Checking Metadata Usage for Enterprise Applications

Yaxuan Zhang

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Application

Na Meng, Chair
Eli Tilevich
Francisco Servant

May 06, 2021
Blacksburg, Virginia

Keywords: Software, Domain Specific Language, XML, Annotation, Spring
Copyright 2021, Yaxuan Zhang

Checking Metadata Usage for Enterprise Applications

Yaxuan Zhang

(ABSTRACT)

It is becoming more and more common for developers to build enterprise applications on top of Spring framework or other other Java frameworks. While the developers are enjoying the convenient implementations of web frameworks, developers should pay attention to configuration deployment with metadata usage (i.e., Java annotations and XML deployment descriptors). Different formats of metadata can correspond to each other. Metadata usually exist in multiple files. Maintaining such metadata is challenging and time-consuming. Current compilers and tools rarely inspect the XML files, not to say the corresponding relationship between Java annotations and XML files. To help developers ensure the quality of metadata, this work presents a Domain Specific Language, RSL, and its engine, MeEditor. RSL facilitates pattern definition for correct metadata usage. MeEditor can take in specified rules and check Java projects for any rule violations.

Developer can define rules with RSL considering the metadata usage. Then, developers can run RSL script with MeEditor to detect rule violations. 9 rules were extracted from Spring specification and are written in RSL. To evaluate MeEditor, we mined 180 plus 500 open-source projects from Github. To evaluate the effectiveness and usefulness of MeEditor, we conducted our evaluation by taking two steps. First, we evaluated the effectiveness of MeEditor by constructing a known ground truth data set (total 180 projects). Based on experiments of ground truth data set, MeEditor can identified the metadata misuse. MeEditor detected bug with 94% precision, 94% recall, 94% accuracy. Second, we evaluate the usefulness of MeEditor by applying it to real world projects (total 500 projects). For the

latest version of these 500 projects, MeEditor gave 79% precision according to our manual inspection. We applied MeEditor to the version histories of rule-adopted projects. MeEditor identified 23 bugs, which later fixed by developers.

Checking Metadata Usage for Enterprise Applications

Yaxuan Zhang

(GENERAL AUDIENCE ABSTRACT)

It is becoming more and more common for developers to build enterprise applications on top of frameworks. While the developers are enjoying the convenient implementations of web frameworks, developers should pay attention to configuration deployment with metadata usage. Maintaining such metadata is challenging and time-consuming. Current compilers and tools do not provide sufficient support on inspecting metadata usage. To help developers ensure the quality of metadata, this work presents a Domain Specific Language, RSL, and its engine, MeEditor. RSL facilitates pattern definition for correct metadata usage. MeEditor can take in specified rules and check projects for any rule violations.

Developer can define rules with RSL considering the metadata usage. Then, developers can run RSL script with MeEditor to detect rule violations. 9 rules were extracted from Spring specification and are written in RSL. To evaluate MeEditor, we mined 180 plus 500 open-source projects from Github. To evaluate the effectiveness and usefulness of MeEditor, we conducted our evaluation by taking two steps. First, we evaluated the effectiveness of MeEditor by constructing a known ground truth data set (total 180 projects). Based on experiments of ground truth data set, MeEditor can identified the metadata misuse with 94% accuracy. Second, we evaluate the usefulness of MeEditor by applying it to real world projects (total 500 projects). MeEditor identified 23 suspicious updates of metadata, which later fixed by developers.

Dedication

To my dad and mom, Qiang Zhang and Guiping Zhang

Acknowledgments

First and most important, I would like to thank my advisor, Dr. Na Meng, for her countless support and advice on this work. Under her advices, I improved my critical thinking ability and problem-solving skills, which are very important if one would like to do high-quality research. She also spent a lot of time on helping me improve my scientific writing, which will benefit for my future career development.

I would like to thank my committee members, Dr. Eli Tilevich, Dr. Francisco Servant for their insightful suggestions, constant patience, and constant support. And I would also like to express my grateful thanks to all the professors and staff members in Computer Science Department. Special thanks to Sharon KinderPotter for giving countless support when I encounter problems. I would like to thank all my former and present colleagues in Computer Science department, Bowen Shen, Yan Zhao, Ying Zhang, Mahir Kabir, Sheikh Shadab Towqir, Sheik Murad Hassan, Tung Dao, Rahul Agarwal, Zijian Jiang, Pronnoy Goswami , Chengyuan Wen for their help and support. They make my research life here in Virginia Tech more enjoyable.

Last but not least, I would like to thank my parents and all my friends for their love, patience, and support. Thank you to Shengchang Lu, for all his love, support, and understanding.

