Mastering Dependencies in Multi-Language Software Applications

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von Dipl.-Inform. Hagen Schink

geb. am 23.11.1984 in Berlin

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Dr. Ralf Lämmel
Prof. Dr.-Ing. Norbert Siegmund

Magdeburg, den 07.03.2018

University of Magdeburg

School of Computer Science

Dissertation

Mastering Dependencies in Multi-Language Software Applications

Author:

Hagen Schink

March 7, 2018

Reviewers:

Prof. Dr. Gunter Saake (University of Magdeburg, Germany)
Prof. Dr. Ralf Lämmel (University of Koblenz - Landau, Germany)
Prof. Dr.-Ing. Norbert Siegmund (Bauhaus-Universität Weimar, Germany)

Schink, Hagen:
Mastering Dependencies in Multi-Language Software Applications
Dissertation, University of Magdeburg, 2018.

Abstract

In software development, developers use *source-code restructurings* or *refactorings* to preserve or improve a source-code's maintainability, that is, the source-code's capability to be adopted to new requirements. For that purpose, a refactoring modifies the source-code's structure without modifying the behavior implemented by the source-code. Practitioners and researchers formulated a number of refactorings for different programming languages and programming paradigms.

Usually, software developers use more than a single programming language to implement a software application. For instance, software developers embed SQL statements in the source-code of a general-purpose language to create, read, modify, and delete datasets in a database. The practice of using more than one programming language is called *polyglot programming*. The result of polyglot programming is a *Multi-Language Software Application (MLSA)* in which source-code of many languages interact.

Though polyglot programming is common in software development, refactorings are defined for source-code of a single programming language or programming paradigm. However, by restructuring source code of a single language the interaction in an MLSA can break. We give examples how refactoring can break a database application comprising the object-oriented, the functional, and the relational paradigm. We also discuss why a general approach to automated refactorings for MLSAs, so called *multi-language refactorings*, are hard to realize.

Our goal is to provide sufficient support for software developers who refactor an MLSA. For that purpose, we introduce an approach that takes the challenges in refactoring MLSAs into account. We present implementations of our approach for two programming-paradigm pairs: the object-oriented and relational paradigm and the object-oriented and functional paradigm. Finally, we report the results of an evaluation of one of our tools. The evaluation studied the effect of our tool on the development performance of 79 participants who are asked to fix an MLSA with broken language interaction.

Zusammenfassung

Software-Entwickler reorganisieren Quellcode mit Hilfe Semantik-erhaltender Strukturveränderungen, genannt Refactorings, um die Wartbarkeit von Quellcode zu erhalten oder zu verbessern. Wartbarkeit beschreibt dabei die Tauglichkeit von Quellcode an neue Anforderungen angepasst zu werden. Software-Entwicklern aus Wissenschaft und Wirtschaft formulieren Refactorings für verschiedene Programmiersprachen als auch Programmierparadigmen.

In der Software-Entwicklung ist es üblich, mehr als eine Programmiersprache zu verwenden. So betten Software-Entwickler SQL-Anweisungen in den in einer General-Purpose-Programmiersprache geschriebenen Quellcode ein, um Datensätze in einer Datenbank anzulegen, zu lesen, zu aktualisieren und zu löschen. Der Einsatz mehr als einer Programmiersprache wird als mehrsprachige Programmierung bezeichnet. Das Ergebnis der mehrsprachigen Programmierung ist eine sogenannte mehrsprachige Software-Anwendung. In mehrsprachigen Software-Anwendungen interagieren die in verschiedenen Programmiersprachen entwickelten Funktionen miteinander.

Obwohl mehrsprachige Programmierung in der Software-Entwicklung verbreitet ist, werden Refactorings bisher nur für einzelne Programmiersprachen beziehungsweise Programmierparadigmen beschrieben. Jedoch kann durch die Semantik-erhaltende Umstrukturierung des Quellcodes einer Programmiersprache durch Refactoring die Interaktion mit Stukturelementen, die in anderen Programmiersprachen definiert worden sind, beschädigt werden. Wir zeigen anhand einer Datenbank-Anwendung beispielhaft, wie die Interaktion von in verschiedenen Programmiersprachen definierten Strukturelementen durch Refactorings beschädigt werden kann. Die Datenbank-Anwendung selbst ist mit Hilfe der Objekt-orientierten, relationalen und funktionalen Paradigmen implementiert. Wir diskutieren ebenfalls, welche Herausforderungen einem allgemeinen Ansatz zur Automatisierung von Refactorings in mehrsprachigen Software-Anwendungen, dem mehrsprachigen Refactoring, entgegenstehen.

Ziel dieser Arbeit ist es, einen Ansatz zur Unterstützung von Software-Entwicklern bei der Umstrukturierung von Quellcode einer mehrsprachigen Software-Anwendung zu entwickeln. Dazu beschreiben wir eine Lösung, die die speziellen Anforderungen beim Refactoring von mehrsprachigen Software-Anwendungen beachtet. Wir stellen außerdem zwei prototypische Implementierungen unseres Ansatzes für die Kombination aus Objekt-orientiertem und relationalem sowie Objekt-orientiertem und funktionalem

Paradigma vor. Schließlich präsentieren wir die Ergebnisse einer Studie, die den Effekt einer unserer Prototypen auf die Produktivität von Software-Entwicklern evaluiert. Die Studie basiert auf Daten von 79 Teilnehmern. Die Aufgabe der Teilnehmer in der Studie ist es, die Interaktion zwischen den Strukturelementen verschiedener Programmiersprachen innerhalb einer mehrsprachigen Software-Anwendung wiederherzustellen.

Acknowledgements

Being an external PhD student poses challenges of its own. Being an external PhD student whose day-to-day job offers only limited connection to his research poses even more challenges. Yet Gunter Saake gave me the opportunity to follow my research interests in his working group while gaining experience in the industry. Gunter, I am very grateful to you for this opportunity.

I also like to thank Ralf Lämmel for his steady support of my research and the warm welcome when I visited his working group. Ralf, though I did not manage to get to Koblenz more often, I really enjoyed the few occasions as well as our remote discussions. Thank you!

One part of research is to evaluate the results. The evaluation presented in this thesis would not have been possible without the expertise and support of Janet Siegmund, Reimar Schröter, David Broneske, and Georg Seibt. It was a pleasure to work with you; thank you!

I would like to thank Wolfram Fenske in particular and Gunter Saake's working group in general for their feedback on my research topic and papers. When we met, I always felt like a regular member of your working group, thank you folks!

The first time I got in touch with my research topic was when Martin Kuhlemann asked me to investigate multi-language refactoring in regard to Refactoring Feature Modules. Since then, Martin supported my research efforts and cheered me up when I raised concerns. Thanks a lot for everything, Martin!

Like so many before me, I depended on numerous tools to develop prototypes, to evaluate my research, even to write this thesis. Therefore, I would like to thank the open-source community for their myriad of contributions.

Finally, I would like to thank my wife for her understanding. Thank you!

Contents

List of Figures

List of Tables

List of Code Listings

List of Acronyms

ANOVA	Analysis of Variance
API	Application Programming Interface
ASP	Active Server Pages
AST	Abstract Syntax Tree
CDIF	CASE Data Interchange Format
DLL	Dynamic-link library
DML	Data Manipulation Language
DSL	Domain Specific Language
FFI	Foreign Function Interface
GPL	General-Purpose Programming Language
HQL	Hibernate Query Language
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
JDBC	Java Database Connectivity
JNI	Java Native Interface
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSP	Java Server Pages
JVM	Java Virtual Machine
LOC	Lines of Code
MLR	Multi-Language Refactoring

MLSA	Multi-Language Software Application
ORM	Object Relational Mapping
SQL	Structured Query Language
UML	Unified Modeling Language
URL	Uniform Resource Locator
VCS	Version Control System
XLL	Cross Language Link
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1. Introduction

As business requirements evolve [Robertson and Robertson, 2012, p. 26], source code may needs to be adapted to new or changed requirements [Fowler, 1999, p. 57; Lehman, 1980]. A source code's capability to be adapted to new requirements is described by its *maintainability* [ISO/IEC/IEEE 24765:2010, E, p. 204]. An approach to preserve the maintainability of source code is to restructure the source code [Griswold and Notkin, 1993]. Today, we usually refer to program restructuring as *refactoring*, that is a modification to the structure of source code which preserves the behavior implemented in the source code. Refactorings exist for all programming paradigms relevant in practice such as object-oriented programming-languages [Fowler, 1999; Opdyke, 1992], functional programming-languages [Li and Thompson, 2008], and relational schemata [Ambler, 2003].

Today, it is common in software development for developers to use more than a single programming language to implement a software application [Chen and Johnson, 2008; Ford, 2008; Grechanik et al., 2004; Jones, 1998; Kullbach et al., 1998; Linos et al., 2006; Mayer and Bauer, 2015; Mayer and Schroeder, 2012; Strein et al., 2006; Tomassetti and Torchiano, 2014; Vetro et al., 2012]. One reason for developers to use more than a single programming language may be the wish of developers to use features of a modern programming languages while a complete migration of legacy code is not practical [Delorey et al., 2007]. Another reason may be that by using a Domain Specific Language (DSL) a developer is able to describe problems of a certain domain in an efficient way. For instance, we use SQL to query relational databases [Ford, 2008, p. 169]. Whatever the reason for using more than a single programming language is, we refer to an application implemented with the help of more than one programming language as Multi-Language Software Application (MLSA) as introduced in [Linos et al., 2006].

In an MLSA, developers let source code of different programming languages interact to accomplish or simplify certain tasks. For instance, developers use SQL queries in a General-Purpose Programming Language (GPL) such as Java to process data stored in a relational database or developers use interfaces to C and C++ to access platform-specific functions. However, a compiler for one language does not check the interaction with source-code written in another language, if the interaction is implemented based on Application Programming Interfaces (APIs). For instance, the Java compiler does not check SQL statements embedded in the Java source code. Likewise, the Java compiler does not check the interaction with source-code written in C and C++. Thus, structural modifications to source code of either of two interacting languages can break the interaction between the languages. For instance, a simple typo in an SQL statement used in a Java application can break the entire application. If the developer misses the typo, she will not notice the typo before running the affected part of the application. In the best case, automated tests detect the broken language interaction by failing. In the worst case, the customer encounters the error in the production environment.

In this thesis, we describe challenges in the refactoring of MLSAs and provide a first attempt to explain why refactoring of MLSAs has to be considered *hard* [Chen and Johnson, 2008]. We present an approach based on tree structures that allows developers of refactoring tools to provide means to support refactoring in MLSAs taking into account the challenges of refactoring MLSAs. We describe a prototypical implementation of our approach for the following combinations of language paradigms: object-oriented and relational paradigm as well as object-oriented and functional paradigm. We also report on a study in which we evaluated the effect of one of our prototypical implementations on the development performance of 79 participants.

1.1 Contribution

In the course of this thesis, we make the following contributions.

1. We describe challenges in refactoring database applications. Additionally, we describe a generalized model of the structure of MLSAs. Based on this model, we formulate general challenges of refactoring in MLSAs.

2. We describe, implement, and evaluate an approach that supports refactoring in MLSAs. The approach is based on tree structures and considers the challenges we describe in our first contribution.

3. We present an experimental design for the evaluation of refactoring tools for MLSAs.

1.2 Outline

The thesis is structured as follows. In Chapter 2, we introduce basic concepts and terms we use in this thesis. In Chapter 3, we describe application-specific and general challenges of refactoring MLSAs. In Chapter 4, we present an approach based on tree structures for supporting refactoring in MLSAs. In Chapter 5, we describe prototypes implementing our approach for two language combinations and, in Chapter 6, we report on an evaluation of one of the prototypes. We present related work in Chapter 7 before we conclude the thesis and give an outlook on future work in Chapter 8.

1.3 Referenced Publications

In this thesis, we share material with the following publications:

Chapter 3 [Schink and Kuhlemann, 2010; Schink et al., 2011]

Chapter 4 [Schink et al., 2016a]

Chapter 5 [Schink, 2013; Schink et al., 2016a]

Chapter 6 [Schink et al., 2016b]

Chapter 3 shares the description of applying nine refactorings on the prototypical application *HRManager* with [Schink and Kuhlemann, 2010] and [Schink et al., 2011]. In Chapter 4, we describe an approach to check the referential integrity between source code of different languages which was published in [Schink et al., 2016a]. The description of the *structure-graph* library, the *sql-schema-comparer*, and the *clojure-java-interface-checker* in Chapter 5 shares material with [Schink, 2013] (only sql-schema-comparer) and [Schink et al., 2016a]. We published parts of Chapter 6 in [Schink et al., 2016b].

2. Background

This chapter covers the concepts we generally depend on throughout this thesis. In Section 2.1, we introduce the term *Refactoring* and explain the different understandings as well as the usage of this term in this thesis. It follows an introduction to the concept of a Multi-Language Software Application (MLSA), that is, an approach to utilize the abilities of more than one programming language to implement a software application in Section 2.2. Then, we introduce the concept of Multi-Language Refactoring (MLR) which relates the discussion about refactoring to MLSAs. At the end, we recapitulate concepts of graph theory that are necessary to understand our approach to support refactoring in MLSAs and summarize this chapter.

2.1 Refactoring

In software development, it is necessary to adapt the functionality of an application to new or changed requirements. However, an application's existing source-code structure may not allow developers to readily extend or change the existing functionality. To preserve the maintainability of the application's source code and to improve the extensibility of its functionality, developers may need to restructure the application's source code [Opdyke, 1992, p. 1]. However, the larger the application grows, the harder it is for developers to ensure the correctness of the source-code after restructuring [Opdyke, 1992, p. 8].

The term *Refactoring* was coined by Opdyke who describes refactoring as methodical approaches for altering the source-code structure of object-oriented software without changing the source-code's behavior. For Opdyke, the source-code's behavior is defined by the output for a given input [Opdyke, 1992, p. 2]. Martin Fowler, who introduced the refactoring term to a broader audience, mentions a more general term of behavior preservation which he calls *observable-behavior* or *semantics* [Fowler, 1999, p. 46 and p. 61]. By observable behavior or semantics, Fowler refers to any kind of behavior, not only the output for a given input as Opdyke does. The term semantics-preservation was also adopted by the scientific community [Balaban et al., 2005; Counsell et al., 2006; Daniel et al., 2007; Lämmel, 2002; Li, 2006; Mayer and Schroeder, 2012; Mens, 2005; Soares et al., 2009; Streckenbach and Snelting, 2004; Strein et al., 2006]. We follow the definition of the term Refactoring according to [Fowler, 1999, p. 46]:

Definition 2.1. *Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*

With the introduction of the term observable behavior or semantics, we also have to consider a different understanding on the effects of refactoring. The term *behavior* is based on the equivalence of the output for a given input. Thus, the term behavior covers only an application's functional aspects. In contrast, the term *semantics* could also include non-functional aspects of an application. For instance, execution time and memory consumption are important aspects for real-time and embedded systems. However, refactorings can affect these properties [Mens and Tourwé, 2004]. That is, we may not call a source-code transformation a refactoring in all application domains depending on whether the refactoring affects properties or not that are important to the application's applicability to a given domain.

A term that is directly related to that of refactoring is *software restructuring*. Software restructuring "is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)" [Chikofsky and II, 1990]. Opdyke emphasizes that while software restructuring focuses on understanding and infusing structure into unstructured code, the purpose of refactoring of object-oriented code is "to refine the design of an already structured program, and make it easier to reuse" [Opdyke, 1992, p. 10]. However, in following works this difference has not been preserved and refactoring became an object-oriented variety of software restructuring [Mens and Tourwé, 2004]. Consequently, we consider the term refactoring and software restructuring to be interchangeable.

2.2 Multi-Language Software Application

Section 2.2 shares material with [Schink and Kuhlemann, 2010; Schink et al., 2011, 2016a].

A software application is an MLSA[1] when it is implemented with the help of different General-Purpose Programming Languages (GPLs) or Domain Specific Languages (DSLs) [Linos et al., 2006]. The usage of different GPLs or DSLs is referred to as *polyglot programming* [Fjeldberg, 2008; Ford, 2008, p. 169].

[1]The term *Multi-Language Software System* is used synonymously [Pfeiffer and Wąsowski, 2012a,b].

Polyglot programming is common in modern software development [Ford, 2008]. However, which GPL or DSL are used for software development differs between software applications. In the following, we introduce common use cases for polyglot programming. A detailed analysis of use cases for polyglot programming in respect to open-source projects can be found in [Mayer and Bauer, 2015].

- Structured Query Language (SQL) is a standardized query language for databases and, therefore, was not intended itself as a GPL [Michels et al., 2003]. It is possible to reference SQL statements in GPLs like C++ or Java [Anderson, Lance, 2006; ISO/IEC 9075-1:2008].
- Extensible Markup Language (XML) is used in different application areas mainly for data exchange purposes [Harold and Means, 2002]. XML is also used for describing configuration files or structured data that can be referenced in GPLs like C++ or Java [Chen and Johnson, 2008; Harold and Means, 2002] in order to access data in databases.
- C++ and different scripting languages, e.g. JavaScript, can be called from or embedded in Java [Grogan, 2006; Liang, 1999]. The interfaces to Java are described by the *Java Native Interface* for C++, and the *Java Specification Request* 223 for scripting languages [Grogan, 2006; Liang, 1999].

Using polyglot programming makes common tasks in software development easier, e.g. database access and data exchange [Fjeldberg, 2008, p. 9-10].

In general, a software application contains source-code written in one programming language that initializes that application. In an MLSA, we call the programming language in which the application's initialization code is written the application's *host language*. From the code of the host language, developers invoke source-code implemented in different languages. We call these languages *guest languages*.

2.2.1 Types of Language Interaction

How host and guest language interact depends on the relation between those languages. We distinguish three kinds of relations:

1. No relation
2. Host and guest language share the same platform
3. The guest language is implemented in the host language

If the host and the guest language interact via an Application Programming Interface (API), then we consider the host and the guest language to have no relation (1st kind). Examples for languages with no relation are Java and SQL via Java Database Connectivity (JDBC) and Java Persistence API (JPA) or Java and C++ via Java Native

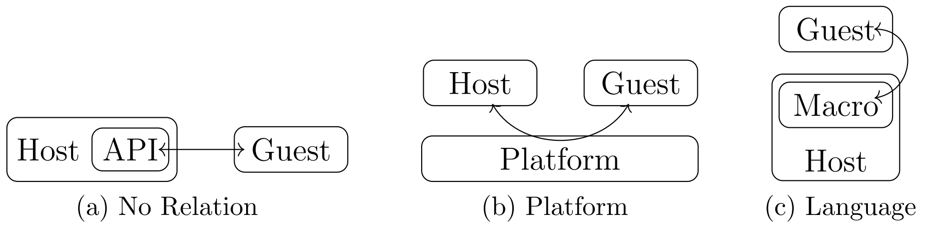


Figure 2.1: Relations between host and guest language.

Interface (JNI) where in both cases Java is the host language which interacts via APIs with the guest languages. In the Java programming language, JDBC and JPA allow to query relational databases via SQL and JNI allows to invoke C/C++ functions. Platforms such as Java and .Net represent the second kind. For instance, the programming languages Java, Clojure, and Scala can all be compiled to Java bytecode [EPFL, 2017; Hickey, 2017] and all languages can invoke Java bytecode [EPFL, 2015; Hickey, 2017]. Other examples are the languages F# and Ceylon which are able to interact with other languages running on the .Net runtime or Java Virtual Machine (JVM), respectively [Sixto Ortiz Jr., 2012]. By compiling the source code of guest languages to the intermediate language of the platform, source-code implemented in the host language can interact with source-code implemented in the guest language and vice versa. The SugarJ language is another example of platform based language interaction[Erdweg, 2012]. In SugarJ, the programming language Java, the grammar formalism SDF [Heering et al., 1989], and the transformation system Stratego [Visser, 2001] provide the common platform on which the guest languages interact with each other and the host language Java. The third kind describes programming languages which provide means to define internal DSLs such as Lisp, Ruby, and Smalltalk [Renggli, 2010]. For instance, developers can use Lisp's macro system and Ruby's support for multiple paradigms to define new language elements or DSLs [Günther, 2010; Sheard, 2001]. The new language elements and DSLs part of the host language and, thus, cannot exist on their own.

In Figure 2.1, we summarize possible relations of a host and guest language. In the figure we emphasize that language interaction depends on an API, a common platform, or facilities of the host language to describe internal DSLs such as a macro system. Without these prerequisites no language interaction takes place.

In [Renggli, 2010, p. 11 ff.], the author distinguishes three approaches for defining DSLs: *internal languages*, *external languages*, and *embedded languages*. In contrast, our definition distinguishes approaches of host and guest language interaction. However, in the following we show how our definition relates to the definition in [Renggli, 2010]. Internal languages are DSLs which are defined by means of a host language. This definition matches our definition of guest languages implemented in a host language (third kind). External languages are DSLs such as SQL which are self-contained. Thus, external languages have no relation to a host language (first kind). [Renggli, 2010, p. 22 ff.] describes five categories of language embedding: extensible compilers, meta-

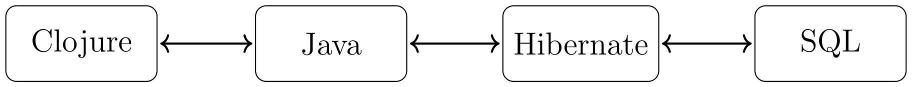


Figure 2.2: The different document types in HRManager and their relation.

programming systems, language workbenches, language transformations, and modeling languages. These categories describe a common basis for implementing DSLs. Thus, we can combine embedded languages by the means of the platform on which the languages are defined (second kind).

In this work, we solely focus on language interaction with no relation because for this kind of interaction no tool support exists to check language interaction [Renggli, 2010, p. 21]. For instance, in case the guest language is implemented in the host language, the guest language's source-code is actually valid source code of the host language. Thus, developers can reuse existing tools for static code analysis of the host language to check the interaction between source code of the guest and the host language. Likewise, in case the guest language's source code can be compiled to the host language's platform, developers can reference the compiled code of the guest language in the host language and the host language's compiler can check the interaction between source code of the guest and the host language. For instance, F# source code can be compiled to a managed Dynamic-link library (DLL) which a developer can reference in C# source code. No such support exists for language interactions with no relation.

In the next section, we introduce an example MLSA that implements language interaction based on APIs.

2.2.2 An Example MLSA

The *HRManager* is a prototypical software application implemented by the author to manage employee data. With the help of HRManager, we show effects of refactoring on MLSAs. The software application has been implemented using the object-oriented, the functional, and the relational paradigm. In particular, we used the programming languages *Java*[2] and *Clojure*[3], the object-relational mapper *Hibernate*[4], and the relational database *SQLite*[5]. Figure 2.2 shows the document types used in HRManager and how respective documents interact. We use HRManager as our running example. For that purpose we will present the different kinds of documents and their relations in detail.

Java is used to declare classes and class hierarchies with which we model the objects of the business domain. For instance, we defined the class hierarchy of class `Employee` including `Manager` and `Salesperson` in Java to model different kinds of employees. Figure 2.3 shows the class hierarchy of `Employee` with the subclasses `Manager` and `Salesperson`.

[2]http://www.oracle.com/technetwork/java/index.html
[3]http://clojure.org
[4]http://www.hibernate.org
[5]https://www.sqlite.org/

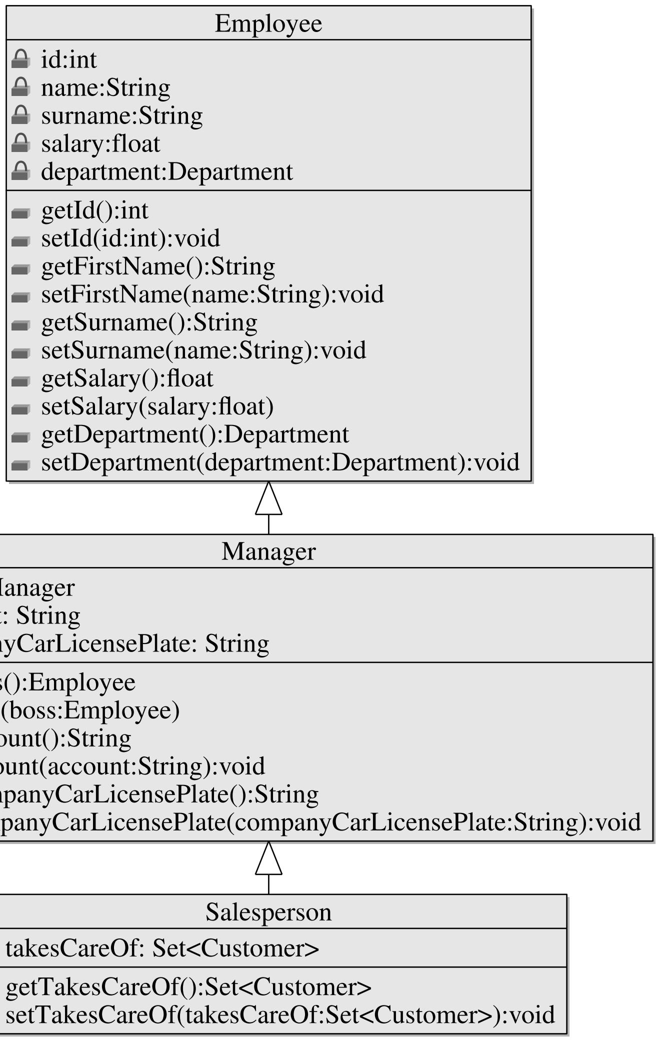

Employee
id:int
name:String
surname:String
salary:float
department:Department
getId():int
setId(id:int):void
getFirstName():String
setFirstName(name:String):void
getSurname():String
setSurname(name:String):void
getSalary():float
setSalary(salary:float)
getDepartment():Department
setDepartment(department:Department):void
Manager
boss: Manager
account: String
companyCarLicensePlate: String
getBoss():Employee
setBoss(boss:Employee)
getAccount():String
setAccount(account:String):void
getCompanyCarLicensePlate():String
setCompanyCarLicensePlate(companyCarLicensePlate:String):void
Salesperson
takesCareOf: Set<Customer>
getTakesCareOf():Set<Customer>
setTakesCareOf(takesCareOf:Set<Customer>):void

Figure 2.3: Class hierarchy of superclass `Employee`.

Each class in HRManager that represents a business object has a counterpart in the relational database schema. For instance, the database schema defines the tables `employees`, `managers` as well as `salespersons` and a column for every field in the respective classes. We have two options to map the class hierarchy to the database schema: in one table or multiple tables. In HRManager, 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. Figure 2.3).

The object-relational mapper Hibernate maps classes and class attributes 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 have to define which classes are part of the ORM and how these classes

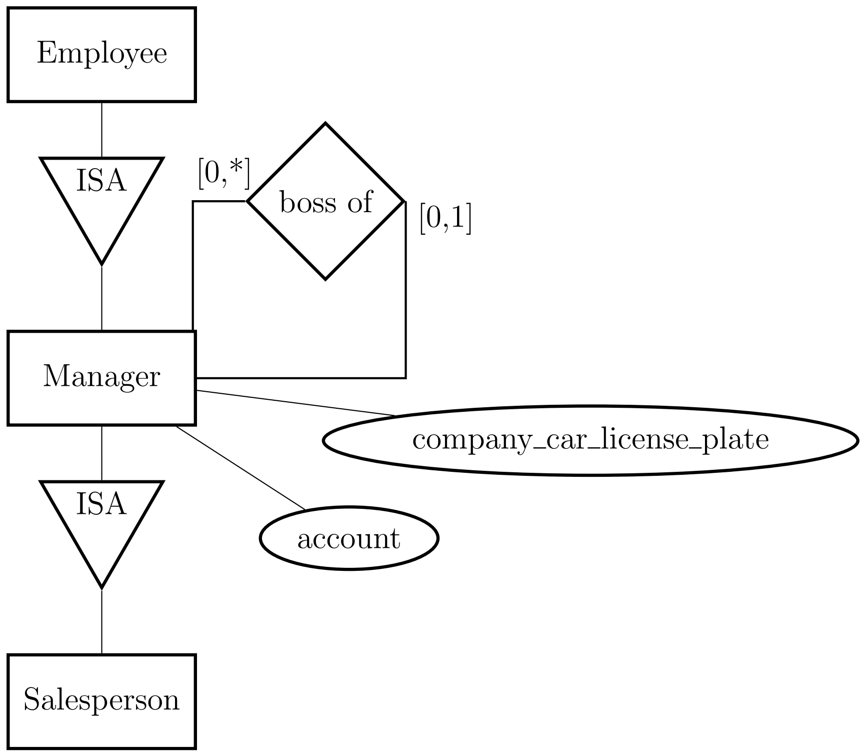


Figure 2.4: Database schema of class hierarchy of superclass `Employee`.

map onto the relational schema in the database. We use the Hibernate configuration file `hibernate.cfg.xml` to define which classes are part of the ORM [Red Hat Inc. and the various authors, 2004]. Listing 2.2 shows an excerpt of the Hibernate configuration file we use in HRManager. In Line 9, we define the class `Department` as part of the ORM. For the definition of the actual mapping, we use Java annotations.[6] Listing 2.1 shows an excerpt of the ORM of class `Employee`. HRManager uses the `@Entity` annotation (Line 1) to let class `Employee` become part of the ORM. 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).

Similar to the way in which classes are mapped to tables, Hibernate also maps class attributes to the respective table columns. By default, Hibernate uses the setter and getter methods to map class attributes to database columns. For instance, Hibernate maps the getter and setter methods `getName` and `setName` of class `Employee` onto the column `name` of table `employees` (cf. Listing 2.1) [Keith and Schincariol, 2006, p. 73]. With the `@Column` annotation we can override the default behavior. Listing 2.3 shows how we map the getter and setter methods of `companyCarLicensePlate` onto the column `company_car_license_plate`.

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

```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 2.1: Excerpt of the ORM of the class `Employee`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="dialect">dialect.SQLiteDialect</property>
        <!-- more properties-->
        <mapping class="hrm.Department"/>
        <!-- more mappings -->
    </session-factory>
</hibernate-configuration>
```

Listing 2.2: Excerpt of the `hibernate.cfg.xml`.

```java
@Column(name = "company_car_license_plate")
public String getCompanyCarLicensePlate() {
    return companyCarLicensePlate;
}
```

Listing 2.3: Application of the `@Column` annotation.

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

Listing 2.4: Referencing the Clojure function `sumSalary` from the Java source code.

Besides Java, the ORM, and the relational database, the HRManager contains source code of the functional programming language Clojure. We use Clojure to add scripting facilities to HRManager, that is, to allow the adaption of program logic without the need to recompile the entire application. More specifically, we use Clojure to compute the overall salary of employees and to find employees with certain attributes. 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` [VanderHart, 2010, p. 149-150].[7] Listing 2.4 shows in Line 1 how the Clojure function `sumSalary` defined in the namespace `scripting` is referenced from Java code. In Line 2, we invoke the function `sumSalary` on a list of managers. The function returns the sum of type float as result of the function invocation.

2.3 Multi-Language Refactoring

We call a refactoring that considers language interaction in an MLSA an MLR[8] [Chen and Johnson, 2008; Mayer and Schroeder, 2012, 2014; Pfeiffer and Wąsowski, 2012b]. In particular, a refactoring aware of language interaction not only considers the source-code written in one language, but also considers interacting source-code implemented in other languages. That is, if the refactoring of the source-code written in one language breaks the language interaction, an MLR will also refactor the interacting source-code implemented in other languages. For instance, in a Java application that is configured with the help of an XML configuration file, a multi-language Class Rename refactoring applied on a Java class referenced in the configuration file will also apply a rename refactoring on the class references in the configuration file [Chen and Johnson, 2008].

In general, we describe an MLR as a set of single-language refactorings for all the languages that may be affected in an MLSA. So, in regard to the previous example, the MLR is a set of Class Rename refactorings for Java class identifiers in Java source code and in XML configuration files. In Figure 2.5, we illustrate the general idea of MLR with respect to an MLSA which consists of n interacting languages. An MLR applies a single-language refactoring to the source-code implemented in language x as well as to all other source-code artifacts of the interacting languages.

We do not set any restrictions on the set of single-language refactorings represented by an MLR. Thus, an MLR either contains only one type of single-language refactorings (like Rename refactorings) or different types of single-language refactorings (like Pull-Up Method and Extract Method refactorings). We call MLRs only containing only one type of single-language refactoring *homogeneous*. Accordingly, we call MLRs containing different types of single-language refactorings *inhomogeneous*. Homogeneous MLRs have been described in respect to the Rename refactoring [Chen and Johnson, 2008; Mayer and Schroeder, 2012, 2014; Pfeiffer and Wąsowski, 2012b; Strein et al., 2006;

[7]Since version 1.6, the preferred way of calling a Clojure function is to use class `Clojure` that returns an instance of class `IFn`. Nevertheless, the basic principle has not changed.

[8]The terms *Cross-Language Refactoring* [Pfeiffer and Wąsowski, 2012a; Strein et al., 2006] and *Deep Refactoring* [Tatlock et al., 2008] are used synonymously with MLR.

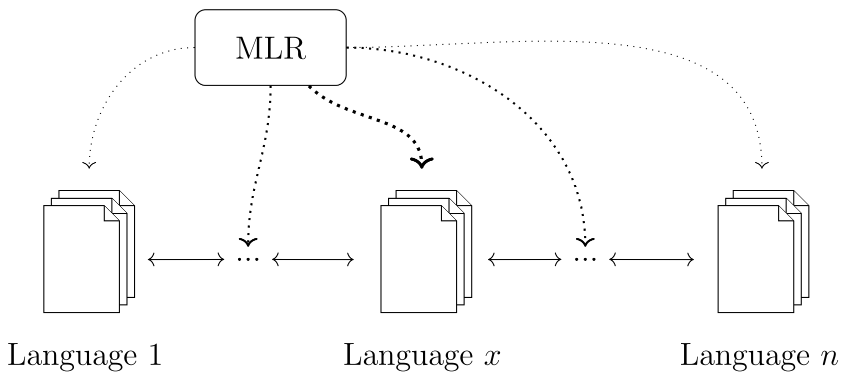


Figure 2.5: Illustration of MLR.

Tatlock et al., 2008]. In a database application, an MLR may be inhomogeneous [Schink et al., 2011].

2.4 Foundations of Graph-Theory

In the course of this work, we will present an approach to support developers who refactor MLSAs. The approach is based on graph and tree structures. In this section, we recapitulate the fundamental terms of graph theory as well as the specific definitions that we use in order to define our approach. The definitions are taken from [Valiente, 2002].

A *graph* is a mathematical structure consisting of vertices and edges. We distinguish *directed* and *undirected* graphs:

Definition 2.2. *A (*directed*) graph $G = (V, E)$ consists of a finite nonempty set V of vertices and a finite set $E \subseteq V \times V$ of edges. An edge $e = (v, w)$ with $v, w \in V$ is incident with vertices v and w, where v is the source and w is the target of edge e.*

In a *directed graph* each edge $e = (v, w)$ defines a direction for the connection from the source v to the target w. An *undirected graph* gives no direction on the edges and is defined as follows:

Definition 2.3. *A graph $G = (V, E)$ is undirected if $(v, w) \in E$ implies $(w, v) \in E$ for all $v, w \in V$.*

Thus, in an undirected graph for each edge (w, v) and edge (v, w), that is an edge with the opposite direction exists. In Figure 2.6, we illustrate a directed graph with the vertices $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$ and the edges $E = \{(v_1, v_2), (v_3, v_1), (v_1, v_4), (v_3, v_5), (v_3, v_6), (v_7, v_4), (v_4, v_8)\}$.

A *subgraph* of graph G is a graph whose vertices and edges are contained in the set of the vertices and edges of G. The term subgraph is defined as follows:

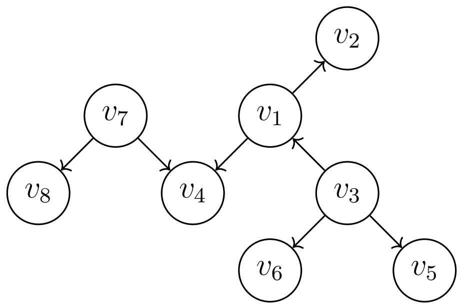


Figure 2.6: A graph with 8 vertices (v_1 - v_8) and 7 edges.

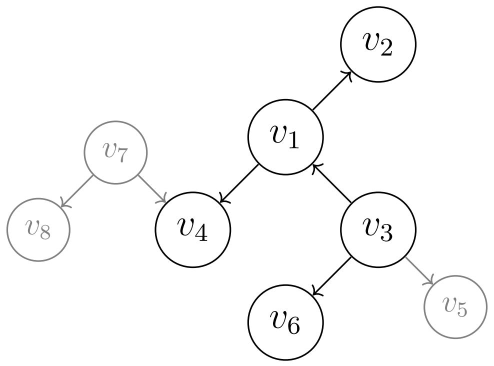


Figure 2.7: $\{v_1, v_2, v_3, v_4, v_6\}$ and their connection edges forming a subgraph of the graph in Figure 2.6.

Definition 2.4. *Let $G = (V, E)$ be a graph, and let $W \subseteq V$. A graph (W, S) is a subgraph of G if $S \subseteq E$.*

In Figure 2.7, we illustrate a subgraph of the graph shown in Figure 2.6 with the vertices $V = \{v_1, v_2, v_3, v_4, v_6\}$ and the edges $E = \{(v_1, v_2), (v_3, v_1), (v_1, v_4), (v_3, v_6)\}$.

A *labeled graph* is a graph whose vertices and edges have additional attributes. The content of the labels depends on the application of the graph. For instance, we can define a graph representing distances between cities (see Figure 2.8). For that purpose, the vertices are labeled with city names and the edges are labeled with the distance between the cities represented by the two connected vertices. We denote the label of node n with $label[n]$.

Our approach to support refactoring in MLSAs is based on *trees*, which are graphs with specific properties. However, before we can define the structure of a tree, we have to introduce the definitions of a *walk* and a *connected* undirected graph:

Definition 2.5. *A walk from vertex v_i to vertex v_j in a graph is an alternating sequence $[v_i, e_{i+1}, v_{i+1}, e_{i+2}, \ldots, v_{j-1}, e_j, v_j]$ of vertices and edges in the graph, such that $e_k = (v_{k-1}, v_k)$ for $k = i+1, \ldots, j$.*

Definition 2.6. *An undirected graph $G = (V, E)$ is connected if for every pair of vertices $v, w \in E$, there is a walk between v and w.*

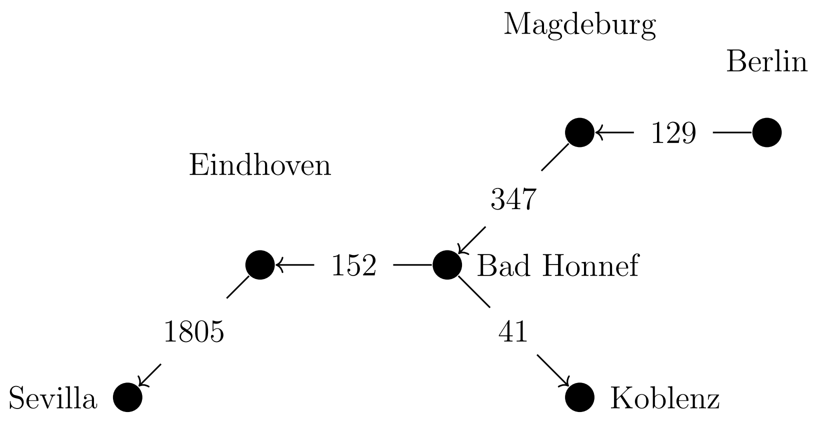


Figure 2.8: A labeled graph with distance information.

Taking these definitions into consideration, we define the structure of a tree as follows:

Definition 2.7. *A connected graph $G = (V, E)$ is said to be a tree T if the underlying undirected graph has no cycles and there is a distinguished node $r \in V$, called the root of the tree and denoted by root$[T]$ such that for all nodes $v \in V$, there is a path in G from the root r to node v.*

In Figure 2.9, we illustrate a tree that contains the same number of vertices, but only a subset of the edges of our initial graph (see Figure 2.6).[9] A tree has a hierarchical structure that allows us to distinguish *parent* and *child* nodes. Parent and child nodes are defined as follows:

Definition 2.8. *Let $T = (V, E)$ be a tree. Node $v \in V$ is said to be the parent of node $w \in V$, denoted by parent$[w]$, if $(v, w) \in E$ and, in such a case, node w is said to be a child of node v.*

For instance, in Figure 2.9, the node v_3 is the parent of nodes v_5 and v_6. Respectively, nodes v_5 and v_6 are the children of node v_3.

The *height* of a node determines how many nodes are between an initial node and another node rooted in the initial node. Accordingly, the height of a tree is the maximum number of nodes between all nodes and the root node of that tree. We will use the height to determine the costs of the algorithms we use in our tree-based approach. We define height as follows:

Definition 2.9. *Let $T = (V, E)$ be a tree. The height of node $v \in V$, denoted by height$[v]$, is the length of a longest path from node v to any node in the subtree of T*

[9]Please note that the underlying undirected graph of the tree in Figure 2.9 matches the undirected graph in Figure 2.6.

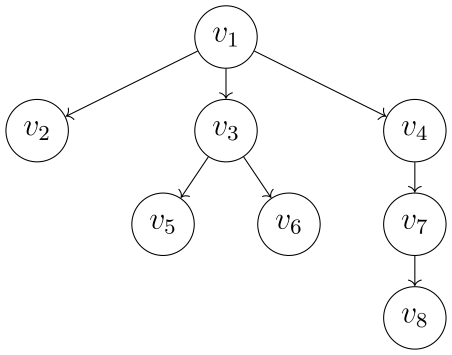


Figure 2.9: A tree based on graph Figure 2.6.

rooted at node v, for all nodes $v \in V$. The height of T is the maximum among the heights of all nodes $v \in V$.

For instance, the tree shown in Figure 2.9 has a height of 3, the node v_7 has a height of 2 in respect to node v_1.

