannot guarantee that the Introduce Default Value refactoring preserves the semantics of HRManager (dealing with `null` values). Furthermore, we need to analyze the semantics of getter/setter methods to set the initializing value for fields correctly.

Introduce Redundant Column Refactoring creates a copy of a column of a source table in a target table, if the column of the source table is queried frequently when a dataset of the target table is queried [1, p. 409]. In Figure 5, the tables `employees` and `departments` are related. Each time we query a dataset from the table `employees` we also query the name of the `department` referenced by the queried dataset. By creating a copy of the column `name` in table `employees`

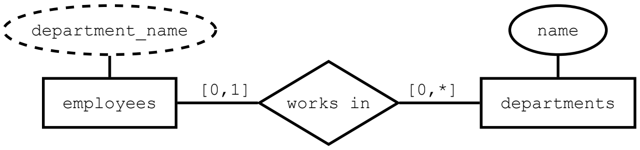


Fig. 5. Extended ER schema showing the entities `employees` and `departments`, whereas attribute `department_name` of `employees` is derived from attribute `name` of `departments`.

no joins remain necessary to retrieve the department an employee is working for. The decrease of join operations may result in a performance gain for certain SQL queries. The following steps are necessary for the Introduce Redundant Column refactoring:

1. create a copy of the column `name` in the table `employees` with the name `department_name`
2. copy all entries from column `name` to the column `department_name`
3. create database triggers to preserve the data consistency between the columns `name` and `department_name`

Additionally, we have to apply the following modifications to make the performance gain available in Java:

4. add a field `department_name` with getters and setters to the class `Employee` as required by Hibernate
5. extend the functionality of the classes `Employee` and `Department` to maintain the consistency between the field `department_name` and `name`

The Steps 4 and 5 are not necessary to preserve the functionality of HRManager. But, if we do not execute Steps 4 and 5 we cannot profit from the performance gain available through the database schema.

The modifications described in Steps 1-3 conform to the steps in the refactoring definition and are semantic preserving [1, p. 409]. Hence, we can call the modifications of Steps 1-3 an MLR.

The extension of functionality described in Step 5 violates the definition of refactorings. The extension of functionality includes securing field `department_name` in class `Employee` against unauthorized writes (only Hibernate and the referenced instance of type `Department` may write the field) and the implementation of the Observer Pattern [11, p. 293] to preserve the consistency between the department name in instances of `Employee` and `Department`. Thus, the modifications described in the Steps 1 to 5 do not adhere to the definition of MLR, because Step 5 does not describe a refactoring. Only the modifications in the Steps 1-3 preserve the semantics of HRManager. Thus, we found two alternate ways to apply the Introduce Redundant Column refactoring on HRManager.

Listing 8. Definition of the function `sumSalary`.

```
1 (def sumSalary (fn [x]
2    (if (and (not (empty? x))
3             (not (instance? hrm.Employee (first x))))
4       (throw (new java.lang.IllegalArgumentException))
5       (if (empty? x) 0 (+ (. (first x) getSalary)
6                           (sumSalary (rest x)))))))
```

Remove Table Refactoring removes a table from a database schema, if the table is deprecated or not used [1].

In HRManager, the table `external_staff` stores information about staff employed through external contractors. Because the Table `external_staff` is not used anymore, we want to remove the table from HRManager. We have to modify the HRManager in the following way:

1. remove the SQL definition of the table `external_staff`
2. remove class `ExternalStaff` from the mapping file of Hibernate
3. ensure that class `ExternalStaff` is not used in conjunction with the ORM

The mapping file modified in step 2 is specific to Hibernate. Thus, the step may be obsolete or different to other ORM frameworks in general.

As long as the class `ExternalStaff` is not used in HRManager no problems arise while we apply the Remove Table refactoring. If `ExternalStaff` is still in use, we have to abort the Remove Unreferenced Class refactoring (Step 3) and undo modifications already applied to HRManager (Step 1).

Introduce New Definition Refactoring defines a local definition for an unnamed expression [24].

In HRManager, we defined the Clojure function `sumSalary` which computes the total salary of all instances of the class `Employee` in `x`. Listing 8 shows the definition of function `sumSalary`. In Line 3, with the unnamed expression `( instance? hrm.Employee (first x))` we test if the first element of list `x` is an instance of the class `Employee` (in the following we call this expression *instance expression*).