Contents

List of Figures x

List of Tables xi

1 Introduction 1

2 Background 4

2.1 Enterprise Applications . . . 4

2.1.1 Web Frameworks . . . 4

2.1.2 IoC Container . . . 5

2.1.3 Dependency Injection . . . 7

2.2 Metadata formats . . . 8

2.2.1 XML-based Metadata . . . 9

2.2.2 Annotation-based and Java-based Metadata . . . 11

2.3 Problem Statement . . . 14

3 Approach 15

3.1 Domain-Specific Language – RSL . . . 15

3.1.1 Design . . . 15

3.2 Automated Engine – MeEditor 19
3.2.1 Implementation 19

4 Rules 23
4.1 Spring ClassPathXmlApplicationContext 24
4.2 Method Existence 26
4.3 <property> Tags and Fields Correspondence 29
4.4 Usage of <context:annotation-config> in XML file 34
4.5 Usage of @ImportResource Annotation 36
4.6 <constructor-arg> Tags and Fields Correspondence 37
4.7 Bean Existence 42

5 Evaluation 45
5.1 Data Sets 45
5.2 Metrics 46
5.3 Effectiveness of Bug Detection with Ground Truth Data Set 47
5.3.1 Bug Detection 48
5.4 Effectiveness and Usefulness of Bug Detection and Fix in Real World Projects 49
5.4.1 Bug Detection for Latest Version 50
5.4.2 Fixed Bugs in History Commits 53

6 Threats To Validity 60

7 Related Works 62

7.1 Domain Specific Language – PBSE 62

7.1.1 PBSE 62

7.2 Existing Researches 63

8 Conclusion 64

Bibliography 66

Appendices 72

Appendix A RSL Core Syntax 73

Appendix B Fix Bugs in History Commits 78

List of Figures

2.1 Spring Ioc Container. 7

3.1 MeEditor implementation. 20

List of Tables

5.1 Rule Examination in 180 Projects. 49

5.2 Rule Examination in 500 Projects. 50

5.3 The 23 erroneous rule use fixed by developers. 59

Listings

2.1 Create a container using XML configuration metadata 6

2.2 Create a container using annotation-based configuration metadata 6

2.3 A simple beans.xml file. 10

2.4 An example of Annotation-based configuration metadata usage. 11

2.5 Equivalent XML configuration example to the List 2.4. 12

2.6 Import annotation into XML configuration. 13

2.7 Import XML configurations into annotation-based configuration. 13

3.1 Main key features of RSL. 17

3.2 Key expressions for inspecting source codes in RSL. 18

3.3 An example of tree building using JJTree in JavaCC. 20

4.1 Example of ClassPathXmlApplicationContext usage. 24

4.2 Pattern of Spring ClassPathXmlApplicationContext API usage 25

4.3 Example of Method Attribute Usage [32]. 26

4.4 Example of init-method and destroy-method [26]. 27

4.5 Example of init-method and destroy-method in corresponding Java class . . . 27

4.6 Pattern of Method Existence 28

4.7 XML configuration of setter-based DI in Spring. 30

4.8 Java file of setter-based DI in Spring. 30

4.9 Pattern of <property> name attribute and field name check 32

4.10 Pattern of <property> name attribute and setter method existence check . . 33

4.11 Usage of <context:annotation-config> in XML configuration. 34

4.12 Pattern of <context:annotation-config> usage 35

4.13 Example of importing XML configuration into Annotation configuration . . 36

4.14 Pattern of importing XML configuration into Annotation configuration . . . 37

4.15 Java code example of constructor-based DI in Spring project 38

4.16 XML configuration example of constructor-based DI in Spring project with type specification 38

4.17 Pattern of <constructor-arg> type attribute and field type check 39

4.18 XML configuration example of constructor-based DI in Spring project with name specification 40

4.19 Pattern of <constructor-arg> name attribute and field name check 41

4.20 Examples of retrieving bean. 42

4.21 Pattern of bean existence check 43

5.1 Changes of project calltag with context-annotation-config-annotation error . 55

5.2 XML file of project unikkit with field-property-map error. 56

5.3 Changes of project unikkit with field-property-map error. 56

5.4 Changes of project featurepub with setter-property-map error. 57

5.5 Changes of project pocket-review with method-exist error. 58
B.1 Changes of project calltag with context-annotation-config-annotation error . 78
B.2 Changes of project calltag with field-property-map and setter-property-map errors 79
B.3 Changes of project contract with context-annotation-config-annotation error 83
B.4 Changes of project LIBRARY with context-annotation-config-annotation error 84
B.5 Changes of project mvc-component-tester with context-annotation-config-annotation error 84
B.6 Changes of project ndx-jersey2 with context-annotation-config-annotation error 85
B.7 Changes of project NewBlog with context-annotation-config-annotation error 86
B.8 Changes of project scaffolding with context-annotation-config-annotation error 87
B.9 Changes of project tmm_phonebook 88
B.10 Changes of project tutorial with context-annotation-config-annotation error . 88
B.11 Changes of project WSSpring with context-annotation-config-annotation error 89
B.12 Changes of project mentor-project-2013 with field-property-map and setter-property-map error 90
B.13 Changes of project soundService with field-property-map error. 91
B.14 Changes of project unikkit with field-property-map error. 93
B.15 XML file of project unikkit with field-property-map error. 94
B.16 Changes of project wl-oauth with field-property-map error. 95