For our tree-based approach, we use the concept of a *subtree*, in particular, a *top-down subtree*. We define a subtree as follows:

Definition 2.10. *Let $T = (V, E)$ be an unordered tree, and $W \subseteq V$. An unordered tree (W, S) is a subtree of T if $S \subseteq E$.*

In Figure 2.10, we give an example for a subtree of the tree given in Figure 2.9 which contains the nodes $W = \{v_3, v_5, v_6\}$. We call a subtree a top-down subtree if the following condition holds additionally to that of a subtree:

Definition 2.11. *A subtree (W, S) is a top-down subtree if $parent[v] \in W$, for all nodes $v \in W$ different from the root [...].*

In Figure 2.11, we present an example for a top-down subtree of the tree given in Figure 2.9 which contains the nodes $W = \{v_1, v_4, v_7\}$. Please note that W contains the parent nodes for the non-root nodes v_7 (v_4) and v_4 (v_1) in W.

2.5 Summary

In this chapter, we introduced refactoring as a methodology for improving the internal structure of source code without affecting the source code's observable behavior. We also discussed how different understandings of source-code semantics affect the applicability of refactoring. We introduced the term MLSA describing software applications implemented in more than one GPL or DSL. For the following discussion of challenges in MLSAs, we introduced the HRManager as a running example for an MLSA. With the introduction of the term MLR, we connected the discussions on refactoring and MLSAs.

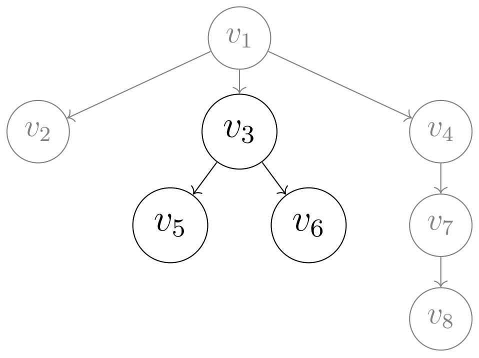


Figure 2.10: A subtree based on tree Figure 2.9.

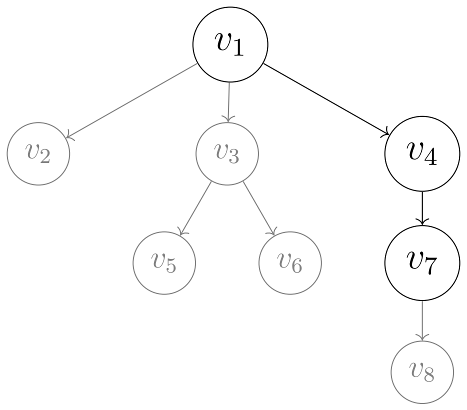


Figure 2.11: A top-down subtree based on tree Figure 2.9.

We described the current approach to refactoring in MLSAs discussed in the scientific community. Finally, we recapitulated definitions of graph theory that are necessary to follow the discussion on our approach to support developers who refactor MLSAs which we will introduce later in this work. More specifically, our approach compares single trees with a graph of trees. Each single tree represents a statement in the source-code written in the host language which contains elements of language interaction. The elements of language interaction in the single trees are expected to exist in the source-code written in the guest language. The graph of trees represents actually existing elements of language interaction in the source-code written in the guest language. With the comparison, the approach checks if the single trees are top-down subtrees of the trees in the graph of trees. If a single tree is a top-down subtree of a tree in the graph of trees, the elements of language interaction referenced in the source-code written in the host language exist in the source-code written in the guest language.

3. Issues of Refactoring Multi-Language Software Applications

Chapter 3 shares material with [Schink et al., 2011] and [Schink and Kuhlemann, 2010].

In this chapter, we illustrate how refactoring an Multi-Language Software Application (MLSA) poses new challenges for developers and how these challenges make it difficult to automate refactorings on MLSAs. For that purpose, we present the diverse effects of single-language refactorings on MLSAs with the help of a prototypical software application.

Apart from the specific challenges single-language refactorings pose, it is considered *hard* to realize Multi-Language Refactorings (MLRs) for MLSAs in general [Chen and Johnson, 2008]. In this chapter, we describe possible reasons why previous approaches have not been able to realize a general approach to automatic MLR and, thus, why MLR has not found widespread adoption, yet.

This chapter is structured as follows. In Section 3.1, we illustrate the effects of refactorings for different programming languages and paradigms that we applied on the prototypical software application HRManager (see Section 2.2.2). In Section 3.2, we describe general challenges of realizing automated MLR approaches before we summarize this chapter.

3.1 Applying Known Refactorings on HRManager may Cause Inconsistent Changes

In the following, we report on effects we observed when we applied nine single-language refactorings on different parts of HRManager. We applied all refactorings manually and evaluated whether the refactoring can be automated. We call a manual refactoring on HRManager *successful*, if a possibly empty set of source-code modifications exists that adheres to the following conditions:

1. Applying the set of source-code modifications preserves the semantics of HRManager.
2. Each source-code modification in the set of source-code modifications is a refactoring.

These conditions describe the definition of MLR given in Section 2.3. By semantics, we refer to the specification of HRManager.[1]

In contrast to source code, relational databases consists of a schema and data. Thus, when refactoring a database, we have to take the effect of the refactoring on the schema as well as the data into account. For that purpose, we distinguish two terms of semantic preservation that describe how undoing a database refactoring affects the data: *reversible* and *symmetrically reversible* [Hainaut, 1996]. A transformation of a database schema and the related data instances is semantic-preserving, if the transformation is reversible [Hainaut, 1996]. That is, for transformation $T1$ a transformation $T2$ exists, that undoes $T1$. However, please note that a reversible transformation is only semantic-preserving in respect to the database schema of $T1$ and not on the data. An example for a reversible transformation can be given by the Replace Column refactoring [Ambler, 2003, p. 416]. The purpose of the Replace Column refactoring is to adapt a column's type to a new definition or usage. For instance, let us assume that we store a customer identifier as an integer. However, the business wants to be able to identify the customer type by the identifier. For that purpose, we change the identifier's definition in such a way that allows to include an alphanumeric type-code [Ambler, 2003, p. 416]. So, after we applied the Replace Column refactoring, the database can include customer identifiers such as `4711` or `4711EXT`. However, if we revert the refactoring, customer identifiers including letters cannot be represented in the database schema and customer identifiers such as `4711EXT` need to be adapted manually to the reverted database schema. Ergo, in the presented example the Replace Column refactoring is a reversible but not a symmetrically reversible transformation.

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 [Hainaut, 1996]. Hence, in contrast to reversible transformations, we can undo symmetrically reversible transformations without losing any data. An example for a symmetrically reversible transformation can be given by the Split Column refactoring [Ambler, 2003, p. 420]. The purpose of the Split Column refactoring is to separate different purpose or data elements from each other. For instance, a column contains an amount of money and its currency. To separate the amount from the actual

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

currency, we split the original column into two columns holding either the amount or the currency. If we revert the refactoring, we do not need any special treatment of data that has been created in the database schema with the split columns, since we only need to merge the amount with the currency information. Thus, the Split Column refactoring is a symmetrically reversible transformation.

3.1.1 Object-oriented Refactorings

At its introduction, practitioners expected the object-oriented paradigm to be a significant leap towards easing the implementation of reusable software. However, researchers and practitioners recognized that object-oriented source code is not reusable by design and, thus, also needs to be restructured to enable or preserve the reusability of the source code [Opdyke, 1992]. In [Opdyke, 1992], Opdyke described the basis for the disciplined refactoring of object-oriented source code. With the widespread adoption of the object-oriented paradigm, also object-oriented refactoring found its way into software development. The adoption and automation of object-oriented refactorings in state-of-the art IDEs like Eclipse or Visual Studio prove the practical importance of object-oriented refactorings in software development.

In the following, we describe the effects of the

- Rename Method [Fowler, 1999, p. 273],
- Pull Up Method [Fowler, 1999, p. 322], and
- Move Class [Opdyke, 1992]

refactoring when applied to the Java source-code of HRManager. We selected these refactorings because of the following reasons: (1) The refactorings modify the class hierarchy which is an important concept in the object-oriented paradigm, (2) the refactorings' usage has been proven in studies [Murphy-Hill et al., 2009; Weißgerber and Diehl, 2006], and (3) we find possible candidates for applying these refactoring in the source code of HRManager.

Rename Method Refactoring

The Rename Method refactoring is used when the name of a method does not describe the purpose of the method correctly. In HRManager, class `Employee` defines the method `getSalary`, but the method's name *getSalary* does not describe the purpose of the method precisely. The method `getSalary` returns the monthly salary, so we rename the method to `getMonthlySalary`. To preserve the semantics of HRManager, we must perform the following actions:

1. Rename `getSalary` to `getMonthlySalary`.
2. Rename `setSalary` to `setMonthlySalary`.

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

Listing 3.1: Utilizing the `@Column` annotation for restoring the ORM.

3. Restore the Object Relational Mapping (ORM) by applying one of the following alternatives.
   (a) Rename column `salary` in table `employees` to `monthlysalary`.
   (b) Add `@Column` annotation to the method `getSalary` with the `name` attribute of the annotation set to the column name `salary`.

Recall that by default, Hibernate maps getter and setter pairs defined in the Java class on columns defined in the database schema, so we need to apply the Item 2 to restore the ORM.

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 Item 1 and Item 2). Second, in Item 3 we have the choice between two actions for restoring the ORM. If we select the first action (Item 3a) we have to rename the column and we must change an unknown number of Structured Query Language (SQL) statements referring to that column. The second action (Item 3b) includes a single modification. Listing 3.1 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 Item 3a and Item 3b differ in their complexity since Item 3b makes the modification of SQL statements unnecessary. Furthermore, without 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 which would force us to abort the MLR.

In summary, the Rename Method refactoring can become a challenging source code modification if the developer chooses to adapt the database schema to the renamed method. Knowledge about the object-relational mapper and its ability to decouple the source code from the database schema can reduce the effort to implement the Rename Method refactoring significantly. The knowledge may even be necessary to prevent name clashes that would perhaps prevent the developer from applying a Rename Method refactoring otherwise.

Pull Up Method Refactoring

The Pull Up Method refactoring unifies identical or similar methods of several subclasses in a superclass. By unifying and moving the methods in the superclass, we reduce code

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

Listing 3.2: Establishing a supervisor relationship between the manager *Greenspan* and the supervisor *Gartner*.

duplicates and promote code reuse. 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` does not provide any methods to manage supervisors. We want to pull-up the methods `getBoss` and `setBoss` from `Manager` to `Employee` in order to be able to reuse code to manage supervisors on all kinds of employees. 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`.

Recall that, since Hibernate maps pairs of getter and setter methods defined in a Java class on columns defined in the database schema, we need to apply the Item 3 to restore the getter and setter pair `getBoss` and `setBoss` inside class `Employee`.

The transformation of the database schema informally described by Items 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`. Therefore, we call the Pull Up Method refactoring an MLR in HRManager. However, the transformation is not symmetrically reversible, because by removing the column `boss` from table `employees` (required when inverting the refactoring) tuples of pure employees lose 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 losing information.

The modification of other SQL statements referencing the column `boss` can be challenging as Listings 3.2 and 3.4 show. In Listing 3.2, the `UPDATE` statement introduces a subordinate-boss-relation between the datasets with the surnames *Gartner* (boss) and

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

Listing 3.3: Establishing a supervisor relationship between the manager *Greenspan* and the supervisor *Gartner* by reusing a modified implementation of Listing 3.2.

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

Listing 3.4: Establishing a supervisor relationship between the manager *Greenspan* and the supervisor *Gartner* with a single string comparison after the Pull Up Method refactoring is applied.

Greenspan (subordinate). One way to adapt the `UPDATE` statement in Listing 3.2 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`) as presented in Listing 3.3. However, this modification only ensures that we can establish a supervisor relationship between managers in the refactored schema. Listing 3.4 shows an additional option to modify the SQL statement that allows to establish a supervisor relationship between any two employees: We can simplify the `WHERE` statement in Listing 3.2, Line 3 by changing the nested `SELECT` statement to a single string comparison (Listing 3.4, Line 3). Therefore, there exist at least two possible modifications of the `UPDATE` statement in Listing 3.2 that differ in the amount of changes to apply and may also differ in their performance (assuming that the single string comparison provides a better performance than the nested `SELECT` statement). However, we argue that the transformations of the nested `SELECT` statement to a single string comparison can only be accomplished by semantic analysis of the source statement (e.g., Listing 3.2). In our opinion, only by the structure of SQL we cannot fathom how to change `UPDATE` statements like the one in Listing 3.2 in general.

We conclude that adapting existing SQL statements to a moved column could be challenging depending on the complexity of the SQL statement at hand. The unknown complexity of the adaption makes it demanding to automate the Pull Up Method refactoring in a database application context.

Move Class Refactoring

The Move Class refactoring changes the superclass of a class to allow reuse of the class's functionality. 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* (cf. Figure 2.3). However, the inheritance relationship between `Salesperson` and `Manager` is purely artificial and only established to enable the re-use of the attribute *company car*. In reality, salespersons are not 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 Items 3 and 5 are reversible, because we can undo the changes described without losing 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 an MLR.

After the Move Class refactoring, `Salesperson` is not a subclass of `Manager` anymore. Thus, code that assumes that all instances of `Salesperson` are also instances of `Manager` breaks. Let us consider for instance Listing 3.5: Before the refactoring, the loop will print all managers and salespersons. After the refactoring, the loop will print only the names of managers, but not of salespersons. 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 code automatically that was built with this assumption in mind. Eventually, only the developer is able to assess if a change of the inheritance relationship affects a piece of code as in Listing 3.5. Thus, we depend on manual or automatic tests that check the a existing dependency on the inheritance relationship between class `Salesperson` and `Manager`.

In summary, the Move Class refactoring can be considered an MLR in the context of a database application. However, we have to carefully investigate if the change in the inheritance hierarchy does not impact implicit assumptions about the types we access in the database. This investigation may be challenging to automate.

3.1.2 Database Refactorings

In a database application, we also need to be able to adapt the database schema to new requirements. In [Ambler, 2003], Ambler distinguishes the following database refactoring categories: Architectural, Data Quality, Performance, Structural, Referential

```java
Query query = sess.createQuery("from␣Manager");
List<Manager> managers = (List<Manager>) query.list();

for (Manager manager : managers) {
  System.out.print(manager.getFirstName() + "␣" + manager.getSurname());
}
```

Listing 3.5: A loop printing names of managers.

Integrity. In [Ambler and Sadalage, 2006], Ambler and Sadalage replace the category performance by two new categories: Transformation and Method Refactoring.

In the following, we describe the effects of the

- Introduce Default Value [Ambler, 2003, p. 408],
- Introduce Redundant Column [Ambler, 2003, p. 409], and
- Remove Table [Ambler, 2003, p. 413]

refactoring when applied to the database schema of HRManager. We selected these refactorings because we are interested in the sole effect of structural changes of the database schema on the database application. For that purpose, all three refactorings belong to the category structural. Furthermore, all three refactorings do either involve no or a minimum effort to migrate existing data.

Introduce Default Value Refactoring

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

In HRManager, we want to set the default value for the column `account` defined in the table `managers` to `"acquisition"`, because a manager has to book 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`.
2. Initialize the field `account` of the class `Manager` with the value `"acquisition"`.

Item 2 is necessary to preserve the semantics of the default value defined in the database for classes defined in Java. Consider that we would not have applied Item 2: In the Java source code, on the creation of a new instance of the class `Manager`, the field `account` is initialized with `null` according to the Java semantics. When we store the instance of

```
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 }
```

Listing 3.6: Method definition `setAccount`.

the class `Manager` in the database, all field values are stored in the database, whether they have been set explicitly or only implicitly during instance construction. Therefore, `null` is written to the column `account`. The default value of the column `account` is never applied to instances of class `Manager`.

The modification described in Item 2 can be semantic-changing, because there can be methods assuming the field `account` is initialized with `null` instead of being set to `"acquisition"` after initialization. Those methods would behave differently after the refactoring. Additionally, since we only know that the column `account` maps onto the getter and setter methods `getAccount` and `setAccount`, setting the initial value in Item 2 requires semantic analysis of the getter and setter to identify the fields accessed and modified by the getter and setter methods. The analysis of the implementation of getter and setter methods is not hard for trivial implementations, but needs advanced treatment for non-trivial getter and setter methods. In Listing 3.6, 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 and can hardly be automated.

During the application of the Introduce Default Value refactoring, we have identified two problems. First, we cannot guarantee that the Introduce Default Value refactoring preserves the semantics of HRManager, since we do not know which behavior the application expects regarding the initialization of field `account` with a `null` value. Furthermore, we need to analyze the semantics of getter and setter methods to set the initializing value for fields correctly.

Introduce Redundant Column Refactoring

The Introduce Redundant Column refactoring creates a copy of a column of a source table in a target table. This refactoring is used to improve the performance of database queries in the case that the column of the source table is queried frequently when a dataset of the target table is queried. In Figure 3.1, 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`, the joins to retrieve the department

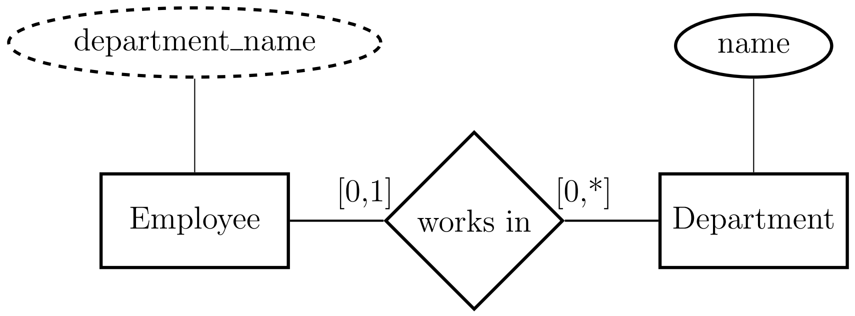


Figure 3.1: Extended ER schema showing the entities `Employee` and `Department`, whereas attribute `department_name` of `Employee` is derived from attribute `name` of `Department`.

an employee is working for become unnecessary. 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 the object-relational mapper.
5. Extend the functionality of the classes `Employee` and `Department` to maintain the consistency between the fields `department_name` and `name` in the Java source code.

Items 4 and 5 are not necessary to preserve the functionality of HRManager. However, if we do not implement Item 4 and Item 5 we cannot benefit from the performance gain available through the database schema. Thus, in this particular case the database refactoring demands an extension of the Java application in order to take advantage of the potential performance improvement.

The modifications described in Items 1 to 3 conform to the steps in the refactoring definition and are semantic preserving [Ambler, 2003, p. 409]. Hence, we can call the modifications of Items 1 to 3 an MLR.

The extension of functionality described in Item 5 violates the refactoring definition. Two modifications are required to implement the extended functionality. First, we have to secure field `department_name` in class `Employee` against unauthorized writes (only the object-relational mapper and the referenced instance of type `Department` may write the field). Second, we have to implement a source-code pattern like the Observer Pattern [Gamma et al., 1993, p. 293] to preserve the consistency between the department name in instances of `Employee` and `Department`. Thus, the modifications described in the Items 1 and 5 do not adhere to the definition of MLR, because Item 5 does not describe a refactoring. Only the modifications in the Items 1 to 3 preserve the semantics of HRManager. Thus, we found two alternative ways to adapt the HRManager to the Introduce Redundant Column refactoring applied to HRManager's database. However, only one of the possible ways to implement the Introduce Column refactoring makes the possible performance gain available in the Java source code. That is, the refactoring of the database schema leads to semantics-changing modifications of the Java source code, if we want to comply with the refactoring of the database schema. However, semantics-changing modifications are not only leaving the refactoring domain, additionally, these modifications have to be done in the context of a specific application and, thus, are hard to automate.

Remove Table Refactoring

The Remove Table refactoring is used to remove a table from a database schema, if the table is deprecated or not used.

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. Assuming that that class `ExternalStaff` is not used in conjunction with the ORM, we have to modify the HRManager in the following way:

1. Remove the table `external_staff` from the database schema.
2. Remove class `ExternalStaff` from the ORM.

To remove the class `ExternalStaff` from the ORM, we have to remove the mapping entry `<mapping>hrm.ExternalStaff</mapping>` from the `hibernate.cfg.xml`. The configuration file `hibernate.cfg.xml` is specific to Hibernate. For instance, Java Persistence API (JPA) specifies the `persistence.xml` for the definition of classes that should be considered in the ORM. Thus, in general, we have to consider framework specific differences when applying this refactoring in a database application which uses an object-relational mapper.

For this refactoring to be successful, we have to ensure that no other parts of the application access the table `external_staff`. Otherwise, since after the refactoring class `ExternalStaff` is not part of the ORM, the object-relational mapper will throw an error. With Hibernate, the developer has many different options to access the database.

```java
// Hibernate native API
Department department1 = session.get(Department.class, 1);

// HQL/JPQL
Query query1 = session.createQuery("from Department d where d.id = 2");
Department department2 = (Department) query1.list().iterator().next();

// Criteria API
List<Department> results = session.createCriteria(Department.class)
                              .add(Restrictions.eq("id", new Integer(3)))
                              .list();
Department department3 = results.get(0);

// Native SQL
String sqlQuery = "SELECT * FROM departments WHERE id = 4";
List<Object[]> departments = sess.createSQLQuery(sqlQuery).list();

Department department4 = new Department();
department4.setId((Integer)departments.get(0)[0]);
department4.setName((String)departments.get(0)[1]);
```

Listing 3.7: Possible options to access persistent data via Hibernate.

Listing 3.7 lists four of the Hibernate-specific options to query the database for a specific dataset[2]. Additionally, the developer may uses the interfaces defined by the JPA specification which Hibernate implements. Thus, to be able to determine whether the developer still accesses the table `external_staff` or not, we have to consider a number of interfaces that developers could possibly use to access data within the application. Additionally, in respect to the Hibernate Query Language (HQL) and the Java Persistence Query Language (JPQL) or the native SQL Application Programming Interface (API), we have to consider possible dynamic string manipulation.

In summary, for the Remove Table refactoring we have to take framework-specific configurations and interfaces into account. Especially, when we check the preconditions for the Remove Table refactoring, we need to consider the API usage and the additional problems that may related to it. For instance, if the API of an object-relational mapper allows access the database based on strings like Hibernate's HQL or native SQL API, we have to consider the dynamic creation of strings as a possible pitfall for correctly determining the tables accessed in the database.

3.1.3 Functional Refactorings

The functional paradigm naturally supports the implementation of parallel algorithms [Lämmel, 2008; Loidl et al., 2003]. Since the importance of parallel execution

[2]The example of the Criteria API in Listing 3.7 is based on a deprecated version of the Criteria API, since the HRManager is using Hibernate 3.3.0.SP1 while the latest Hibernate version is 5.2. However, the current version of Hibernate still provides a Criteria API which is based on the JPA specification. Thus, also recent Hibernate releases provide diverse options to access specific datasets.

```
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)))))))
```

Listing 3.8: Definition of the function `sumSalary`.

increased in the recent years, also functional programming languages received increasing interest. We assume that the increasing interest will result in an increasing amount of functional code in MLSAs and, thus, increased need to adapt functional code to new or changed requirements.

In the following, we describe the effects of the

- Introduce New Definition [Li, 2006, p. 18],
- Promote Definition [Li, 2006, p. 16], and
- Move Definition [Li, 2006, p. 20]

refactoring when applied to the Clojure source code of HRManager. We selected these refactorings because other refactorings described in Li [2006] are either specific to the Haskell programming language or we already applied the object-oriented equivalents to the Java source code of HRManager.

Introduce New Definition Refactoring

The Introduce New Definition refactoring defines a local definition for an anonymous expression. The purpose of this refactoring is to enable the re-use of an otherwise anonymous function. Thus, this refactoring may be compared to the Extract Method refactoring [Fowler, 1999, p. 110] for object-oriented source code.

In HRManager, we defined the Clojure function `sumSalary` which computes the total salary of all instances of the class `Employee` in list `x`. Listing 3.8 shows the definition of function `sumSalary`. In Line 3, 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*) with the anonymous expression `(instance? hrm.Employee (first x))`.

We want to apply the Introduce New Definition refactoring in order to create a function `isEmployee?` from the instance expression. To this end, we need to apply the following modifications to HRManager:

1. Enclose the instance expression with a `letfn` statement.
2. Within the parameter list of the `letfn` statement defined in Item 1, define the instance expression as the body of 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))))))))
```

Listing 3.9: The function `sumSalary` with the additional `letfn` (Line 3) statement defining the function `isEmployee?`.

3. Within the body of the `letfn` statement, replace the instance expression by a call to the new function `isEmployee?`.

With the help of the `letfn` statement introduced in Item 1, we can define local functions. Local functions defined with `letfn` are visible within the body of the `letfn` statement. Listing 3.9 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. Admittedly, the local function `isEmployee?` may even reduce the readability of the actual source code. We tackle this issue in the next section.

In this example, we are not able to reference the refactored code, that is, the anonymous function `(instance? hrm.Employee (first x))`. Thus, there cannot exist any reference to this expression from source code of other languages. Hence, since there cannot be effects on source code of other languages, we can call the Introduce New Definition refactoring an MLR.

We conclude that being an MLR is inherent in the Introduce New Definition refactoring: The refactoring does not affect the interaction with source code of other languages, since the refactoring introduces structural elements that cannot have been part of the language interaction before the refactoring. Thus, we call refactorings which only create new structural elements MLRs.

Promote Definition Refactoring

The Promote Definition refactoring increases the scope or visibility of a definition. By increasing its scope, the definition can be re-used by other definitions.

In HRManager, we defined the function `isEmployee?` with a `letfn` statement, as shown in Listing 3.9, Lines 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?` in such a way that we are able to reuse `isEmployee?` in other functions. To promote the definition of `isEmployee?` into a new, globally visible function `isEmployee?`, we need to apply the following modifications to HRManager:

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

Listing 3.10: The function `sumSalary` with the globally visible definition of `isEmployee?`.

1. Introduce the new function definition `isEmployee?` in the global scope.

2. Let the body of the `isEmployee?` function defined in the `letfn` statement be the new body of the function `isEmployee?` introduced in Item 1.

3. Remove the `letfn` statement from the function `sumSalary`.

Listing 3.10, Line 1, shows the function `isEmployee?` introduced by the Promote Definition refactoring. The `letfn` statement is removed, only the body is preserved (Listing 3.10, Line 5).

Because the function `isEmployee?` was not visible before the Promote Definition refactoring, again, source code of other languages could not reference the function `isEmployee?`. Thus, we do not need to apply further modifications to Java source code. For that reason we call the Promote Definition refactoring an MLR.

Move Definition Refactoring

Clojure provides namespaces to group functions [VanderHart, 2010, p. 24]. The Move Definition refactoring describes how functions can be moved between different namespaces.

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`. Instead we want to move the function to the namespace `management`. We need to perform the following modifications to change the namespace:

1. Move function `managersWithBoss` from the namespace `salary` to namespace `management`.

2. Modify calls to `managersWithBoss` from Java source code.