We want to apply the Introduce New Definition Refactoring in order to create a function `isEmployee?` out of the instance expression. Therefore, we need to apply the following modifications to HRManager:

1. enclose the instance expression with a `letfn` statement
2. define the Function `isEmployee?` with the instance expression as body within the `letfn` statement defined in step 1
3. within the body of the `letfn` statement, replace the instance expression by a call to the new function `isEmployee?`

Listing 9. The function `sumSalary` with the additional `let` (Line 3) statement defining the function `isEmployee?`.

```
1 (def sumSalary (fn [x]
2     (if (and (not (empty? x))
3               (not (letfn [(isEmployee? [x]
4                     (instance? hrm.Employee x))]
5                       (isEmployee? (first x)))))
6     (throw (new java.lang.IllegalArgumentException))
7     (if (empty? x) 0 (+ (. (first x) getSalary)
8                          (sumSalary (rest x))))))))
```

With the `letfn` statement introduced in Step 1 we can define named expressions. The named expression defined with `letfn` is visible within the body of the `letfn` statement. Listing 9 shows the refactoring result, i.e., the definition of the function `isEmployee?` in Line 3 and the body of the function `isEmployee?` in Line 4. We can use the function `isEmployee?` within the body of the `letfn` statement as shown in Line 5.

After the Introduce New Definition refactoring we do not apply further modifications on the Java artifacts because the instance expression itself was missing a name which could be referenced by Java or other documents. Hence, since there are no other effects, we can call the Introduce New Definition refactoring an MLR.

Promote Definition Refactoring increases the scope or visibility of a definition, so the definition can be used by other functions [24].

In HRManager, we defined the function `isEmployee?` with a `letfn` statement, as shown in Listing 9, Line 3 and 4. That is, the function `isEmployee?` is only visible within the scope of the `letfn` statement (Line 5). We want to increase the visibility of `isEmployee?`, such that we can reuse `isEmployee?` in other functions, too. To promote the definition `isEmployee?` into a new, globally visible function `isEmployee?` we need to apply the following modification to HRManager:

1. introduce the new function definition `isEmployee?` in the global scope
2. let the body of the `letfn` statement be the new body of the Function `isEmployee?` introduced in step 1
3. remove the `letfn` statement from the Function `sumSalary`

The Listing 9, Line 1, shows the function `isExternalStaff?` introduced by the Promote Definition refactoring. The `letfn` statement is removed, only the body is preserved (Listing 10, Line 5).

Because the function `isEmployee?` was not visible before the Promote Definition refactoring, there are no Java documents which reference the function `isEmployee?`. Thus, we do not need to apply further modifications to Java code, so we call the Promote Definition refactoring an MLR.

Listing 10. The function `sumSalary` with the globally visible definition of `isEmployee?`.

```
(def isEmployee? (fn [x] (instance? hrm.Employee x)))

(def sumSalary (fn [x]
   (if (and (not (empty? x))
            (not (isEmployee? (first x))))
   (throw (new java.lang.IllegalArgumentException))
   (if (empty? x) 0 (+ (. (first x) getSalary)
                       (sumSalary (rest x)))))))
```

Listing 11. An excerpt of the reference to the function `managersWithBoss` in Java after the application of the Move Definition refactoring.

```
RT.var("management","managersWithBoss");
```

Move Definition Refactoring describes how functions can be moved between different namespaces [24]. Clojure provides namespaces to group functions [38, p. 24].

In HRManager, the function `managersWithBoss` is defined in the namespace `salary`. The namespace `salary` defines functions for the computation of salaries. The function `managersWithBoss` computes employees who have a supervisor. Thus, the function `managersWithBoss` is not related to the namespace `salary`, we want to move the function to the namespace `management`. We need to perform the following modifications to change the namespace:

1. copy function `managersWithBoss` to namespace `management` and remove the function from the namespace `salary`
2. modify calls to `managersWithBoss` from Java documents

Listing 11 shows how calls to `managersWithBoss` must look like in the Java source code after performing Step 2.

In Java, we resolve dependencies to missing classes by using Java's `import` statement. For functions defined in Clojure we have to use the Java class `RT` and the method `var`, respectively. Hence, we use the Clojure-specific Java class `RT` to reference functions defined in Clojure instead of `import` statements. With this, we have to take language-specific functions into account for MLRs.