B.17 Changes of project wl-solr-search with field-property-map error. 96

B.18 Changes of project pocket-review with method-exist error. 97

B.19 Changes of project pocket-review with setter-property-map error. 99

B.20 Changes of project featurepub with setter-property-map error. 100

B.21 Changes of project spatialsearch with setter-property-map error. 100

Chapter 1

Introduction

When building at-scale applications with software frameworks (e.g. Spring, Java EE platforms), developers often use metadata to configure the system [31]. Metadata is usually in XML or annotation-based formats. Wrong usage of metadata can result in abnormal runtime behaviors [39] or confusing errors [15]. Debugging such metadata can be challenging and time-consuming for two reasons. First, metadata can be located at any Java class or XML file. At mean time, some metadata usage might follow the same rule. It is tedious and error-prone for developers to check the metadata usage by scanning the Java classes and XML files line by line. Second, current compilers and research tools mostly examine source code instead of XML files. Although XML validators can be built to validate syntax based on XML Schemas or DTDs, the validators do not semantically examine the correspondence between XML files and Java class.

Existing researches provide quite limited support for checking metadata usage (i.e. XML-based and annotation-based configuration) [49, 50, 51]. For example, XQuery is a domain-specific language (DSL), which provides features to find elements and attributes in XML documents [49]. Tools are also built to check the annotation usage [50, 51]. These research works defined individual DSLs for users to specify and convert the constraints to XQuery path expressions. Then, validate the constraints on annotation usage. It can be time-consuming for some developers to learn more than one DSL in order to validate the

constraints on both XML-based and annotation-based metadata formats. We believe that with a simpler approach to (1) define rules considering constrains in both Java classes and XML files and (2) apply those rule definitions to reveal erroneous metadata usages, we can provide quality assurance for metadata usages without the needs of much human effort.

This thesis presents a domain-specific language, RSL, and its engine, MeEditor. RSL facilitates pattern/rule definition for correct metadata usage, including both XML-based and annotation-based configuration. MeEditor can take in specified rules and check Java projects for any rule violations.

For evaluation, we manually extracted 9 rules from Spring specification. Furthermore, based one our observation, we built a ground truth data set in 180 projects. We followed the mechanisms (discussed in Section 5.3) to fairly construct the ground truth data set. For each rule, we randomly assigned 20 projects (in those 180 projects). Our evaluation shows that on average, MeEditor detected bugs with 94% precision, 94% recall, 94% accuracy.

Additionally, we applied MeEditor to latest commit in another 500 open-source projects. MeEditor reported that there are totally 443 usages of these 9 rules in 500 projects. MeEditor revealed that 80 incorrectly updated Java source code or XML file, 63 of which were identified by manual inspection. We later applied MeEditor to history commits in projects which contain 363 (comes from 443-80) usages of these 9 rules. These 363 usages are reported as matched usage and identified as correct usage. Among these 363 usages, MeEditor revealed 23 incorrectly updated Java source code or XML DD.

In summary, this thesis makes the following contributions:

- We designed a domain-specific language, RSL, which is used to define rules of correct metadata usage (i.e. XML DD and annotations) in enterprise-level applications. Different from most prior DSL, it can prescribe rules of both XML DD and annotations at the same time.
- We developed an engine, MeEditor, which can take in specified rules and check projects for any rule violations. MeEditor can detect rule violations with high accuracy (94%).
- We conducted a comprehensive evaluation on MeEditor. Our evaluation indicates that MeEditor detected incorrect metadata usage, which developers applied fixes to the incorrect updates.

The rest of this thesis is structured as follows. In Chapter 2, we introduce the background of our research. Next, in Chapter 3, we explain the approach of our research, including the design of RSL and the implementation of MeEditor. Afterwards, in Chapter 4, we present the 9 rules in RSL, which were extracted from Spring specification. Then, in Chapter 5, we show our evaluation results of these 9 rules in both ground truth data set and open-source projects. Finally, in Chapter 6 and 7, we discuss the threats to validity of this work and the related works, respectively.

Chapter 2

Background

2.1 Enterprise Applications

2.1.1 Web Frameworks

Enterprise applications is one of the the major success of Java. Many developers would like to choose Java as their primary programming language when they build enterprise-level applications [47]. Especially, when the enterprise application is large-scale. The adoption of framework would save time and effort for developers. They provide interfaces for working with databases, for communicating with front-end services, for distributed setting, and many others technologies.