```
1 RT.var("salary", "managersWithBoss"); // before
2 RT.var("management", "managersWithBoss"); // after
```

Listing 3.11: An excerpt of the reference to the function `managersWithBoss` in Java after the application of the Move Definition refactoring.

Line 1 in Listing 3.11 shows how calls to `managersWithBoss` look before before and Line 2 shows how calls to `managersWithBosss` must look like in the Java source code after performing Item 2.

In Java, we resolve dependencies to missing classes by using Java's `import` statement. For dynamic calls of 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 Java's `import` statements. Thus, we have to take Clojure's language-specific functions into account, if we want to preserve the language interaction between Java and Clojure code.

3.2 Challenges of Multi-language Refactoring

Chen and Johnson stated that: "While finding a general solution for extending refactoring across multiple languages is *hard*, it is *simple* and *possible* to support automated refactorings for some common cases that programmers already encounter in their programs today" [Chen and Johnson, 2008]. In this section, we attempt to explain why it is hard to find a general approach to MLR.

Chen's and Johnson's statement actually describes one challenge of refactoring MLSAs: *How does a refactoring affect the interaction between languages*? Obviously, as Chen and Johnson stated, we can answer this question for known cases. However, tracking all possible language interactions affected by a refactoring poses an entirely different challenge.

Besides the information if two programming languages interact with each other, for an MLR, we need to know how the interaction between two programming languages is implemented. Otherwise, we are unable to adapt the source code written in the host language to the refactored source-code written in the guest programming language and vice versa. Recall that we defined an MLR as a possible empty set of refactorings that preserve the semantics of the application. However, in order to preserve the semantics of the overall application, we may need to apply semantics-changing modifications (cf. Section 3.1.1 and Section 3.1.2). Thus, in an MLSA, we cannot generally assume that a refactoring of source code written in one language will either have no affect on source code written in other languages or lead to only semantics-preserving modifications, that is, additional refactorings, on interacting source code. Consequently, in order to preserve the semantics of the entire software application, a refactoring of one part of the application may forces us to apply semantics-changing modifications to other parts of the application.

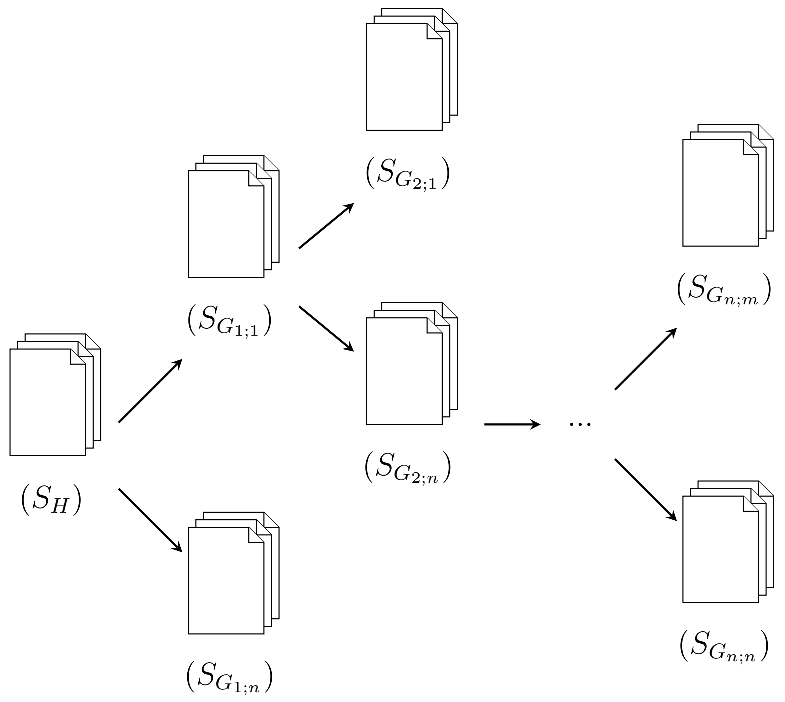


Figure 3.2: Illustration of the general structure of an MLSA.

So far, we addressed issues of MLRs only considering a pair of programming languages. However, in an MLSA a guest language may be itself a host language interacting with another guest language. Thus, a refactoring of source-code written in the host language may not only affect source code written in one but any number of different guest languages. That is, assuming that all programming languages may be used as host language, we have to address all possible host and guest language combinations. Additionally, we have to take into account how refactorings affect different host- and guest-language interactions and how we can adapt the source code written in the guest language to refactorings in the host language and vice versa. We describe the set of all possible host and guest language combinations as the transitive closure of all language interactions. We assume that the extent of the transitive closure of all language interactions renders a general approach to automated MLR practically infeasible.

In Figure 3.2, we illustrate the general structure of an MLSA. We have a single host language S_H and interacting guest languages $S_{G_{m;n}}$. Guest languages that directly interact with the host language have the index $(1; n)$ and guest languages indirectly interacting with the host language have an index $(x; n)$ with $x > 1$. A tool developer has to know not only the host language S_H, but also all possible interacting programming languages $S_{G_{1;1}}$ to $S_{G_{n;n}}$. Let us assume that a tool developer implements an MLR which only considers the languages used in $S_{G_{1;1}}$ to $S_{G_{1;n}}$. Applying the refactoring may cause changes to the source code structure of $S_{G_{1;1}}$ to $S_{G_{1;n}}$ which may break the interaction with $S_{G_{2;1}}$ or $S_{G_{2;n}}$. Thus, an MLR that considers only certain language interactions can break an MLSA in the general case.

In practice, tool developers seem to follow the advice given by Chen and Johnson and integrate refactoring support for specific language interactions. For instance, the refactoring tool ReSharper integrates support for the Rename refactoring on HTML, CSS, JavaScript, and TypeScript [JetBrains s.r.o., 2017] and the Google Plugin for Eclipse supports the Rename refactoring on Java and interacting JavaScript source code [Google Inc., 2017] for applications implemented with the Google Web Toolkit. To the best of our knowledge, also all existing attempts in the scientific community focus on specific language interactions [Kempf et al., 2008; Mayer and Schroeder, 2012; Strein et al., 2006; Tatlock et al., 2008]. The advantage of focussing on a specific language interaction is that we can seize all oportunities for automating MLRs between the two languages of the language interaction. However, only developers of these two languages can take advantage of the automated MLRs. In the next chapter, we will present an approach that introduces a common scheme for representing language interaction. We will show that we can provide already basic support for refactoring in an MLSA for any two languages based on that common scheme. Our approach does not focus on automated refactorings in MLSAs. However, based on the observations outlined in this chapter, we argue that automation can only be achieved in certain cases and, thus, a more general support for refactoring MLSA is necessary.

3.3 Summary

In this chapter, we presented specific challenges that arise when developers apply single-language refactorings on an MLSA. In particular, we have shown issues in the context of refactoring a prototypical database application. The prototypical database application comprises code of four General-Purpose Programming Languages (GPLs) and Domain Specific Languages (DSLs) of different programming paradigms. We assume that the issues we encountered on our prototypical database application can be generalized to other MLSAs that employ different GPLs and DSLs.

Furthermore, we illustrated the challenges that prevent the practical realization of automatic MLR in general. Because of the general challenges we face in the realization of automatic MLR, we not only consider a general approach to automatic MLR *hard*, but we question the practical feasibility of it at all. Especially the need for semantic-changing code modifications contradicts approaches which are based on the assumption that refactoring an MLSA is the application of a set of refactorings on different interacting source-code documents. Thus, as Chen and Johnson already pointed out, automatic MLRs are possible and useful for specific use cases, but do not provide the basis for improving the refactoring experience in MLSAs in general.

Nevertheless, if the current approach to MLRs is only applicable to specific MLSA setups, how could we support refactoring in arbitrary MLSAs? Based on the results of this chapter, we have to conclude that we cannot apply the same approach for refactoring MLSAs that proved to be successful for single-language refactorings: In general, we cannot rely on MLR, that is a set of single-language refactorings applied to a number of interacting source-code documents written in different programming languages.

Since automation is not as easy to accomplish for MLR as compared to single-language refactorings, we propose to step back and to start thinking about tools that support developers in the manual refactoring of an MLSA. An automated MLR would preserve the interaction between the considered programming languages by design. However, without any tool support developers who manually apply refactorings on MLSAs need to check the preservation of language interaction manually as well. Even if developers can fall back on a suite of automated tests, there is a chance that broken language interaction will find its way into production code. Thus, first and foremost, tool support for manual refactoring of MLSAs should help developers to determine if language interaction is affected by a refactoring. Based on this insight, we present an approach to highlight the mechanics of language interaction in the following chapter.

4. Structure Graphs

Chapter 4 shares material with [Schink et al., 2016a].

In Chapter 3, we learned about the diverse effects of refactorings within Multi-Language Software Applications (MLSAs). In summary, refactorings defined for single languages or language paradigms do not respect language interaction in MLSAs and, thus, single language refactorings can break the interaction of languages in MLSAs. A possible approach to improve refactoring in MLSAs would be the definition of multi-language refactorings [Chen and Johnson, 2008; Kempf et al., 2008; Mayer and Schroeder, 2012; Strein et al., 2006]. However, because of the high number of possible language interactions and techniques involved in establishing an interaction between two languages, Multi-Language Refactorings (MLRs) only provide a practical approach for specific language combinations and refactorings. Thus, because of the practical limitations, we do not expect MLRs to become as pervasive as single-language refactorings.

In [Chen and Johnson, 2008], the authors state that "[...] refactoring across multiple languages is hard [...]". Since we cannot expect refactorings to preserve behavior in an MLSA, developers have to rely on a sufficient suite of tests to ensure that after the refactoring of the source code of one language the interaction with other languages is not affected. However, even if the test suite fails, because a single-language refactoring broke the language interaction within an MLSA, developers still need to investigate the reason for the broken language interaction to be able to provide a manual fix. Thus, it is desirable for developers not only to detect broken language interactions, but to get information describing the possible reason for the broken language interaction. Thus, we formulate the following two abilities a tool must provide to ease a developer's effort to refactor an MLSA: (1) The tool presents interactions between artifacts within an MLSA in such a way that developers are able to fix broken interactions manually and (2) the tool provides developers with sufficient information about the modification that caused the broken language interaction, so the developer can fix the broken language interaction correctly.

In this section, we introduce an approach to support refactoring in MLSAs based on graphs of trees. We selected an approach based on graphs for two reasons:

1. Since graphs or more specifically trees are already used to represent source-code elements, we assume that graphs are a practical choice for representing source-code elements involved in language interaction as well.
2. Graph theory provides the necessary theoretical and practical foundation to analyze and implement our approach to support refactoring in MLSAs.

In the following, we present two algorithms which detect broken language interactions and compute the changes that preceded a broken language interaction, respectively. These pieces of information support developers in fixing broken language interactions. Since both algorithms are based on graphs, we start with discussing the representation of source-code elements involved in language interaction with graphs.

4.1 Modeling Elements of Language Interaction

We implement language interaction between a host language and a guest language (see Section 2.2): A guest language defines structure elements like methods and classes and a host language invokes or references these structure elements. Thus, the host language *expects* certain structure elements in the source code implemented in the guest language. The interaction of source code written in two programming languages is preserved if the actual structure elements defined in the source code written in the guest language match the structure elements expected, that is invoked or referenced, in the source code written in the host language. For the comparison of the actual with the expected structure elements, we need to model the respective structure elements involved in language interaction.

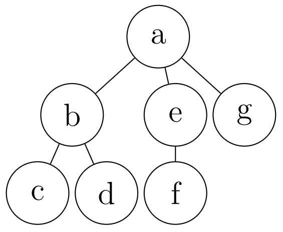
For the source-code structure S_G written in guest language G and the source-code structure S_H written in host language H, we call the references to S_G extracted from the host language's structure $R_{S_H \rightarrow S_G}$. Additionally, we call the structure elements in the source code written in the guest language involved in a language interaction R_{S_G}. Since $R_{S_H \rightarrow R_G}$ and R_{S_G} represent not all structure elements, but those that are involved in language interaction, the following relations hold:

$$R_{S_H \rightarrow S_G} \subseteq S_H \tag{4.1}$$

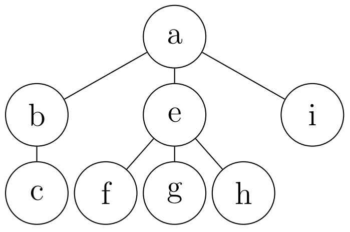
and

$$R_{S_G} \subseteq S_G \tag{4.2}$$

Elements involved in language interaction are source-code elements. We can represent source-code elements in Abstract Syntax Tree (AST) [Aho et al., 2006, p. 91 ff.]. Based



(a) Labeled tree G_a representing the actual source-code structure.



(b) Labeled tree G_e representing the expected source-code structure.

Figure 4.1: Illustrating example for different node types.

on the general representation of source-code elements in tree structures, we represent $R_{S_H \to S_G}$ and R_{S_G} as sets of labeled trees. In this context, we call a set of labeled trees *structure graphs*. Consequently, $r_{S_H \to S_G} \in R_{S_H \to S_G}$ is a tree representing a single invocation of a structure element defined in the guest language from the host language. Likewise, $r_{S_G} \in R_{S_G}$ is a tree representing a single structure element defined in the guest language that is involved in language interaction.

Before developers are able to invoke any element in the guest language from the host language, they must define elements in the source code written in the guest language. Hence, on the one hand, R_{S_G} contains only definitions of elements involved in a language interaction. On the other hand, $R_{S_H \to S_G}$ contains only invocations of the source-code elements defined in R_{S_G}. Thus, a representation for modeling language interaction must be able to describe two elements involved in language interaction: element definitions and element invocations. Element invocations explicitly define all parts involved in the invocation. For instance, a function call contains the function name and a set of parameters. However, element definitions may describe variability or define required information that must be present when the element is invoked. For instance, function definitions can define required parameters as well as an arbitrary number of parameters in the case of variadic functions. Thus, a structure graph contains one or more of the following types of nodes: *plain*, *mandatory*, and *optional list*. These three node types exist to control the comparison of two given structure graphs of which one describes element definitions and the other describes element invocations. We explain their purpose in detail in the following by means of the trees G_a (see Figure 4.1a) and G_e (see Figure 4.1b).

Plain nodes trigger a comparison of the labels of nodes. The comparison of a node's label is also done for the mandatory and optional list nodes. However, if a plain node is defined in the graph with the actual source-code structure, but is missing in the graph with the expected source-code structure, we will still assume that the expected source-code structure matches the actual source-code structure. For instance, in the labeled tree G_a, the node g exists. However, the node g does not exist in labeled tree G_e at the same depth as in node G_a. However, node i, which does not have the same label as node g, causes a mismatch between the two graphs G_a and G_e. In contrast, in G_a the node d exists which has no counterpart in G_e and, thus, does not trigger a mismatch between the two graphs G_a and G_e. Thus, in summary a plain node in the expected source-code structure (G_e) must exist in the actual source-code structure (G_a). However, a plain node in G_a does not need to have a counterpart in G_e.

Mandatory nodes enforce a check of their existence during the comparison. Thus, in contrast to a plain node, a mandatory node must exist if the mandatory node's parent exists. For instance, let us assume node d in G_a is a mandatory node. Then, the missing node d in G_e causes a mismatch between the two graphs G_a and G_e.

Optional list nodes represent a variable number of nodes whereas plain and mandatory nodes represent a single node. For instance, let us assume that node f in G_a is an optional list node. Then, during a comparison between G_a and G_e, the nodes f, g, and h in G_e are matched by f in G_a.

4.2 Referential Integrity Between Languages

We check the referential integrity between languages by comparing each individual tree in $R_{S_H \to S_G}$ with the trees in R_{S_G}. To ensure the referential integrity, for all $r_{S_H \to S_G} \in R_{S_H \to S_G}$ there must be one $r_{S_G} \in R_{S_G}$, so that the following condition is satisfied:

$$r_{S_H \to S_G} \text{ is a top-down subtree of } r_{S_G} \tag{4.3}$$

However, this precondition is not sufficient because nodes may be missing in $r_{S_H \to S_G}$ that are mandatory for language interaction. Hence, for the set of mandatory nodes r_M, we additionally have to check if

$$r_M \subseteq r_{S_H \to S_G} \tag{4.4}$$

The set r_M is defined as follows for mandatory nodes m and the function $parent[x]$ which returns x's parent node:

$$r_M = \{m \mid m \in r_{S_G} \land parent[m] \in r_{S_H \to S_G}\} \tag{4.5}$$

Thus, we check Equation (4.4) only for mandatory nodes whose parents exist in $r_{S_H \to S_G}$.

In Figure 4.2, we illustrate the process of checking language interaction: First, we need to extract $R_{S_H \to S_G}$ from S_H and R_{S_G} from S_G. In accordance with Equation (4.6), we compute the set of mandatory nodes R_M for R_{S_G} and $R_{S_H \to S_G}$:

$$R_M = \{m \mid m \in R_{S_G} \land parent[m] \in R_{S_H \to S_G}\} \tag{4.6}$$

Then, we can compare the extracted references $R_{S_H \to S_G}$ and R_{S_G}. The comparison returns a result which contains the possibly empty sets

$$N_G = R_{S_H \to S_G} \setminus R_{S_G} \tag{4.7}$$

and

$$N_M = R_M \setminus R_{S_H \to S_G} \tag{4.8}$$

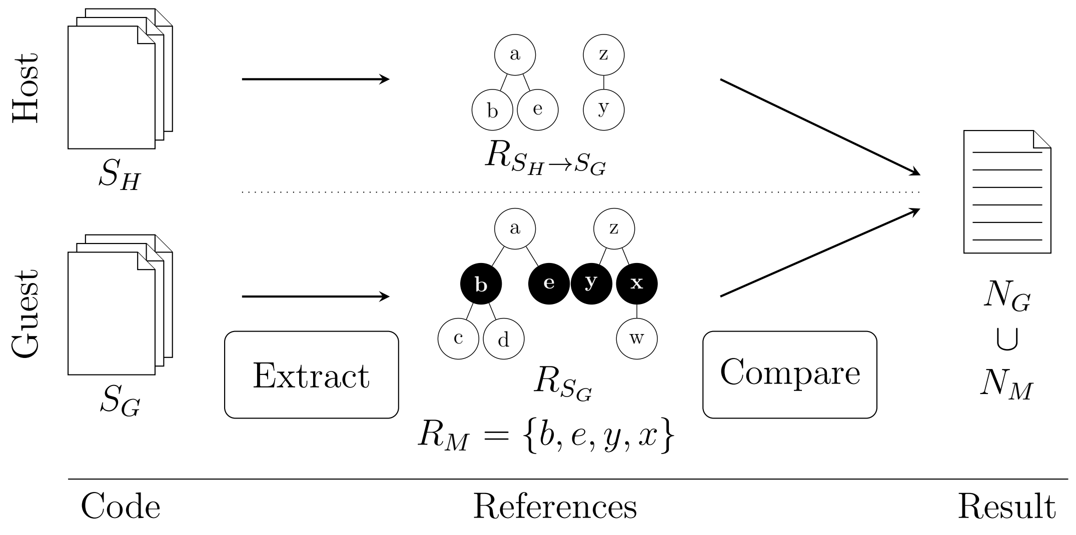


Figure 4.2: The process of checking language interaction.

N_G contains the elements which were extracted from the source code written in the host language, but are missing in the source code written in the guest language. N_M is the set of mandatory elements defined in the source code written in the guest language that are not referenced in the source code written in the host language. In short, N_G is the set of missing guest language nodes and N_M is the set of missing mandatory nodes. Language interaction is preserved if both sets N_G and N_M are empty. Otherwise, the sets contain the structure elements which are involved in the broken language interaction.

4.3 Changes to the Source-Code Structure

In Section 4.2, we present a method which allows developers to detect broken language interaction. However, developers still need to detect the changes that are responsible for the broken language interaction. For instance, in a development team, different developers may be responsible for the host- and guest-language structures S_H and S_G. Let us assume that one team changes S_G in such a way that the references for language interaction R_{S_G} are changed, too. The development team of S_H detects the broken language interaction between S_H and S_G. However, without knowing the changes applied by the development team of S_G, the team of S_H will not be able to fix the broken language interaction adequately. In this section, we present an approach based on structure graphs to detect the applied changes.

In the following, we consider changes to source code S in general. However, the following approach may be especially useful for the developers of the source code S_H: Developers of source code S_H may not be involved in or do not immediately detect changes to the source code S_G. This forces developers of S_H to refer to the developers of S_G or to the source code S_G itself to adapt S_H accordingly. Thus, detecting changes in S_G is sufficient to support developers of S_H in restoring language interaction. Nevertheless, the following method may also be applied to detect changes to S_H.

We detect changes to S by extracting and comparing different revisions of R_S. We may extract different revision of R_S from a version control system, IDE, or any other appropriate source. However, the following approach is independent from the actual source of the revisions.

We distinguish different revisions of R_S by the index rev, so that the current revision is $R_{S;rev}$ and the previous revision is $R_{S;rev-1}$. The result of the comparison is a set of tree modifications containing the added nodes N_a with

$$N_a = R_{S_{rev}} \setminus R_{S_{rev-1}} \tag{4.9}$$

and the removed nodes N_r with

$$N_r = R_{S_{rev-1}} \setminus R_{S_{rev}} \tag{4.10}$$

In Figure 4.3, we illustrate the process of extracting added and removed nodes by comparing the revisions $R_{S_{rev}}$ and $R_{S_{rev-1}}$.

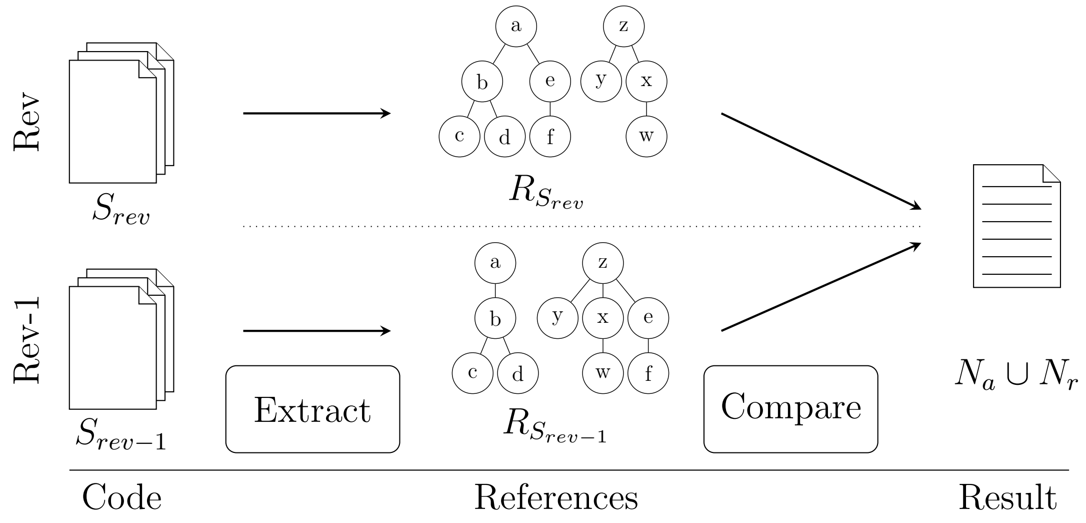
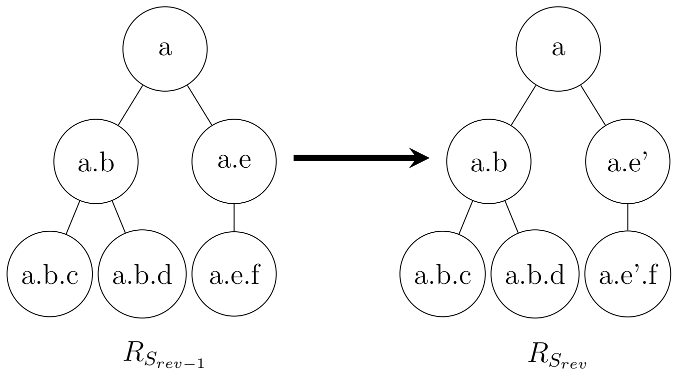


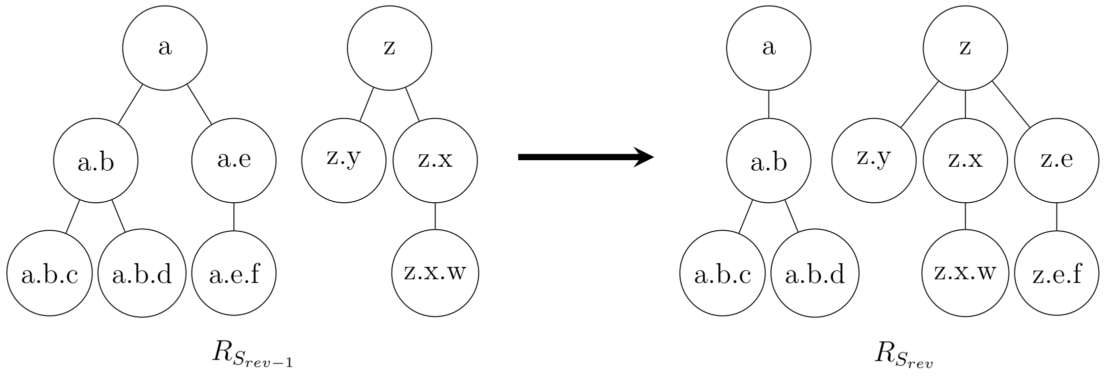
Figure 4.3: The process of detecting changes to the guest language structure.

Addition and *removal* are elementary tree-edit operations [Valiente, 2002, p. 56].[1] However, addition and removal only allow to detect new or missing nodes. Thus, a change to an existing node would be represented as a removal and a subsequent addition. But having a removal and an addition for a change obfuscates that the *same* node has changed. Hence, the developer may not be sure whether the added node is a new element or a modification of an existing one. Therefore, additionally, we introduce the *move* and *rename* modification to preserve information about changes to an existing node that would be lost otherwise.

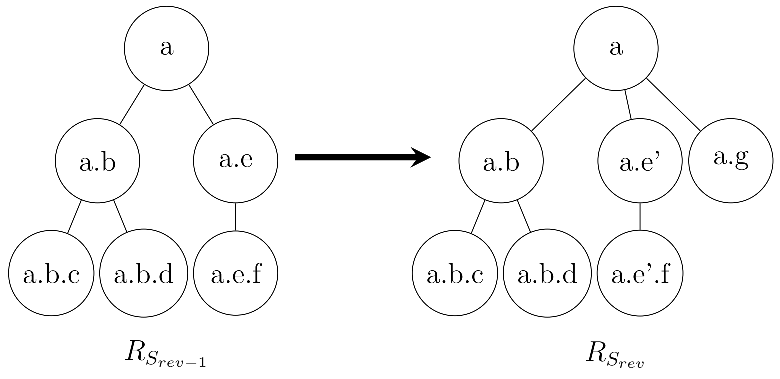
[1]Another elementary tree-edit operation is *substitution* [Valiente, 2002, p. 56]. We consider substitution in the *move* and *rename* operation.



(a) Renamed node detection.



(b) Moved node detection.



(c) Ambiguous node modification.

Figure 4.4: Detection of renamed or moved nodes.

We consider a node $n \in R_{S;rev-1}$ renamed in $R_{S;rev}$ if the following equation holds:

$$n \in R_{S;rev-1}, \exists! n' \in R_{S;rev}(parent[n] = parent[n'] \wedge label[n] \neq label[n']) \tag{4.11}$$

That is, for n exactly one node $n' \in R_{S;rev}$ exists which has the same parent but not the same label. Figure 4.4a presents how the node $n_{a.e} \in R_{S;rev-1}$ is renamed to $n_{a.e'}$ in $R_{S;rev}$. We consider a node $n \in R_{S;rev-1}$ moved in $R_{S;rev}$ if holds

$$n \in R_{S;rev-1}, \exists! n' \in R_{S;rev}(parent[n] \neq parent[n'] \wedge label[n] = label[n']) \tag{4.12}$$

That is, for n exactly one node $n' \in R_{S;rev}$ exists which has the same label but not the same parent. Figure 4.4b presents how the node $n_{a.e} \in R_{S;rev-1}$ is moved to $n_{z.e}$ in $R_{S;rev}$.

The Equations (4.11) and (4.12) hold only for unambiguous node modifications. However, ambiguous node modifications may appear. For instance, in Figure 4.4c, node $n_{a.e} \in R_{S;rev-1}$ is renamed to $n_{a.e'}$ in $R_{S;rev}$, additionally, a new node $n_{a.g}$ is added to $R_{S;rev}$. According to Equation (4.11), no renaming occurred, because $n_{a.e'} \in R_{S;rev}$ and $n_{a.g} \in R_{S;rev}$ share the same parent with $n_{a.e} \in R_{S;rev-1}$. In practice, two source-code revisions $R_{S;rev}$ and $R_{S;rev-1}$ can differ in more than one modification. Thus, due to ambiguity, we may not be able to detect renamed and moved node modifications that actually occurred. However, approaches to schema matching offer (semi-)automatic techniques to find suitable matchings. In [Rahm and Bernstein, 2001; Shvaiko and Euzenat, 2005], the authors provide a classification and a comparison of existing approaches to schema matching. We assume that a composite matcher, which extends our graph-based approach by other matching approaches, will enable developers to deal with ambiguous node modifications. In Section 4.4, we classify the integrity check and change detection algorithm according to Rahm and Bernstein [2001] to underpin this assumption.

In this section, we present an approach to extract changes between source-code revisions. The purpose of this approach is to enable developers to detect the source-code changes that led to a broken language interaction. For instance, based on the change-detection algorithm, developers are able to distinguish whether a broken language interaction is caused by a missing or renamed element in the S_G. Thus, developers do not need to consult the source code written in a maybe unknown guest language nor do they need to check the revision history of the source code.

4.4 Classification of Matching Algorithms

The referential-integrity check and the change-detection algorithm compare two structure graphs with each other and check if both structure graphs match. Thus, in simple terms, we can classify both algorithms as matching algorithms. According to the definition in [Rahm and Bernstein, 2001], a *match* algorithm[2] takes two *schemas* and returns

[2]In [Rahm and Bernstein, 2001], the authors use the term *match operator*. Since we defined algorithms and not operators, we use the term *match algorithm* instead to stay consistent.

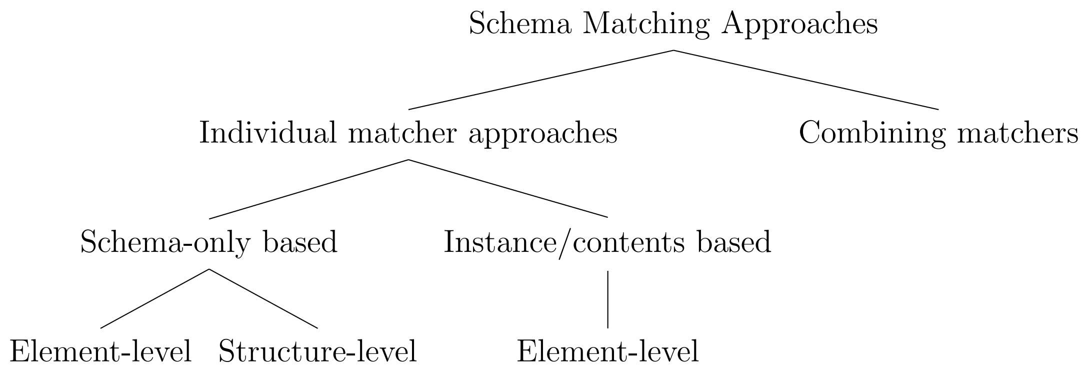


Figure 4.5: Excerpt of a taxonomy of schema matching approaches as defined by Rahm and Bernstein [2001].

a *match result*. A schema as defined by the authors "is a set of elements connected by some structure". The authors mention directed graphs among others as valid representations of such schemas. The match result returns possibly matching candidates. The *outermatch* algorithm is an extension of the match algorithm that, additionally, considers elements with no matching counterpart [Bernstein and Rahm, 2000; Rahm and Bernstein, 2001]. In contrast, our algorithms return only the elements that have no matching counterpart. In the following, we classify our algorithms according to the taxonomy defined in [Rahm and Bernstein, 2001] (cf. Figure 4.5) to show opportunities for further extending the existing algorithms.

We distinguish two matching granularities: *element-level* and *structure-level*. Element-level matchers compare two elements by the elements' properties. Structure-level matchers compare two elements based on their position in a higher-level structure. That is, in a structure-level approach also the surrounding elements are considered. We represent source-code elements involved in language interaction as structure graphs, that is, graphs of labeled trees. The nodes and edges of a structure graph define its schema and the labels define the actual instance of the schema element as defined by the source code. For instance, in source code of an object-oriented programming language, nodes may represent structure elements like namespaces, classes, and methods. Then, the node labels represent the namespace identifiers, class and method names, respectively. Our matching algorithms use structure information to check if the elements of two structure graphs are composed identically and instance information to check if the actual instances represented by two nodes at the same position match. Thus, according to the taxonomy, our matching algorithms possess properties of structure-level and element-level approaches.

The matching cardinality describes how many matching elements a matcher can assign to a single element. For instance, an element of one schema may represents a single or a number of elements in another schema. The change detection algorithm possesses a match cardinality of 1 : 1, that is, for each element in one structure graph it finds at most one matching element in the other structure graph. The integrity check algorithm needs to match optional list nodes, that is, the algorithm must be able to match a number of nodes in one structure graph with a single optional list node in another structure graph.

Our two matching algorithms do not possess any other approaches like linguistic and constraint-based approaches. Thus, both matching algorithms represent hybrid matchers with a cardinality of $1:1$ (change detection) and $1:n$ (integrity check). Nevertheless, the algorithms to not integrate any auxiliary information like user input. Auxiliary information can be used to solve ambiguous node modifications.

4.5 Performance and Generality

In Section 4.2, we discussed an algorithm for checking the interaction between a host and a guest language that builds on a tree-based representation of syntax elements involved in language interaction. In the following, we discuss the theoretical performance of the integrity check discussed in Section 4.2 and the change detection approach presented in Section 4.3 as well as the generality of our approach.

4.5.1 Performance of the Integrity Check

For Condition (4.3), we check that for each node in $R_{S_H \rightarrow S_G}$ a node exists in R_{S_G}. In a tree structure, each node has a unique path, thus, we need to check at most h_G nodes in R_{S_G} for each node in $R_{S_H \rightarrow S_G}$, where h_G is the height of R_{S_G}. Hence, we get each missing node in $O(n_H h_G)$, where n_H is the number of nodes in $R_{S_H \rightarrow S_G}$.

For Condition (4.4), we check that for each mandatory node in R_M a node exists in $R_{S_H \rightarrow S_G}$. Again, we need to check at most h_H nodes in $R_{S_H \rightarrow S_G}$ for each node in R_M, where h_H is the height of $R_{S_H \rightarrow S_G}$. Hence, we get each missing mandatory node in $O(n_M h_H)$, where n_M is the number of nodes in R_M.

Checking both Conditions 4.3 and 4.4 results in a complexity of $O(n_H h_G + n_M h_H)$. The heights of the trees $R_{S_H \rightarrow S_G}$ and R_{S_G} depend on how the host and the guest language are modeled in the structure graph, respectively. However, since we know the model, we also know the maximum height a tree can have in regard to that model. There will be no tree with a height above the maximum height the model defines. Thus, for a specific model, we can consider the maximum height to be constant. This consideration leads to a complexity of $O(n_H + n_M)$ for checking all conditions.

4.5.2 Performance of the Change Detection

To get the set of all added nodes N_a (see Equation (4.9)), we check for each node $r_{S_{rev}} \in R_{S_{rev}}$, if $r_{S_{rev}}$ exists in $R_{S_{rev-1}}$. Again, in a tree structure each node has a unique path. Thus, we need to check at most $h_{S_{rev-1}}$ nodes in $R_{S_{rev-1}}$. Hence, we get each added node in $O(n_{S_{rev}} h_{S_{rev-1}})$, where $n_{S_{rev}}$ is the number of nodes in $R_{S_{rev}}$ and $h_{S_{rev-1}}$ is the height of $R_{S_{rev-1}}$.

To get the set of all removed nodes N_r (see Equation (4.10)), we check for each node $r_{S_{rev-1}} \in R_{S_{rev-1}}$, if $r_{S_{rev-1}}$ exists in $R_{S_{rev}}$. According to the discussion in the preceding paragraph, we get each removed node in $O(n_{S_{rev-1}} h_{S_{rev}})$, where $n_{S_{rev-1}}$ is the number of nodes in $R_{S_{rev-1}}$ and $h_{S_{rev}}$ is the height of $R_{S_{rev}}$.

Additionally, we need to check for each node $r \in N_a$ if r has been renamed or moved. Let's assume that we need constant time to check whether a node has been renamed or moved, then we find all renamed or moved nodes in $O(n_{N_a})$ with n_{N_a} being the number of nodes in N_a.

In summary, the complexity of the change detection approach is $O(n_{S_{rev}} h_{S_{rev-1}} + n_{S_{rev-1}} h_{S_{rev}} + n_{N_a})$. Since, again, the maximum height of a tree is known, we can assume the height to be constant. Thus, we get a complexity of $O(n_{S_{rev}} + n_{S_{rev-1}} + n_{N_a})$ for the detection of changes between two revisions.

4.5.3 Generality of the Approach

Until now, we discussed our approach based on an MLSA consisting of one host and one guest programming language. However, MLSAs can consist of more than two programming languages. Thus, we need to discuss our approach to check the integrity of an MLSA in the context of generalized MLSA setups to show the generality of our approach.

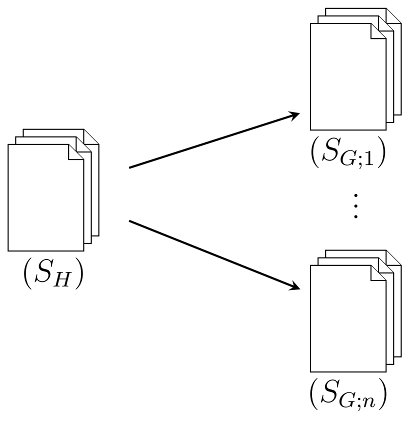
In this section, we only consider the integrity check as presented in Section 4.2 because in an MLSA we may have more than two languages interacting with each other. In contrast, the approach to detect changes in the language structure as presented in Section 4.3 is defined for exactly two source-code revision. Thus, the change detection approach is already defined for the general case.

In the following, we describe three different MLSA setups and how our approach handles these setups (cf. Figure 4.6). To the best of our knowledge, all existing MLSAs constitute special cases of the three MLSA setups shown in Figure 4.6a to 4.6c. Since our approach supports all three MLSA setups, we conclude that our approach supports the refactoring of arbitrary MLSAs.

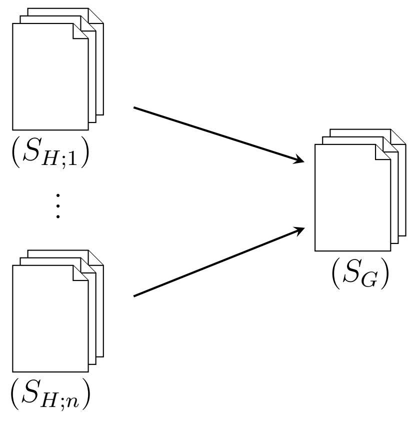
Multiple Guest and Single Host Programming Languages

The first generalization we consider is to assume not only one, but an arbitrary number of guest languages. Then, in the source code written in a single host programming language S_H, different interfaces are used by developers to interact with source code written in multiple guest languages $S_{G;n}$ with n being the nth guest language of the MLSA (cf. Figure 4.6a). Thus, to check the referential integrity of the language interaction, we need to extract and compare the references in the source code written in the host language to the source code written in the nth guest language $R_{S_H \to S_{G;n}}$ and all possible guest language structures for the nth language involved in language interaction $R_{S_{G;n}}$. That is, for guest language n, we compute the set of missing guest language nodes $N_{G;n}$ and the set of missing mandatory nodes $N_{M;n}$ as follows:

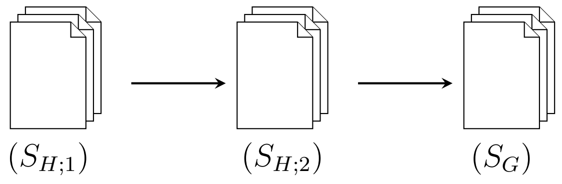
$$N_{G;n} = R_{S_H \to S_{G;n}} \setminus R_{S_{G;n}} \tag{4.13}$$



(a) Single host and multiple guest programming languages.



(b) Multiple host and single guest programming languages.



(c) A host is also a guest programming language.

Figure 4.6: MLSA setups.

and

$$N_{M;n} = R_M \setminus R_{S_H \rightarrow S_{G;n}} \tag{4.14}$$

Single Guest and Multiple Host Programming Languages

The second generalization we consider is to assume not only one, but an arbitrary number of host languages. Then, in the source code written in multiple host programming language $S_{H;n}$, the same elements of the guest programming language are used by developers to interact with source code written in a single guest language S_G with n being the nth host language of the MLSA (cf. Figure 4.6b). Thus, to check the referential integrity of the language interaction, we need to compute and compare the references in the source code written in the nth host language to the source code in the single guest language $R_{S_{H;n} \rightarrow S_G}$ with all possible guest language structures involved in language interaction R_{S_G}. That is, for host language n, we compute the set of missing guest language nodes $N_{G;n}$ and the set of missing mandatory nodes $N_{M;n}$ as follows:

$$N_{G;n} = R_{S_{H;n} \rightarrow S_G} \setminus R_{S_G} \tag{4.15}$$

and

$$N_{M;n} = R_M \setminus R_{S_{H;n} \rightarrow S_G} \tag{4.16}$$

A Host is also a Guest Programming Language

Given source code of three programming languages S_G, $S_{H;1}$, $S_{H;2}$ involved in an MLSA where S_G is accessed by $S_{H;2}$ and $S_{H;2}$ is accessed by $S_{H;1}$ (cf. Figure 4.6c). Hence, this MLSA contains code of two guest programming languages (S_G and $S_{H;2}$) and code of two host programming languages ($S_{H;1}$ and $S_{H;2}$). In other words, we have multiple guest and multiple host programming languages. However, having multiple guest and multiple host programming languages corresponds to a combination of the preceding cases.

Let us index the different languages used in an MLSA by depth d and a language counter n, so that $S_{d;n}$ describes the source-code structure of the nth language at language depth d. That is, $S_{0;n}$ represents the source-code structure of the nth host language and $S_{d;n}$ with $d > 0$ the source-code structure of the nth guest language. Please note that for the language structure $S_{d;n}$ only language structures $S_{d+1;o}$ represent possible guest languages for language interaction. According to this convention, for $S_{H;1}$, $S_{H;2}$, and S_G in Figure 4.6c, we get $S_{0;0}$, $S_{1;0}$, and $S_{2;0}$, respectively. That is, for the general case, we compute the set of missing guest language nodes $N_{G_{d;n;o}}$ and the set of missing mandatory nodes $N_{M_{d;n;o}}$ for the nth host and oth guest language as follows:

$$N_{M_{d;n;o}} = R_{S_{d;n} \rightarrow S_{d+1;o}} \setminus R_{S_{d+1;o}} \tag{4.17}$$

and

$$N_{M_{d;n;o}} = R_M \setminus R_{S_{d;n} \rightarrow S_{d+1;o}} \tag{4.18}$$

4.6 Summary

In this chapter, we introduced structure graphs as a basis to describe source-code elements involved in language interaction. Developers can use structure graphs to represent elements defined in the source code written in the guest language as well as invocations of those elements in the source code written in the host language.

Guest languages do not define a uniform set of elements involved in language interaction. Additionally, we neither can consider all current structure elements, nor foresee all future elements involved in language interaction. Consequently, we cannot use a single type of labeled tree to represent a definition or invocation of an element involved in language interaction. Thus, we need to define specialized types of trees which only represent the structure elements in the guest language that are actually involved in language interaction.

We presented two algorithms based on structure graphs to support developers in their endeavor to refactor MLSAs. The first algorithm checks the referential integrity between the source code written in the host language and the source code written in the guest language. The second algorithm uncovers changes to the structure of the source code written in the guest language, that may lead to broken language interaction. We showed that the algorithms' complexity has a linear dependency on the number of nodes.

Though we used a simplified MLSA setup consisting only of one host and one guest language to introduce our approach to support refactoring MLSAs, we demonstrated that our approach is generally applicable.

5. Implementation and Application of Structure Graphs

Chapter 5 shares material with [Schink, 2013; Schink et al., 2016a].

In Chapter 4, we introduced an approach to support developers who refactor an Multi-Language Software Application (MLSA). The approach consists of two algorithms on graph structures: The first algorithm detects broken language interactions between source code implemented in two programming languages by checking the referential integrity of the source-code elements involved in language interaction. The second algorithm detects changes of the source-code elements involved in language interaction. These changes may help the developer to fix the broken language interaction. In this chapter, we present the prototypical realization of the two algorithms.

We present the prototypical realization of the two algorithms in respect to three different programming paradigms: object-oriented, declarative, and functional. For that purpose, we implemented our approach for the object-oriented programming language Java, the declarative language SQL, and the functional programming language Clojure.

We begin with the presentation of the structure-graph library that implements the two algorithms introduced in Chapter 4. We also describe a general approach to utilize the library for the MLSA at hand. Then, we present two applications of the structure-graph library for database applications implemented in Java and a relational database as well as for Java applications dynamically accessing code written in the programming language Clojure.

5.1 The Structure-Graph Library

The structure-graph library provides a prototypical implementation of a framework for checking language interaction in MLSAs in general. Therefore, the structure-graph

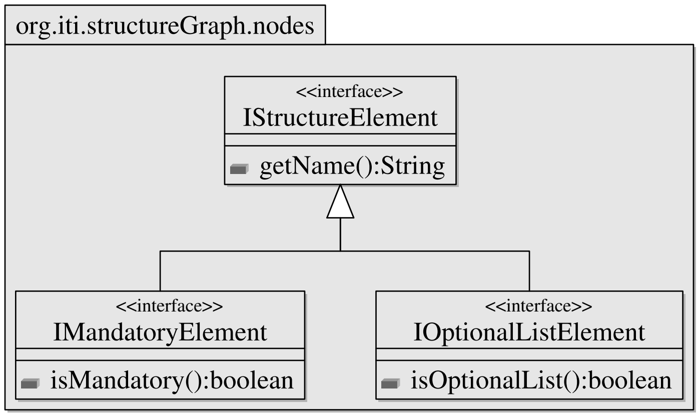


Figure 5.1: Package `org.iti.structureGraph.nodes` with interfaces for implementing structure element representations.

library does not contain any assumptions about language interaction, but focuses on the graph-based representation and comparison of structure elements involved in language interaction. The implementation of the structure-graph library follows the concept presented in Chapter 4.

In this section, we first describe how a developer can utilize the structure-graph library to implement tools for checking language interaction for the MLSA at hand. We, then, describe details of the implementation of the structure-graph library.

5.1.1 Implementation

The purpose of the structure-graph library is to ease the implementation of tools for checking and fixing language interaction. To represent structure elements involved in language interaction, the structure-graph library provides the `IStructureElement` interface. The interface must be implemented by all representations of structure elements. The `IStructureElement` interface extends the `IMandatory` and `IOptionalList` interfaces. These interfaces allow the comparison algorithm to distinguish mandatory nodes and nodes that represent optional parameters. Figure 5.1 presents the Java package `org.iti.structureGraph.nodes` that contains the aforementioned interfaces.

Within the structure-graph library, we use JGraphT[1] for representing structure graphs. By using JGraphT, we are able to re-use certain graph algorithms already implemented in JGraphT like different algorithms to determine the shortest path between two nodes. In particular, we use the class `DirectedGraph` to build graphs of labeled trees representing either $R_{S_{H;n} \to S_G}$ or R_{S_G} and $R_{S_{rev}}$ or $R_{S_{rev1}}$, respectively. A labeled tree itself consists of nodes that are represented by instances of the interface `IStructureElement`. These nodes are connected by edges of JGraphT's type `DefaultEdge`. `DefaultEdge` connects exactly two nodes.

[1]http://jgrapht.org/, visited 15.12.2016

Structures represented with JGraphT are encapsulated with the help of the structure-graph library's class `StructureGraph`. The class `StructureGraph` provides a convenient interface to implement the graph comparison algorithms presented in Chapter 4. For instance, the class `StructureGraph` provides the function `getIdentifiers()` which returns the identifiers for each structural element in the underlying labeled tree. A node's identifier is determined by the concatenation of the node's label and the node's ancestors' labels and the labels of the edges in between. The sets of identifiers is used to determine the difference between two graphs.

The structure-graph library provides three different classes for graph comparison: `SimpleStructureGraphComparer`, `StatementStructureGraphComparer`, and `StructureGraphComparer`. The `StatementStructureGraphComparer` implements the algorithm for checking the referential integrity involved in language interaction as presented in Section 4.2. The `StructureGraphComparer` implements the algorithm to detect changes the structure-elements involved in language interaction as presented in Section 4.3. The `SimpleStructureGraphComparer` provides the `StatementStructureGraphComparer` and `StructureGraphComparer` with the difference between two structure-graphs (added and removed nodes).

SimpleStructureGraphComparer

The class `SimpleStructureGraphComparer` compares two structure graphs with each other. The result of the comparison is an instance of class `StructureGraphComparisonResult` which contains a list of all added and removed nodes. Let us assume we pass two graphs G_S and G_T to an instance of `SimpleStructureGraphComparer`. Then, a node v with its identifier $id(v)$ is *added* if $id(v) \in G_T \wedge id(v) \notin G_S$. Respectively, a node v is *removed* if $id(v) \notin G_T \wedge id(v) \in G_S$.

StatementStructureGraphComparer

The class `StatementStructureGraphComparer` allows developers to check if a source graph G_S that represents a statement referencing a structural element in the source code implemented in the guest language (represented by the target graph G_T) is actually valid. Therefore, the class performs the following modifications on the result of the `SimpleStructureGraphComparer`:

1. Remove all added nodes that are not mandatory or optional list nodes,
2. Remove all added mandatory nodes if the parent of the mandatory node does not exist in the source graph G_S, and
3. Remove all added nodes that share the path with an optional list in the target graph G_T node and do not belong to a mandatory node in the target graph.

If the modifications leave an empty result, then G_S represents a valid invocation to structural elements in G_T. However, if the result is not empty, either one or both of the following problems exist:

1. The result contains removed nodes, thus, G_S references elements that do not exist in G_T.
2. The result contains added nodes, thus, G_S misses nodes that are mandatory for the invocation of structure elements in G_T.

In the first case, the developer either misspelled the structure elements' identifiers or referenced structure elements that either stopped to exist or have not existed at all. Issues of the first kind can be fixed by correcting misspelled structure-element identifiers, removing non-existing structure-element identifiers, or updating the referenced source code. In the second case, the developer forgot to reference structure elements that are mandatory for the invocation of a parent structure element. For instance, an invocation of a function misses the mandatory parameters and, thus, cannot be called. Adding the missing mandatory structure elements or removing the invocation of the parent structure elements with missing mandatory nodes fixes issues of the second kind.

StructureGraphComparer

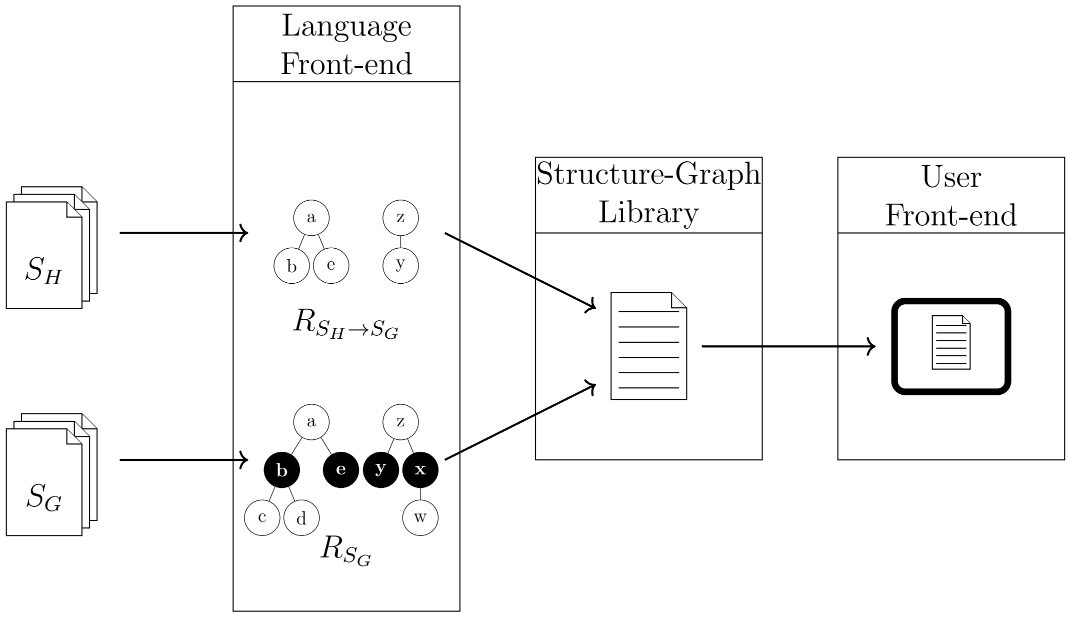
The class `StructureGraphComparer` allows developers to compare two structure graphs G_S and G_T with each other. Like the `StatementStructureGraphComparer`, the class `StructureGraphComparer` takes the result of the `SimpleStructureGraphComparer`, that is a set of added and removed nodes, and tries to detect renamed or moved nodes. We consider a removed node n to be renamed, if we can find a single added node n', so that $path(n) = path(n') \wedge name(n) \neq name(n')$ applies (cf. Section 4.3). Accordingly, we consider a removed node n to be moved, if we can find a single added node n', so that $path(n) \neq path(n') \wedge name(n) = name(n')$ applies (cf. Section 4.3).

5.1.2 Usage of the Framework

Developers need to embed the structure-graph library into another tool that translates language-specific elements into structure graphs and interprets the results generated by the structure-graph library. In the following, we call the part responsible for the translation of language-specific elements *language front-end* and the part responsible for the interpretation of the comparison results *user front-end* (cf. Figure 5.2).

The purpose of the language front-end is to translate the elements present in an application's source code involved in language interaction into structure graphs. For checking the language interaction (cf. Section 4.2), the language front-end must be able to parse source code of at least two languages, a host and a guest language. The framework provides the necessary means to represent structure elements of the parsed languages as structure graphs.

Certain information may need to be generalized to allow the comparison of the structure graphs. For instance, function or method parameters are only present in the source code written in the guest language. In the source code written in the host language, the function or method parameters are replaced by actual values (see the call of method



S_H... Source code written in the host language

S_G... Source code written in the guest language

$R_{S_H \to S_G}$... References from S_H to S_H

R_{S_G}... Structure elements in S_G involved in language interaction

Figure 5.2: General implementation approach for tools with the structure-graph library.

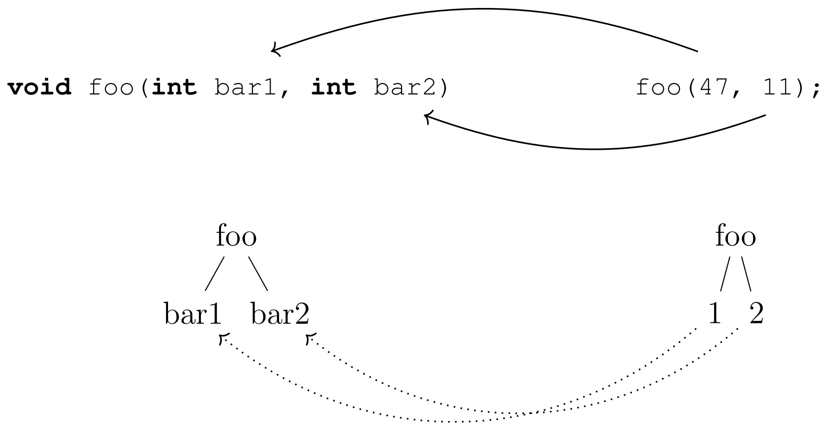


Figure 5.3: Method definition and invocation, and their respective structure graphs.

`foo` in Figure 5.3). Accordingly, in the source code written in the host language only the order of the parameters is known, and, thus, also the structure graph for the function or method call can contain only the order of the parameters (see the structure graph for the call of method `foo` in Figure 5.3). Hence, in order to ensure that we can compare the structure graph when building the structure graph for the source code written in the guest language, we may not use the parameter names like in the structure graph for the method definition of method `foo`, but the order to identify a function's or method's parameters like in the structure graph of the invocation of method `foo`.

Since the developer creates customized classes for representing the source-code elements involved in language interactions, these customized classes can be augmented by additional information useful for the interpretation of results. For instance, developers can store position information to identify elements in the source code. This information may be useful when presenting the results in the user front-end.

The user front-end translates the results of the integrity check (cf. Section 4.2) or the detection of structural modifications (cf. Section 4.3) into language-specific results. For instance, whereas the integrity check and the detection of the structural modifications are based on labeled graphs with nodes and edges, programming languages provide different elements like functions and parameters. Therefore, in the user front-end developers take the results in the instances of the class `StructureGraphComparisonResult` returned by the graph comparer classes (see Section 5.1.1) and map the nodes listed in the results to actual source-code elements that can be recognized by the end user. For instance, developers can use position information stored in the classes which represent structural elements to highlight affected source-code elements in an Integrated Development Environment (IDE).

5.2 The Sql-Schema-Comparer Library

Based on the structure-graph library, We implemented two tools which assist developers with refactoring MLSAs. In this section, we present the first of these tools: The *sql-*

schema-comparer, is a software library to compare and check SQL schemes.[2] The library allows to compare:

1. two Structured Query Language (SQL) schemata with each other, or
2. an SQL statement in respect to an SQL schema

with each other. The library provides support for `SELECT` statements and schemata of SQLite[3] and H2[4] databases and elementary support for Java Persistence API (JPA) annotations in Java files.

According to Section 5.1.2, the sql-schema-comparer uses the structure-graph library to check the integrity of SQL statements and JPA entity definitions in respect to a given database schema and to detect changes between two given database schemata. Therefore, the sql-schema-comparer implements language front-ends for SQLite and H2 database schemata, for SQL's `SELECT` statement, and for JPA entity definitions. The sql-schema-comparer also extends the structure-graph library's comparison algorithms by providing SQL-specific comparisons based on the results of the structure-graph comparison.

In this section, we describe the implementation of the sql-schema-comparer and how we realized SQL-specific checks on the existing graph representation. We also describe the integration of the library into Eclipse.

5.2.1 Implementation Details

The language front-end provided by the sql-schema-comparer library translates database schemata and SQL statements into a structure-graph representation by implementing the `IStructureElement` interface and extending JGraphT's `DefaultEdge` class for the different elements of a database schema. Figure 5.4 presents a generalized representation of a structure graph with the possible node and edge types as defined in the sql-schema-comparer. In the sql-schema-comparer library, edge types are defined in package `org.iti.sqlSchemaComparison.edges` (cf. Figure A.4) and nodes types are defined in package `org.iti.sqlSchemaComparison.vertex` (cf. Figure A.5). We will omit the package identifier in the following description. The root of each tree (tab_1 in Figure 5.4) is represented by instances of class `SqlTableVertex`. Edges of type `TableHasColumn` ($e_{Tab2Col}$ in Figure 5.4) connect the root node with one or more nodes representing columns (col_1 to col_n in Figure 5.4) of type `SqlColumnVertex`. Each column node is connected with one ore more nodes representing the column type (typ_1 in Figure 5.4) or the column constraints (con_1 to con_n in Figure 5.4) like `NOT NULL`, `DEFAULT`, and `PRIMARY KEY` constraints. Column type nodes are represented by instances of type `ColumnTypeVertex` and column constraint nodes are represented by instances of type `ColumnConstraintVertex`. Edges of type `ColumnHasType` (represented

[2] https://github.com/hschink/sql-schema-comparer
[3] https://www.sqlite.org/
[4] http://www.h2database.com/

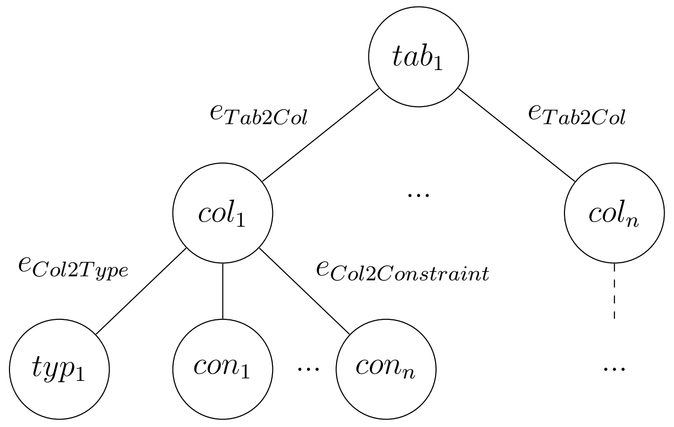


Figure 5.4: Generalized representation of a table schema as structure graph.

by $e_{Col2Type}$ in Figure 5.4) and `ColumnHasConstraint` (represented by $e_{Col2Constraint}$ in Figure 5.4) connect column nodes with column type and column constraint nodes, respectively.

Besides tables, columns, column types, and column constraints, the sql-schema-comparer considers foreign-key relationships defined in a relational schema. For that purpose, instances of the class `ForeignKeyRelationEdge` contain a reference to the table that defines the referenced primary key. Figure 5.5 shows an example structure graph that contains two table nodes tab_1 and tab_2 of type `SqlTableVertex`. Each table node references two columns: Columns col_1 and col_{12} from table tab_1 and columns col_{21} and col_2 from table tab_2 of type `SqlColumnVertex`. Each column references a single type node: typ_1, typ_{12}, typ_{21}, and typ_2, respectively. The type nodes are instances of class `ColumnTypeVertex`. Table and column nodes are connected by edges labeled as $e_{Tab2Col}$ of type `TableHasColumnEdge`. Column and column type nodes are connected via edges labeled $e_{Col2Typ}$ of type `ColumnHasType`. Additionally, the structure graph contains an edge labeled e_{FK} of type `ForeignKeyRelationEdge` with an outgoing path and two ingoing paths. This edge represents a foreign-key relationship between the columns col_{12} and col_{21}. As stated above, a foreign-key relationship also contains information about the foreign-key table. In Figure 5.5, the table involved in the foreign-key relationship is tab_2.

The sql-schema-comparer library is able to create structure graphs of H2 and SQLite database schemata, SQL `SELECT` statements, and JPA entity source files. JPA entity source files are regular Java source files in which the defined Java classes are annotated with `@Entity` from JPA's namespace `javax.persistence`. All the classes in the sql-schema-comparer that implement front-ends, that is `H2SchemaFrontend`, `SqliteSchemaFrontend`, `JPASchemaFrontend`, and `SqlStatementFrontend`, implement the `ISqlSchemaFrontend` interface (see Figure A.6). Table A.1 lists all front-end implementations available in the sql-schema-comparer library.

For the language front-ends for database schemata and JPA entities a type deduction is possible by analyzing the database schema or the return types of the methods in

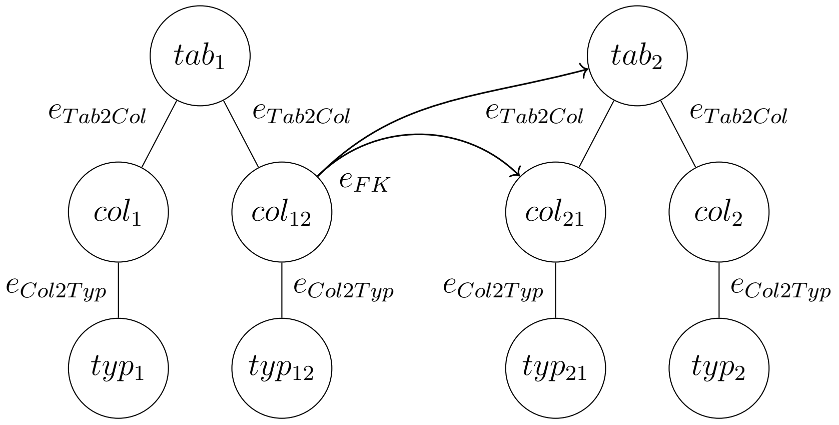


Figure 5.5: Database schema with a foreign-key relationship.

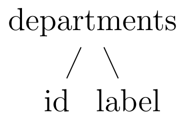


Figure 5.6: Structure graph for JPA entity and `SELECT` statement in Listing 5.1 and Listing 5.2, respectively.

the JPA entity definition, respectively. In contrast, for SQL `SELECT` statements, a type interference is not possible since the statement does not contain any type information.[5] However, it is possible to pass the constructor of the class `SqlStatementFrontend` a structure graph representing a database schema. The structure graph is then used to augment the schema graph of the `SELECT` statement with type information.

Though we use different programming and domain-specific languages to define JPA entities and SQL `SELECT` statements (cf. Listing 5.1 and Listing 5.2), we can understand both, entities and statements, as SQL queries. Thus, JPA entity definitions and `SELECT` statements create the same type of structure graph. For example, let us consider the definition of the JPA entity `Department` in Listing 5.1 and the `SELECT` statement on table `departments` in Listing 5.2. Both definitions query the table `departments` (see Line 2 on the JPA entity definition) with the columns `id` and `label`. Consequently, the language front-ends for JPA and `SELECT` create the same structure graph as presented in Figure 5.6. Thus, we can use the same structure-graph representation for SQL statements and JPA entities.

The sql-schema-comparer supports the check of referential integrity between SQL `SELECT` statements and a database schema and between JPA entities and a database schema, respectively (cf. Section 4.2), as well as the detection of changes between two versions of a database schema (cf. Section 4.3). The check of referential integrity is implemented in class `SqlStatementExpectationValidator` (see Figure A.7). The con-

[5]If a number of SQL statements are available, the type interference approach described in [Weisheng, 2004] may be applicable to deduce the types of columns though.