4 Evaluation of MLRs

We implemented an MLR version of the Rename Method and the Push Down Method refactorings for programs written in Java, Hibernate, and SQL.[6] The

---

[6] Currently sophisticated tools for the modification of Clojure source code are missing. Therefore, we have not automated any refactorings for Clojure.

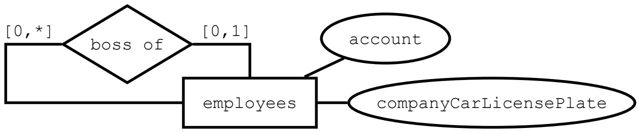


Fig. 6. Representation of the class hierarchy shown in Figure 4 by a single entity.

Push Down Method refactoring removes a method definition from a superclass and copies the method definition to all subclasses.

We evaluated the refactorings on applications which use the Rich Internet Framework JBoss Seam.[7] We applied the refactorings on a Seam project created by the Rapid Application Development (RAD) tool Seam-gen and on the demonstration projects *Seam Space* and *DVD Store* delivered with Seam. All classes that have been refactored are part of an ORM with hibernate, so there are always at least two document types involved, documents of Java and of Hibernate.

In HRManager, each class instance of a class hierarchy is stored in a separate table in the database (see Figure 4(b)). In contrast, in Seam Space, DVD Store, and the generated Seam-gen project, classes of a class hierarchy are stored in a single table. Figure 6 visualizes the single table approach for the class hierarchy in Figure 4(a).

Generated Seam project First, we refactored the Seam project generated by the RAD tool Seam-gen. With Seam-gen, we also added basic user management functionality to our Seam project which adds additional classes. Then, the project consists of 54 different files of 5 different file types with 3266 lines of code (LOC) altogether. Thereof, 6 lines of SQL source code and 257 lines of Java source code.[8] Because the additional classes are not part of a class hierarchy, we could not apply the Method Push Down refactoring. The Seam project and the user management is usable right after the generation, so we are able to evaluate the correctness of our refactoring implementation. One of the classes added by Seam-gen is `UserRole`. In `UserRole`, the method `getName` is defined which we renamed to `getRoleName` automatically with our tool. After the refactoring, the generated project is as usable as before. The preservation of the getter/setter pair `getName` and `setName` as well as the preservation of the correct reference to the column `name` in the database is done automatically by the implementation of the Rename Method refactoring. We do not have to apply additional modifications because of the Seam-specific `@RoleName` annotation which labels the original method `getName`. `@RoleName` labels the method that returns the role name of instances of `UserRole`. Due to the `@RoleName` annotation, we cannot break references to the original method `getName`. Hence, our MLR implementation may only work without further modifications if the tool-specific `@RoleName`

---

[7] http://seamframework.org

[8] All measurements of LOC were taken with cloc (http://cloc.sourceforge.net).

Listing 12. The original and refactored HQL statement in the DVD Store demonstration project.

```
-- original statement
select sum(i.quantity) from Inventory i
```

```
-- refactored statement
select sum(i.numberOfProducts) from Inventory i
```

annotation is used. Therefore, we have to consider the effect of the tool-specific `@RoleName` annotation when applying the Rename Method refactoring on the the generated Seam project.

DVD Store The demonstration project *DVD Store* is an online DVD store implementation. The project consists of 73 different files of 6 different file types with 6886 lines of code (LOC) altogether. Thereof, 3794 lines of SQL source code and 1828 lines of Java source code. In *DVD Store*, an instance of the class `Inventory` stores the amount of dvds available for a certain movie. In `Inventory`, the method `getQuantity` is defined. The method `getQuantity` returns the amount of dvds possessed by the dvd store. We renamed the method `getQuantity` to `getNumberOfProducts`. Besides the automatic modifications, we had to modify a HQL[9] statement manually. This modification is semantic preserving because we renamed a method reference [35]. Listing 12 shows both, the original and the refactored HQL statement querying the amount of all dvds in the database.