There are many frameworks that can be used to develop enterprise applications, such as Spring Boot, Spring MVC, JSF, Struts, and so on. Among these frameworks, Spring Boot has been rated as the most popular web framework[58]. 40% developers use Spring Boot to write their backend web applications. 36% developers in the survey use Spring MVC in their web applications. In 2020, the Spring framework has been listed as the most popular java framework for building backend web applications [55]. Clearly, the Spring frameworks is the best solution for developing at-scale web applications. Especially for beginners, it is much easier for them to utilize the framework instead of reinventing the wheels.

Web frameworks always use Dependency Injection (DI) to emphasize the abstraction. The goal of abstraction is try to be as general as it can, when the framework is applied by all kinds of applications [47]. Especially, since enterprise-level projects often heavily depend on the web frameworks these days. The dynamic techniques have been employed to achieve this goal in many frameworks. DI is one of the popular dynamic techniques. To implement DI, it would arise the needs for developers to configure the metadata of applications.

When building at-scale web applications with web frameworks, there are always needs for deploying metadata configurations. Generally, there are two major metadata formats. First one is XML configurations, which developers usually create deployment descriptors (e.g. application-context.xml, beans.xml) to configure deployment options in both Spring [10] and JavaEE projects [1]. The second one is annotation-base configurations, which developers use annotations (e.g. @Bean, @Autowired, @Configuration) provided by the frameworks to deploy configurations. It is not necessary that developers can only apply one of them to their applications. Developer can also use both of them with importing the other configuration details Since Spring framework is much more popular than others in building Enterprise web application nowadays, this work will focus more on Spring related configuration introductions.

2.1.2 IoC Container

Inversion of Control (IoC) is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. It is often used in object-

oriented programming (OOP) [4]. In Spring/Spring Framework, it implements the Inversion of Control [10]. When running the application, the IoC container creates beans/objects according to metadata configuration like XML-based or annotation-based metadata. Meanwhile, the IoC container also inject dependencies or establishing communication between its components following the metadata definition. Those dependencies always include the essential information of dependent object instances, which current bean is working with. The process is shown in Figure 2.1. As we can see from the figure, all configuration metadata and objects will be read by the Spring IoC container. After processing all the configurations, the fully configured application is generated. The package of org.springframwork.beans, org.springframwork.context are the packages that provide basic functionalities toward this dependency injection. org.springframwork.beans is built for manipulating Java beans [35]. org.springframework.context is built based on the beans package, which provides supporting functionalities in obtaining or applying resources. For example, the ApplicationContext is the central interface that providing the methods for configuring an application. In List 2.1 and List 2.2, they are examples of instantiating a Spring IoC container with XML-based and Java-based metadata formats, respectively. In List 2.1, the created container reads XML-based configuration specified inside file applicationContext.xml. In List 2.2, the created container reads annotation-based configuration specified in the Class Config.

Listing 2.1: Create a container using XML configuration metadata

```
1       ApplicationContext context
2  = new ClassPathXmlApplicationContext("applicationContext.xml");
```

Listing 2.2: Create a container using annotation-based configuration metadata

```
1       ApplicationContext context = new AnnotationConfigApplicationContext(
          ↪ Config.class);
```

IoC is also known as another highly frequent used name, dependency injection (DI). However, DI is only one of the patterns or mechanisms that can implement the IoC. There are other patterns or mechanisms, including Strategy design pattern, Service Locator pattern, and Factory pattern. Since DI is the one that Spring is primarily using, DI will be primarily discuss here.

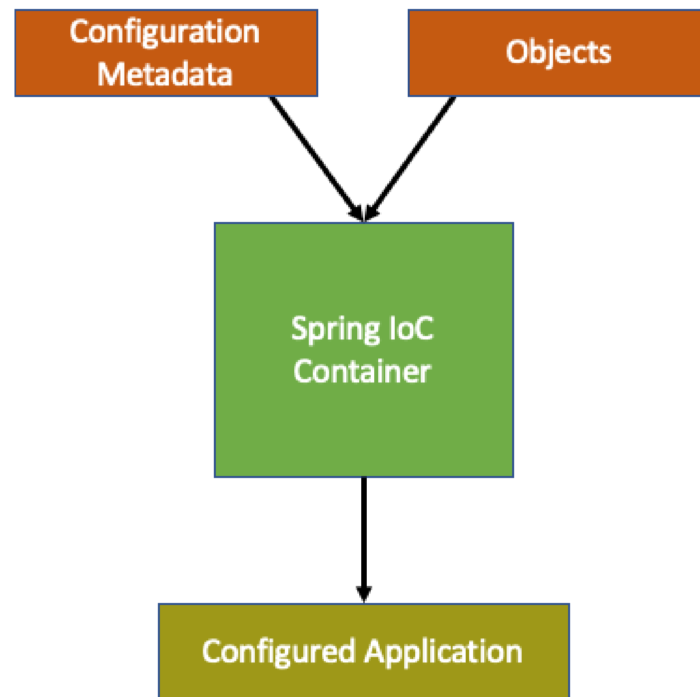

Figure 2.1: Spring Ioc Container.