```java
@Entity
@Table(name="departments")
public class Department implements Serializable {

    private int id;
    private String label;

    public void setId(int id) { this.id = id; }

    @Id
    public int getId() { return id; }

    public void setLabel(String label) { this.label = label; }

    public String getLabel() { return label; }
}
```

Listing 5.1: JPA entity `Department`.

```sql
SELECT id, label FROM departments;
```

Listing 5.2: Select statement on table `departments`.

```
SELECT col_11, col_12 FROM tab_1, tab_4;
```

Listing 5.3: Select statement including missing tables and columns.

```
SELECT col_x FROM tab_1;
```

Listing 5.4: Select statement including a missing but reachable column.

structor of class `SqlStatementExpectationValidator` takes one argument: The structure graph of the database schema against which the `SqlStatementExpectationValidator` checks the integrity of SQL `SELECT` statements and JPA entities. To perform the actual integrity check, we call the method `computeGraphMatching` of class `SqlStatementExpectationValidator` with the structure graph of an SQL `SELECT` statement or JPA entity as parameter. Internally, the `SqlStatementExpectationValidator` uses the structure-graph library's `StatementStructureGraphComparer` class to perform the comparison. The method `computeGraphMatching` returns a result of type `SqlStatementExpectationValidationResult` that contains a list of all table and column nodes that are present in the SQL `SELECT` statement or JPA entity, respectively, but are missing in the database schema.

For instance, let us assume that we want to check the integrity of the SQL statement in Listing 5.3 against the structure graph of a database schema in Figure 5.7. The integrity check detects one missing table and one missing column, because neither col_{12} nor tab_4 are defined in the database schema. Additionally, if a column is moved from a source table to a target table and source and target table are connected by a foreign key reference, this column is marked as missing but reachable. For instance, in Figure 5.7, in the initial database schema column col_x is part of table tab_1, but, in the current state, column col_x is part of table tab_2. However, the SQL statement in Listing 5.4 is still based on the initial state. Thus, the integrity check would detect column col_x as missing. Nevertheless, in the current state, table tab_2 is connected to table tab_1 by a foreign-key relationship. Hence, column col_x is still reachable from table tab_1 by joining table tab_2. For the purpose of detecting missing but reachable columns, the integrity check uses the class `SqlColumnReachabilityChecker` (cf. Figure A.9). The `SqlColumnReachabilityChecker` uses Dijkstra's algorithm as implemented in JGraphT[6] to check the reachability of a column node. The developer can use the results in the `SqlStatementExpectationValidationResult` to highlight erroneous `SELECT` statements and JPA entities.

[6]http://jgrapht.org/javadoc/org/jgrapht/alg/DijkstraShortestPath.html, visited 15.12.2016

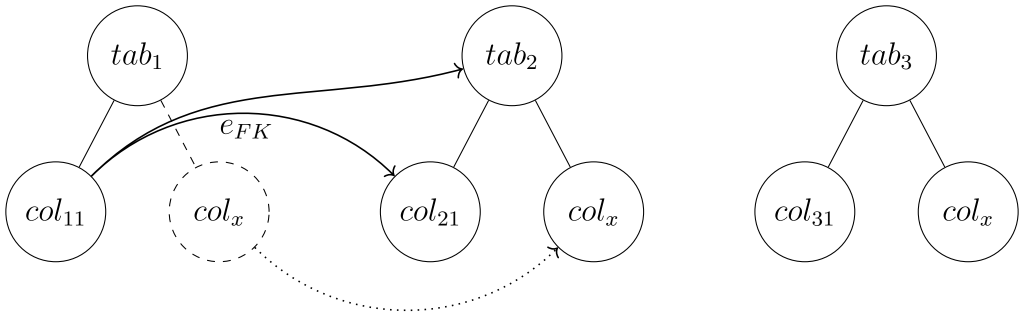


Figure 5.7: Resolving moved nodes by foreign-key relationships.

The sql-schema-comparer supports the detection of changes to the language structure as described in Section 4.3. That is, the sql-schema-comparer can detect changes between two revisions of a database schema. The detection of database schema changes is implemented in class `SqlSchemaComparer` (cf. Figure A.7). The implementation is able to detect the following database schema modifications (cf. Figure A.8):

- Table
  - Create
  - Delete
  - Rename
- Column Type
  - Create
  - Delete
  - Change
- Column
  - Create
  - Delete
  - Rename
  - Move
- Column Constraint
  - Create
  - Delete

All database schema modifications refer directly to the node modifications described in Section 4.3. That is, *create* modifications refer to added nodes, *delete* modifications refer to removed nodes, *rename* modifications refer to renamed nodes, and *move* modifications refer to moved nodes. Thus, for instance, the sql-schema-comparer refers to an added table node as create table modification. However, for the column type one exception exists: The *change* column type modification refers to a renamed column type node. In a structure graph for a database schema, a column does always have a type node. The label of the type node is the name of the type the node presents. Thus, a change of a column type is reflected as a change of the respective type node's label. However, we consider the change of a node label to be a renaming.

5.2.2 Application of the Sql-Schema-Comparer Library

The sql-schema-comparer library allows developers to compare two SQL schemata with each other or an SQL statement or JPA entity definition with an SQL schema, re-

```
1 ISqlSchemaFrontend frontend
2     = new SqliteSchemaFrontend(SQLITE_FILE);
3 Graph<ISqlElement, DefaultEdge> schema
4     = frontend.createSqlSchema();
```

Listing 5.5: Create a schema instance of an SQLite file

```
1 SqlSchemaComparer comparer
2     = new SqlSchemaComparer(oldSchema, currentSchema);
3 SqlSchemaComparisonResult result = comparer.comparisonResult;
```

Listing 5.6: Compare two schema instances

spectively. In this section we describe the library's application within a third-party application. In this section, we first describe how developers can integrate the sql-schema-comparer library in their development tools. Then, we describe a prototypical integration of the sql-schema-comparer library into the Eclipse IDE.

Integration of the Sql-Schema-Comparer Library

In the following, we describe how a developer can utilize the Application Programming Interface (API) of the sql-schema-comparer library (1) to detect differences between two SQL schemata and (2) to check the referential integrity between an SQL statement or JPA entity and an SQL schema.

For schema comparison, we need to create a structure graph representation for each schema. Structure graphs are instances of class `DirectedGraph` (JGraphT) with the node type `IStructureElement` and edge type `DefaultEdge` (JGraphT). We may create instances manually or by parsing SQLite or H2 files. The latter is described in the following based on the SQLite front-end. The schema creation for the H2 front-end works accordingly.

First of all, we need to create an instance of the SQLite front-end as described in Listing 5.5, Line 2. The method `createSqlSchema` of the front-end returns the structure graph instance for the SQLite file (see Listing 5.5, Line 4).

The schema comparison is implemented in class `SqlSchemaComparer`. The class takes two structure-graph instances on construction (see Listing 5.6, Line 2). The field `comparisonResult` of an `SqlSchemaComparer` instance contains the match result. The match result contains the affected tables, columns, column types, and column constraints.

In the following, we describes the comparison of an SQL statement with a database schema. However, the comparison of a JPA entity with a database schema works accordingly. Just use the class `JPASchemaFrontend` instead of the class `SQLStatementFrontend`.

```
1 ISqlSchemaFrontend frontend
2     = new SqlStatementFrontend(STATEMENT, null);
3 Graph<ISqlElement, DefaultEdge> statementSchema
4     = frontend.createSqlSchema();
```

Listing 5.7: Create a schema instance of an SQL statement

```
1 SqlStatementExpectationValidator validator
2     = new SqlStatementExpectationValidator(databaseSchema);
3 SqlStatementExpectationValidationResult result
4     = validator.computeGraphMatching(statementSchema);
```

Listing 5.8: Compare a database and a statement schema

To compare an SQL schema with an SQL statement, we need to extract a structure graph for the database schema (cf. Listing 5.5) as well as for the SQL statement. Latter is possible with class `SqlStatementFrontend` as shown in Listing 5.7, Line 2. The front-end takes at least an SQL statement. Additionally, it is possible to pass a database schema, for augmenting the columns referenced in the SQL statement with type and constraint information.

The database and statement comparison is implemented in class `SqlStatementExpectationValidator` (see Listing 5.8, Line 2). The class takes a structure-graph instance on construction. Calling the method `computeGraphMatching` with the structure graph of the SQL statement's schema returns a matching result (see Listing 5.8, Line 4). The result contains missing tables or columns and columns that appear to be moved to different tables.

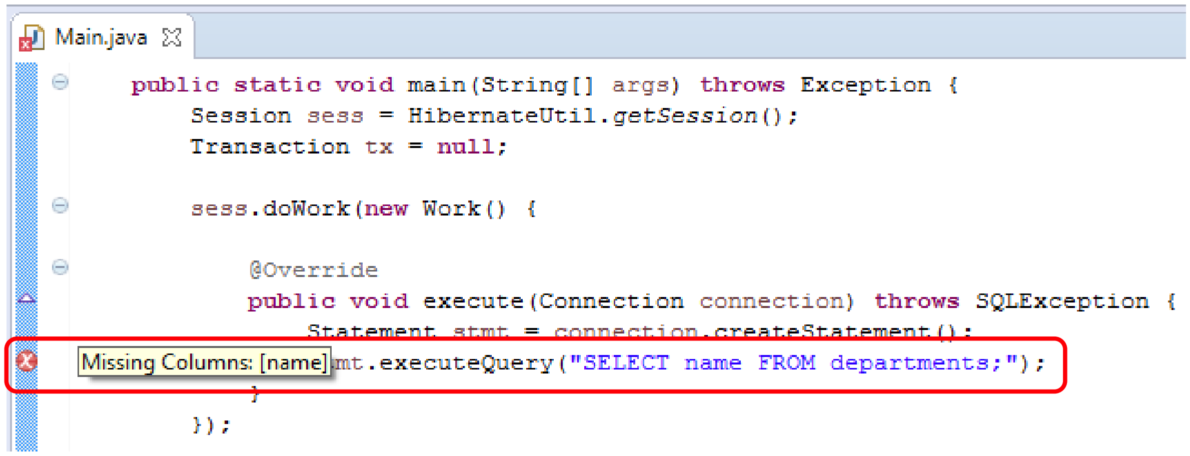
The Sql-Schema-Comparer Eclipse Plug-in

Since the usage of IDEs is common for software development in Java [Murphy et al., 2006], developers would benefit from an integration of the sql-schema-comparer library into a Java IDE, because it allows developers to check the integrity of their MLSA within their usual development environment.

We implemented the sql-schema-comparer Eclipse Plug-in[7] to integrate the sql-schema-comparer into the Eclipse IDE[8]. The Eclipse Plug-in provides an interface for the integrity check between a relational schema and an SQL statement and for the comparison of the two most recent revisions of a relational schema. In the following, we describe how developers can access both functionalities with the sql-schema-comparer Eclipse Plug-in.

The sql-schema-comparer Eclipse Plug-in checks the integrity of an MLSA on each build of a Java project. If the plug-in detects a violation of the integrity, it marks

[7]https://github.com/hschink/sql-schema-comparer-eclipse-plugin
[8]http://www.eclipse.org/



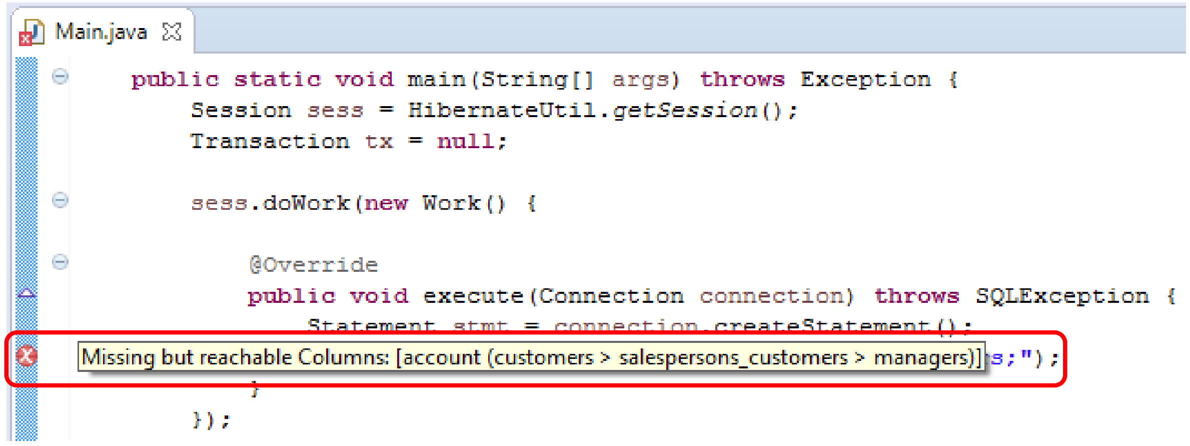
```
    public static void main(String[] args) throws Exception {
        Session sess = HibernateUtil.getSession();
        Transaction tx = null;

        sess.doWork(new Work() {

            @Override
            public void execute(Connection connection) throws SQLException {
                Statement stmt = connection.createStatement();
Missing Columns: [name]mt.executeQuery("SELECT name FROM departments;");
            }
        });
```

Figure 5.8: Missing column `name` for SQL statement **SELECT** name **FROM** departments.

the affected positions in the source code. For instance, in Figure 5.8, the SQL statement references a column `name`. However, since a column with this identifier does not exist in table `departments`, the plug-in marks the statement as incorrect. The sql-schema-comparer Eclipse Plug-in also supports the integrity check for JPA entities. For instance, in Figure 5.10, the Eclipse plug-in marks the method `getName`[9] of entity `Department` as incorrect to make the developer aware of the missing column in the database schema. Additionally, the plug-in detects missing but reachable columns as described for the sql-schema-comparer in Section 5.2.1: In Figure 5.9, the query `SELECT account FROM customers` references the column `account` which does not exist on table `customers`. However, the column `account` exists on table `managers` that is connected to table `customers` via table `salespersons_customers`. Accordingly, the plug-in presents a possible `JOIN` path which the developer can use to access the column `account`.

Besides the integrity check of SQL statements and JPA entities, the sql-schema-comparer Eclipse Plug-in provides information about the last schema changes applied to a project's database. For that purpose, on the first build of a Java project, the sql-schema-comparer Plug-in retrieves and saves the database schema from the Java project's database. On the subsequent builds of the Java project, the plug-in compares the current database schema with the saved database schema. If the current schema is different to the saved one, the plug-in replaces the saved one by the current one. Additionally, the plug-in displays a list of changes retrieved by the comparison of the saved and the current database schema in a separate Eclipse view. Figure 5.11 presents the *Schema Changes* View which contains a list of modifications for each of the three subsequent builds of the Java project. On the first subsequent build, the plug-in detected a change of the column `title` and the column `title`'s default value in table `departments`. On the second subsequent build, a change to the `title` column's type

[9]Actually, both methods, that is getter and setter, are incorrect. However, we assume that the developer is aware of the fact that both methods need to be renamed to preserve the mapping to column `name`. It is possible to mark both methods, though.



```
    public static void main(String[] args) throws Exception {
        Session sess = HibernateUtil.getSession();
        Transaction tx = null;

        sess.doWork(new Work() {

            @Override
            public void execute(Connection connection) throws SQLException {
                Statement stmt = connection.createStatement();
 Missing but reachable Columns: [account (customers > salespersons_customers > managers)]s;");
            }
        });
```

Figure 5.9: Missing but reachable column `account` for SQL statement **SELECT** account **FROM** customers.

was detected in table `departments`. And, on the last subsequent build, an addition of the `NOT NULL` constraint to the `title` column was detected in table `departments`.

Because of the sql-schema-comparer Eclipse plug-in being a prototype, a number of limitations exist. First of all, the plug-in does only support SQLite and H2 databases for now. Second, the database must be part of the Java project in Eclipse. If several SQLite or H2 database files exist, the plug-in will only consider the first detected database file. Finally, the plug-in does not save the changes to a database schema nor the most recent database schema on the disk. Thus, after Eclipse is closed, the SQL schema comparison will be lost. A solution exists for all of the above mentioned limitations. However, the current prototype shows how developers can integrate the sql-schema-comparer in their daily work and, thus, is sufficient for our purpose at its current state.

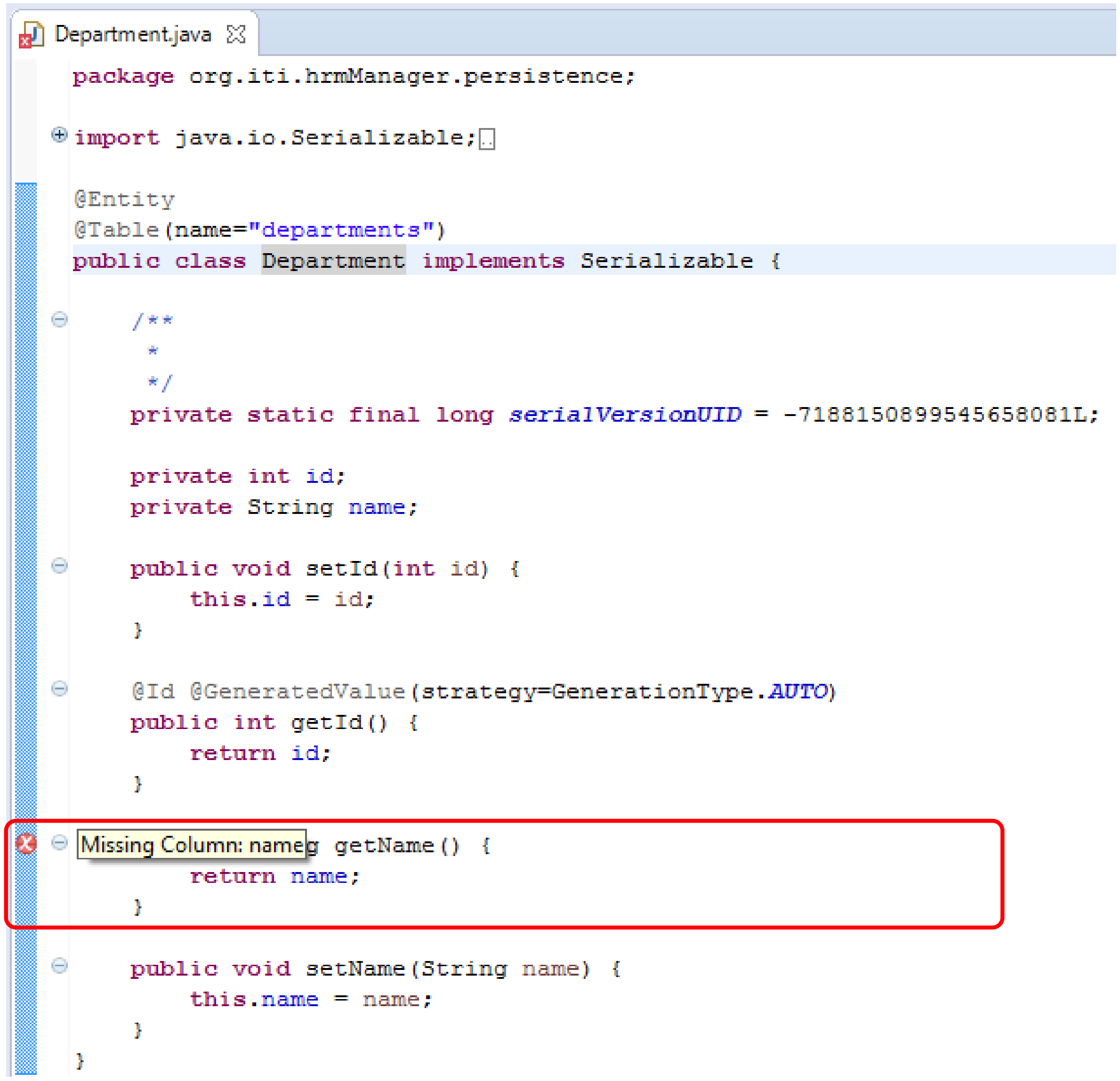
5.3 The Clojure-Java-Interface-Checker Library

The sql-schema-comparer library checks the interaction between a relational database and an application implemented in Java. However, Java not only offers means to interact with relational databases, but with source code of a number of other programming languages like C/C++, JavaScript, and Clojure. We implemented the *clojure-java-interface-checker*[10] to show that our structure-graph based approach is also applicable to interactions of other languages than Java and SQL. In the following, we introduce the programming language Clojure and describe details of the implementation of the clojure-java-interface-checker.

5.3.1 The Clojure Programming Language

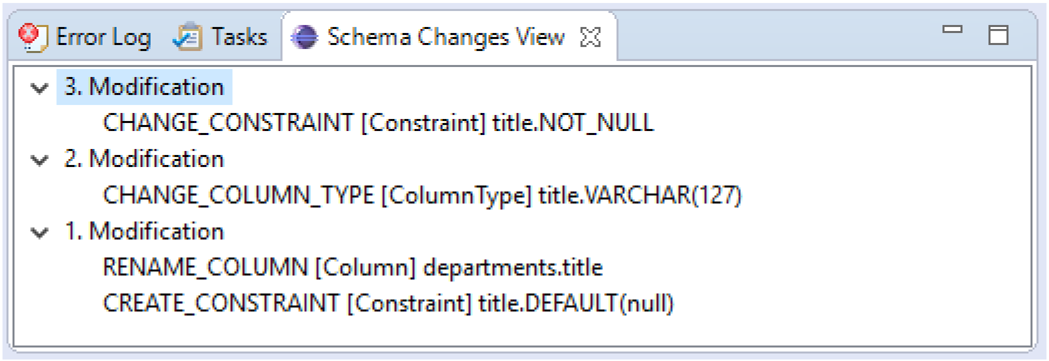
Clojure is a functional programming language that runs on the *Java Virtual Machine (JVM)*. Syntactically, Clojure is a Lisp dialect. Since Clojure runs on the JVM, devel-

[10] https://github.com/hschink/clojure-java-interface-checker



```
package org.iti.hrmManager.persistence;

import java.io.Serializable;

@Entity
@Table(name="departments")
public class Department implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = -7188150899545658081L;

    private int id;
    private String name;

    public void setId(int id) {
        this.id = id;
    }

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    public int getId() {
        return id;
    }

    Missing Column: nameg getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Figure 5.10: Missing column name on table departments.



```
3. Modification
    CHANGE_CONSTRAINT [Constraint] title.NOT_NULL
2. Modification
    CHANGE_COLUMN_TYPE [ColumnType] title.VARCHAR(127)
1. Modification
    RENAME_COLUMN [Column] departments.title
    CREATE_CONSTRAINT [Constraint] title.DEFAULT(null)
```

Figure 5.11: Three change sets applied to a database schema.

opers can directly use libraries available for the programming language Java. Additionally, compiled Clojure source code can be directly called from the Java programming language.

Since Clojure and Java share the same platform, developers do not need additional means to call Clojure source code from Java and vice versa. However, calling Clojure functions in Java without additional means works only for compiled Clojure source code. However, additionally, the Clojure library provides means to dynamically load Clojure source code directly from Java source code.

For dynamically calling Clojure functions from Java source code, developers must provide the namespace and the name of the function to be called. For instance, to call the function `add2` in namespace `i.o.c.Test` (see Listing 5.9), developers use the class `RT`[11] shown in Listing 5.10.

```
(ns o.i.c.Test)

(defn add2 [x]
  (+ x 2))
```

Listing 5.9: Definition of a namespace and a function in Clojure.

```
Var f = RT.var("o.i.c.Test", "add2");

f.invoke(2);
```

Listing 5.10: Invocation of Clojure function in Java.

5.3.2 Implementation Details

Unlike the sql-schema-comparer library, the clojure-java-interface-checker's only purpose is to show the applicability of the structure-graph approach on a different set of programming paradigms. Thus, the clojure-java-interface-checker does not provide any interfaces for the integration into third-party tools. Therefore, we do not describe how to integrate the clojure-java-interface checker into third-party tools.

The clojure-java-interface-checker is a Java library that checks the dynamic invocation of Clojure functions in Java source code (see Section 5.3.1). Therefor, the library creates a structure graph for the function invocations defined in the Java source code and for the actual functions defined in the Clojure source code. The structure graph contains a tree for each namespace defined in the Clojure source code. Each namespace has a child node for each function defined in that namespace. Additionally, each node

[11]Since version 1.6, the preferred way of calling a Clojure function is to use class `Clojure` that returns an instance of class `IFn`. Our implementation is based on version 1.5. Nevertheless, the basic principle has not changed.

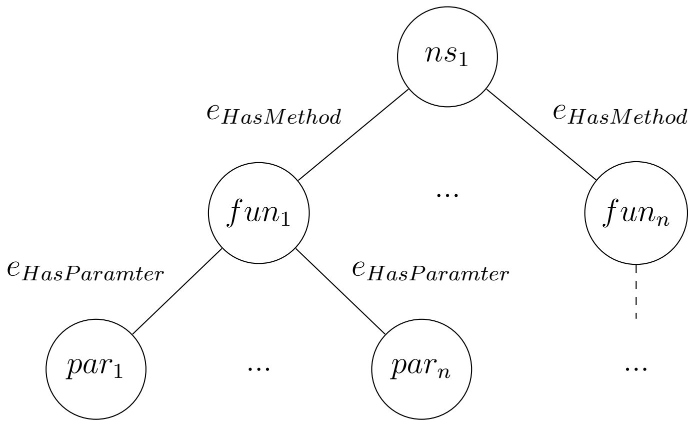


Figure 5.12: Generalized representation of Clojure namespaces as structure graph.

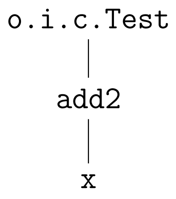
representing a function has a child node for each function parameter. Having a node for each parameter allows to check that the function invocation in the Java source code contains the correct number of parameters. Figure 5.12 shows the generalized structure of a tree for a Clojure namespace definition.

In an SQL statement, the statement specifies all tables and columns that the statement accesses. Thus, when we extract the structure graph from an SQL statement, we get the identifiers as defined in the relational schema of the respective database. However, in contrast to SQL statements, function calls only specify the function name, but not the parameter name.[12] Thus, the argument's position is the only information we can extract from the function invocation. For instance, for the function definition in Listing 5.9 and function invocation in Listing 5.10, the library creates two different trees (see Figure 5.13): The only difference between the two trees is that, in Java, we have no information about the called parameter but its position. Therefore, in Figure 5.13b the parameter `x` is represented by the parameter's position `0`. Thus, accordingly, before we can compare two structure graphs representing a function declaration and a function invocation, we need to replace the parameter names in the structure graph of the function declaration Figure 5.13a by their position in the function definition. This allows us to compare the structure graph of the function declaration with the structure graph of the function call without the information of the actual parameter name.

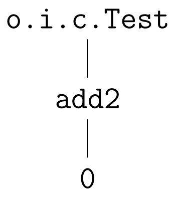
5.4 Summary

In this chapter, we presented three libraries and an Eclipse plug-in which check the integrity between the source code of two languages and detect changes between two revisions of the source code of one language. In particular, we first presented the structure-graph library which provides all the necessary data structures and algorithms

[12]Some languages like Clojure and Groovy support keyword arguments, that is, the ability to specify a parameter regardless of its position. For brevity and generality, we ignore keyword arguments in the discussion.



(a) Graph representing the function in Listing 5.9.



(b) Graph representing the invocation in Listing 5.10.

Figure 5.13: Function graphs.

to create language front-ends and to compare structure graphs created in the language front-ends with each other. Then, we introduced the sql-schema-comparer library as the first implementation of a language front-end which allows to check the integrity between a database application implemented in the programming language Java and a relational database. Furthermore, we showed the library's ability to compare two revisions of a relational schema with each other. We also presented the sql-schema-comparer Eclipse plug-in which integrates the sql-schema-comparer library's functionality into the Eclipse IDE. Finally, we introduced the clojure-java-interface-checker library which allows developers to check the integrity between source code written in Clojure and dynamic invocations of the Clojure source code in a Java application. The clojure-java-interface-checker also confirms the applicability of the structure-graph approach for different language interactions.

With the sql-schema-comparer, we show two aspects: (1) The adaption of the structure-graph library to the interaction of Java and SQL and (2) the augmentation of the structure-graph library's functionality by language-specific features. The second point is illustrated by the sql-schema-comparer's ability to use foreign-key relationships to detect possibly moved columns. The sql-schema-comparer Eclipse plug-in highlights the feasibility to integrate our approach into state-of-the-art IDEs. With the clojure-java-interface-verifier, however, we show that we can apply the structure-graph library to different interacting languages and language paradigms.

The structure-graph library provides and implements all the necessary data structures and algorithms to build and compare structure graphs. However, the developer is still required to implement the language front-ends for the interacting programming languages at hand. In particular, the developer needs to parse the source code of the languages and create suitable structure graphs. Additionally, the developer needs to implement a user front-end which interprets the node changes detected by the structure-graph library. A possible approach to ease the implementation for language-specific tools is to automatically create parsers for the language front-ends that process the interacting languages at hand with the help of a parser generator and a syntax definition. Also the mapping of the source-code elements to structure-graph nodes could be automated by a mapping description. Thus, eventually, the developer needs to provide an interpretation of the results of the structure-graph library for the user front-end. However, the last may not be necessary, if special cases like the detection of moved columns in the sql-schema-comparer (see Section 5.2.1 on Section 5.2.1) do not exist for the interaction of the programming languages at hand.

The structure-graph approach as presented in Chapter 4 describes a generalized approach to support refactoring in MLSAs. Though, developers still need to implement language-specific front-ends to use the structure-graph approach as implemented by the structure-graph library for the interacting languages at hand. However, when we introduced the structure-graph approach, we outlined that we do not think a generally applicable approach to automated Multi-Language Refactoring (MLR) exists because of the number of different structure-elements and refactorings. Thus, as stated above, we do not question that approaches exist to automate the language-specifics to a certain extend, but we do not assume that we can find an approach completely independent of any language specifics. Following this assumption, the structure-graph library only implements a limited set of features and is dependent on language-specific extensions to be useful for the language interaction at hand.

6. Evaluation

Chapter 6 shares material with [Schink et al., 2016b].

In this chapter, we present an evaluation of the sql-schema-comparer library and its Eclipse plug-in which we described in Chapter 5. The sql-schema-comparer library supports the refactoring of database applications that are implemented in the Java programming language and access a relational database. Specifically, the sql-schema-comparer detects mismatches between the schema of a relational database and the schema expected by the Java code that accesses the relational database. We used Eclipse's plug-in infrastructure to integrate the sql-schema-comparer into the Eclipse Integrated Development Environment (IDE) and called the result the sql-schema-comparer Eclipse plug-in. The plug-in allows developers to detect mismatches at compile time. To investigate if the sql-schema-comparer can improve the productivity of developers who refactor a database application, we conducted a controlled experiment.

The goal of our experiment was to evaluate if participants supported by the sql-schema-comparer Eclipse plug-in achieve a significant higher development productivity than without the plug-in's support. Additionally, in our experiment we considered development experience as a cofounding parameter to distinguish between improvements of the development productivity induced by the tool and by the participants' programmming experience. To the best of our knowledge, no other experiment for evaluating refactoring tools in an Multi-Language Software Application (MLSA) setup exists. Thus, we devised a novel experimental design for the evaluation of the sql-schema-comparer library and its Eclipse plug-in. The experiment consists of two refactorings and a control task on two open-source projects. The open-source projects consists of several thousand lines of code. With the help of this experimental setup, we gathered data of 79 undergraduate and graduate students.

This chapter is divided in six parts. In the first part in Section 6.1, we describe the experimental design in detail. In the second part in Section 6.2, we report on the execution of the experiment. We analyze the results of the experiment in the third part in Section 6.3 and interpret the results in Section 6.4. We discuss threats to validity in Section 6.5 before we summarize this chapter in the sixth section.

6.1 Experimental Design

We designed the experiment according to the guidelines proposed by [Wohlin et al., 2012]. In the following, we describe the details of the experimental design. First, we describe which hypotheses we want to falsify and the material on which we built the experiment. Then, we describe the participants who took part in the experiment and the tasks the participants had to work on. Finally, we describe the tools with which we gathered the experimental data.

6.1.1 Hypotheses, and Variables

We expect that developers supported by the sql-schema-comparer Eclipse plug-in adapt Java source code to a refactored database schema more productively than developers without its support. Additionally, we hypothesize that our tool increases productivity independently of the developers' programming experience. In other words, we assume that inexperienced users who use the sql-schema-comparer Eclipse plug-in are more productive than inexperienced user who do not use the tool and that experienced user with tool support are as well more productive with the Eclipse plug-in than without.

We consider one measure of productivity[1]: *Development time* (μ_T). Development time describes the time a developer needs to successfully adapt Java source code to a refactored database schema. We assume that development time is lower for developers having tool support than for developers without tool support.