We applied the Push Down Method refactoring to *DVD Store*. In the class `User` the method `getFirstName` is defined. In *DVD Store*, the method `getFirstName` is only used in conjunction with the subclass `Customer`. Therefore, we want to push down the method `getFirstName` from the class `User` into the subclasses `Customer` and `Admin` (Figure 7 shows the class hierarchy of `User`). After the push no further modifications are necessary for several reasons. First, no modification of the database is necessary because the entire class hierarchy is represented by a single table and, therefore, we do not need to move the column *firstname* between tables. Second, the getter/setter pair `getFirstName` and `setFirstName` is preserved by the refactoring implementation through renaming the setter when the getter has changed and vice versa. At last, in all documents the method `getFirstName` is called only on instances of the static type `Customer`, so we do not have to change the static type of these instances by casts.

Seam Space The *Seam Space* project implements a rudimentary social network. The project consists of 53 different files of 6 different file types with 7899 lines of code (LOC) altogether. Thereof, 36 lines of SQL source code and 1956 lines

---

[9] The *Hibernate Query Language* (HQL) allows us to query objects instead of relations from a database.

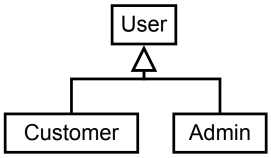


Fig. 7. The class hierarchy of `User` in the Seam DVD project.

Listing 13. The original and refactored JSF EL statement in the Seam Space demonstration project.

```
/* original statement */
register.member.dob
```

```
/* refactored statement */
register.member.dateOfBirth
```

of Java source code. The information about the users of *Seam Space* is stored in instances of the class `Member`, which in turn are stored in the database. The class `Member` defines the method `setDob`. Because the purpose of `setDob` is not obvious on the first sight, we renamed `setDob` to `setDateOfBirth`. The preservation of the getter/setter pair `setDob` and `getDob` as well as the preservation of the correct reference to the column `dob` in the database is done automatically by our tool. Besides the automatic modifications, we had to modify *JSF Expression Language* (EL)[10] statements in the unit test `testRegister` defined in the class `RegisterTest` and in `register.xhtml`. The modifications of the EL statements are semantic-preserving because we changed the method reference accordingly to the renamed methods `getDateOfBirth` and `setDateOfBirth`. Listing 13 shows both, the original and the refactored JSF EL statement calling the method `getDateOfBirth`.

Table 1 summarizes the evaluation results. Within Table 1 we distinguish fully automatic refactorings (only A checked) and refactorings where we made manual adjustments during refactoring (A and M checked). The table also labels refactorings we have not applied.

5 Related Work

In the following we present different approaches to MLR. We argue, that all the different approaches consider language features which exist in all of the different documents or document types. To give an example, consider the Rename Method refactoring. We can apply the Rename Method refactoring to source code of

---

[10] With the JSF Expression Language we can access fields of managed beans [3]. As a simplification, with JSF EL we can call Java methods from within (X)HTML.

	Refactorings			
	Rename Method		Push Down	
Application	*A*	*M*	*A*	*M*
Seam-gen	×		n.a.	
DVD Store	×	×	×	
Seam Space	×	×	n.a.	

(A) *automatic*; (M) *manual modification*; (n.a.) *not applied*

Table 1. Results of the evaluation of multi-language refactorings.

object-oriented programming languages. Furthermore, we can apply the Rename Method to JSF documents because these documents also have a notion of method calls [4]. Therefore, they do not have to discuss effects as presented in our paper.

The main idea of all approaches presented in the following for describing MLRs is to find commonalities between all considered document types. This idea appears for instance in the term *Generic Refactoring* [21]. Generic refactorings modify language features that all programming languages share. For instance, we can describe the Rename Method refactoring as a Generic Refactoring because most modern programming languages share the notion of functions or methods to define behavior of programs. Generic refactorings only consider documents of programming languages [21]. Furthermore, the application of generic refactorings is limited to features shared by all programming languages.

An approach to describe a refactoring in an abstract way is to use meta models of source code. The meta models FAMIX and MOOSE are used for describing refactorings of object-oriented programming languages independently from the OOP language at hand [7, 27, 28, 36, 27]. Therefore, FAMIX as well as MOOSE cannot be used to abstract the diverse documents of a multi-language software application. Another meta model based approach is used in the IDE *X-Develop* [33]. X-Develop realizes MLR on top of a *Common Meta-Model.* X-Develop uses Front-Ends to transform source code of different programming and special purpose languages, e.g. C#, Java, and ASP, to the common meta-model. The authors evaluate the Rename Method refactoring implemented in X-Develop on a project that utilizes C#, J#, Visual Basic, and the Common Intermediate Language (CIL). C#, J#, and Visual Basic are object-oriented programming languages, moreover, all three languages can be compiled to CIL code. Obviously, C#, J#, and Visual Basic share common properties and are already related from beginning, and, therefore, belong to the same document type. Refactorings of other artifact types are not considered by the authors.