2.1.3 Dependency Injection

Dependency Injection (DI) is a highly frequently used technique in enterprise-level programming. DI is a technique that can be used in many programming languages, such as Java, C#, PHP, and many others [18]. Dependency injection containers are usually offered by

web frameworks. For example, in Spring, we have the Spring IoC container. Using DI can provide many benefits. It provides loose coupling instead of tight coupling when there is one component depending on other components. For example, component A depends on component B. Instead of instantiating a new object into A that establishes a connection to B, the configuration metadata can connects the two by injecting the dependency into A. The object of B is created by Spring and injected for the application according to the configuration definition. It is more convenient to have the loose coupling and it can be achieved by introducing DI. It also provides convenience during the testing. For example, a developer is testing a class that is related to a database object. It is a good practice to create a mock object when testing a class. It is useful because developers would not worry about influencing the database.

2.2 Metadata formats

There are two major formats of metadata in Spring. One is XML-based metadata and the other one is annotation-based metadata. Nowadays, developers prefer annotation-based metadata. However, XML-based metadata still has its place in Spring applications. Many projects in this thesis are containing XML-based metadata, as we will discuss them in Section 5.4. XML-based metadata can offer a clear separation between beans/objects and its behaviors. It is usually easier to search objects in XML-based configuration than search in all source code.

2.2.1 XML-based Metadata

XML is a eXtensible Markup Language that helps store and transport data [46]. XML is built for storing and transporting data. Deployment Descriptor (DD) is often used to store those information. DD is a special kind of XML files to configure deployment options [2].

XML has its own syntax, which it defines how an XML file can be written [13]. According to the syntax rules, each XML file can define one or more XML elements, which are organized in a tree structure. Specifically, there is only one root element in any XML file, and the root element can have one or more child elements. For example, List 2.3 is an simple example of beans.xml file. <beans> is the root. The urls in its attributes are schema, which defines the rules and restrictions of constructing this XML metadata. The root <beans> has its child elements, which are three <bean>s in List 2.3. In this beans.xml file, it creates three beans/instances, which are beans with id ”mathObjectValue”, ”scienceObjectValue”, and ”stu”, respectively. This XML-based metadata establishes dependency of bean ”stu” on bean ”mathObjectValue” by indicating it under the ”ref” attribute. It is possible that XML files locate in various file folders and named differently (e.g., beans.xml, applicationContext.xml...). The location and the name of XML file are depending on the types of applications and modules, Developers usually specify the location at the start application class, where they instantiate the Spring IoC container.

In general, an XML document consists of markups and data. Markups are provided in the form of tags and attributes. Data is usually input as values for attributes. XML elements are represented by tags. An element usually consists of an opening tag (e.g., <bean ...>), a closing tag (e.g. </bean>). Between the tags, there are attributes and the value for the at-

tributes (e.g., id = ”stu”), child elements between the its tags (e.g., <property>). An element can also has only a single tag with a ”/” at the end (e.g., <bean id = ”mathObjectValue” class=”spring.fieldsMatching.correct.Math”/>). The ”/” at the end is a substitution for the closing tag. The legal format of XML file is defined in XML schema file [12]. The XML schema defines the number and the data type of elements and attributes that can be stated in the XML file, the number of child elements, default and fixed values for elements and attributes. For example, Spring framework has its own XML schema, which is specified at the root of beans.xml in List 2.3.

Listing 2.3: A simple beans.xml file.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="
        ↪ http://www.springframework.org/schema/context"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5     http://www.springframework.org/schema/beans/spring-beans.xsd
6     http://www.springframework.org/schema/context
7     http://www.springframework.org/schema/context/spring-context.xsd">
8
9   <bean id = "mathObjectValue" class="spring.fieldsMatching.correct.Math"/>
10  <bean id ="scienceObjectValue" class="spring.fieldsMatching.correct.
        ↪ Science"/>
11
12   <bean id="stu" class ="spring.fieldsMatching.correct.Student">
13    <property name="action" ref="mathObjectValue"/>
14    <property name="studentname" value="Yaxuan Zhang"/>
```