We can summarize our hypotheses as follows:

H_0**:** $\mu_{T_{with\ plug\text{-}in}} \geq \mu_{T_{without\ plug\text{-}in}}$
H_1**:** $\mu_{T_{with\ plug\text{-}in}} < \mu_{T_{without\ plug\text{-}in}}$

That is, we expect that the development time μ_T with the sql-schema-comparer Eclipse plug-in is less than the development time without the plug-in (cf. H_1). If we cannot reject the null hypothesis H_0, the experiment provides no evidence for our expectation and, as long as we cannot prove otherwise, we have to assume that the μ_T with the tool is either equal or larger to the development time without the tool.

We consider programming experience as a major confounding parameter. That is, we assume that programming experience influences the development time μ_T. Assuming

[1]Initially, we conducted the experiment with two measures of productivity. Since unit testing is an important part of software development, we considered the number of unit-test runs as the second measure of productivity. The assumption was that with tool support developers are able to decrease the number of unit-test runs and, thus, the amount of time to fix broken language interaction in an MLSA. However, based on the unit-test runs, we have not been able to draw reasonable conclusions.

that we would not consider programming experience as a confounding parameter, we would not be able to distinguish whether an improvement in productivity is induced by the tool or by the programming experience of the participant. To control the influence of programming experience, we measure it based on a questionnaire [Feigenspan et al., 2012] (see Table A.2 for the questions defined in that questionnaire) and analyze its effect on the results.

6.1.2 Material

We selected the two open-source applications *Apache Syncope*[2] and *AppFuse*[3] as objects for our experiment. Apache Syncope is a web application for identity management and AppFuse is a framework for building web applications based on the Java Virtual Machine (JVM). Both applications access a relational database via the Java Persistence API (JPA). Additionally, we used the MLSA *HRManager* (cf. Section 2.2.2) as a minimal example for the training session.

We selected Apache Syncope and AppFuse for several reasons. First of all, both projects implement a realistic use case for the interaction between Java source code and a relational database via JPA. Second, both projects contain a code base of realistic size: Apache Syncope and AppFuse consist of 77 400 and 24 331 Lines of Code (LOC), respectively. The implementation of the database interface consists of 10 169 LOC in Apache Syncope and 527 LOC in AppFuse. This is an important attribute since effects of tool support may easily be diminished by a small code base that can be easily overseen by the participants. In regard to our participants it is important that both projects are implemented in the programming language Java, since Java is the only language with which the participants have sufficient experience. Finally, both projects provide a set of unit tests. Based on the unit tests, we checked if the participants have been able to successfully complete the tasks.

In the experiment, we allow participants to run the unit tests of Apache Syncope and AppFuse, so the participants are able to check their source-code modifications. However, by default each unit-test run in Apache Syncope and AppFuse re-creates the database schema according to the JPA entity definition in the Java source code. Thus, executing the unit tests would overwrite the refactored database schema. We solved this issue by switching off the automatic schema creation in the unit-test configuration of Apache Syncope[4] and AppFuse[5] accordingly. Changing the configuration has no effect on the results of the unit-test runs.

Specific experience with Apache Syncope or AppFuse may affects the outcome of the experiment. Thus, we have to consider experience with the experimental objects as well. To assess the participants' experience with Apache Syncope and AppFuse, we extended the questionnaire from [Feigenspan et al., 2012]. Table 6.1 summarizes the additional questions.

[2]http://syncope.apache.org, visited 15.12.2016
[3]http://appfuse.org, visited 15.12.2016
[4]https://github.com/hschink/syncope/commit/96553ad7061a361ff0f49f623c54834822efe5fe
[5]https://github.com/hschink/appfuse/commit/bd93bc5cc401b423b10df2082c70ce6cedff014b

Question	**Scale**
Do you know the Java Persistence API (JPA)?	Yes or No
How experienced are you with libraries implementing the JPA (e.g., Hibernate or EclipseLink)?	-1 (I don't know any of them) to 4 (very experienced)
Do you know Apache Syncope and AppFuse, respectively?	Yes or No
How familiar are you with the source code of Apache Syncope?	0 (not at all) to 3 (much)
How familiar are you with the source code of AppFuse?	0 (not at all) to 3 (much)

Table 6.1: Questions about the experience with Apache Syncope and AppFuse.

6.1.3 Participants

We recruited 79 students of the University of Magdeburg as participants for our experiment. All students attended a database course. The sample consisted of 76 undergraduate and 3 graduate students. As compensation for their participation, we offered all participants a bonus point for their homework assignments. We informed all participants that they take part in an experiment and that the participation is entirely voluntary. All participants have been assured that they can leave the experiment at any time. We gathered all data anonymously.

All but one participants stated that they know neither the two open-source projects Apache Syncope and AppFuse, nor the code base of the two projects. Only a single participant stated to know both projects. This participant also stated to have minor knowledge of the projects' code bases. However, the performance of this single participant did not suggest any particular familiarity with either Apache Syncope or AppFuse. We, thus, included the participant's data in the analysis.

We considered programming experience as a confounding parameter in our experiment. We determined the participants' programming experience based on the questionnaire and analysis of [Feigenspan et al., 2012]. The computation of the experience involves a participant's answer to the following two questions:

$s.ClassMates$ How do you estimate your programming experience compared to your class mates?

$s.Logical$ How experienced are you with the logical programming paradigm?

The scale of the questions range from *1* (*very inexperienced*) to *5* (*very experienced*). The value for the programming experience is then computed by the formula: $0.441 * s.ClassMates + 0.286 * s.Logical$ [Feigenspan et al., 2012]. Figure 6.1 presents the distribution of the programming experience for the participants with and without tool support ranging from 0 to 2.908.

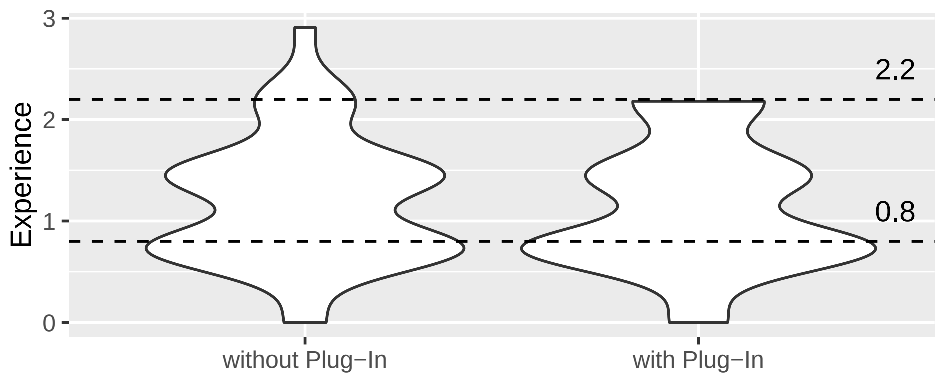


Figure 6.1: Distribution of programming experience.

The distribution shows that participants accumulate on three experience levels. The first level includes experience values less or equal to 800. The second level contains experience values greater than 800 and less than 2200 and the third level contains experience values greater or equal to 2200. Based on the experience distribution, we formed two almost equally-sized programming-experience groups: A group *inexperienced* with experience values less or equal to 800 and a group *experienced* with experience values greater than 800. The group inexperienced contains 40 participants and the group experienced contains 39 participants. To have two almost equally-sized experience groups can be beneficial for the expressiveness of statistical tests.

6.1.4 Tasks

We designed four tasks for the experiment: An introductory task for the training session and three experimental tasks. However, we introduced the third experimental task only as a means to check that no other confounding parameter is affecting the experiment's outcome. In the introductory, we used the source code of HRManager (cf. Section 2.2.2). The first experimental task is based on Apache Syncope's code base and the second and third experimental tasks are based on AppFuse's code base.

All tasks follow the same central theme (cf. Appendix A.2.3): A team of software developers agrees on refactoring the database schema of an application. However, the refactoring of the database schema led to failing unit tests. It is the participant's task to fix the failing unit test by adapting the Java source code to the refactored database schema.

For our experiment, we selected the *Rename Column* and *Move Column* refactoring. We selected these refactorings based on the following criteria:

1. Does the refactoring break the database application?
2. How much JPA and application specific knowledge is necessary to understand the effect of the refactoring?
3. How much effort is necessary to fix the database application?

We selected these refactorings, because, considering the participants knowledge, the effects of these refactorings require to know only the fundamental concepts of JPA. Furthermore, for fixing the effects of these refactorings, the participants only need to know a minimum of application-specific details.