Refactoring Unified Modeling Language (UML) models is another approach to MLR in two respects. First, UML provides a set of diagrams to describe the different aspects of a software application. If we refactor an instance of one diagram, we have to modify instances of other diagrams accordingly [34]. But there exist known limitations of the UML meta-model, e.g. missing relations between different models or missing specification, that prevent the application of certain refactorings [34]. Second, UML class diagrams are used to describe and create classes for a software application. By refactoring a UML class diagram we may want to refactor the created classes accordingly [37]. In [37], the authors focus on the interaction of UML with documents of object-oriented programming languages and the application of object-oriented refactorings. Refactorings of different documents or artifact types are not considered or discussed.

Some authors analyze and implement renaming for different artifact types [4, 19, 35]. We analyzed and implemented refactorings beyond renaming.

Coupled Software Transformations or *Co-transformations* are modifications of different interacting document types [22, 5]. Co-transformations describe semantic-preserving as well as semantic-changing modifications [22]. Based on our findings we argue that a general application of semantic-changing modifications is irreconcilable with the term refactoring. But co-transformations exists for semantic-preserving database schema transformations and the associated program transformations [6, p. 231 ff.]. These co-transformations are driven by database schema transformations [5][6, p. 237]. Database schema transformations driven by application transformations as shown in this paper are not discussed. Moreover, not all possible semantic-preserving transformations are considered [6, p. 242]. Therefore, problems as presented in this paper are not discussed or even discovered.

6 Summary

We applied several object-oriented, database, and functional refactorings on an example application implemented by means of different general-purpose and domain-specific programming languages. When we applied the refactorings, we observed the following:

1. A refactoring of one artifact can lead to semantic-changing modifications in other artifacts.
2. Tool-specific documents must be considered, whose structure cannot be generalized.
3. There can be alternative approaches to realize a refactoring on different document types. These modification can differ substantially in the amount of modifications or differ in whether they preserve program-semantics or not.

Hence, we argue that a general approach to automatic multi-language refactorings (MLR) covering all possible multi-language software applications cannot exist.

We automated the Rename Method and the Push Down Method refactoring for programs written in Java, Hibernate, and SQL to some extend. We evaluated the implementation on different case studies. The implemented refactorings do not realize a general approach to MLR but cover documents of a number of general-purpose and special-purpose programming languages.

In our case studies, we have also shown that an MLR of one software application is not semantic-preserving on another software application.

7 Future Work

So we argue that there is no general approach to MLR, we and others show that certain refactorings perform an MLR [4, 19, 35]. The next step is to find more combinations of refactorings performing an MLR and to specify the conditions under which the successful application of identified MLR is possible. Then, commonalities between specifications of different MLRs must be identified. These commonalities may help to decrease the effort for implementing MLR further.

Acknowledgments

The authors like to thank Gunter Saake for his comments on earlier drafts of this paper.

References

1. Ambler, S.: Agile Database Techniques: Effective Strategies for the Agile Software Developer. John Wiley & Sons, Inc., New York, NY, USA (2003)
2. Andersen, L.: JDBC $^{\text{TM}}$ 4.0 Specification. Sun Microsystems, Inc., Santa Clara, USA, final edn. (2006)
3. Bergsten, H.: JavaServer Faces. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2004)
4. Chen, N., Johnson, R.: Toward Refactoring in a Polyglot World: Extending Automated Refactoring Support across Java and XML. Workshop on Refactoring Tools pp. 1–4 (2008)
5. Cleve, A., Henrard, J., Hainaut, J.: Co-transformations in Information System Reengineering. Electronic Notes in Theoretical Computer Science 137(3), 5–15 (2005)
6. Cleve, A.: Program Analysis and Transformation for Data-Intensive System Evolution. Ph.D. thesis, University of Namur (2009)
7. Ducasse, S., Lanza, M., Tichelaar, S.: MOOSE: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. International Symposium on Constructing Software Engineering Tools pp. 24–30 (2000)
8. Fjeldberg, H.C.: Polyglot Programming. Master thesis, Norwegian University of Science and Technology, Trondheim/Norway (2008)
9. Ford, N.: The Productive Programmer. O'Reilly (2008)
10. Fowler, M.: Refactoring: Improving the Design of existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Abstraction and reuse of object-oriented design. Springer, London (1993)
12. Grechanik, M., Batory, D., Perry, D.: Design of Large-Scale Polylingual Systems. International Conference on Software Engineering, Edinburgh, Scotland, UK (2004)
13. Grogan, M.: JSR-223 Scripting for the Java $^{\text{TM}}$ Platform. Sun Microsystems, Inc., Santa Clara, USA, final edn. (2006)
14. Hainaut, J.L.: Specification Preservation in Schema Transformations – Application to Semantics and Statistics. Data & Knowledge Engineering 19, 99–134 (1996)