```
15     <property name="id" value="2"/>
16   </bean>
17 </beans>
```

2.2.2 Annotation-based and Java-based Metadata

XML-based metadata is not the only allowed form of configuration metadata. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. These days many developers choose Java-based/Annotation-based configuration for their Spring applications [10]. Annotation-based and Java-based configuration were introduced after the XML configuration metadata. Developers can use @Configuration annotation with a class to indicate Spring IoC container that this class is served as a source of bean definitions. Developers can use @Bean annotation with a method to indicate that the object returned by this method should be registered as a bean in the Spring IoC container [40]. In List 2.4, the class AnnotationExample is annotated with @Configuration annotation, which indicates that this class contains bean definition. The method hello() is annotated with @Bean annotation, which indicates the object returned by this method should be considered as a bean. It can give equivalent configuration as a XML configuration as shown in the List 2.5. Both of the configuration define a bean of class Hello. By default, the @Bean will use the return type (with first letter change to lowercase) as its id. In this case, it would be "hello", which is the same as the XML-configuration definition of this bean id. One can have multiple Java files that are annotated with @Configuration as one can have multiple XML-defined beans in XML-based configuration files.

Listing 2.4: An example of Annotation-based configuration metadata usage.

```
1 package examples.annotation;
```

```
2 import org.springframework.context.annotation.*;
3
4 @Configuration
5 public class AnnotationExample {
6    @Bean(name="hello")
7    public Hello hello(){
8       return new Hello();
9    }
10 }
```

Listing 2.5: Equivalent XML configuration example to the List 2.4.

```
1 <beans>
2    <bean id = "hello" class = "example.annotation.Hello" />
3 </beans>
```

The choice of choosing Java-based configuration or XML configuration is depending on the developers. Both of them can configure application with the same configuration results. The debates of XML or Java-based configuration is still existed. But still, more people think annotation-based or Java-based configuration can provide more flexibility to the application. Along this transition process, spring also provide the function that one can combine the use of XML and annotation-based configuration. In List 2.6, it adds an element $< context : annotation-config >$ to the file and it would enable annotations in beans, which are already defined in the Spring IoC container. In List 4.14, the use of @ImportResource annotation with the class would enable XML configuration in beans, which are already defined in the Spring IoC container. These two methods are the most commonly used methods to combine two metadata formats.

Listing 2.6: Import annotation into XML configuration.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config>
    </context:annotation-config>
<beans/>
```

Listing 2.7: Import XML configurations into annotation-based configuration.

```
package spring.annoImportXML.correct;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;

@Configuration
@ImportResource(locations = { "classpath:spring/annoImportXML/correct/BeanB.
    ↪ xml" })
public class ConfigC {

  @Bean
```

```
12   public BeanC getBeanC() {
13     return new BeanC();
14   }
15
16 }
```

2.3 Problem Statement

When using web frameworks, there are different rules that developers have to follow in order to correctly configure the metadata. At the meantime, the project might use this rule for multiple times in multiple files. It is difficult for developers to inspect all of them one by one and locate them quickly. Especially for the needs of inspecting projects that have both XML-based and annotation-based configuration. It is much better to reveal the potential bugs before the run time. As stated in a research study on StackOverflow [56], numerous developers posted various questions on how to correctly configure web applications (e.g. Spring Boot) and expressed frustration with XML or annotation debugging. The necessity and usefulness of static code analysis tool has been considered for building enterprise applications effectively. However, there is limited tool developed for bug detection for metadata usage. Especially, for bug detection in projects which use two metadata formats at the same time. Many research works only focus on one type of metadata formats. To address the need of efficient analysis of inspecting the metadata, we introduce our domain-specific-language RSL and its engine, MeEditor. We keep the format of RSL as simple as possible for people to quickly pick up. And enable the users to check both metadata formats.

Chapter 3

Approach

3.1 Domain-Specific Language – RSL

In this section, the DSL of this work, RSL, will be introduced. Compared to PBSE, RSL includes more clauses, expressions, and statements. However, the design of RSL still remain concise and straightforward. RSL would be straightforward to developers that have experience in Objective-Oriented Programming (OOP) and Declarative Query Programming. The design of RSL is discussed in Section 3.1.1. In order to let developers use RSL immediately, an automated engine was implemented, called MeEditor. The details of MeEditor will be discussed in Section 3.2.

3.1.1 Design

We keep the design of RSL as simple as possible for easy understanding and learning. In List 3.1, the main features are represented. Key statements include **Rule**, **for**, **where**, **assert**, **msg**. The symbols used in the RSL script is depending on the developer himself or herself. In this case, we add flexibility to the usage of RSL. However, it is important to make sure the symbols are consistent within the pattern script. If it is not consistent, the engine can not check the information correctly or fail to check any information.

It is easy to create an engine-readable RSL script. Simple usage of RSL is discussed in this section. You can always find more complete usage of RSL in 9 rule definitions located at Chapter 4. As a start of constructing a RSL script, user should define the rule name for rule. User can use **Rule** keyword, followed by a customised rule name. User can use **for** statement to instantiate the iterators of desired variables in collections. The engine of RSL will store all the information into a storage pool. The collection can be project, file, class, element. These can be collections since they can have child elements inside them. For examples, in a project, it can have XML files, Java classes; in an XML file, it can have elements; in an element, it can have its child elements; in a class, it can have methods, fields. User can use **where** statement to refined the already existed variable collections. It is important to remember that it will be an error to use it without **for** statement before it. It is straightforward to understand why it is incorrect. Users can no refine variable collections if they do not already exist in the storage pool. Next, user can use **assert** statement to assert the expressions inside the assert statement. Last, based on the assertion result, user can use **msg** statement to print out the bug.