In the following, we introduce the Rename Column and Move Column refactoring in detail. Then, we describe the introductory task and the three experimental tasks.

```
@Entity
@Table(name="departments")
public class Department implements Serializable {

    private int id;
    private String name;

    public void setId(int id) {
        this.id = id;
    }

    @Id
    public int getId() {
        return id;
    }

    public void setName(String name) {
        this.name = name;
    }

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

Listing 6.1: JPA entity `Department`

Rename Column Refactoring

The purpose of the Rename Column refactoring is to give an inappropriately named table column a suitable identifier.

Let us assume we have a database table `departments` with a column `name` (cf. Figure 6.2). The column `name` stores a department's name. We rename the column `name` to `label` to make the usage consistent with columns in other tables. However, renaming the column breaks the Object Relational Mapping (ORM): We cannot use the methods `getName` and `setName` of class `Department` (cf. Listing 6.1) to access the column `label`.

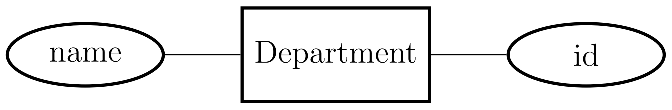


Figure 6.2: Entity relationship model of table `departments`.

We have two options to adapt the Java source code in Listing 6.1 to the refactoring:

- Rename the methods `getName` and `setName` of the class `Department` to `getLabel` and `setLabel` or

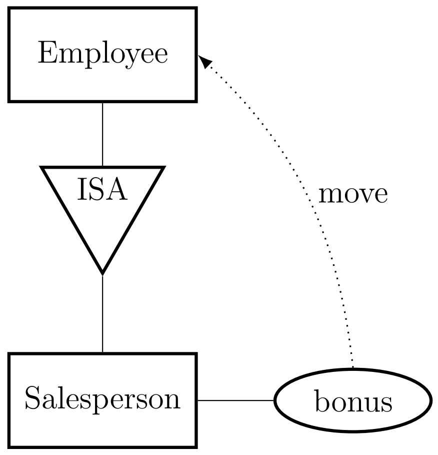


Figure 6.3: Entity relationship model of tables `Employee` and `Salesperson`.

- annotate the method `getName` with `@Column` and assign the value `label` to the annotation's name attribute.

The first option allows us to preserve the naming of related elements between source code and database schema. The second option allows us to rename the database column without the need to rename all occurrences of the methods `getName` and `setName`. Though, the first would preserve a common naming convention across the different parts of the application, we may prefer the second option which has less impact on the entire code base and may also avoid other issues like name clashes (cf. Section 3.1.1).

Move Column Refactoring

The purpose of the Move Column refactoring is to move a column from one table to a different table if the target table is more appropriate to store the column's values.

Let us assume we have two database tables `salespersons` and `employees` (see Figure 6.3). The table `salespersons` contains a column `bonus` that stores the amount of the bonus that a salespersons gets. Now, assume that management decides that all employees should get a bonus. To avoid redundancies, we want to manage the bonus of all employees in the table `employees`. Hence, we want to move the column `bonus` from table `salespersons` to table `employees`. However, after moving the column `bonus`, the ORM is broken and we cannot use the methods `getBonus` and `setBonus` of class `Salesperson` to manage the bonus of a salesperson. Furthermore, we cannot manage the bonus from the class `Employee`, because the class misses the required getter and setter methods.

For adapting the ORM in the Java source code, we have to move the methods `getBonus` and `setBonus` from class `Salesperson` to class `Employee`. Additionally, we may have to adapt the application logic, such as the user interface that expects bonus information in the class `Salesperson` instead of class `Employee` (cf. Section 3.1.1).

Experimental Tasks

In this section, we introduce the experimental tasks in detail and describe possible approaches to adapt the Java source code to the refactored database schema.

Introductory Task

The introductory is based on a Rename Column refactoring that renames column `surname` of table `customers` to `lastname`. The renaming breaks the ORM between the database and the source code of HRManager. The task of the participants is to fix the ORM in HRManager's source code. The following steps describe a possible solution:

1. Find the methods `getSurname` and `setSurname` in the class `Customer`.
2. Fix the ORM by either
   (a) renaming the methods to `getSurname` and `setSurname` or
   (b) applying JPA's `@Column` annotation on the method `getSurname` with the attribute `name` set to `lastname`.

For participants with tool support, the sql-schema-comparer highlights the method `getSurname` of class `Customer`. In contrast, participants without tool support need to use the information in the task description or unit-test log to identify the methods mentioned in Step 1. We explained the participants how to detect and fix the error in respect to the assigned group. For participants without tool support, we described how to use the task description, unit-test log, and the built-in search function in Eclipse to find the erroneous spot. For participants with tool support, we explained how to use the sql-schema-comparer to locate the defective source code.

Task 1

Task 1 describes a Rename Column refactoring that renames column `CHANGEPWDDATE` of table `SYNCOPEUSER` to `CHANGEPASSWORDDATE`. Due to renaming, the ORM between the database and the Java source code breaks. The task for the participants is to fix the ORM in Apache Syncope's Java source code. The following steps describe a possible solution to fix the ORM:

1. Find the methods `getChangePwdDate` and `setChangePwdDate` in the class `SyncopeUser`.
2. Fix the ORM by either
   (a) renaming the methods to `getChangePasswordDate` and `setChangePasswordDate` or
   (b) applying JPA's `@Column` annotation on the method `getChangePwdDate` with the attribute `name` set to `CHANGEPASSWORDDATE`.

For participants with tool support, the sql-schema-comparer highlights the method `getChangePwdDate` of class `SyncopeUser`. In contrast, participants without tool support need to use the information in the task description or unit-test log to identify the methods mentioned in Step 1.

Task 2

Task 2 describes a Move Column refactoring that moves the column `POSITION` from table `ROLE` to table `APP_USER`. Due to moving the column, the ORM between the database and the Java source code breaks. The participant is asked to fix the ORM in AppFuse's Java source code. The following steps describe a possible solution to fix the ORM:

1. Find the methods `getPosition` and `setPosition` in class `Role`.
2. Find the class `User` that maps onto the table `APP_USER`.
3. Fix the ORM by moving the methods `getPosition` and `setPosition` to class `User`.

For participants with tool support, the sql-schema-comparer highlights the method `getPosition` of class `Role`. In contrast, participants without tool support need to use the information in the task description or unit-test log to identify the methods mentioned in Step 1. However, the sql-schema-comparer does not mark the class that maps to the target table of the Move-Column refactoring (cf. Step 2). Thus, the sql-schema-comparer does not provide complete support for this refactoring task.

Originally, AppFuse's database schema does not contain an obvious candidate for the Move Column refactoring. To not let participants waste time on irrelevant details, we decided to introduce the column `POSITION` on table `ROLE` as candidate for the refactoring.

Task 3

The purpose of Task 3 is to serve as a sanity check for the results of Task 1 and Task 2. That is, the results should give us a hint whether other effects than tool support influenced the development productivity for Task 3. For this purpose, no database refactoring is applied to the relational schema in Task 3 and, thus, the sql-schema-comparer does not provide any support for this task.

This task asks participants to ensure the data validity of the Uniform Resource Locators (URLs) stored in column `WEBSITE` of table `APP_USER` in the application's Java source code. The relational database H2[6] used in the experiment does not provide a dedicated data type for URLs. Thus, the database cannot distinguish URLs from other strings. The following steps describe a possible approach to ensure the data validity:

[6] http://www.h2database.com

1. Find the methods `getWebsite` and `setWebsite` in class `User`.
2. Change the parameter of `getWebsite` and the return type of `setWebsite` from `java.lang.String` to `java.net.URL`.

Since no database refactoring or other source-code modification is applied beforehand to AppFuse's code base, the unit tests complete successfully from the beginning. Thus, participants are asked to do the necessary modifications without breaking the application.

6.1.5 Tooling

We used Prophet[7] to present the questionnaire and the tasks to the participants and to collect the data [Feigenspan et al., 2011]. For browsing the source code and executing the unit tests, the participants used Eclipse. We provided scripts that started a pre-configured Eclipse environment for each task. The pre-configured Eclipse environment only included the Java source code of the respective task. The scripts also logged their invocation time. Furthermore, we configured Eclipse to log the results and the completion time of the unit-test runs.

In the questionnaire, we asked participants how familiar they are with the Eclipse IDE. The question allowed answers ranging from 0 (very inexperienced) to 4 (very experienced). Table 6.2 shows the possible answers related to the number of participants who chose the answer. We could not verify any relation between the participants' experience with Eclipse and the results of the experiment.

Experience with Eclipse	Number of Answers
0	6
1	10
2	37
3	24
4	2

Table 6.2: Number of answers to the question about the experience with Eclipse.

For the participants who were assigned to work on the tasks with tool support, we provided an Eclipse with the sql-schema-comparer Eclipse plug-in pre-configured. Although the plug-in was pre-configured, it was not activated. Thus, we let the participants activate the plug-in. We considered the time to activate the plug-in in the measurement of the development time. The activation triggers the comparison of the Java source code and the relational schema. In case the interaction between the Java application and the database is broken, the plug-in immediately provides error markers in the Java source

[7]https://github.com/feigensp/Prophet, visited 15.12.2016

code. We have not considered measuring the performance of the sql-schema-comparer Eclipse plug-in on checking the referential integrity. Thus, we do not know how much time participants spent to wait for the sql-schema-comparer Eclipse plug-in.

6.2 Execution

We offered seven appointments that students could choose according to their convenience. We randomly assigned all students of an appointment to a group with or without tool support. Thus, always all students of an appointment have been working either with or without tool support.

The laboratory where we conducted the experiment had a limited number of 16 work stations. Some appointments were chosen by more students than we had work stations available for the experiment. So, we randomly selected participants by lot. Students who could not take part nevertheless received their bonus point.

After every participant successfully logged into their work stations, we continued with the introduction. First, we introduced the general topic before we presented the necessary details of the ORM with JPA. Additionally, the participants assigned to the experiment with tool support got an introduction to the sql-schema-comparer Eclipse plug-in. To perform the introduction similarly in all groups, we created a presentation that set the content for the introduction (cf. Appendix A.2.1). The introduction was closed with an example refactoring on the HRManager (see Section 6.1.4) and a question and answer session. We ensured that all participants successfully completed the introductory example before we asked the participants to start the experiment by answering the questionnaire and solving the tasks. Additionally, we asked all participants to initially run the unit tests on each task to ensure that all participants have the same starting point. When a participant had finished the experiment, she could leave without disturbing the others.

Though we checked that the sql-schema-comparer works correctly on the laboratory's work stations, when conducting the experiment, the participants were not able to successfully activate the plug-in due to a technical problem. We offered participants help to initialize the plug-in correctly by an additional manually activation of the plug-in.

In some cases, unit tests were failing although no obvious error was found. We asked the participants affected by these unit tests to continue with the next task. For Task 2, some participants applied incorrect source-code modifications that left the source code in a broken and hard to fix state. For instance, participants applied Eclipse's built-in Move Method refactoring on the methods `getPosition` and `setPosition`. However, the participants did not move the methods to the class `User`, but rather to an automatically created new class. We explained the technical reasons for the broken source code shortly to the participants and asked the participants to continue with the next task. We did not consider tasks with failing unit-tests or broken source-code for the analysis.

6.3 Analysis

The result set contains data of 79 participants. Before we analyzed the data we cleaned the data set (see Table 6.3 for an overview). We started with removing measurements of unsuccessful tasks. For Task 1 and Task 2, we identified an unsuccessful task by a missing successful unit-test run in the unit test log. Hence, we removed also incomplete tasks, because these miss a successful unit-test run, too. For Task 3, the application is not broken, that is, initially, the unit tests complete successfully. Thus, we analyzed the participants' actual source-code modifications to detect successful completions of Task 3.

We checked the development times for soundness by comparing the times measured in Prophet with the timestamps created in Eclipse. Since we did not enforce to start Eclipse after reading the task in Prophet, we use the development times measured in Prophet for the following analysis.

Though we asked the participants to run the unit tests at the beginning of each task, the gathered data shows that some participants ran unit tests only once. Especially participants with tool support did not invoke an initial unit-test run. However, running unit tests can take a reasonable amount of time: In case of Task 1, the first unit-test run can take more than a minute. For Task 2, unit-test runs take about seven seconds. Thus, the missing initial unit-test runs can bias the results in favor for the evaluated tool: For each task and participant, we extracted the execution time for the first unit-test run and calculated the mean of the extracted execution times for each task. To correct the bias, we applied a penalty to the development times of the results with only one unit test run: We added the mean of the execution times of the first unit-test runs of the respective task to the development time of the results with only one unit-test run.

After the removal of unsuccessful tasks and the correction of time measurements, we removed outliers, that is, development times that deviate more than 1.5 standard deviations from the mean. Table 6.3 summarizes the number of successfully completed tasks in column *Valid*. Additionally, for development time μ_T, the table presents the number of outliers in the column *Outliers* and the total number of results in column *Total*. You find the adjusted dataset that we used in our analysis in Table A.4 in Appendix A.2.4.

	Valid	Outliers	Total
Task 1	69	2	67
Task 2	72	1	71
Task 3	77	3	74

Table 6.3: Available number of results for each task.

For the analysis, we applied the two-way Analysis of Variance (ANOVA). The two-way ANOVA allows us to evaluate the influence of two independent variables on one dependent variable. Thus, in our case we are able to evaluate the effect of tool support and programming experience on the development times of the participants. The two-way ANOVA assumes the data to be normally distributed and to have the same variance. We applied the Shapiro-Wilk test to test for normality and Bartlett's test to test for variance. We found that based on the Shapiro-Wilk test we can assume normality for the data of Task 1 and Task 3, but not for Task 2. Thus, we need to be especially careful making conclusions based on the results of Task 2. Based on Bartlett's test we can assume variance equality for Task 2 and Task 3, but not Task 1. However, we found that the correction of data leads the Bartlett's test to reject variance homogeneity. Recall that we had to correct some of the development times because of a missing initial unit-test run. The data without the correction adheres to the variance homogeneity assumption. To evaluate how intense the effect of the missing variance homogeneity is on the analysis result of Task 1, we compared the results of the two-way ANOVA for the development times with and without the corrections. We found that the result of the corrected data is in line with the unmodified data.

Figures 6.4, 6.6 and 6.8 present the development times grouped by tool support and programming experience. We applied a two-way ANOVA on the results of each task with the independent variable tool support (*plg*) and the confounding variable programming experience (*exp*) as the two factors. The results of the two-way ANOVA for tool support, programming experience, and the interaction of tool support and experience (*plg:exp*) are summarized in Figures 6.5, 6.7 and 6.9.

For Task 1, the groups with tool support were faster than the groups without tool support. The result of the two-way ANOVA in Figure 6.5 shows a significant effect of the variable *plg* (p-value in column Pr(>F): 0.0089) with a small effect size (η^2: 0.12). Considering the means of the different experience groups, the performance differs for inexperienced participants by 284.3 seconds and for experienced participants by 53.7 seconds. The results show no significant interaction of *plg* and *exp* (cf. Figure 6.4).

For Task 2, the groups with tool support were slower than the experienced group without tool support, but faster than the inexperienced group without tool support. The results of the two-way ANOVA in Figure 6.7 show no significant effect of *plg*, but of *exp* on the development time. Since the data for Task 2 does not adhere to the normality assumption, we have to be especially careful relying on results of Task 2. To examine the effect of experience, we calculated the Kendall rank correlation coefficient (Kendall's τ) to evaluate the correlation between development time and development experience for Task 2. The Kendall rank correlation coefficient does not make any assumptions on the distribution of the data. For Task 2, Kendall's τ is -0.1688977 with a p-value of 0.032944. That is, a small negative correlation between development time and development experience.

For Task 3, the groups with tool support were faster than the groups without it. However, neither *plg* nor *exp* show a significant effect on development time (cf. Figure 6.9). Furthermore, there is no significant interaction of *plg* and *exp*.

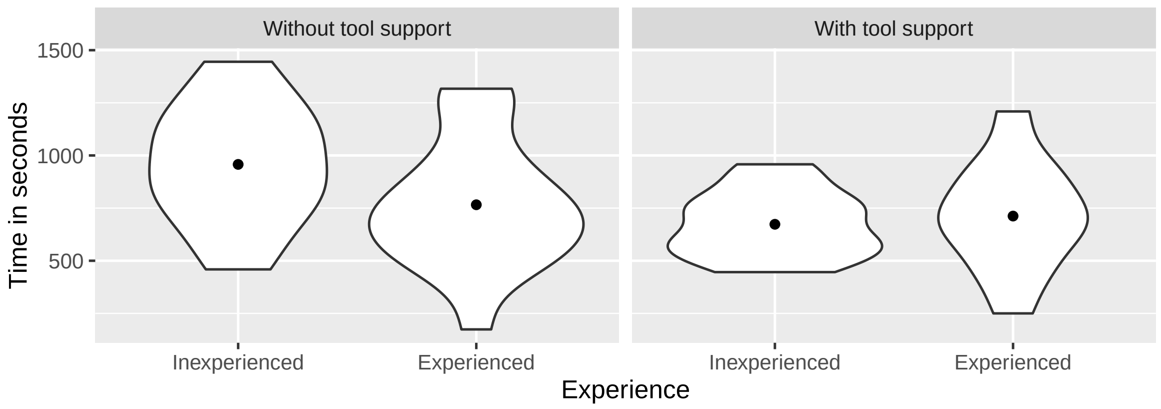


Figure 6.4: Distribution of development times for Task 1.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	η^2
plg	1	470751.26	470751.26	7.29	0.0089	0.12
exp	1	125622.11	125622.11	1.94	0.1680	0.031
plg:exp	1	215142.02	215142.02	3.33	0.0727	0.053
Residuals	63	4069218.23	64590.77			

Figure 6.5: ANOVA results for Task 1.

In summary, for Task 1 we can reject the null-hypothesis H_0 regardless of the participants' programming experience. We cannot reject the null-hypothesis for Task 2 and 3.

6.4 Interpretation

For Task 1, tool support allowed participants significant faster adaption of the broken source code regardless of their programming experience. Since all participants successfully completed the introductory task, which shares the same rationale with Task 1, we assume that all participants were equally prepared for Task 1. Thus, we conclude that the sql-schema-comparer actually improved the participants' performance with its ability to mark the getter method that maps to the renamed column. A plain Eclipse does not provide such information.

Tool support has no significant influence on Task 2. Thus, we cannot reject H_0 based on the results of Task 2. However, programming experience shows a significant effect on the results of Task 2. The additionally calculated result of the Kendall rank correlation coefficient ($\tau = -0.1688977$ with a p-value of 0.032944) suggests that there is a certain probability that a very small negative correlation between development time and development experience in Task 2 exists (cf. Section 6.3). Based on this finding, the reason for sql-schema-comparer plug-in having no effect on the outcome of Task 2

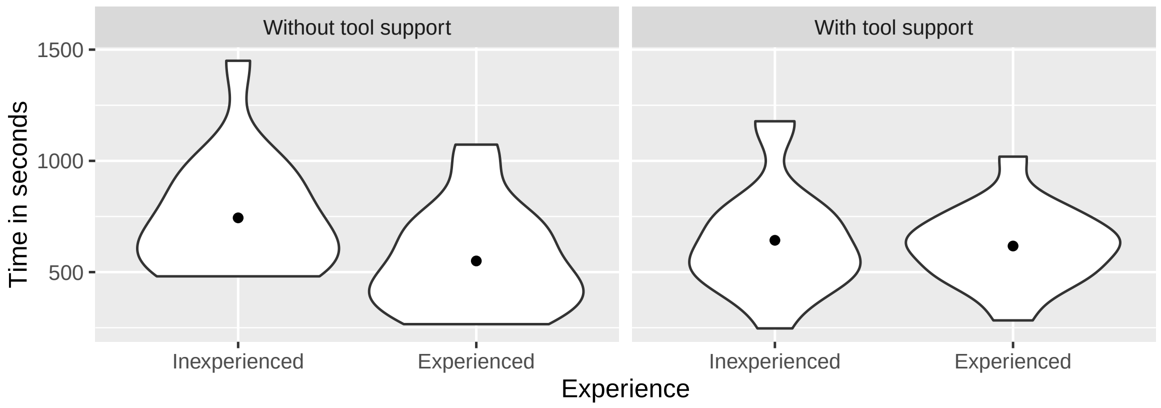


Figure 6.6: Distribution of development times for Task 2.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	η^2
plg	1	1322.72	1322.72	0.03	0.8731	0.00038
exp	1	207182.00	207182.00	4.03	0.0488	0.06
plg:exp	1	124933.44	124933.44	2.43	0.1238	0.036
Residuals	67	3445465.16	51424.85			

Figure 6.7: ANOVA results for Task 2.

could be the following: Two relational tables `ROLE` and `APP_USER` are affected by the Move Column refactoring and, thus, the two Java classes `Role` and `User` needed to be adapted. The Eclipse plug-in helps developers to find the method `getPosition` in class `Role` which, after the Move Column refactoring, has no matching column in the related database table. However, the plug-in does not highlight the Java class `User` which needs to contain the methods `getPosition` and `setPosition` in order to fix the ORM. In contrast, participants without tool support just continued to use Eclipse's built-in search functionality and the refactoring description given with each task to detect the affected methods and classes. Participants with tool support and an higher programming experience might have been able to adapt faster to a refactoring without complete tool support and to use the built-in search of Eclipse instead.

In Task 2, the advantage of the plug-in could have been diminished by showing only one affected class. Though the sql-schema-comparer may help developers to find the class that maps to the source table of the Move Column refactoring, our tool does not indicate the class that maps to the target table of the refactoring. However, technically the sql-schema-comparer is able to detect moved columns in a relational schema. By integrating this feature to the Eclipse front-end, we may be able to support the Move Column refactoring entirely.

For Task 3, participants with tool support required less time to adapt the Java source code to the database refactoring compared to participants with tool support (cf. Fig-

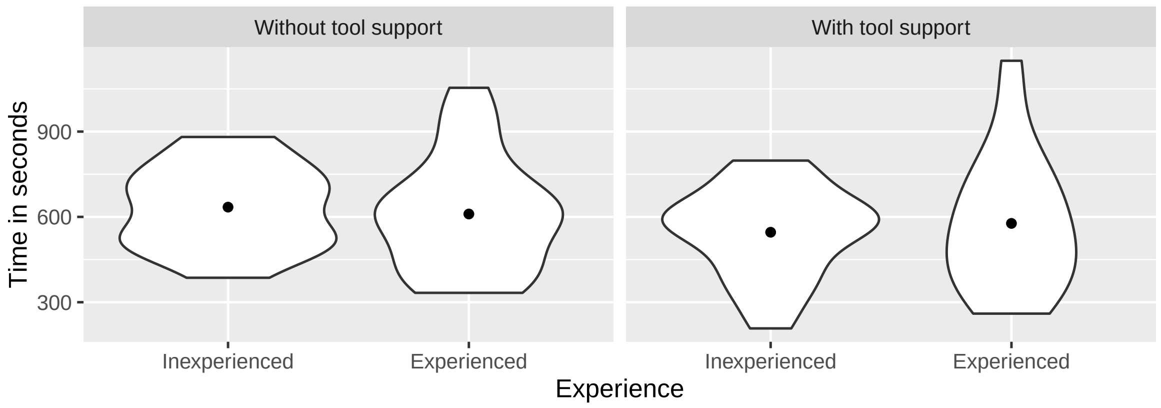


Figure 6.8: Distribution of development times for Task 3.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	η^2
plg	1	66967.13	66967.13	1.83	0.1799	0.026
exp	1	350.78	350.78	0.01	0.9222	0.00014
plg:exp	1	13953.30	13953.30	0.38	0.5384	0.0055
Residuals	70	2555216.15	36503.09			

Figure 6.9: ANOVA results for Task 3.

ure 6.8). However, the difference in the development time is not significant. We expected this result for variable tool support, because our plug-in does not support the refactoring of Task 3.

6.5 Threats to Validity

When an experiment is planned, experimenters have to consider the different effects that may influence the outcome of the experiment and risk its validity. In literature, either two or four different types of threats to validity are distinguished [Wohlin et al., 2012]. In the following, we classify the threats to validity of our experiment by two types according to [Campbell and Stanley, 1963]: the internal and the external validity.

6.5.1 Internal Validity

A steady performance by the participants is an important attribute to preserve the informational value of the experiment: An abnormal performance can have a positive or negative effect that cannot be reproduced in different settings. Participants in need for a bonus point may not share the same motivation than participants who take part voluntarily, and, thus, do not show their usual performance. Though all participants took part voluntarily, they may be forced indirectly to take part in the experiment in need for the promised bonus point. Thus, participants in need of a bonus point may not take part voluntarily, but are forced to participate indirectly. According to the exercise instructor of the database course, the bonus point for the experiment was not crucial to any of the participants.

Wrong assumptions about the participants' behavior can lead to wrong conclusions. In the experiment, we assumed that all participants do an initial unit-test run. However, we did not enforce an initial run of the tasks' unit-tests run. Thus, participants were able to and actually did finish a task only with a single, successful unit test. This can affect the validity of our results. We minimized this threat by applying the mean execution time of the first unit-test runs to the results with only a single unit-test run.

The Rosenthal effect describes the influence of the experimenters' expectations on the participants' performance. To avoid it, we developed standardized instructions for the introduction. We ensured that the instructions are not biased towards the sql-schema-comparer by discussion among the authors until reaching interpersonal consensus.

6.5.2 External Validity

We did a convenience sampling and selected only students attending the exercise of a database course as participants for the experiment. Thus, the results can only be carefully applied to different populations such as Ph.D. students or practitioners. Nevertheless, our results are useful for comparison with the many experiments that researchers conduct with students.

We modified the unit-test configuration of Apache Syncope and AppFuse, so unit tests do not overwrite the refactored database schema. Thus, the results of the experiment cannot be applied to applications that depend on the automatic database-schema creation. Nevertheless, the automatic database-schema creation just drops a database schema and does not migrate data from an old to a new database schema. Hence, the results are still relevant for applications that use JPA to access legacy instances of relational databases.

For Task 2, we added an artificial column `POSITION` to table `ROLE` and the artificial methods `getPosition` and `setPosition` to class `Role`. One may argue that because of the artificial extension the results of Task 2 cannot be applied to different code bases. Nevertheless, the purpose of the experiment is to evaluate the influence of tool support on adapting source code to a database refactoring, i.e., the experiment focuses on fixing the ORM between the Java source code and the database schema. Thus, further adaption of the application logic is not considered by the experiment. Additionally, we could imagine that in an initial design a position is associated with a role in the application. Thus, we argue that having a position information associated to a role is not entirely unusual.

6.6 Summary

We conducted a controlled experiment to evaluate whether the Eclipse plug-in of the sql-schema-comparer can improve the productivity of developers who adapt Java source code to a refactored database schema. The sql-schema-comparer's Eclipse plug-in allows developers to detect broken language interactions before executing the application's unit tests or the application itself.

We conducted an experiment with 79 student participants based on two open source projects (Apache Syncope and AppFuse). The task of the participants was to adapt the Java source code of the two open source applications to a Rename Column (Task 1) and a Move column (Task 2) refactoring which we applied to the applications' database back-end beforehand. We found that depending on the task the sql-schema-comparer can improve the development productivity of inexperienced as well as experienced participants. For instance, for Task 1 we could provide evidence that using the sql-schema-comparer Eclipse plug-in results in a significant better performance. However, in Task 2 we could not detect any significant improvement of the development times for participants using the sql-schema-comparer Eclipse plug-in. But we found possible causes for the missing improvement: We conclude that the Eclipse plug-in needs to expose the full feature-set of the sql-schema-comparer and the structure-graph library to support refactorings that involve incomplete but valid ORMs. However, according to the outcome of Task 1, the plug-in can already be considered useful for the common rename refactoring.

Furthermore, since all participants in the group with tool support used the sql-schema-comparer Eclipse plug-in, we conclude that the plug-in performed reasonably well on code bases up to the size of Apache Syncope and AppFuse to make a significant difference on the participants' development time on Task 1. This is a relevant result because (1) it supports our claim of the practicability of the approach and (2) it suggests that a prototypical implementation can already provide a significant benefit. Especially the last suggestion may ease the practical adoption of the structure-graph approach.

7. Related Work

Since its introduction in [Opdyke, 1992], the scientific community explored different topics in the field of *refactoring*. Among these topics we find the following:

- Search for new refactorings [Ambler and Sadalage, 2006; Fowler, 1999; Li, 2006]
- Automation of refactorings [Dig, 2007; Li and Thompson, 2008; Roberts, 1999; Streckenbach and Snelting, 2004]
- Detection of applied refactorings [Taneja et al., 2007; Xing and Stroulia, 2005, 2008]
- Proof of the behavior-preservation property of existing refactorings [Bannwart and Müller, 2006; Mens et al., 2005; Soares et al., 2011]
- Improvement of the tool support for refactoring [Murphy-Hill and Black, 2008; Xing and Stroulia, 2008]
- Re-use of refactorings for different languages [Ducasse et al., 2000; Strein et al., 2007; Tichelaar, 2001]
- Refactoring of Multi-Language Software Applications (MLSAs) [Mayer and Schroeder, 2012; Pfeiffer and Wąsowski, 2011, 2012b; Strein et al., 2006]

This list is not exhaustive. However, the list emphasizes the diversity of the field of refactoring. In this chapter, we focus on a subset of the topics listed above. In particular, we introduce attempts for re-using refactorings, since these attempts influenced how refactoring was realized in MLSAs initially. We present work that represents the state-of-the-art in Multi-Language Refactoring (MLR) and work that is related to our attempt to support refactoring in MLSAs. Last but not least, we present evaluations of refactoring tools and how these evaluations are related to the one presented in this thesis.

In this thesis, we focus on language interactions in which no relation between host and guest language[1] exists, because this kind of interaction lacks tool support for checking language interaction (cf. Section 2.2.1). In [Renggli, 2010], the authors provide an overview about different approaches to define Domain Specific Languages (DSLs) and a list of techniques to embedd languages.

We structured this chapter as follows. First, we present approaches to re-use refactorings in different languages. Second, we give an overview about work on MLR. Then, we present other evaluations of refactoring tools and relate these evaluations to ours. In the last section, we present approaches to change detection that are related to the change detection with structure graphs (cf. Section 4.3).

7.1 Re-use of Refactorings

In this section, we discuss approaches which allow to re-use one refactoring implementation for different languages to a greater or lesser extent. Nevertheless, these approaches are and remain noteworthy in the context of MLSAs since they provide refactoring solutions for MLSAs that are build of programming languages which share a common ground like programming languages that are based on a common platform (cf. Figure 2.1).

The tool environment *MOOSE* provides means to reverse engineer and re-engineer object-oriented systems [Ducasse et al., 2000]. For that purpose, MOOSE is able to import source code of different languages with the help of common exchange formats like CASE Data Interchange Format (CDIF) and XML Metadata Interchange (XMI) [Tichelaar et al., 2000]. The source code of different languages is represented in a meta model called *FAMIX*. The meta model FAMIX can represent source code written in object-oriented as well as procedural programming languages [Tichelaar, 2001, p. 37]. The meta model is specifically designed to allow tools contained in MOOSE a uniform treatment of source code written in different programming languages. One of these tools is the *MOOSE Refactoring Engine* which provides 15 language-independent implementations [Tichelaar, 2001, p. 65] of refactorings described in [Fowler, 1999; Opdyke, 1992; Roberts, 1999; Werner, 1999]. Though, language independence is not achieved entirely [Tichelaar, 2001, p. 115 ff.].

With FAMIX, the tool environment MOOSE contains a new meta-model for representing object-oriented programming languages. Although, the authors of FAMIX considered the Unified Modeling Language (UML), they found version 1.3 of UML [Object Management Group (OMG), 2000] to be inadequate in respect to the considered re-engineering use cases [Tichelaar, 2001, p. 46 ff.]. Also the next version 1.4 of UML [Object Management Group (OMG), 2001] was considered inadequate for refactoring purposes [Gorp et al., 2003]. However, Gorp et al. [2003] propose extensions to UML called

[1]According to the classification in [Renggli, 2010], we focus on language interaction with external languages.

GrammyUML and evaluate if GrammyUML allows the authors to provide an abstract description of two refactorings for the programming language *C*. Although the authors showed the feasibility of their approach, like for FAMIX the refactoring description in GrammyUML is not entirely independent of language-specific properties.

Lämmel presents a refactoring framework based on generic functional programming [Lämmel, 2002]. The framework provides generic interfaces for different refactorings. By implementing the generic interfaces for a specific programming language, developers are able to re-use the generic refactoring implementation provided by the refactoring framework. The author presents two generic interfaces for the *extraction* and the *introduction* refactorings and implements these interfaces for the method definition of the academic programming language JOOS[2]. By providing these implementations, the author is able to implement the Extract Method refactoring (extraction) and Create Method refactoring (introduction) for JOOS by re-using the definition in the refactoring framework.

In [Strein et al., 2007], the authors describe a meta model called *Common Meta-Model* that uses trees to represent a software application. The Common Meta-Model consists of different node types and relations between these node types. Language-specific front-ends parse source code and create instances of front-end specific meta-models. Front-end specific meta-models are an abstract representation of the source code written in the programming language, e.g., an Abstract Syntax Tree (AST). The Common Meta-Model defines mappings between elements of front-end specific meta-models and elements of the Common Meta-Model. By applying these mappings on instances of front-end specific meta-models, developers can create an instance of the Common Meta-Model for a specific software application. Like the approaches above, the Common Meta-Model allows developers to describe refactorings on an abstract level. However, the authors note that the Common Meta-Model only supports refactorings that can be described based on the elements included in the Common Meta-Model.

Although the authors emphasize the possibility of re-using refactoring implementations with the Common Meta-Model, they also recognized how useful the representation of MLSAs with the help of the Common Meta-Model [Strein et al., 2006] is. We consider this a major milestone in the effort to refactor MLSAs which culminated in the advent of MLR. We discuss related work in respect to MLR in the following section.

7.2 Multi-language Refactoring Approaches

In this section, we present works that describe approaches to MLR and related terms like Cross-Language Refactoring, Deep Refactoring, and Coupled Software Transformations. Coupled Software Transformations is related to MLR as Coupled Software Transformations also describe changes that may affect more than one artifact of a software application, e.g., source code of different programming languages and documentation [Lämmel, 2004; Lämmel, 2016]. However, in contrast to MLR Coupled Software

[2]JOOS represents a subset of the programming language Java and was originally designed by Laurie Hendren.

Transformations do not define any restrictions on the kind of change and, thus, changes may also modify behavior. Nevertheless, like MLRs, Coupled Software Transformations need a notion of consistency between artifacts, though the allowed modifications may differ.

In the previous section, we presented the work of Gorp et al. [2003] as an example for using UML as means to generalize the refactoring implementation across different languages. In [Bottoni et al., 2003], the authors provide a different perspective: The authors consider UML diagrams as equivalent target for refactoring and present an approach to preserve the consistency between source code and different UML diagrams based on distributed graph transformations. In particular, the authors considered the effects of refactorings on source code as well as UML's class, sequence, and state diagrams. The authors demonstrate how distributed graphs propagate changes between source code and the UML diagrams with the help of the Extract Method and Move Method refactoring.

Also in the previous section, we presented the Common Meta-Model that captures the commonalities of different programming languages [Strein et al., 2007]. In [Strein et al., 2006], the authors of the Common Meta-Model present the Integrated Development Environment (IDE) *X-Develop* which represents the source code of a software application with the help of the Common Meta-Model. X-Develop provides front-ends for the following General-Purpose Programming Languages (GPLs) and DSLs: C#, Java, Visual Basic, J#, Hypertext Markup Language (HTML), Extensible Markup Language (XML), Active Server Pages (ASP), Java Server Pages (JSP), and JavaScript. The Common Meta-Model represents a subset of the elements extracted by the front-ends among which are class, method, and type information from programming languages like C# and Java as well as element and attribute information from markup languages like HTML and ASP.

The Common Meta-Model and X-Develop comprise many characteristics of the structure-graph approach we presented in Chapter 4. Hence, it is appropriate to discuss the differences in detail. First, though both approaches use graphs to represent an MLSA, the approaches differ in the way of representing language interaction: X-Develop uses a single meta-model to capture all kinds of language interaction, while the structure-graph approach allows to use different models to represent language interaction between two languages. Second, the front-ends in X-Develop do not expose any information about *how* language interaction is established. Accordingly, also the Common Meta-Model does not contain any information about the mechanics of language interaction. In contrast, the structure-graph approach exposes how language interaction is established by modeling both the expected structure implemented in the host language and the actual structure implemented in the guest language. Based on the information about the mechanics of language interaction, developers are able to work on software applications with broken language interaction. Third and last, X-Develop is based on the assumption that the Common Meta-Model allows us to readily re-use once implemented refactorings even in an MLSA. However, the authors do not investigate other MLR setups apart from web applications that are based on the GPLs and