15. Harold, E.R., Means, W.S.: XML in a nutshell. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2002)
16. ISO/IEC: International Standard ISO/IEC 9075-1 Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework). ISO/IEC, third edn. (2008)
17. Jones, T.C.: Estimating software costs. McGraw-Hill, Inc., Hightstown, NJ, USA (1998)
18. Keith, M., Schincariol, M.: Pro EJB 3: Java Persistence API (Pro). Apress, Berkely, CA, USA (2006)
19. Kempf, M., Kleeb, R., Klenk, M., Sommerlad, P.: Cross Language Refactoring for Eclipse plug-ins. Companion to the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications pp. 1–4 (2008)
20. Kullbach, B., Winter, A., Dahm, P., Ebert, J.: Program Comprehension in Multi-Language Systems. Working Conference on Reverse Engineering pp. 135–143 (1998)
21. Lämmel, R.: Towards Generic Refactoring. ACM SIGPLAN Workshop on Rule-based Programming pp. 15–28 (Oct 2002)
22. Lämmel, R.: Coupled Software Transformations. Workshop on Software Evolution Transformations pp. 31–35 (2004)
23. Leyderman, R.: Oracle ® C ++ Call Interface. Oracle Corporation (2005)
24. Li, H.: Refactoring Haskell Programs. Ph.D. thesis, University of Kent, Canterbury, Kent, UK (2006)
25. Liang, S.: The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
26. Linos, P.K., Lucas, W., Myers, S., Maier, E.: A Metrics Tool for Multi-Language Software. International Conference on Software Engineering and Applications pp. 324–329 (2007)
27. López, C., Marticorena, R., Crespo, Y., Pérez, F.: Towards a Language Independent Refactoring Framework. International Conference on Software and Data Technologies pp. 165–170 (2006)
28. Marticorena, R.: Analysis and Definition of a Language Independent Refactoring Catalog. Conference on Advanced Information Systems Engineering. Doctoral Consortium pp. 8–16 (2005)
29. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Transactions on software engineering 30(2), 126–139 (2004)
30. Michels, J.E., Kulkarni, K., Farrar, M.C., Eisenberg, A., Mattos, N., Darwen, H.: The SQL Standard. it – Information Technology 45(1), 30–38 (2003)
31. Opdyke, W.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
32. Schrijvers, T., Serebrenik, A., Demoen, B.: Refactoring Prolog Code. Workshop on (Constraint) Logic Programming pp. 115–126 (2004)
33. Strein, D., Kratz, H., Lowe, W.: Cross-Language Program Analysis and Refactoring. IEEE International Workshop on Source Code Analysis and Manipulation pp. 207–216 (Sep 2006)
34. Sunyé, G., Pollet, D., Traon, Y.L., Jézéquel, J.: Refactoring UML Models. UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools pp. 134–148 (2001)
35. Tatlock, Z., Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: Refactoring UML Models. Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications pp. 37–52 (Oct 2008)

36. Tichelaar, S.: Modeling Object-Oriented Software for Reverse Engineering and Refactoring. Ph.D. thesis, University of Berne, Switzerland (2001)
37. Van Gorp, P., Stenten, H., Mens, T., Demeyer, S.: Towards Automating Source-Consistent UML Refactorings. UML 2003 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools pp. 144–158 (2003)
38. VanderHart, L.: Practical Clojure. Apress (2010)