RSL also introduces the substitution feature for **msg** statement. This feature can give a user a fancier output. The first argument in the **msg** statement would be a string literal. It can include modulo operator (%) followed by a symbol. User will also need to apply value/expression to this symbol after this string literal in sequence if more than one symbols were input into the string literal. The engine will recognize this value/expression and substitute the symbol in the string literal with actual value in the error message. For example, in Pattern Method Existence 4.6, user can specify msg("The referenced method name %s in XML is not defined", s). If the project has one method undefined, which called "init". The

engine will print out the error message in the following way. ”The referenced method name init is not defined”.

Listing 3.1: Main key features of RSL.

```
Rule rule_name
    Define a rule name
for ([class | method | string | field | element | file] var in collection)
    An iterator for a collection of program
where (Expressions)
    A statement that refines collection information of the program
assert (Expressions)
    A statement that asserts current information of the program
msg (string literals, vars)
    A statement that prints the error messages
```

Relational operations, **AND** and **OR** are also introduced to RSL. The usage of **AND** is the same as **AND** in SQL or **&&** in Java. The engine ensures the truth of expressions on both side of this **AND**. The usage of **OR** is the same as **OR** in SQL or || in Java. The engine will consider the expression as false only if the expressions on both side of **OR** are false. Otherwise, the engine will consider it as true.

MeEditor also enable 12 methods to let users get started easier, which are listed in List 3.2. Note, these methods are the bottom expressions to the RSL script. They will be executed first and store the information in the temporary storage pool. And then the engine will pass the information to its outer expression or statement. The return value for each method is

listed in the list. If you are interested in knowing more details about this language, the full syntax of RSL is located at Appendix **??**.

Listing 3.2: Key expressions for inspecting source codes in RSL.

```
1  callExists(api):
2      return true if there is certain API "api" called. Otherwise, return
           ↪ false.
3  getArg(api, #arg):
4      return the argument #arg in called API "api".
5  pathExist(path):
6      return true if "path" exists. Otherwise, return false.
7  getFQN(c):
8      return the fully qualified name of class "c".
9  getAttr(eleName, attrName):
10     return the attribute value of attribute "attrName" in element "eleName".
11 upperCase(string):
12     return the "string" in upper case.
13 substring(string, #start, [#end]):
14     return the substring of "string" from #start to #end or from #start to
           ↪ the end of "string" if #end does not present.
15 getAnnoAttr(annotation, attr):
16     return the attribute value of attribute "attr" in "annotation".
17 isEmpty(expression):
18     return true if expression are presented in the source code. Otherwise,
           ↪ return false.
```

```
19 join(expression1, expression2,...):
20     join the expressions as a list.
21 getName(field | method):
22     return the name of field or method.
23 getType(field):
24     return the type of field.
```

3.2 Automated Engine – MeEditor

To enable the use of RSL, an automated engine, MeEditor, was implemented for immediate use. In this section (Section3.2), the implementation of MeEditor will be discussed thoroughly.

3.2.1 Implementation

Figure 3.1 represents the implementation of MeEditor. There are two sources of the inputs. One part is the source codes of project, which needs to be examined. The projects we used for evaluation in this thesis usually contain both Java files and XML files. To offer convenience and completeness to these projects, we enable supports for parsing both Java files and XML files. To parse the Java file, Abstract Syntax Tree (AST) is used. Eclipse JDT is used to implement the AST [28]. It offers great convenience when traversing the tree. The Document Object Model (DOM) API is used to generate tree structure for XML file [19]. DOM is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document.