DSLs mentioned above. In Chapter 3, we present diverse challenges that can arise when refactoring MLSAs. Consequently, we focus on support for developers who refactor an MLSA rather than automating refactorings in the first place.

In [Chen and Johnson, 2008] and [Kempf et al., 2008], the authors present solutions for the Rename Refactoring in two different MLSA setups. Both solutions are implemented in the Eclipse IDE. The refactoring tool by Chen and Johnson [2008] focuses on the interaction between the Java frameworks Struts, Hibernate, and Spring and their respective XML-based configuration files. As soon as the developer triggers a Rename Class refactoring, the tool adapts corresponding references in the configuration files. Additionally, the refactoring tool also considers JSP files. Kempf et al. [2008] present an idea on how to implement refactoring support for MLSAs written in Java and Groovy. The idea is based on the Java search-engine which is part of the core components of Eclipse's Java development tools[3]. The approaches of Chen and Johnson [2008] and Kempf et al. [2008] are based on the development and refactoring infrastructure provided by the Eclipse IDE. Ideas for supporting MLR beyond Rename refactorings are not part of these works. However, Chen and Johnson [2008] recognize that supporting MLR in general is hard to achieve, though the authors do not further underpin this statement (cf. Section 3.2).

Tatlock et al. [2008] present the idea of *Deep Typechecking* to support developers who use the Java Persistence API (JPA) for accessing databases. More specifically, the approach checks types in respect to queries written in the Java Persistence Query Language (JPQL)[4]. For instance, Deep Typechecking checks the types of parameters which are passed to a JPQL query. In contrast to the sql-schema-comparer, the Deep Typechecking approach considers the interaction between Java and JPQL queries and not the interaction between the ORM defined with JPA and the database schema. Thus, in comparison with Deep Typechecking, the sql-schema-comparer currently checks a different level of language interaction. Based on Deep Typechecking, the authors also present a refactoring approach called *Deep Refactoring*. More specifically, the authors implemented the Rename Class and Rename Method refactoring. Due to the focus on the interaction between Java and JPQL, the authors do not investigate possible effects of other refactorings on the interaction between Java, JPQL, and the database.

Another approach to type checking that considers programming languages with automated memory management is presented in [Furr and Foster, 2008]. In particular, the authors examine OCaml's Foreign Function Interface (FFI) to C: In C source code the types of the values passed via OCaml's FFI need to be determined at runtime because all the OCaml types passed to C share the same memory layout and, thus, cannot be distinguished statically. Based on the examination, the authors present a multi-lingual type interference system that is able to check the correct usage of OCaml types in C source code. Additionally, the approach is also able to check if C source code references

[3]https://eclipse.org/jdt/core/index.php, visited 15.12.2016

[4]JPQL is a Data Manipulation Language (DML) which is based on the Object Relational Mapping (ORM) defined by JPA. The syntax of JPQL is based on SQL.

data on OCaml's heap and to determine whether the reference is safe in respect to the garbage collection of the OCaml runtime system. The type interference system uses dataflow analysis to check the correct usage of OCaml types in the C source code.

In his thesis, Cleve [2009] presents coupled software transformations[5] between a database schema and source code of a procedural programming language called LDA. Though the author uses the term coupled software transformation, he actually refers to refactoring because he only considers semantic-preserving transformations [Cleve, 2009, p. 237]. The definition of semantic-preservation used in the thesis is based on Hainaut [1996] (cf. Section 3.1). The author states that by using semantic-preserving transformations "Program conversion mainly consists in adapting the related [Data Management System] statements to the modified data structure" [Cleve, 2009, p 240]. This statement is based on the coupled software transformations (refactorings) the author defines in the course of his thesis. Nevertheless, the author does not consider refactorings defined elsewhere (for instance in [Ambler, 2003] and [Ambler and Sadalage, 2006]). Consequently, the author does not take into account semantic-changing effects of database refactorings (cf. Section 3.1.2). However, these effects may lead to more complex modifications of the refactored software application than only adapting the related DML statements.

In [Mayer and Schroeder, 2012], the authors present an approach called Cross Language Link (XLL) that is based on so called *semantic links* between artifacts. Semantic links relate artifacts of two GPLs or DSLs with each other. Based on the relations between artifacts, XLL supports program understanding, code analysis, and refactoring. The links are described with the help of relations as defined by the QVT specification [Object Management Group (OMG), 2011, p. 13]. By parsing artifacts like source code and configuration files, the XLL framework resolves links between artifacts of different languages and returns a set of semantic links. In [Mayer and Schroeder, 2014], the authors emphasize the refactoring use case and replace the QVT-based approach by custom binding resolvers for each supported language pair. The result is not a set of semantic links, but a linking model.

The XLL framework with semantic links as well as the custom binding resolver with the linking model is very closely related to our work. Thus, we want to hightlight two important differences in the following. The first difference between XLL and custom binding resolvers and our approach is about the assumed methods of language interaction. We explicitly take different methods of language interaction into account. For instance, there exists different frameworks to establish an interaction between Java source code and a relational database [Anderson, Lance, 2006; Oracle America, Inc., 2013]. For this reason, the check of referential integrity between two interacting languages is not based on specifics of the involved framework. In contrast, with the XLL framework developers define links directly on the involved structure elements like classes, methods, and XML

[5]Throughout the thesis the author uses the term *co-transformation*. However, the author explicitly refers to the definition of coupled software transformation given in [Lämmel, 2004]. Thus, we can use the terms interchangeably.

elements. Thus, our approach includes a further abstraction between structure elements and the actual check of the referential integrity. The second difference is about the possible propagation of refactorings in an MLSA. For the XLL framework and the custom binding resolver approach the authors assume that refactorings can be distributed over artifacts of different GPLs and DSLs as soon as the set of semantic links or the linking model is available. Additionally, the authors only take the Rename refactoring into account. In contrast, in Chapter 3 we showed that we cannot generally propagate a refactoring for one programming languages on artifacts implemented in a different GPL or DSL. Consequently, we extracted the mechanics of language interaction by representing all elements involved in language interaction in structure graphs. In the XLL framework and with custom binding resolvers the mechanics of language interaction are hidden in the QVT-based description of semantics links and the implementation of binding resolvers, respectively. However, we would like to emphasize that structure graphs as presented in Chapter 4 are an attempt to approach refactoring in MLSAs in general. If only a fraction of language interactions may be relevant, XLL and custom binding resolvers still represent an feasible approach to this issue.

In [Pfeiffer and Wąsowski, 2012b], the authors present a so called *Multi-Language Development Environment (MLDE)* called *TexMo*. Like XLL, TexMo supports program understanding, code analysis, and refactoring. The relation between artifacts of different GPLs and DSLs is represented by a *Relation Model (RM)*. The RM is text-based and, thus, depends solely on the textual elements contained in the source code. In the version of TexMo presented in [Pfeiffer and Wąsowski, 2012b], the RM has to be established manually. However, the authors plan to integrate *GenDeMoG* [Pfeiffer and Wąsowski, 2011] in TexMo. GenDeMoG generates programs that search for dependency patterns in source code and create a dependency graph based on the findings. Developers create dependency patterns between elements of language-specific meta-models.

TexMo with GenDeMoG and XLL implement similar approaches on a different technology stack. Thus, the discussion on differences between XLL and our approach can be applied to TexMo and GenDeMoG as well. However, due to the fact that GenDeMoG allows developers to implement the detection of language interaction automatically based on meta-models and dependency patterns, GenDeMoG may eases the burden for the actual tool implementation.

7.3 Studies on Refactoring Tools and MLSAs

In this section, we present studies which investigate either the effect of tool support on refactoring performance or properties of MLSAs. We relate the former to our experiment which we describe in Chapter 6. The latter give insight into how MLSAs are implemented.

7.3.1 Studies on Refactoring Tools

In an empirical study, Murphy-Hill et al. [2009] evaluated data from different sources to answer the general question how refactorings are used in practice. In particular, the authors considered data from different usage reports and source code repositories. In the curse of the evaluation, the authors also investigated how refactoring tools are used in practice. The authors report two findings: First, developers do not often configure refactoring tools before the developers use the tools. Second, developers tend to prefer manual refactoring. However, the authors state that the underlying analysis for both findings suffer several limitations. Thus, the findings have to be considered with caution.

In [Negara et al., 2013], the authors report on a study about refactoring-tool usage that is based on $1,520$ hours of work recorded from 23 developers. The recorded work consists of continuous changes which the developers applied to their code base. Based on the continuous changes, the authors are able to infer refactorings applied to the code base. Moreover, the authors are able to distinguish manually and automatically applied refactorings. The authors affirm the finding of Murphy-Hill et al. [2009] that refactoring tools are underused. Additionally, the authors found that refactoring-tool usage changes with development experience. For instance, novice developers as well as developers with more than 10 years of experience tend to use automated refactorings less often than developers with $5 - 10$ years experience. The authors hypothesize that novice developers have not yet learned to use refactoring tools while experienced users have learned to apply refactorings manually before tool support was even available.

In [Murphy-Hill and Black, 2008], the authors first report about the support for the Extract Method refactoring in the Eclipse IDE. In a pre-study, the authors found that participants faced two main challenges when applying the Extract Method refactoring: (1) Participants made avoidable mistakes when applying the Extract Method refactoring and (2) error messages missed the necessary expressiveness to help participants to recover from an erroneous Extract Method refactorings. Based on these findings, the authors evaluated two new refactoring tools which implement additional support for the Extract Method refactoring. In the evaluation, the authors gathered data of 16 participants. The authors conclude that the additional information provided by the two new refactoring tools can significantly improve the participants' performance. Furthermore, the authors give general recommendations on how refactoring tools can improve the refactoring performance.

Pfeiffer and Wąsowski [2012a] present an evaluation of the tool TexMo [Pfeiffer and Wąsowski, 2012b] conducted with 22 participants on the open-source bug-tracker JTrack[6]. The participants were asked to perform three tasks: In the first task, participants are asked to fix an instance of JTrac on which a rename refactoring was applied only on one artifact, but not on the interacting artifacts. In the second task, the participants are requested to apply a Rename refactoring. In the third task, participants are asked to replace a code block by another one and to report on the artifacts affected by the replacement. All participants were evenly distributed among two groups of which one group had support for MLSAs and the control group had no specific support for MLSAs. No unit or acceptance tests were provided, so that participants without tool support had to start JTrack and detect errors manually. The authors found that TexMo im-

[6]http://www.jtrac.info/

proves refactoring productivity. Additionally, the authors found that TexMo improves the refactoring productivity regardless of the participants' experience.

The study of Pfeiffer and Wąsowski [2012a] is the first one which evaluates development performance in the context of an MLSA. In the following, we highlight the differences in the experimental designs of the study in [Pfeiffer and Wąsowski, 2012a] and our evaluation presented in Chapter 6. The first difference is the number of experimental objects. In [Pfeiffer and Wąsowski, 2012a], the experimental design considers the source code of JTrac. In our study, the experimental design considers the source code of Apache Syncope and AppFuse to mitigate threats to the external validity caused by a non-representative code-base. The second difference is the number of measures. Pfeiffer and Wąsowski [2012a] measure four different metrics while we measure only the time to fix the error. The third difference is the number of participants: 22 participants took part in [Pfeiffer and Wąsowski, 2012a] and 79 participants took part in our experiment. As fourth difference we consider the derivation of programming experience. The participants' professional experience is derived from the participants' years of work experience in [Pfeiffer and Wąsowski, 2012a]. In contrast, we derived the participants' programming experience based on the results reported by Feigenspan et al. [2012]. Fifth, the tutorials of both studies considered different information. In [Pfeiffer and Wąsowski, 2012a], participants received no information about the specific mechanics of language interaction in JTrac. In our experiment, we gave all participants an introduction to JPA (cf. Slide 6). Sixth, the tasks differ between the two studies. In [Pfeiffer and Wąsowski, 2012a], all three tasks contain a variation of the Rename Field refactoring: The first task is based on source code with renamed fields, the second task asks the participant to rename specific fields, and the third task asks the participant to replace one code block with a code block with renamed field names. In contrast, the tasks in our experimental design comprise different refactorings. Finally, the features available in the tools TexMo and the sql-schema-comparer Eclipse Plug-in are different in respect to both experiments. In [Pfeiffer and Wąsowski, 2012a], the cross-language relation model for JTrac was established manually before the experiment started. In contrast, the sql-schema-comparer Eclipse Plug-in automatically created the structure graphs for Apache Syncope and AppFuse during the experiment. In summary, participants of our experiment with tool support had less informational advantage than those in [Pfeiffer and Wąsowski, 2012a]. Furthermore, the time to retrieve the language interaction model (cross-language relations and structure graphs, respectively) was part of our experiment, but the time was not considered in [Pfeiffer and Wąsowski, 2012a]. Thus, our experimental design focuses on the evaluation of the refactoring tool itself. In contrast, the study in [Pfeiffer and Wąsowski, 2012a] also considers the performance of the participants without tool support.

In [Mayer and Schroeder, 2014], the authors present an evaluation of the XLL framework [Mayer and Schroeder, 2012] in respect to seven software applications which are implemented in Java and either Spring, Hibernate, Wicket, or any possible combination of these three frameworks. In particular, the authors evaluate three different use cases: artifact discovery, artifact binding resolution, and refactoring change closures.

The purpose of the artifact discovery use case is to evaluate how many artifacts involved in language interaction can be correctly identified by XLL. With the artifact binding resolution use case, the authors determine how many links between discovered artifacts are established correctly. At last, the purpose of the refactoring change closures use case is to evaluate how many Rename refactorings XLL applies correctly. Due to limitations of the static analysis of certain artifacts, XLL was not able to detect all artifacts involved in language interaction. This problem has to be considered for the structure-graph approach as well. However, we would not relate the issue of non-detected artifacts involved in language interaction to the approach itself, but to the availability of language front-ends which are able to extract the information necessary for deducing the language interaction.

In summary, only a few studies exist on the evaluation of refactoring tools in the context of MLSAs. These studies either do not take development performance into account [Mayer and Schroeder, 2014] or do not entirely focus on the effect induced by the tool [Pfeiffer and Wąsowski, 2012a]. Furthermore, none of the studies takes experience as a confounding parameter into account. To the best of our knowledge, our evaluation is the only one in the context of MLSA which, on the one hand, solely focuses on the evaluation of a refactoring tool and, on the other hand, takes the participants' development experience into account.

7.3.2 Studies on MLSAs

A study on language interaction between Java and either the Android UI framework, the Spring Inversion-Of-Control framework, or the Hibernate Query Language (HQL) is presented in [Mayer and Schroeder, 2013]. The authors derived patterns of language interaction from the three frameworks. Overall, the authors identified eight different patterns assigned to three categories. The authors conclude that the implementation of language interaction can be complex. Furthermore, the authors evaluated the correctness of the patterns by applying them to software applications implemented with the help of one of the frameworks. For the evaluation, the authors implemented a prototypical tool which extracts a meta-model representation for Java and the frameworks in use, respectively. Then, the tool establishes links between elements of the Java and the framework meta-model based on the derived patterns.

The patterns derived in [Mayer and Schroeder, 2013] are relevant for our approach as well because we also depend on the extraction of structure element involved in language interaction. However, as the authors pointed out, these patterns need to be considered when creating language front-ends (cf. Section 5.1.2). Nevertheless, by distinguishing the actual source-code structure present in the guest language from the source-code structure expected by the host language (cf. Section 4.1), we preserve the elements involved in language interaction. A linking model as described in [Mayer and Schroeder, 2013, 2014; Pfeiffer and Wąsowski, 2012b] does not contain information about the mechanics of linking. Thus, the mechanics of linking remain an implementation detail of the dependency patterns.

In order to provide support for specific language combinations, it is useful to know which language interactions exist in practice. The most convenient way to investigate language combinations used in practice is to examine open-source projects because the source code of free software is typically readily available. For instance in [Delorey et al., 2007], the authors describe an investigation on the structure of about 10,000 open-source projects hosted on SourceForge[7]. The authors found that 32% of the investigated projects use two or three different programming languages. Additionally, the authors identified C and Perl, C and C++, and JavaScript and PHP as most used language combinations for projects implemented with two programming languages.

In [Vetro et al., 2012], the authors investigate three questions: (1) How much language interaction is present in software applications, (2) which GPLs and DSLs interact most extensively, and (3) are MLSAs more defect-prone than applications implemented with a single programming-language. The study takes the source code and the issue tracker of the Hadoop project[8] as basis for the examination. The results show that 56% of the commits to the Hadoop repository involve artifacts implemented with the help of more than one programming language. Additionally, the source code of Hadoop contains interactions between artifacts of most of the GPLs and DSLs in use within Hadoop. Finally, the authors observed that certain interactions are more error-prone than others. For instance, the interaction between C and XML is more error prone than the interaction between Java and XML. The authors did not investigate possible effects which could explain the differences in error-proneness for different language combinations.

Tomassetti and Torchiano [2014] report on a study about the application of polyglot programming in 15,216 open-source projects hosted on GitHub[9]. In particular, the authors assess how many languages are used within a single project and which languages are typically used together. In the study, the authors distinguish programming, documentation, presentation, data, and automation languages. The authors found that six languages are used on average in a single project. However, without considering documentation languages which are used for describing programming artifacts such as architecture and design decisions, the number decreases to about five languages per project on average. Additionally, the authors discovered that certain languages typically appear together. For instance, the build management tool *make* is mainly used in projects with C and C++ as main languages while XML typically appears with GPLs such as Objective-C and Java. Though the study revealed that developers use a couple of GPLs and DSLs, it is not clear whether the languages actually interact as described in Section 2.2. For instance, the only interaction between the build management tool *make* and source code written in C is that *make* builds C source code at build time. However, we have to assume that, in general, no interaction exists between *make* and an application written in C at runtime.

In [Mayer and Bauer, 2015], the authors present one of the most recent studies on the structure of open-source projects in respect to polyglot programming. In this study,

[7]https://sourceforge.net/
[8]http://hadoop.apache.org/
[9]https://www.github.com/

the authors examine $1,150$ open-source projects hosted on GitHub[10]. The authors found that on average developers use five different programming languages within an open-source project. Based on the results, the authors also observed a relation between certain project attributes and the number of languages used in that project. In particular, the authors observed a direct relation between number of commits, project size, and some of the GPLs used in the examined projects. For instance, there is a direct relation between C as the project's main language and the number of other languages used in that project. The authors observed the same effect for languages used for web development such as JavaScript and Ruby. Finally, in accordance with Tomassetti and Torchiano [2014] the authors discovered that certain language combinations are more frequently used than others. For instance, the build management tool *make* is mainly used in projects with C and C++ as main languages while shell scripts are also used within projects that use Python and Perl as main languages. However, also this study does not investigate whether the languages just appear together or interact with each other.

7.4 Change Detection

In this section, we list approaches which are related to our approach to change detection based on structure graphs (cf. Section 4.3). We focus on approaches related to refactoring.

UMLDiff returns a change set based on the comparison of two class models [Xing and Stroulia, 2005]. UMLDiff is part of a development tool suite called *JDEvAn* [Xing and Stroulia, 2008]. JDEvAn is able to extract a data model of a software application's class hierarchy directly from the application's source-code repository. The data model is based on UML. The UMLDiff approach comprises different algorithms and heuristics to detect renamed, added, and removed elements. These algorithms and heuristics may be useful to improve change detection between structure graphs because our current approach is based on the assumption that we only detect unambiguous node modifications between two revisions of a structure graph.

In [Dig et al., 2006; Taneja et al., 2007], the authors present tools for detecting refactorings applied between two versions of a framework or library, respectively: *RefactoringCrawler* and *RefacLib*. The RefactoringCrawler uses shingles [Broder, 1997] to find similar methods, classes, and packages. Then, RefactoringCrawler applies the strategies of each of the supported refactorings until it cannot detect any new refactorings. The refactorings supported by RefactoringCrawler are Rename Package, Rename Class, Rename Method, Pull Up Method, Push Down Method, Move Method, and Change Method Signature. The refactoring strategies apply a combination of syntactic and semantic checks to identify a refactoring. The semantic checks are based on reference counts. For instance, the Rename Method strategy checks syntactically, if a method shares the same package and class but not name between two versions of a framework.

[10]https://www.github.com/

Then, the strategy checks semantically, if the method is referenced as often in version one as in version two. If both conditions hold, the strategy considers the method to be renamed. The refactoring strategies share a refactoring log which contains already detected refactorings. By sharing the log, strategies are able to detect compound refactorings such as a Rename Class and Rename Method refactoring. Without the log the Rename Method refactoring would go undetected because the classes between two versions do not share the same name. The RefacLib replaces the semantic check based on references implemented in the RefactoringCrawler by heuristics. Taneja et al. [2007] show that heuristics detect more refactorings in libraries where references to a library's Application Programming Interface (API) are missing because a library exposes, but does not use its API.

In [Robbes, 2007], the author presents the idea of *change-based software repositories* as a technical approach to improve the assessment of the evolution of software applications. A change-based software repository represents software systems not as a consecutive list of file states, but as a list of changes to the software application's AST. The repository actively logs and distinguishes atomic change operations like node creation and removal as well as composite change operations which aggregate a number of atomic change operations like refactorings. In [Robbes and Lanza, 2008], the authors implement a tool called *SpyWare* based on the idea of change-based software repositories. The sql-schema-comparer Eclipse Plug-in actively logs changes on the structure elements involved in language interaction, too (cf. Section 5.2.2 and Figure 5.11). Thus, our approach to change detection could be seen as a first attempt to a multi-language change-based software repository. Adding support for composite change operations and refactoring detection as proposed in [Robbes, 2007] would be an interesting next step to further improve the refactoring support in MLSAs for developers.

In [Negara et al., 2012, 2013], the authors present a tool called *CodingTracker* which allows the authors to track source-code edits in the Eclipse IDE. Moreover, the recorded data allows the authors to reproduce the exact state of the source code at any point in time. Apart from the actual source-code modifications, CodeTracker records other development activities such as interactions with the Version Control System (VCS) and unit-test runs. Though CodingTracker implements a notion of a change-based software repository similar to SpyWare, CodingTracker does not rely on an AST, but on actual source-code changes. The authors of CodingTracker emphasize that recording source-code changes allows for a more complete tracking of the coding activities and, thus, for a more fine-grained analysis of source-code evolution.

8. Conclusion and Future Work

Polyglot programming is a paradigm that emphasizes the importance of *using the right tool for the right job* in contrast to using only one tool to implement the requirements of a software application. To be able to efficiently use different programming languages in a software project, developers must be able to call program logic implemented in one programming language from program logic implemented in another programming language. That is, developers need to be able to implement an interaction between program logic implemented in different programming languages. The desired result is a Multi-Language Software Application (MLSA), that is an application in which developers achieve optimal results in respect to maintainability and performance for all implemented requirements.

With each programming language, may it be a General-Purpose Programming Language (GPL) or Domain Specific Language (DSL), the developer may be faced with a new tool chain which allows developers to compile or to interpret, to debug, and to analyze that language. However, in general all these tools are not aware of the different programming languages developers use to implement an MLSA. Thus, developers have no means to statically check language interaction when they refactor or extend an application. Without any support of compilers and other code-analysis tools, developers depend on a sufficient test coverage for instance by integration tests to check the functionality of the language interaction. However, in practice we cannot rely on a sufficient coverage by such tests.

In this thesis, we described specific challenges of refactoring MLSAs in respect to a database application and we introduced a generalized structure of MLSAs to show the diverse challenges developers of automated refactoring solutions have to solve in respect to polyglot programming and MLSAs. We introduced an approach to support developers who refactor MLSAs. The approach considers the possible diversity of languages present in an MLSA and the unknown effects which source-code modifications could have on the interaction between program logic implemented in different languages.

The main idea of our approach is to allow specialized abstractions based on graphs of trees for modeling the interaction between a number of programming languages. With graphs, we can use a single mechanism for checking the interaction between languages.

8.1 Summary of the Thesis

In Chapter 3, we described the diverse challenges developers may face by refactoring a database application implemented with the object-oriented programming language Java and the functional programming language Clojure as well as the relational database SQLite. The interaction between Java and Clojure is implemented by dynamic invocations of functions defined in Clojure from program logic implemented in Java. The interaction between Java and SQLite is implemented with the object-relational mapper Hibernate. Additionally, we theoretically discussed general challenges in respect to a generalized model of MLSAs.

In Chapter 4, we introduced a tree-based approach to support refactoring in an MLSA. The approach focusses on developer support rather than refactoring automation. We also described how developers can follow-up on changes that affect language interaction based on our approach. Furthermore, we justified the general applicability and discussed the performance of our approach.

In Chapter 5, we presented the implementation of our approach for two language combinations: Java and SQL as well as Java and Clojure. We also presented a single library which implements the integrity check for both of the language combinations and a prototypical integration of our approach into the Eclipse IDE. In Chapter 6, we reported on results of an evaluation of our approach for Java and SQL conducted with 79 participants.

Finally, in Chapter 7 we drew a line from approaches to re-use refactorings for different languages to the current approaches to Multi-Language Refactoring (MLR). We also discussed studies on refactoring tools for MLR and studies on the structure of open-source MLSAs. This chapter contains also a discussion of other works related to our approach or its implementation.

8.2 Contribution

In this thesis, we described specific as well as general challenges implementors of automated MLRs are faced with in an MLSA. We presented, implemented, and evaluated an approach taking the possible diversity of MLSAs into account by providing direct support for developers who refactor an MLSA. Furthermore, we provided a experimental design for evaluating refactoring tools' impact on development performance in respect to different refactoring tasks in an MLSA.

Challenges for automated refactoring in MLSAs.

We reported on specific challenges for refactoring within a database application implemented with the help of the object-oriented programming language Java, the functional programming language Clojure, and the relational database SQLite. In particular, we found that (1) not only different options exist to adapt an MLSA to a refactoring, but that the complexity of the different options can be significantly different, (2) a refactoring of one artifact can lead to a semantic-changing adaption of another artifact, and (3) the implementation of language interaction depends on the framework that implements the interaction. With the help of a generalized description of the structure of MLSAs, we deduced the general challenges for realizing automated MLR.

Approach to general refactoring support in MLSAs.

We presented an approach based on tree structures to provide developers with refactoring support in MLSAs. In particular, based on tree structures, which we call *structure graphs* in this context, we are able to implement checks for language interaction and to provide information on previous changes of structure elements involved in language interaction. We described two tools that provide specific refactoring support for the interaction between the programming languages Java and Clojure and between Java and SQL. Both tools re-use the same implementation for the check of language interaction which we implemented in a separate library. Additionally, we evaluated one of our tools and provided a differentiated analysis of the tool's effect on developers' performance.

Experimental design for evaluating refactoring tools for MLSAs.

We introduced an experimental design for the evaluation of the tool which checks the interaction between Java and SQL. The experimental design considers different refactoring tasks and, additionally, the participants' development experience as a confounding parameter. Thus, with the help of this experimental design we are able to compare the effect of different tools on the same language interaction on the one hand. On the other hand, by modifying the tasks and experimental objects we are able to re-use the experimental design for different language combinations.

8.3 Future Work

In Chapter 3, we described effects of different refactorings on an MLSA to emphasize the diversity of refactoring challenges. In order to guide the future development of refactoring tools for MLSAs, we think it is beneficial to extend this investigation to other combinations of GPLs and DSLs. Furthermore, we think that with such a catalog of implementations it becomes easier for researchers and practitioner to further investigate commonalities and differences of language interaction. Such information could be used to further improve the implementation of existing language interactions and of development tools that explicitly consider MLSAs. For instance, in Section 5.2 we augmented the our model for language interaction between Java and SQL with information about foreign-key relationships. Based on this language-specific extension our tool can provide additional information about broken language interaction.

We described how to use specialized tree structures to enable the check of diverse language combinations (cf. Section 4.2). Specialized tree structures allow tool developers to model the specific structure elements which are involved in language interaction. In practice, only a small number of structure elements may be relevant. For instance, for the interaction with a relational database only structure elements such as tables, columns, and column types may be of interest. Thus, we believe that only structure elements present in popular language paradigms like the aforementioned elements of the relational model, classes, and methods are relevant in practice. Further investigation of this question could potentially lead to a small number of models which are sufficient to represent the language interaction between most of the languages relevant in practice.

With the change detection algorithm (see Section 4.3), we described a possible approach to track changes to structure elements involved in language interaction based on structure graphs. We stated that information about changes of the elements involved in language interaction can help developers to fix broken language interactions. The sql-schema-comparer Eclipse plug-in in fact tracks change information, but does not persist this information yet. Furthermore, since not all developers use IDEs, it could be beneficial to extract such information from source-code repositories.

In Chapter 5, we described our implementation of the structure-graph approach. Based on our experience with the implementation, we recognized the effort for creating the language front-ends. However, we believe that significant steps for creating language front-ends can be automated. For instance, existing parsers and parser generators can easily retrieve the structure elements involved in language interaction. Based on a mapping between language-specific types of structure elements and node types for a structure graph, an automated processing can build instances of structure graphs automatically. By automating these steps, tool developers gain time to extract further information like type information that may not straight-forward to deduce by static code analysis.

In respect to the evaluation of refactoring tools for MLSAs, a number of opportunities exist to further improve and extend the experimental design presented in Chapter 6. For instance, different measures of productivity might be relevant and should be considered. A differentiated evaluation of productivity will become even more important the more tools mature. In our study, we considered two experimental objects, that is two open-source projects implemented in Java which access a relational database via an object-relational mapper. We believe that a catalogue of experimental objects is necessary to reflect the diversity of MLSA implementations. A catalogue of experimental objects would also help us to assess the generality of the implementation in respect to different language combinations. However, an interesting question is how differences in the quality of language front-ends could be considered as a confounding parameter. As for the experimental objects, a catalogue of refactoring tasks for different MLSA setups is helpful to implement an easily reproducible experimental design.

A. Appendix

A.1 Implementation

A.1.1 Structure Graph Library

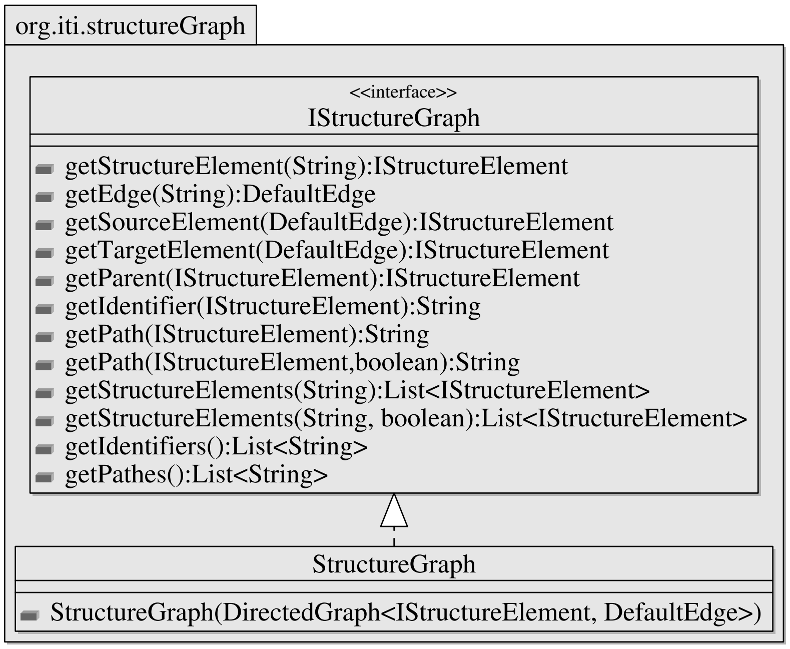

org.iti.structureGraph
<<interface>>
IStructureGraph
getStructureElement(String):IStructureElement
getEdge(String):DefaultEdge
getSourceElement(DefaultEdge):IStructureElement
getTargetElement(DefaultEdge):IStructureElement
getParent(IStructureElement):IStructureElement
getIdentifier(IStructureElement):String
getPath(IStructureElement):String
getPath(IStructureElement,boolean):String
getStructureElements(String):List<IStructureElement>
getStructureElements(String, boolean):List<IStructureElement>
getIdentifiers():List<String>
getPathes():List<String>
StructureGraph
StructureGraph(DirectedGraph<IStructureElement, DefaultEdge>)

Figure A.1: Package `org.iti.structureGraph` with interfaces and classes for implementing structure graphs.

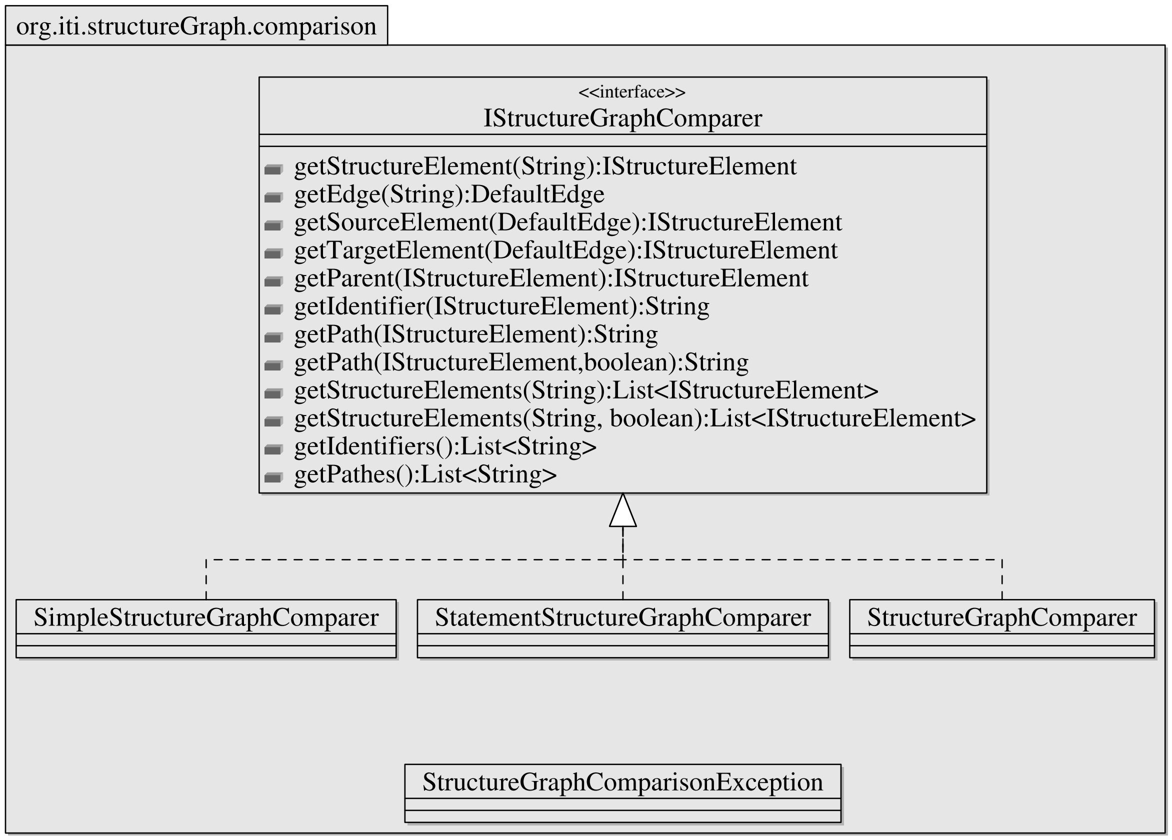

org.iti.structureGraph.comparison
<<interface>>
IStructureGraphComparer
getStructureElement(String):IStructureElement
getEdge(String):DefaultEdge
getSourceElement(DefaultEdge):IStructureElement
getTargetElement(DefaultEdge):IStructureElement
getParent(IStructureElement):IStructureElement
getIdentifier(IStructureElement):String
getPath(IStructureElement):String
getPath(IStructureElement,boolean):String
getStructureElements(String):List<IStructureElement>
getStructureElements(String, boolean):List<IStructureElement>
getIdentifiers():List<String>
getPathes():List<String>
SimpleStructureGraphComparer
StatementStructureGraphComparer
StructureGraphComparer
StructureGraphComparisonException

Figure A.2: Package `org.iti.structureGraph.comparison` with interfaces and classes for implementing structure-graph comparison.

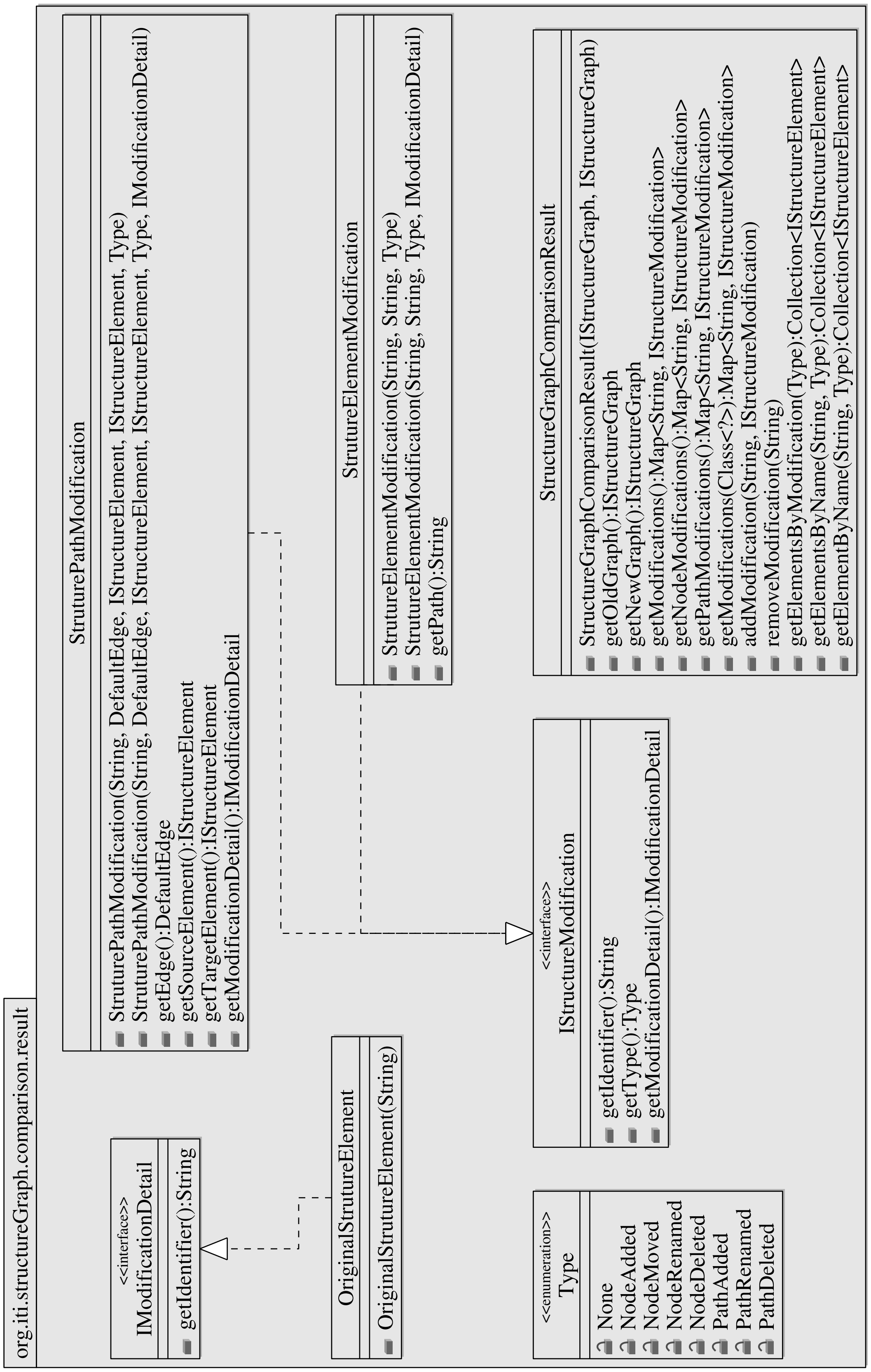

org.iti.structureGraph.comparison.result
<<interface>>
IModificationDetail
getIdentifier():String
OriginalStrutureElement
OriginalStrutureElement(String)
<<enumeration>>
Type
None
NodeAdded
NodeMoved
NodeRenamed
NodeDeleted
PathAdded
PathRenamed
PathDeleted
StruturePathModification
StruturePathModification(String, DefaultEdge, IStructureElement, IStructureElement, Type)
StruturePathModification(String, DefaultEdge, IStructureElement, IStructureElement, Type, IModificationDetail)
getEdge():DefaultEdge
getSourceElement():IStructureElement
getTargetElement():IStructureElement
getModificationDetail():IModificationDetail
<<interface>>
IStructureModification
getIdentifier():String
getType():Type
getModificationDetail():IModificationDetail
StrutureElementModification
StrutureElementModification(String, String, Type)
StrutureElementModification(String, String, Type, IModificationDetail)
getPath():String
StructureGraphComparisonResult
StructureGraphComparisonResult(IStructureGraph, IStructureGraph)
getOldGraph():IStructureGraph
getNewGraph():IStructureGraph
getModifications():Map<String, IStructureModification>
getNodeModifications():Map<String, IStructureModification>
getPathModifications():Map<String, IStructureModification>
getModifications(Class<?>):Map<String, IStructureModification>
addModification(String, IStructureModification)
removeModification(String)
getElementsByModification(Type):Collection<IStructureElement>
getElementsByName(String, Type):Collection<IStructureElement>
getElementByName(String, Type):Collection<IStructureElement>

Figure A.3: Package `org.iti.structureGraph.comparison.result` with interfaces and classes for implementing structure-graph comparison results.

A.1.2 Sql-Schema-Comparer Library

Source Type	Class
H2	`H2SchemaFrontend`
SQLite	`SqliteSchemaFrontend`
`SELECT` statement	`SqlStatementFrontend`
JPA entity	`JPASchemaFrontend`

Table A.1: Available front-ends for structure-graph creation in the sql-schema-comparer library.

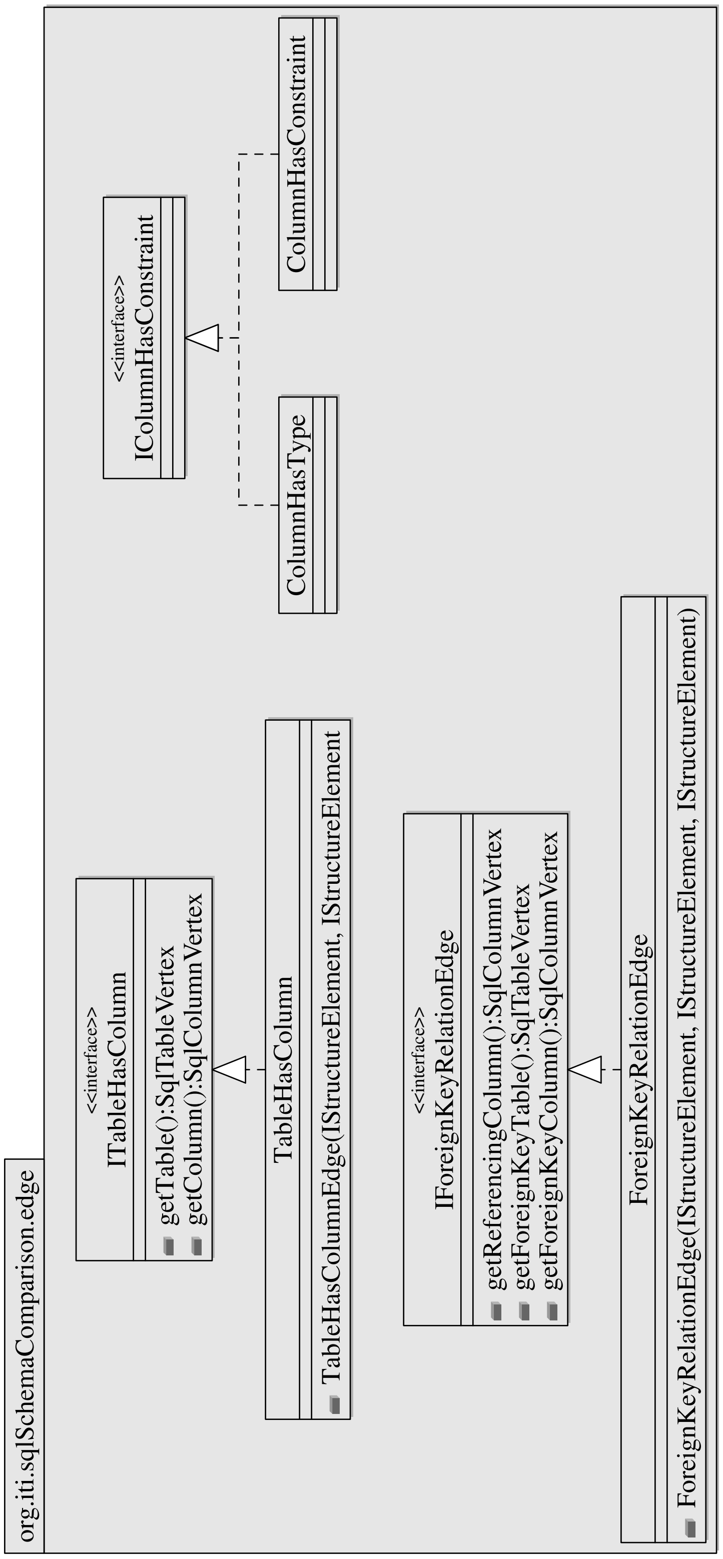

org.iti.sqlSchemaComparison.edge
<<interface>>
ITableHasColumn
getTable():SqlTableVertex
getColumn():SqlColumnVertex
TableHasColumn
TableHasColumnEdge(IStructureElement, IStructureElement
<<interface>>
IForeignKeyRelationEdge
getReferencingColumn():SqlColumnVertex
getForeignKeyTable():SqlTableVertex
getForeignKeyColumn():SqlColumnVertex
ForeignKeyRelationEdge
ForeignKeyRelationEdge(IStructureElement, IStructureElement, IStructureElement)
<<interface>>
IColumnHasConstraint
ColumnHasType
ColumnHasConstraint

Figure A.4: Package `org.iti.schemaComparison.edge` with interfaces for implementing edges between structure elements for database schemata.

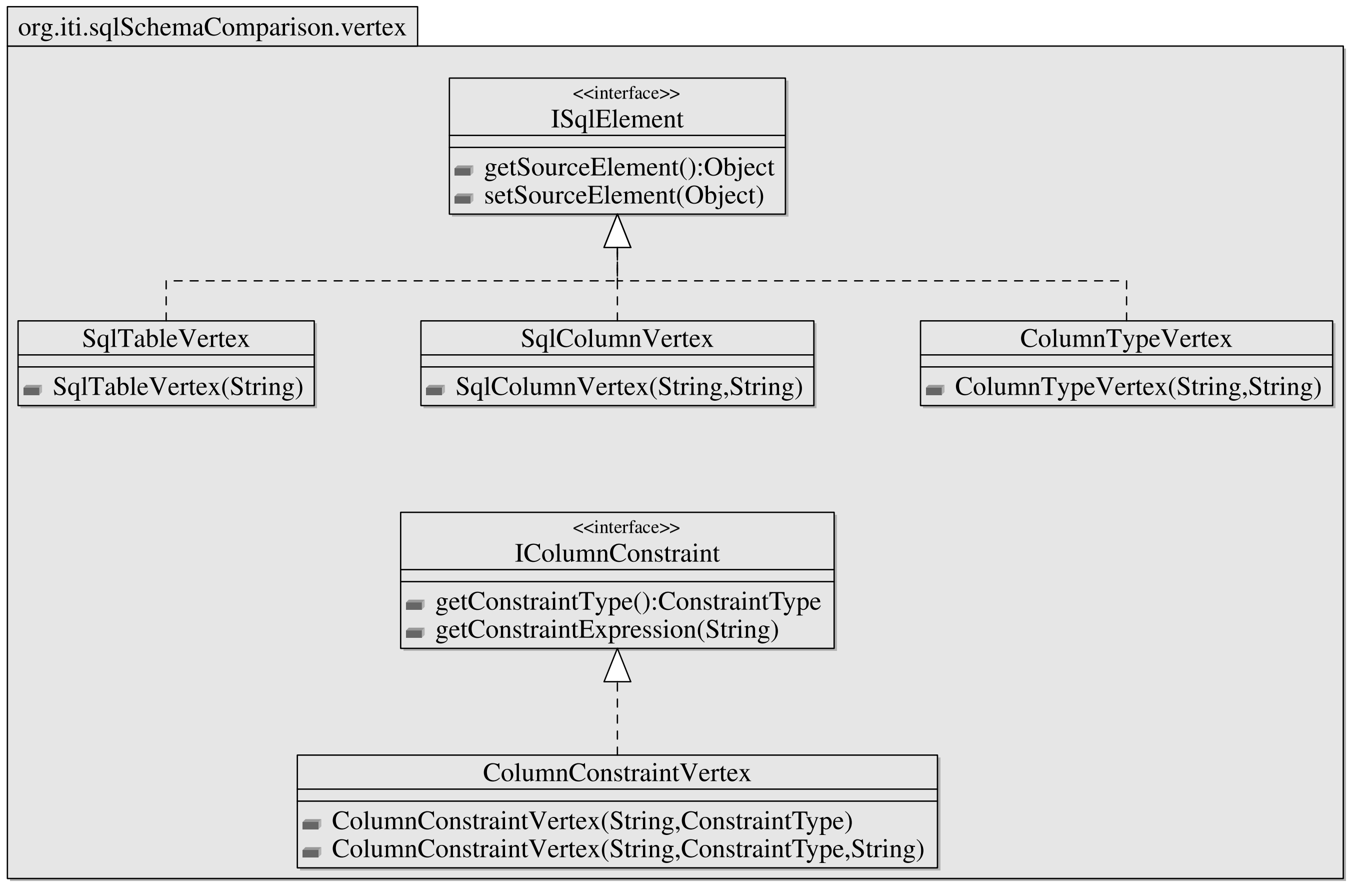

org.iti.sqlSchemaComparison.vertex
<<interface>>
ISqlElement
getSourceElement():Object
setSourceElement(Object)
SqlTableVertex
SqlTableVertex(String)
SqlColumnVertex
SqlColumnVertex(String,String)
ColumnTypeVertex
ColumnTypeVertex(String,String)
<<interface>>
IColumnConstraint
getConstraintType():ConstraintType
getConstraintExpression(String)
ColumnConstraintVertex
ColumnConstraintVertex(String,ConstraintType)
ColumnConstraintVertex(String,ConstraintType,String)

Figure A.5: Package `org.iti.schemaComparison.vertex` with interfaces for implementing structure elements for database schemata.

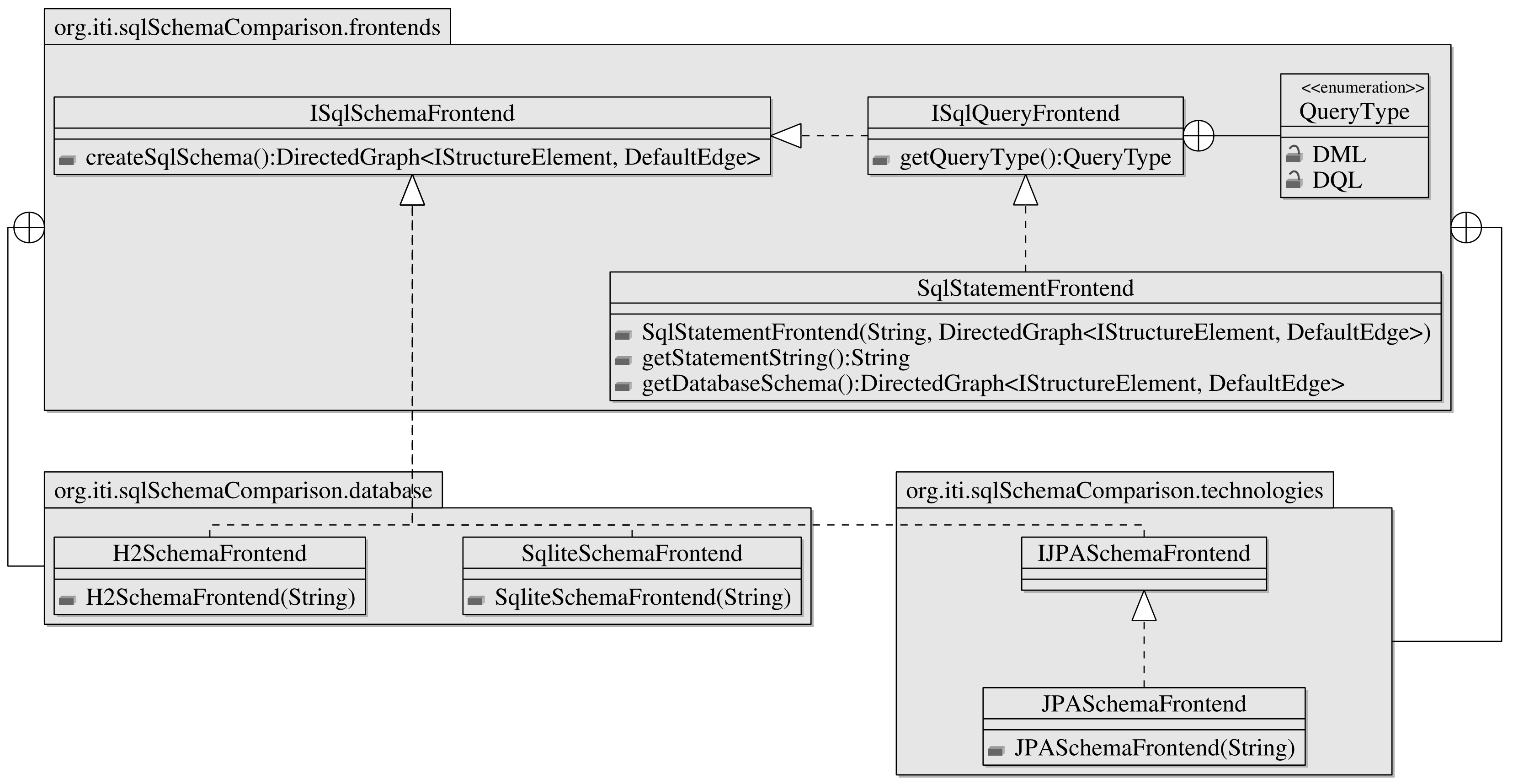

org.iti.sqlSchemaComparison.frontends
ISqlSchemaFrontend
createSqlSchema():DirectedGraph<IStructureElement, DefaultEdge>
ISqlQueryFrontend
getQueryType():QueryType
<<enumeration>>
QueryType
DML
DQL
SqlStatementFrontend
SqlStatementFrontend(String, DirectedGraph<IStructureElement, DefaultEdge>)
getStatementString():String
getDatabaseSchema():DirectedGraph<IStructureElement, DefaultEdge>
org.iti.sqlSchemaComparison.database
H2SchemaFrontend
H2SchemaFrontend(String)
SqliteSchemaFrontend
SqliteSchemaFrontend(String)
org.iti.sqlSchemaComparison.technologies
IJPASchemaFrontend
JPASchemaFrontend
JPASchemaFrontend(String)

Figure A.6: Package `org.iti.schemaComparison.frontends` with interfaces for implementing front-ends for extracting database schemata.

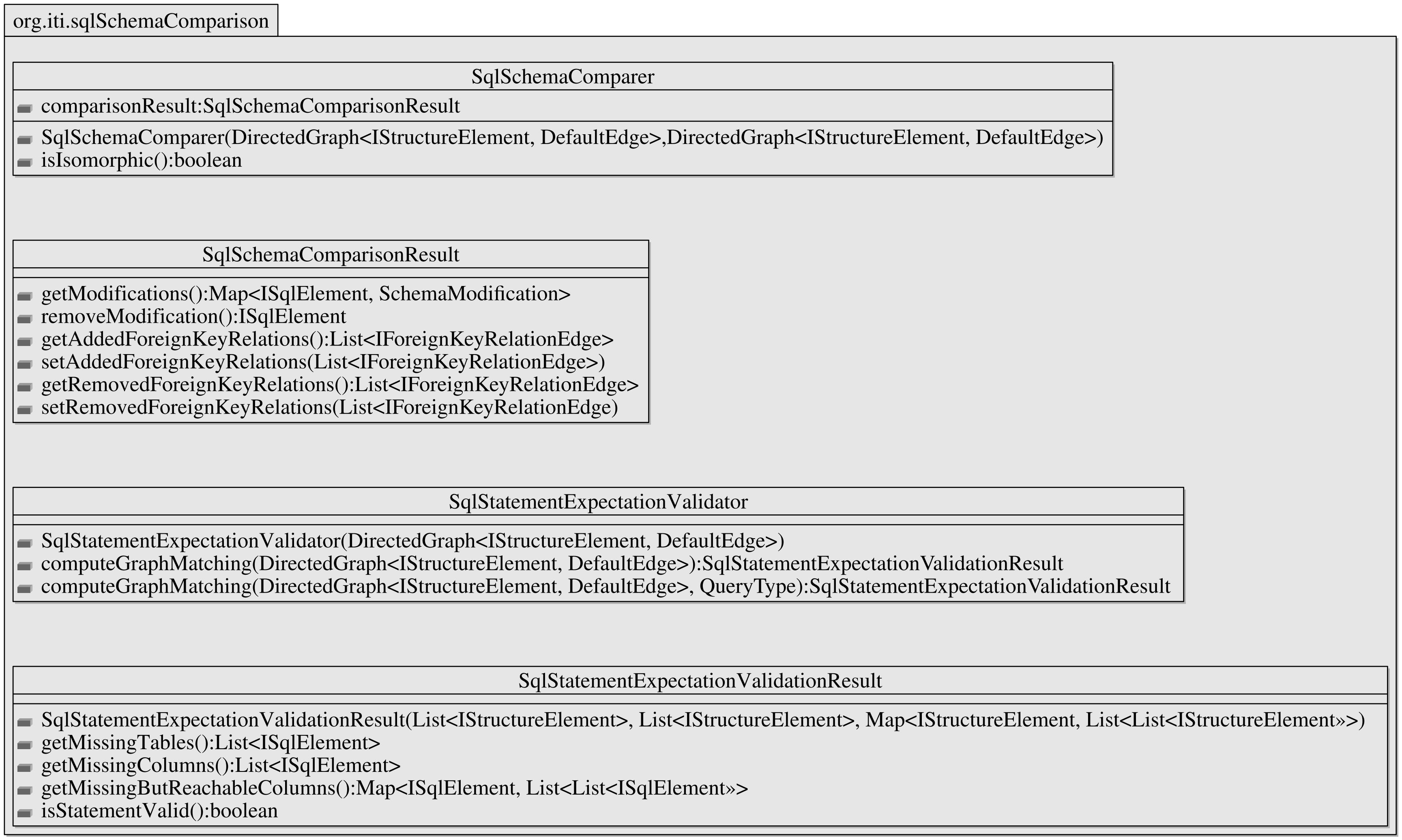

org.iti.sqlSchemaComparison
SqlSchemaComparer
comparisonResult:SqlSchemaComparisonResult
SqlSchemaComparer(DirectedGraph<IStructureElement, DefaultEdge>,DirectedGraph<IStructureElement, DefaultEdge>)
isIsomorphic():boolean
SqlSchemaComparisonResult
getModifications():Map<ISqlElement, SchemaModification>
removeModification():ISqlElement
getAddedForeignKeyRelations():List<IForeignKeyRelationEdge>
setAddedForeignKeyRelations(List<IForeignKeyRelationEdge>)
getRemovedForeignKeyRelations():List<IForeignKeyRelationEdge>
setRemovedForeignKeyRelations(List<IForeignKeyRelationEdge)
SqlStatementExpectationValidator
SqlStatementExpectationValidator(DirectedGraph<IStructureElement, DefaultEdge>)
computeGraphMatching(DirectedGraph<IStructureElement, DefaultEdge>):SqlStatementExpectationValidationResult
computeGraphMatching(DirectedGraph<IStructureElement, DefaultEdge>, QueryType):SqlStatementExpectationValidationResult
SqlStatementExpectationValidationResult
SqlStatementExpectationValidationResult(List<IStructureElement>, List<IStructureElement>, Map<IStructureElement, List<List<IStructureElement»>)
getMissingTables():List<ISqlElement>
getMissingColumns():List<ISqlElement>
getMissingButReachableColumns():Map<ISqlElement, List<List<ISqlElement»>
isStatementValid():boolean

Figure A.7: Package `org.iti.schemaComparison` with classes for comparing database schemata with each other and database schemata with SQL `SELECT` statements.

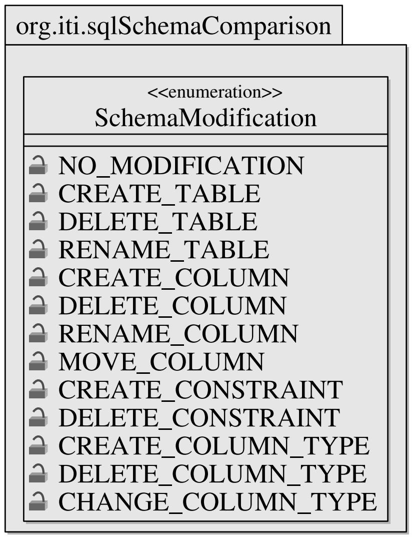


Figure A.8: Package `org.iti.schemaComparison` with an enumeration to distinguish different database schema modifications.

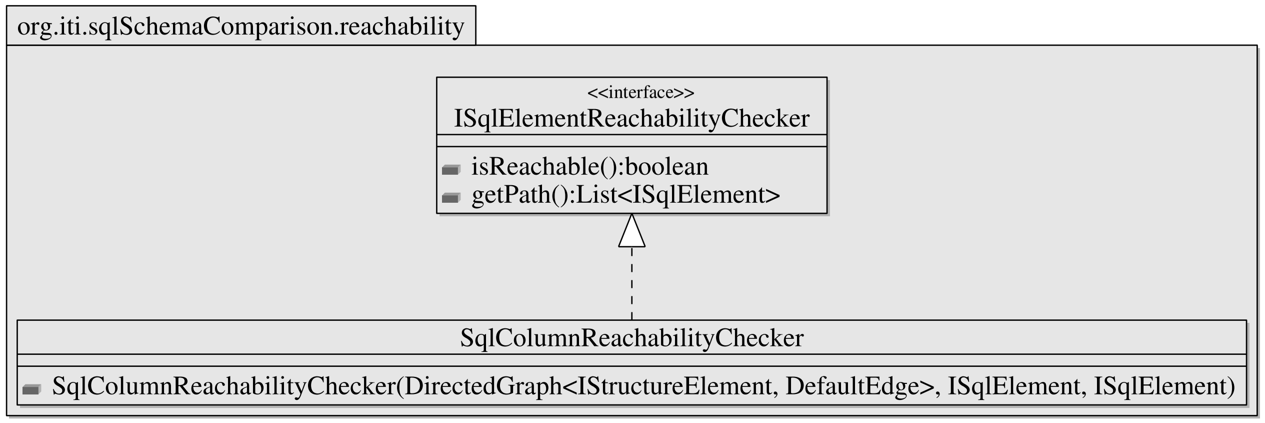

org.iti.sqlSchemaComparison.reachability
<<interface>>
ISqlElementReachabilityChecker
isReachable():boolean
getPath():List<ISqlElement>
SqlColumnReachabilityChecker
SqlColumnReachabilityChecker(DirectedGraph<IStructureElement, DefaultEdge>, ISqlElement, ISqlElement)

Figure A.9: Package `org.iti.schemaComparison.reachability` with utility class to check reachability of column nodes.

A.1.3 Clojure-Java-Interface-Checker Library

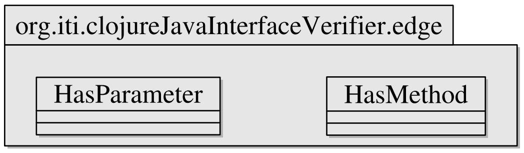
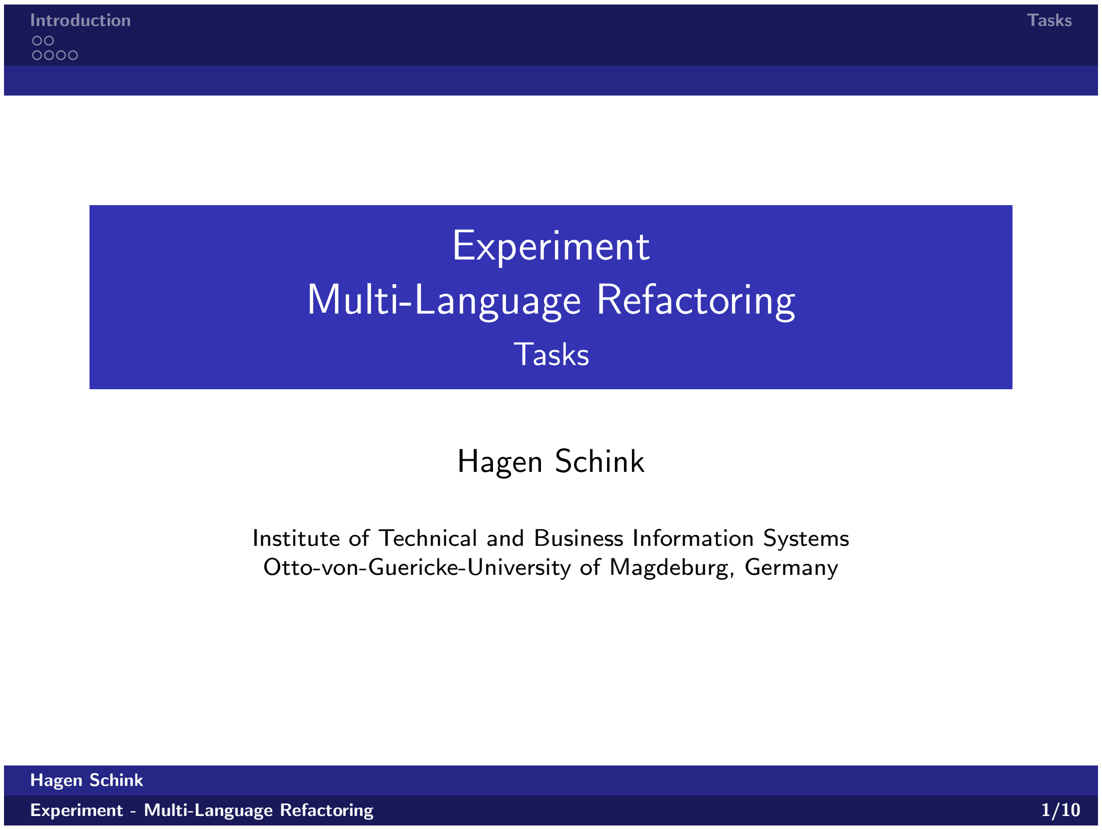


Figure A.10: Package `org.iti.clojureJavaInterfaceVerifier.edges` with edges for connecting elements of Clojure functions.

A.2 Evaluation

A.2.1 Introduction



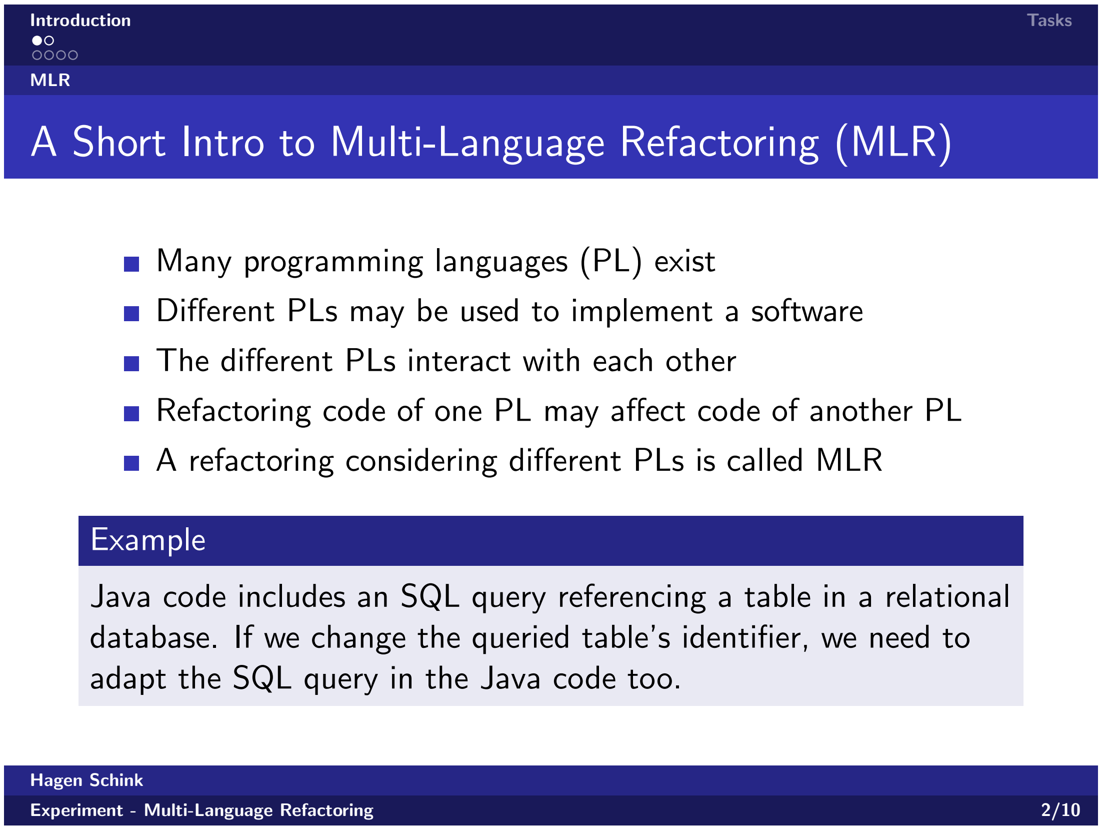
Introduction Tasks

Experiment
Multi-Language Refactoring
Tasks

Hagen Schink

Institute of Technical and Business Information Systems
Otto-von-Guericke-University of Magdeburg, Germany

Hagen Schink
Experiment - Multi-Language Refactoring 1/10



Introduction ●○ ○○○○ Tasks

MLR

A Short Intro to Multi-Language Refactoring (MLR)

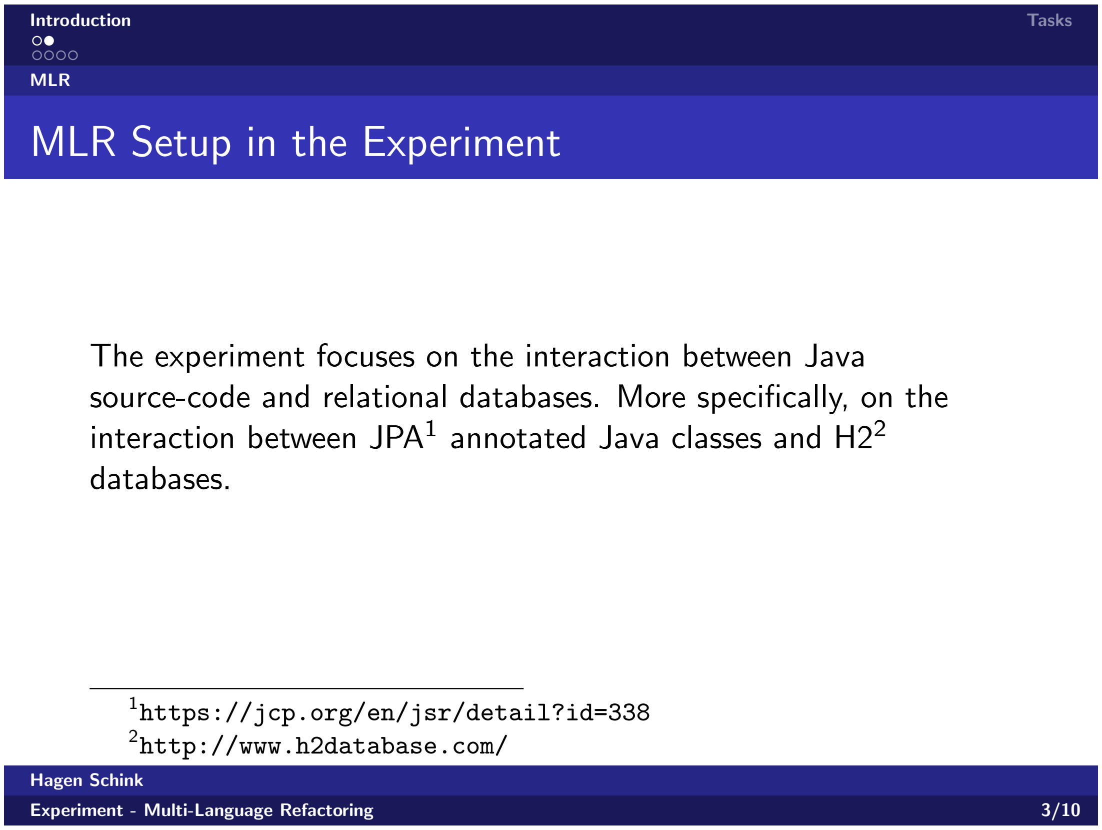
- Many programming languages (PL) exist
- Different PLs may be used to implement a software
- The different PLs interact with each other
- Refactoring code of one PL may affect code of another PL
- A refactoring considering different PLs is called MLR

Example

Java code includes an SQL query referencing a table in a relational database. If we change the queried table's identifier, we need to adapt the SQL query in the Java code too.

Hagen Schink

Experiment - Multi-Language Refactoring 2/10



Introduction ○● ○○○○ Tasks

MLR

MLR Setup in the Experiment

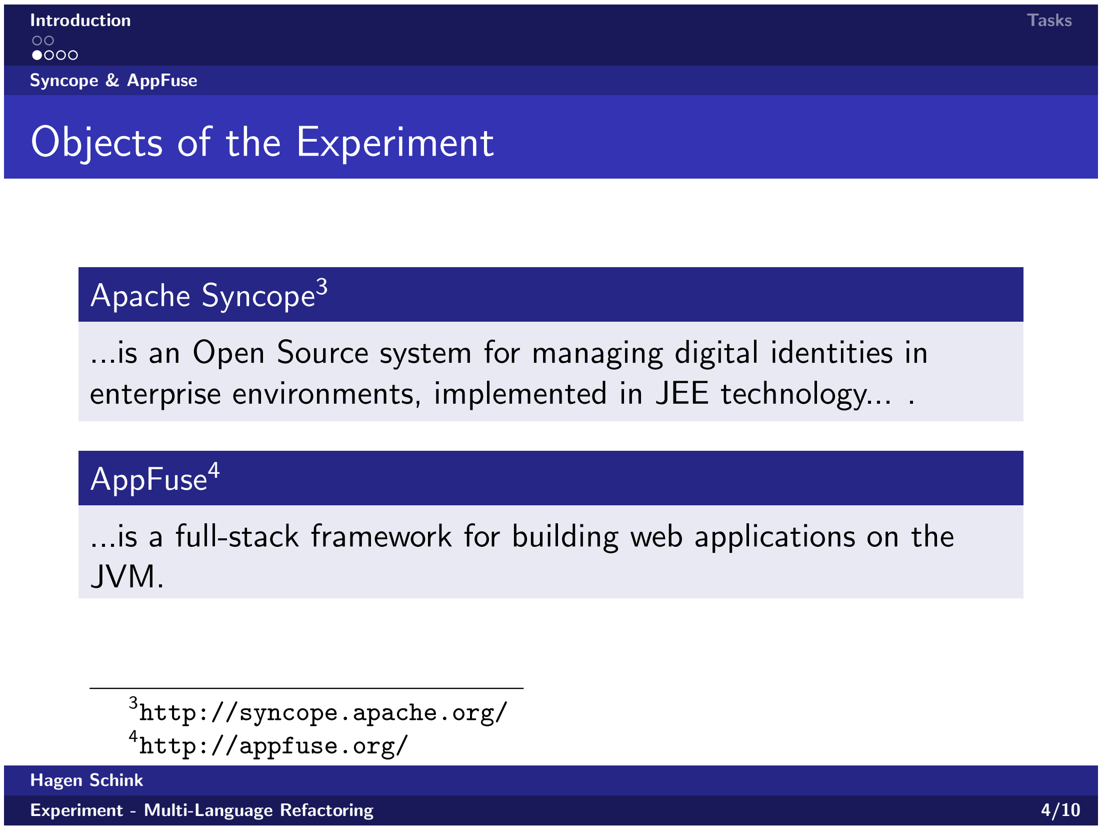
The experiment focuses on the interaction between Java source-code and relational databases. More specifically, on the interaction between JPA[1] annotated Java classes and H2[2] databases.

[1] `https://jcp.org/en/jsr/detail?id=338`

[2] `http://www.h2database.com/`

Hagen Schink

Experiment - Multi-Language Refactoring 3/10



Introduction Tasks

Syncope & AppFuse

Objects of the Experiment

Apache Syncope[3]

...is an Open Source system for managing digital identities in enterprise environments, implemented in JEE technology... .

AppFuse[4]

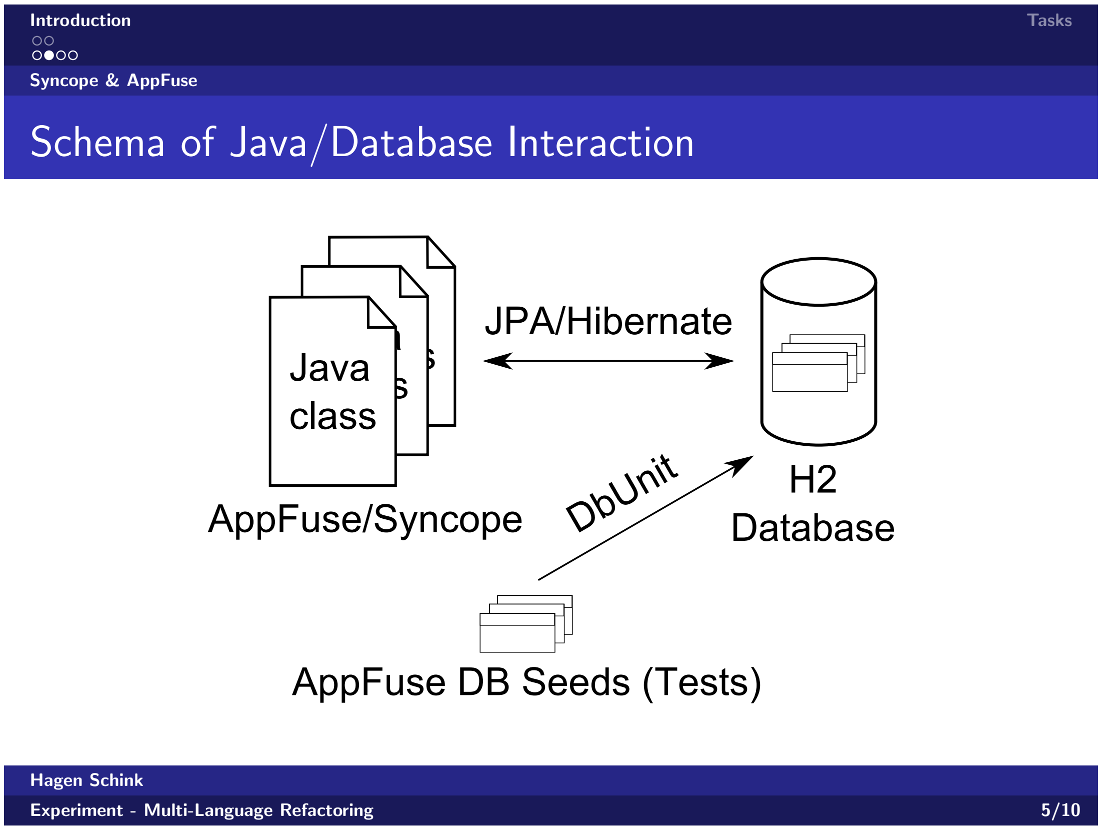
...is a full-stack framework for building web applications on the JVM.

[3] `http://syncope.apache.org/`

[4] `http://appfuse.org/`

Hagen Schink

Experiment - Multi-Language Refactoring 4/10



Introduction Tasks

Syncope & AppFuse

Schema of Java/Database Interaction

Hagen Schink

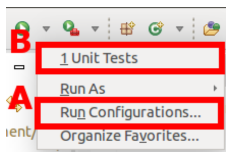
Experiment - Multi-Language Refactoring 5/10

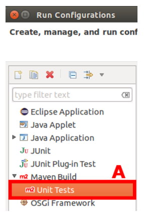
A short Introduction to JPA

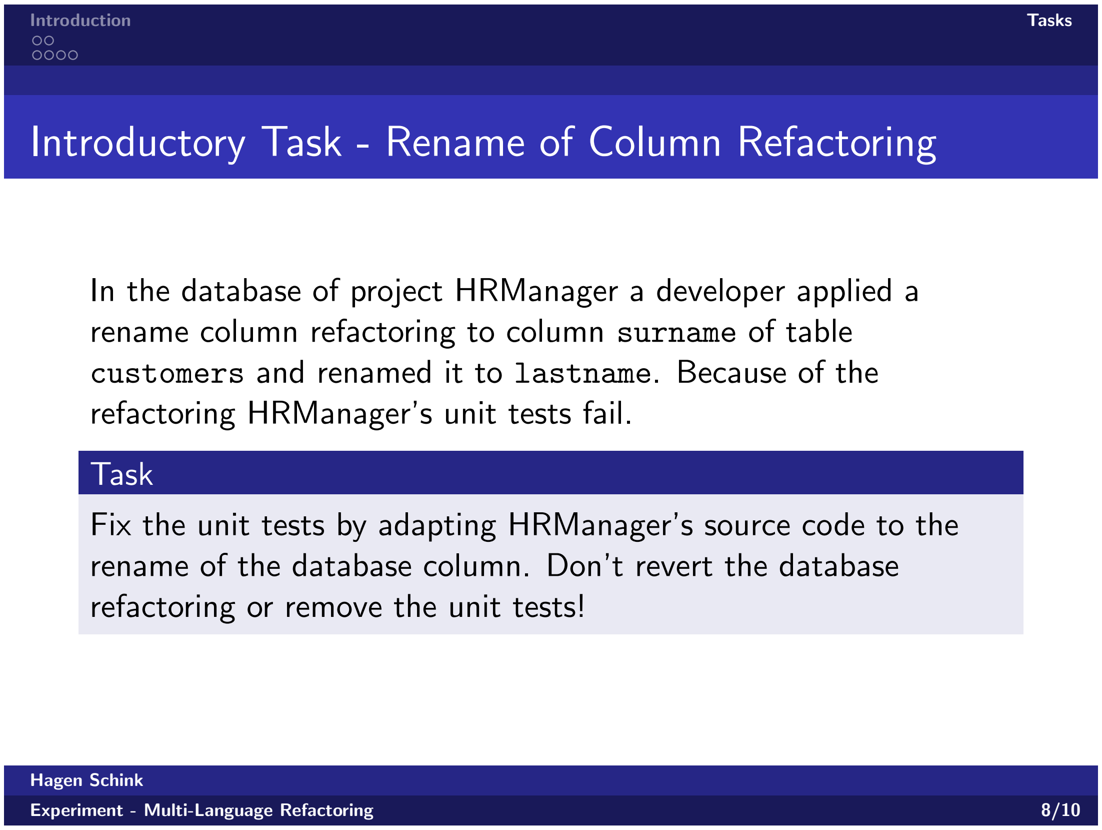
- A Java class annotated with `@Entity` is persisted to a table
- The mapped table's name is...
  - either derived from the class name
  - or set with the `@Table` annotation
- Persisted class' getter/setter methods map to table columns
- Methods annotated with `@Transient` are ignored
- Getter/Setter method identifiers must match (`getName`/`setName`)
- The mapped column's name is...
  - either derived from the method name (`getName` → `name`)
  - or set with the `@Column` annotation

Organization of Tasks and how to run Unit Tests

- Each task has a link on the desktop
- Each link starts a pre-configured Eclipse
- You start unit tests using the *Run* menu, more specifically
  - A the run configuration dialog
  - B the run history







Introduction Tasks

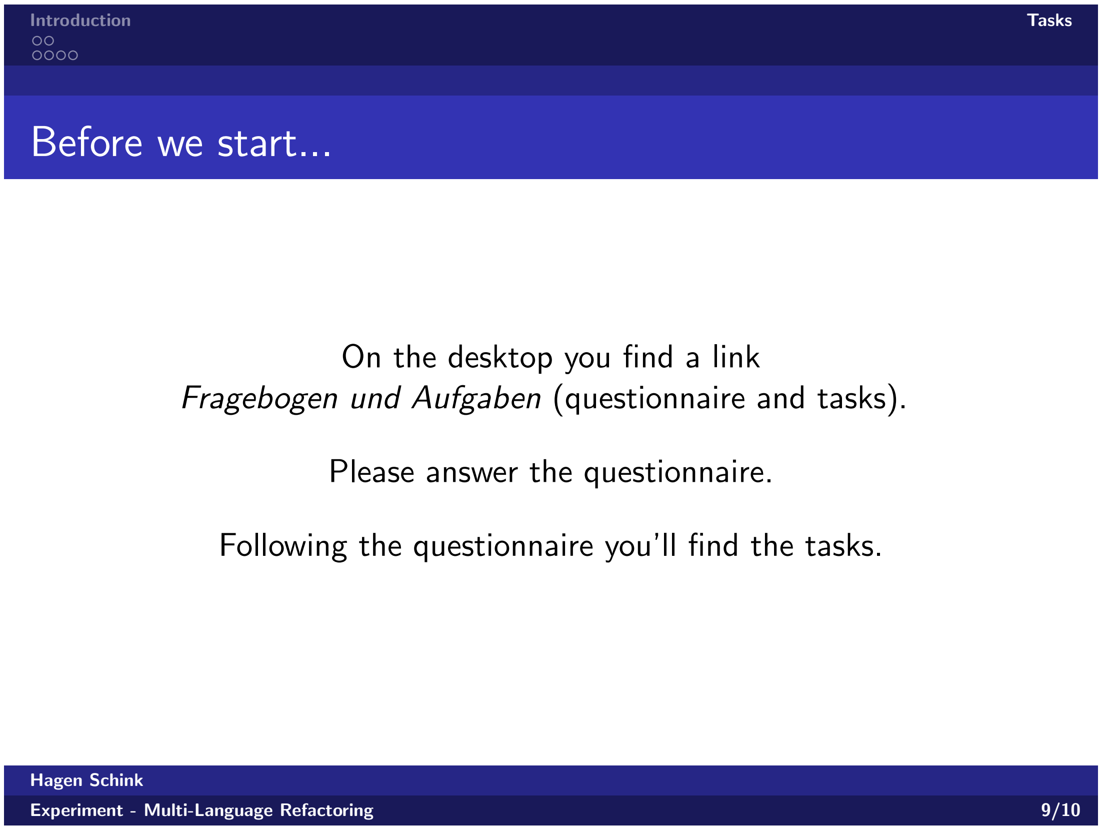
Introductory Task - Rename of Column Refactoring

In the database of project HRManager a developer applied a rename column refactoring to column `surname` of table `customers` and renamed it to `lastname`. Because of the refactoring HRManager's unit tests fail.

Task

Fix the unit tests by adapting HRManager's source code to the rename of the database column. Don't revert the database refactoring or remove the unit tests!

Hagen Schink
Experiment - Multi-Language Refactoring 8/10



Introduction Tasks

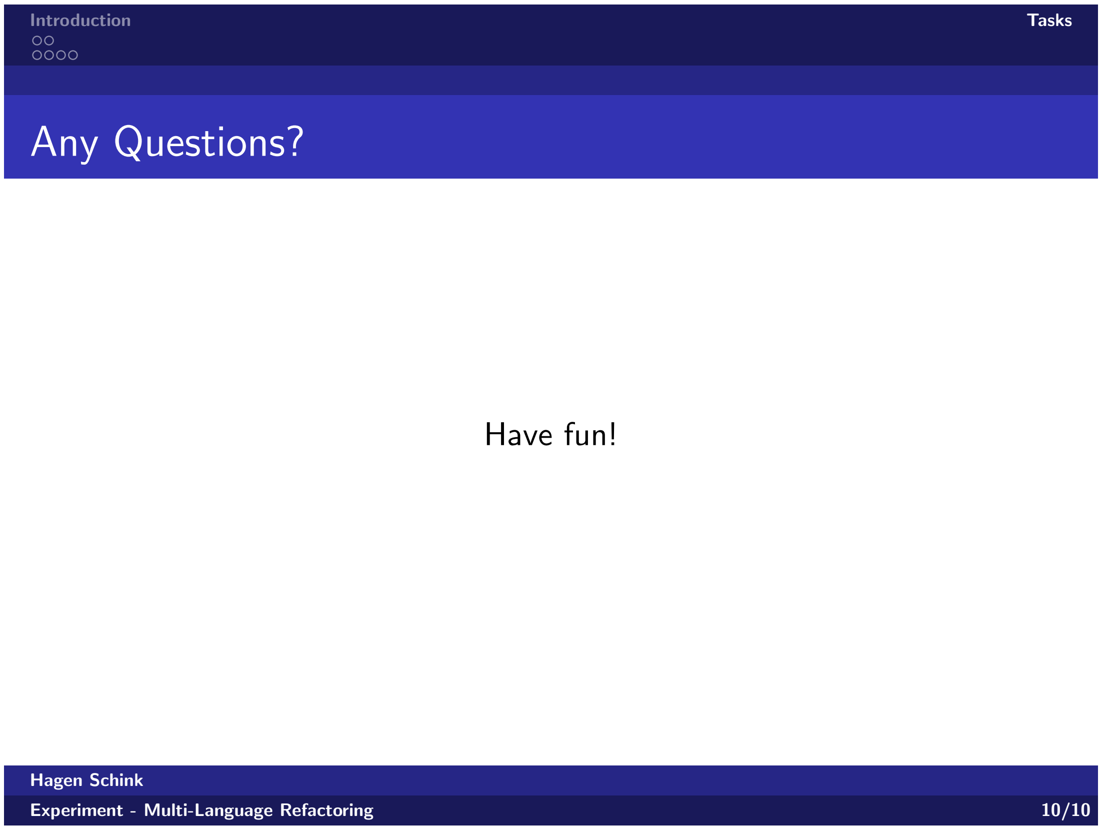
Before we start...

On the desktop you find a link
Fragebogen und Aufgaben (questionnaire and tasks).

Please answer the questionnaire.

Following the questionnaire you'll find the tasks.

Hagen Schink
Experiment - Multi-Language Refactoring 9/10

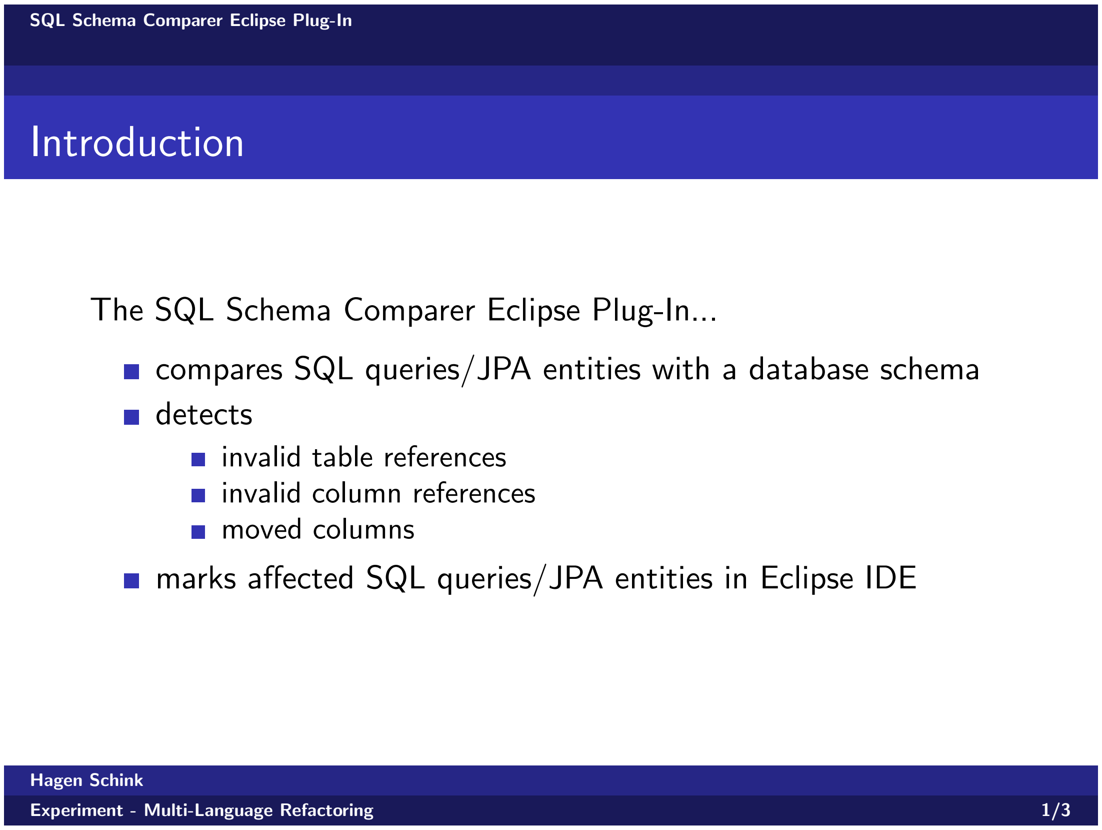


Introduction Tasks

Any Questions?

Have fun!

Hagen Schink
Experiment - Multi-Language Refactoring 10/10



SQL Schema Comparer Eclipse Plug-In

Introduction

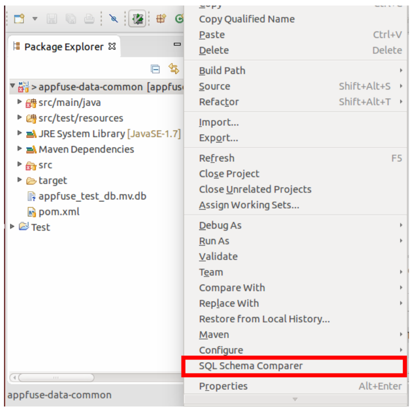
The SQL Schema Comparer Eclipse Plug-In...

- compares SQL queries/JPA entities with a database schema
- detects
  - invalid table references
  - invalid column references
  - moved columns
- marks affected SQL queries/JPA entities in Eclipse IDE

Hagen Schink
Experiment - Multi-Language Refactoring 1/3

Usage

Activate the SSCEP by using the switch in the context menu



Notes about the Plug-In Behavior

If you close Eclipse while the plug-in is activated, you need to set trigger the switch in the context menu twice on the next start.

A.2.2 Questionnaire

	Question	Scale
Q1.	How old are you?	
Q2.	What year did you enroll at university?	
Q3.	Gender	Female or Male
Q4.	What is your major?	
Q5.	How do you estimate your programming experience?	1 (very little) to 10 (very much)
Q6.	For how many years have you been programming?	
Q7.	How many courses did you take in which you had to implement source code?	
Q8.	How experienced are you with the following languages? (Java, C, Haskell, Prolog)	0 (very inexperienced) to 4 (very experienced)
Q9.	How many additional languages do you know (medium experience or better)?	
Q10.	How experienced are you with the following programming paradigms? (logical, functional, imperative, object-oriented)	0 (very inexperienced) to 4 (very experienced)
Q11.	Are you or have you been working on a larger programming project in a company or at the university?	Yes or No
Q12.	For how many years have you been programming for larger software projects?	
Q13.	What domain were these projects related to (Databases, Operating Systems, Web Applications)?	
Q14.	How large were the professional projects typically (in lines of code)?	< 900; $900 - 40,000$; $> 40,000$
Q15a.	How do you estimate your programming experience compared to your class mates?	0 (considerably worse) to 4 (considerably better)
Q15b.	How do you estimate your programming experience compared to to experts with 20 years of practical experience?	0 (considerably worse) to 4 (considerably better)

Table A.2: Questions in the questionnaire as defined by [Feigenspan et al., 2012].

A.2.3 Tasks

Task 1 - Rename Column Refactoring

The Syncope team decided to rename the column `CHANGEPWDDATE` of table `SYNCOPEUSER` to `CHANGEPASSWORDDATE` to make the purpose of the column easier to understand. One of the developers went ahead and changed the column name already in the development database. Because of the renaming the Java application's unit tests fail.

Prerequisite

Start Eclipse with `C:\path\to\eclipse\executable\for\Task\1`

Task

Fix the unit tests by adapting Syncope's source code to the rename of the database column. Don't revert the database refactoring or remove the unit tests!

Task 2 - Move Column Refactoring

Because of an error during the creation of the AppFuse database the colum `POSITION` was initially created in table `ROLE` and not in table `APP_USER`. Another developer already moved the column and also adapted the unit tests to the new database schema. But now the adapted unit tests fail because the actual application code remained unchanged.

Prerequisite

Start Eclipse with `C:\path\to\eclipse\executable\for\Task\2`

Task

Adapt AppFuse's source code to the rename of the database column. Please note: The unit tests are already adapted to the new database schema and you must not change them! Don't revert the database refactoring or remove the unit tests!

Task 3 - Change Java Attribute's Type

In the AppFuse database the table `APP_USER` has a column `WEBSITE` that holds an URL to the user's website. Nevertheless, this column is treated as plain string in AppFuse's Java code. As H2's Structured Query Language (SQL) dialect has no special type for URLs, we need to ensure the validity of the URL in the Java code.

Prerequisite

Start Eclipse with `C:\path\to\eclipse\executable\for\Task\3`

Task

Change the type of the Java class attribute corresponding to column WEBSITE of table APP_USER to class URL and adapt the corresponding code and unit tests if necessary.

A.2.4 Data

Participant	Year of Matriculation	Semester	Tool Support	Experience	Is Experienced	Task 1	Task 2	Task 3
1	2013	3	1	727	0		477	627
2	2014	7	1	0	0	595		570
3	2013	3	1	2181	1	916	622	260
4	2013	3	1	727	0	767	759	208
5	2013	3	1	727	0	744	783	798
6	2013	3	1	727	0	543		633
7	2013	3	1	2181	1		629	626
8	2012	5	1	0	0	763	795	446
9	2013	3	1	2181	1		478	408
10	2013	3	1	1454	1	776	637	481
11	2012	5	1	1454	1	1209	496	278
12	2012	5	1	727	0		650	594
13	2013	3	1	1454	1	695	673	963
14	2013	3	1	727	0	800	557	426
15	2014	7	1	727	0		699	
16	2013	3	1	1454	1		561	
17	2012	5	1	1454	1		500	487
18	2012	5	1	727	0	726	400	374
19	2013	3	1	1454	1		714	584
20	2014	7	1	727	0	581		691
21	2013	3	1	1454	1	849	613	767
22	2013	3	1	727	0	624		636
23	2012	5	1	727	0	891	588	586
24	2014	7	1	727	0	926	547	502
25	2013	3	1	727	0	523	1178	
26	2013	3	1	2181	1	612	1019	630

Participant	Year of Matriculation	Semester	Tool Support	Experience	Is Experienced	Task 1	Task 2	Task 3
27	2013	3	1	1454	1	670	664	785
28	2008	13	1	1454	1	975	779	759
29	2013	3	1	2181	1	442	476	313
30	2013	3	1	727	0	958	499	285
31	2012	5	1	727	0	613	247	349
32	2013	3	1	1454	1	250	393	387
33	2012	5	1	727	0	501	567	552
34	2012	5	1	727	0	446	476	768
35	2013	3	1	0	0	470	440	541
36	2013	3	1	2181	1	474	283	406
37	2012	5	1	2181	1		808	481
38	2012	5	1	1454	1	680	767	1149
39	2014	7	1	727	0	578	752	744
40	2013	3	1	727	0	743	1160	593
41	2013	3	1	1454	1			626
42	2012	5	0	1454	1	703	669	902
43	2014	7	0	727	0	1193	484	539
44	2012	5	0	1454	1		714	612
45	2012	5	0	1454	1	1295	315	368
46	2014	7	0	727	0	663	982	779
47	2009	11	0	727	0	1445	481	720
48	2014	7	0	727	0	1374	560	739
49	2013	3	0	1454	1	174	436	791
50	2011	7	0	727	0			
51	2013	3	0	2908	1	1240	284	349
52	2014	7	0	727	0	1119	685	501
53	2013	3	0	727	0	1153	619	500
54	2012	5	0	727	0	883	1007	511

Participant	Year of Matriculation	Semester	Tool Support	Experience	Is Experienced	Task 1	Task 2	Task 3
55	2013	3	0	2181	1	983	702	685
56	2013	3	0	727	0	1000	758	881
57	2011	7	0	727	0	530	512	492
58	2013	3	0	727	0	831		386
59	2012	5	0	1454	1	609	985	430
60	2014	7	0	727	0	850	658	648
61	2014	7	0	1454	1	872	266	333
62	2013	3	0	2181	1	714	698	449
63	2014	7	0	727	0	716	626	861
64	2012	5	0	2181	1	513	411	600
65	2012	5	0	727	0	780	1450	
66	2014	7	0	1454	1	814		481
67	2012	5	0	727	0	958	911	528
68	2013	3	0	1454	1	667	468	634
69	2012	5	0	2181	1	617	326	400
70	2013	3	0	2181	1	767	408	954
71	2012	5	0	1454	1	1317	1073	564
72	2013	3	0	1454	1	1005	560	640
73	2012	5	0	1454	1	454	356	736
74	2013	3	0	0	0	459	481	671
75	2013	3	0	1454	1	785	767	604
76	2013	3	0	727	0	1081	901	743
77	2012	5	0	1454	1	537	533	621
78	2012	5	0	1454	1	488	483	1054
79	2014	7	0	0	0	1245	789	652

Table A.4: Base data used for the analysis.

Bibliography

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811. (cited on Page 40)

Scott Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, Inc., New York, NY, USA, 2003. ISBN 9780471202837. (cited on Page 1, 20, 25, 26, 28, and 100)

Scott Ambler and Pramodkumar Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, Boston, MA, USA, 2006. ISBN 9780321293534. (cited on Page 26, 95, and 100)

Anderson, Lance. JSR 211, JDBC 4.1 Specification. http://download.oracle.com/otndocs/jcp/jdbc-4.0-fr-eval-oth-JSpec/, 2006. (cited on Page 7 and 100)

Ittai Balaban, Frank Tip, and Robert M. Fuhrer. Refactoring Support for Class Library Migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), San Diego, CA, USA, October 16-20, 2005*. ACM, 2005. (cited on Page 6)

Fabian Bannwart and Peter Müller. Changing Programs Correctly: Refactoring with Specifications. In *Proceedings of the 14th International Symposium on Formal Methods (FM 2006), Hamilton, Canada, August 21-27, 2006*. Springer, 2006. (cited on Page 95)

Philip A. Bernstein and Erhard Rahm. Data Warehouse Scenarios for Model Management. In *Proceedings of the 19th International Conference on Conceptual Modeling (ER 2000), Salt Lake City, Utah, USA, October 9-12, 2000*. Springer, 2000. (cited on Page 47)

Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Specifying Integrated Refactoring with Distributed Graph Transformations. In *Proceedings of the 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003), Charlottesville, VA, USA, September 27 - October 1, 2003*. Springer, 2003. (cited on Page 98)

A. Broder. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences (SEQUENCES 1997), Salerno, Italy, June 11 - 13, 1997*. IEEE Computer Society, 1997. (cited on Page 106)

Donald T. Campbell and Julian C. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Rand McNally College Publishing, Skokie, IL, USA, 1963. ISBN 9780395307878. (cited on Page 91)

Nicholas Chen and Ralph E. Johnson. Toward Refactoring in a Polyglot World Extending Automated Refactoring Support Across Java and XML. In *Proceedings of the 2nd ACM Workshop on Refactoring Tools (WRT 2008), Nashville, TN, USA, October 19, 2008*. ACM, 2008. (cited on Page 1, 2, 7, 13, 19, 34, 39, and 99)

Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1), 1990. (cited on Page 6)

Anthony Cleve. *Program Analysis and Transformation for Data-Intensive System Evolution*. PhD thesis, University of Namur, Namur, Belgium, October 2009. (cited on Page 100)

Steve Counsell, Robert M. Hierons, Rajaa Najjar, George Loizou, and Youssef Hassoun. The Effectiveness of Refactoring, Based on a Compatibility Testing Taxonomy and a Dependency Graph. In *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART 2006), Windsor, United Kingdom, 29-31 August 2006*. IEEE Computer Society, 2006. (cited on Page 6)

Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated Testing of Refactoring Engines. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. ACM, 2007. (cited on Page 6)

Daniel Delorey, Charles D. Knutson, and Christophe Giraud-Carrier. Programming Language Trends in Open Source Development: An Evaluation Using Data from All Production Phase SourceForge Projects. In *Proceedings of the 2nd International Workshop on Public Data about Software Development (WoPDaSD 2007), Limerick, Ireland, June 11-14, 2007*. ACM, 2007. (cited on Page 1 and 105)

Danny Dig. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, Champaign, IL, USA, 2007. (cited on Page 95)

Danny Dig, Can Comertoglu, Darko Marinov, and Ralph E. Johnson. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP 2006), Nantes, France, July 3-7, 2006*. Springer, 2006. (cited on Page 106)

Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET 2000), Limerick, Ireland, June 4-11, 2000*, 2000. (cited on Page 95 and 96)

EPFL - École Polytechnique Fédérale de Lausanne. A Scala Tutorial for Java Programmers – Interaction with Java. http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html#interaction-with-java, 2015. Accessed: 2017-01-30. (cited on Page 8)

EPFL - École Polytechnique Fédérale de Lausanne. Getting Started – Compile it! http://www.scala-lang.org/documentation/getting-started.html#compile-it, 2017. Accessed: 2017-01-30. (cited on Page 8)

Sebastian Thore Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, University of Marburg, Marburg, Germany, November 2012. (cited on Page 8)

Janet Feigenspan, Norbert Siegmund, Andreas Hasselberg, and Markus Köppen. PROPHET : Tool Infrastructure To Support Program Comprehension Experiments. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM 2011), Banff, Alberta, Canada, September 22-23, 2011*. IEEE Computer Society, 2011. (cited on Page 85)

Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring Programming Experience. In *Proceedings of the 20th International Conference on Program Comprehension (ICPC 2012), Passau, Germany, June 11-13, 2012*. IEEE Computer Society, 2012. (cited on Page 77, 79, 103, and 132)

Hans-Christian Fjeldberg. Polyglot Programming. Master's thesis, Norwegian University of Science and Technology, Trondheim, Norwegian, June 2008. (cited on Page 6 and 7)

Neil Ford. *The Productive Programmer*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2008. ISBN 9780596519780. (cited on Page 1, 6, and 7)

Martin Fowler. *Refactoring: Improving the Design of existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 9780201485677. (cited on Page 1, 5, 6, 21, 31, 95, and 96)

Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. *ACM Transactions on Programming Languages and Systems*, 30(4), 2008. (cited on Page 99)

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Springer Berlin Heidelberg, Berlin, Germany, 1993. ISBN 9783540479109. (cited on Page 29)

Google Inc. Working with JSNI. https://developers.google.com/eclipse/docs/gwt_jsni, 2017. Accessed: 2017-01-11. (cited on Page 36)

Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards Automating Source-Consistent UML Refactorings. In *Proceedings of the 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML 2003), San Francisco, CA, USA, October 20-24, 2003*. Springer, 2003. (cited on Page 96 and 98)

Mark Grechanik, Don S. Batory, and Dewayne E. Perry. Design of Large-Scale Polylingual Systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), Edinburgh, United Kingdom, May 23-28 2004*. IEEE Computer Society, 2004. (cited on Page 1)

William G. Griswold and David Notkin. Automated Assistance for Program Restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3), 1993. (cited on Page 1)

Mike Grogan. JSR-223, Scripting for the Java ™ Platform. https://jcp.org/aboutJava/communityprocess/final/jsr223/index.html, 2006. (cited on Page 7)

Sebastian Günther. Multi-DSL Applications with Ruby. *IEEE Software*, 27(5), 2010. (cited on Page 8)

Jean-Luc Hainaut. Specification Preservation in Schema Transformations - Application to Semantics and Statistics. *Data & Knowledge Engineering*, 19(2), 1996. (cited on Page 20 and 100)

Elliote Rusty Harold and W. Scott Means. *XML in a nutshell*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2002. ISBN 9780596002923. (cited on Page 7)

Jan Heering, P. R. H. Hendriks, Paul Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11), 1989. (cited on Page 8)

Rich Hickey. Ahead-of-time Compilation and Class Generation. https://clojure.org/reference/compilation, 2017a. Accessed: 2017-01-30. (cited on Page 8)

Rich Hickey. Ahead-of-time Compilation and Class Generation. https://clojure.org/reference/libs, 2017b. Accessed: 2017-01-30. (cited on Page 8)

ISO/IEC 9075-1:2008. Plastics – Determination of fracture toughness – Linear elastic fracture mechanics (LEFM) approach. Standard, International Organization for Standardization, Geneva, Switzerland, July 2008. (cited on Page 7)

ISO/IEC/IEEE 24765:2010(E). Systems and software engineering – Vocabulary. Standard, International Organization for Standardization, Geneva, Switzerland, December 2010. (cited on Page 1)

JetBrains s.r.o. Refactorings for HTML. https://www.jetbrains.com/help/resharper/2016.3/Refactorings_for_HTML.html, 2017. Accessed: 2017-01-11. (cited on Page 36)

T. Capers Jones. *Estimating Software Costs*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1998. ISBN 9780079130945. (cited on Page 1)

Mike Keith and Merrick Schincariol. *Pro EJB 3: Java Persistence API (Pro)*. Apress, Berkely, CA, USA, 2006. ISBN 9781590596456. (cited on Page 11)

Martin Kempf, Reto Kleeb, Michael Klenk, and Peter Sommerlad. Cross language refactoring for Eclipse plug-ins. In *Proceedings of the 2nd ACM Workshop on Refactoring Tools (WRT 2008) Nashville, TN, USA, October 19, 2008*. ACM, 2008. (cited on Page 36, 39, and 99)

Bernt Kullbach, Andreas Winter, Peter Dahm, and Jürgen Ebert. Program Comprehension in Multi-Language Systems. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE '98), Honolulu, Hawai, USA, October 12-14, 1998*. IEEE Computer Society, 1998. (cited on Page 1)

Ralf Lämmel. Towards Generic Refactoring. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Rule-based Programming (RULE 2002), Pittsburgh, PA, USA, October 5, 2002*. ACM, 2002. (cited on Page 6 and 97)

Ralf Lämmel. Coupled Software Transformations. In *Proceedings of the 1st International Workshop on Software Evolution Transformations, Delft, Netherlands, November 9, 2004*. IEEE Computer Society, 2004. (cited on Page 97 and 100)

Ralf Lämmel. Google's MapReduce programming model - Revisited. *Science of Computer Programming*, 70(1), 2008. (cited on Page 30)

Ralf Lämmel. Coupled software transformations revisited. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, Netherlands, October 31 - November 1, 2016*. ACM, 2016. (cited on Page 97)

M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1980. (cited on Page 1)

Huiqing Li. *Refactoring Haskell Programs*. PhD thesis, University of Kent, Canterbury, Kent, United Kingdom, September 2006. (cited on Page 6, 31, and 95)

Huiqing Li and Simon J. Thompson. Tool support for refactoring functional programs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2008), San Francisco, California, USA, January 7-8, 2008*. ACM, 2008. (cited on Page 1 and 95)

Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 9780201325775. (cited on Page 7)

Panos K Linos, Whitney Lucas, Sig Myers, and Ezekiel Maier. A Metrics Tool for Multi-Language Software. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications (SEA 2006), Dallas, TX, USA, November 13-15, 2006*. ACTA Press, 2006. (cited on Page 1 and 6)

Hans-Wolfgang Loidl, Fernando Rubio, Norman Scaife, Kevin Hammond, Susumu Horiguchi, Ulrike Klusik, Rita Loogen, Greg Michaelson, Ricardo Pena, Steffen Priebe, Álvaro J. Rebón Portillo, and Philip W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-Order and Symbolic Computation*, 16(3), 2003. (cited on Page 30)

Philip Mayer and Alexander Bauer. An Empirical Analysis of the Utilization of Multiple Programming Languages in Open Source Projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering (EASE 2015), Nanjing, China, April 27-29, 2015*. ACM, 2015. (cited on Page 1, 7, and 105)

Philip Mayer and Andreas Schroeder. Cross-Language Code Analysis and Refactoring. In *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2012), Riva del Garda, Italy, September 23-24, 2012*. IEEE Computer Society, 2012. (cited on Page 1, 6, 13, 36, 39, 95, 100, and 103)

Philip Mayer and Andreas Schroeder. Patterns of Cross-Language Linking in Java Frameworks. In *Proceddings of the 21st IEEE International Conference on Program Comprehension (ICPC 2013), San Francisco, CA, USA, May 20-21, 2013*. IEEE Computer Society, 2013. (cited on Page 104)

Philip Mayer and Andreas Schroeder. Automated Multi-Language Artifact Binding and Rename Refactoring between Java and DSLs Used by Java Frameworks. In *Proceedings of the 28th European Conference Object-Oriented Programming (ECOOP 2014), Uppsala, Sweden, July 28 - August 1, 2014*. Springer, 2014. (cited on Page 13, 100, 103, and 104)

Tom Mens. On the Use of Graph Transformations for Model Refactoring. In *Generative and Transformational Techniques in Software Engineering, International Summer School (GTTSE 2005), Braga, Portugal, July 4-8, 2005*. Springer, 2005. (cited on Page 6)

Tom Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2), 2004. (cited on Page 6)

Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing Refactorings with Graph Transformations. *Journal of Software Maintenance*, 17(4), 2005. (cited on Page 95)

J.-E. Michels, K. Kulkarni, M. C. Farrar, A. Eisenberg, N. Mattos, and H. Darwen. The SQL Standard. *it — Information Technology*, 45, 2003. (cited on Page 7)

Gail C. Murphy, Mik Kersten, and Leah Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4), 2006. (cited on Page 66)

Emerson R. Murphy-Hill and Andrew P. Black. Breaking the Barriers to Successful Refactoring : Observations and Tools for Extract Method. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. ACM, 2008. (cited on Page 95 and 102)

Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), Vancouver, Canada May 16-24, 2009*. IEEE, 2009. (cited on Page 21, 101, and 102)

Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP 2012), Beijing, China, June 11-16, 2012*. Springer, 2012. (cited on Page 107)

Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A Comparative Study of Manual and Automated Refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP 2013), Montpellier, France, July 1-5, 2013*. Springer, 2013. (cited on Page 102 and 107)

Object Management Group (OMG). OMG Unified Modeling Language Specification Version 1.3. OMG Document Number formal/2000-03-01 (http://doc.omg.org/formal/2000-03-01.pdf), 2000. (cited on Page 96)

Object Management Group (OMG). OMG Unified Modeling Language Specification Version 1.4. OMG Document Number formal/2001-09-67 (http://doc.omg.org/formal/2001-09-67.pdf), 2001. (cited on Page 96)

Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification Version 1.1. OMG Document Number formal/2011-01-01 (http://www.omg.org/spec/QVT/), 2011. (cited on Page 100)

William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Champaign, IL, USA, 1992. (cited on Page 1, 5, 6, 21, 95, and 96)

Oracle America, Inc. JSR-338, Java Persistence API, Version 2.1. http://download.oracle.com/otndocs/jcp/persistence-2_1-fr-eval-spec/index.html, 2013. (cited on Page 100)

Rolf-Helge Pfeiffer and Andrzej Wąsowski. Taming the Confusion of Languages. In *Proceedings of the 7th European Conference Modelling Foundations and Applications (ECMFA 2011), Birmingham, United Kingdom, June 6 - 9, 2011*. Springer, 2011. (cited on Page 95 and 101)

Rolf-Helge Pfeiffer and Andrzej Wąsowski. Cross-Language Support Mechanisms Significantly Aid Software Development. In *Proceedings of the 15th International Conference Model Driven Engineering Languages and Systems (MODELS 2012), Innsbruck, Austria, September 30 - October 5, 2012*. Springer, 2012a. (cited on Page 6, 13, 102, 103, and 104)

Rolf-Helge Pfeiffer and Andrzej Wąsowski. TexMo: A Multi-language Development Environment. In *In Proceedings of the 8th European Conference Modelling Foundations and Applications (ECMFA 2012), Kgs. Lyngby, Denmark, July 2-5, 2012*. Springer, 2012b. (cited on Page 6, 13, 95, 101, 102, and 104)

Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The International Journal on Very Large Data Bases*, 10(4), 2001. (cited on Page 46 and 47)

Red Hat Inc. and the various authors. Chapter 1. Setting up an annotations project. https://docs.jboss.org/hibernate/stable/annotations/reference/en/html/ch01.html, 2004. Accessed: 2017-01-26. (cited on Page 11)

Lukas Renggli. *Dynamic Language Embedding With Homogeneous Tool Support*. PhD thesis, University of Berne, Berne, Switzerland, October 2010. (cited on Page 8, 9, and 96)

Romain Robbes. Mining a Change-Based Software Repository. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR 2007), Minneapolis, MN, USA, May 19-20, 2007*. IEEE Computer Society, 2007. (cited on Page 107)

Romain Robbes and Michele Lanza. SpyWare: a change-aware development toolset. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. ACM, 2008. (cited on Page 107)

Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, Champaign, IL, USA, 1999. (cited on Page 95 and 96)

Suzanne Robertson and James Robertson. *Mastering the Requirements Process: Getting Requirements Right*. Addison-Wesley Professional, Boston, MA, USA, 2012. ISBN 9780321815743. (cited on Page 1)

Hagen Schink. sql-schema-comparer: Support of Multi-Language Refactoring with Relational Databases. In *Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2013), Eindhoven, Netherlands, September 22-23, 2013*. IEEE Computer Society, 2013. (cited on Page 3 and 53)

Hagen Schink and Martin Kuhlemann. Hurdles in refactoring multi-language programs. Technical Report FIN-007-2010, University of Magdeburg, Germany, November 2010. (cited on Page 3, 6, and 19)

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 Data Technologies (ICSOFT 2011), Seville, Spain, July 18-21, 2011*. SciTePress, 2011. (cited on Page 3, 6, 14, and 19)

Hagen Schink, David Broneske, Reimar Schröter, and Wolfram Fenske. A Tree-Based Approach to Support Refactoring in Multi-Language Software Applications. In *Proceedings of the 2nd International Conference on Advances and Trends in Software Engineering, Lisbon, Portugal, February 21-25, 2016*. IARIA, 2016a. (cited on Page 3, 6, 39, and 53)

Hagen Schink, Janet Siegmund, Reimar Schröter, Thomas Thüm, and Gunter Saake. A Study on Tool Support for Refactoring in Database Applications. *Softwaretechnik-Trends*, 36(2), 2016b. (cited on Page 3 and 75)

Tim Sheard. Accomplishments and Research Challenges in Meta-programming. In *Proceedings of the 2nd International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2001), Florence, Italy, September 6, 2001*. Springer, 2001. (cited on Page 8)

Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics IV*, 3730(1), 2005. (cited on Page 46)

Sixto Ortiz Jr. Computing Trends Lead to New Programming Languages. *Computer*, 45, 2012. (cited on Page 8)

Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Generating Unit Tests for Checking Refactoring Safety. In *Proceedings of the 13th Brazilian Symposium on Programming Languages (SBLP 2009), Gramado, RS, Brazil, August 19-21, 2009*. n.p., 2009. (cited on Page 6)

Gustavo Soares, Melina Mongiovi, and Rohit Gheyi. Identifying overly strong conditions in refactoring implementations. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011), Williamsburg, VA, USA, September 25-30, 2011*. IEEE Computer Society, 2011. (cited on Page 95)

Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with KABA. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), October 24-28, 2004, Vancouver, BC, Canada*. ACM, 2004. (cited on Page 6 and 95)

D. Strein, R. Lincke, J. Lundberg, and W. Löwe. An extensible meta-model for program analysis. *IEEE Transactions on Software Engineering*, 33(9), 2007. (cited on Page 95, 97, and 98)

Dennis Strein, Hans Kratz, and Welf Löwe. Cross-Language Program Analysis and Refactoring. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), Philadelphia, Pennsylvania, USA, September 27-29, 2006*. IEEE Computer Society, 2006. (cited on Page 1, 6, 13, 36, 39, 95, 97, and 98)

Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of api refactorings in libraries. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. ACM, 2007. (cited on Page 95, 106, and 107)

Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Deep Typechecking and Refactoring. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008), Nashville, TN, USA, October 19-23, 2008*. ACM, 2008. (cited on Page 13, 14, 36, and 99)

Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, Berne, Switzerland, December 2001. (cited on Page 95 and 96)

Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. FAMIX: Exchange Experiences with CDIF and XMI. In *Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000), Limerick, Ireland, June 4-11, 2000*. ACM, 2000. (cited on Page 96)

Federico Tomassetti and Marco Torchiano. An empirical assessment of polyglot-ism in GitHub. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014), London, England, United Kingdom, May 13-14, 2014*, 2014. (cited on Page 1, 105, and 106)

Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer, Berlin, Germany, 2002. ISBN 9783540435501. (cited on Page 14 and 44)

Luke VanderHart. *Practical Clojure*. Apress, Berkely, CA, USA, 2010. ISBN 9781430272311. (cited on Page 13 and 33)

Antonio Vetro, Federico Tomassetti, Marco Torchiano, and Maurizio Morisio. Language Interaction and Quality Issues: An Exploratory Study. In *Proceedings of the 6th ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2012), Lund, Sweden, September 19-20, 2012*. ACM, 2012. (cited on Page 1 and 105)

Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA 2001), Utrecht, Netherlands, May 22-24, 2001*. Springer, 2001. (cited on Page 8)

Lin Weisheng. Type Inference in SQL. Master's thesis, Concordia University, Montréal, Québec, Canada, April 2004. (cited on Page 61)

Peter Weißgerber and Stephan Diehl. Are Refactorings Less Error-prone Than Other Changes? In Stephan Diehl, Harald C. Gall, and Ahmed E. Hassan, editors, *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006), Shanghai, China, May 22-23, 2006*. ACM, 2006. (cited on Page 21)

Michael M Werner. *Facilitating Schema Evolution With Automatic Program Transformations*. PhD thesis, Northeastern University, Boston, MA, USA, July 1999. (cited on Page 96)

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Germany, 2012. ISBN 9783642290435. (cited on Page 76 and 91)

Zhenchang Xing and Eleni Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, November 7-11, 2005*. ACM, 2005. (cited on Page 95 and 106)

Zhenchang Xing and Eleni Stroulia. The JDEvAn Tool Suite in Support of Object-Oriented Evolutionary Development. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. ACM, 2008. (cited on Page 95 and 106)

E h r e n e r k l ä r u n g

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.“

Magdeburg, den 14.05.2017

Hagen Schink