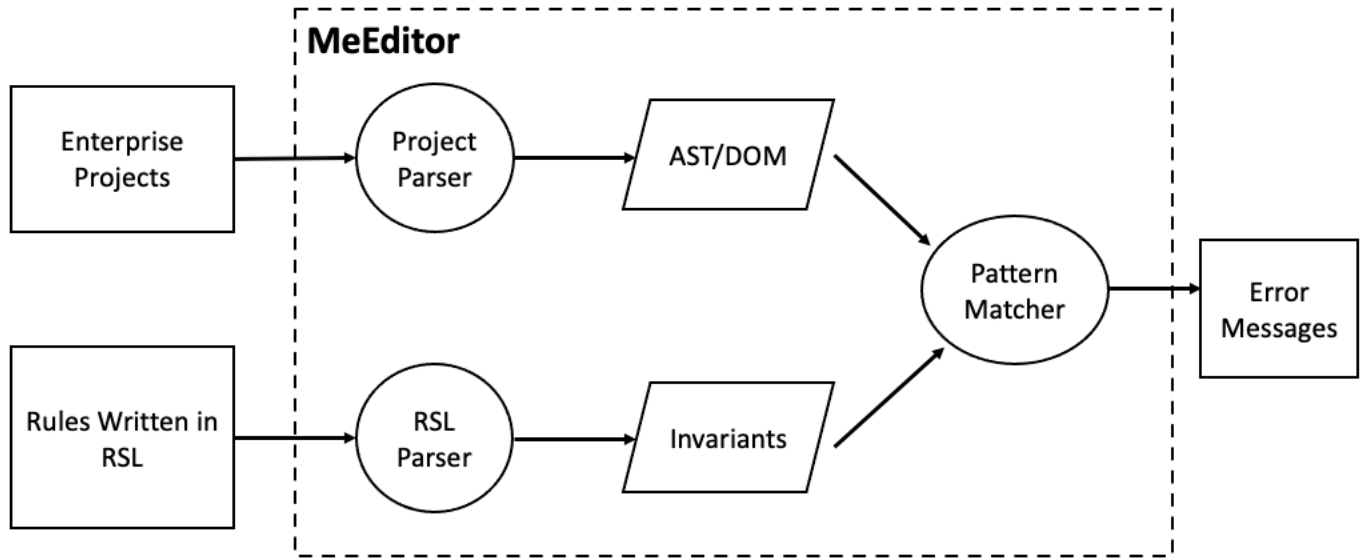

Figure 3.1: MeEditor implementation.

The other part is the input pattern script, which should be written according to RSL specification (specified in Section 3.1 and Appendix A). To generate the parser for RSL, Java Compiler Compiler (JavaCC) is used [27]. JavaCC is the most popular parser generator for use with Java applications. It can not only generate a basic parser but also include other standard capabilities related to parser generation. For example, tree building is a very useful implementation of JavaCC and it is used in this work. Tree building is an essential implementation in this work since the engine needs to traverse the tree to establish the relationship between each command. The tool implementated in JavaCC, JJTree, is the key feature that is used to fulfill the tree building in this work. For example, in List 3.3, it is the generated tree for List 4.14. As we can see in the tree, it contains all essential elements we need to implement the engine. Tree structure will allow us to access parent nodes, child nodes, child nodes number, and so on. Information from the RSL script is passed to the engine. Engine would process the information accordingly.

Listing 3.3: An example of tree building using JJTree in JavaCC.

```
1 Reading from standard input...
2 Start
3  Rule
4   JavaName
5    Identifier: import-resource-path
6   For
7    Class
8     Identifier: c
9    Project
10    Identifier: p
11   For
12    String
13     Identifier: s
14    GetAnnoAttr
15     AnnotationName
16      Identifier: ImportResource
17     StringLiteral: "location"
18    Assert
19     PathExist
20      JavaName
21       Identifier: s
22     Msg
23      StringLiteral: "The @ImportResource path %s does not exist"
24      JavaName
```

```
25        Identifier: s
26 Thank you.
```

Chapter 4

Rules

In this chapter, we discuss the 9 extracted rules. We pull out these 9 rules in separate subsections. In Section 4.1, we discuss the rule about the usage of ClassPathXmlApplicationContext. Aforementioned class is an implementation of ApplicationContext and it reqires a correct classpath of the XML file. In Section 4.2, we discuss the rule about the usage of "*method" attribute usage in Spring bean definition. It requires the corresponding Java class to have the exact method included. In Section 4.4, we discuss the rule about importing annotations into XML configuration files. The related element tag usually searches for the annotations in the Java sources code, which requires the existence of those annotations. Likewise, in Section 4.5, we discuss the rule about importing XML files into annotation. It requires correctness of classpath of the XML file. In Section 4.3 and 4.6, we discuss two ways to do Denpendency Injection in Spring. They all need the consistence in Java source code and the XML bean definition. In Section 4.7, we discuss the rule of predefined bean usage. It requires the correct bean definition in either XML configuration or annotation configuration. In each section, we first explain the usage of rule. Second, the RSL script for the rule will be discussed. Note, there is one rule for Section 4.1, 4.2, 4.4, 4.5, and 4.7. There are two rules for Section 4.3, and 4.6.

4.1 Spring ClassPathXmlApplicationContext

Interface ApplicationContext is a frequently used interface in the org.springframework.context package. It has several implementations, such as ClassPathXmlApplicationContext and AnnotationConfigApplicationContext. In this rule, we will discuss the ClassPathXmlApplicationContext. ClassPathXmlApplicationContext is the class which can load an XML configuration from a classpath and manage its beans. The classpath is usually input as the first argument. For example, in the List 4.1, the code use the ClassPathXmlApplicationContext implementation of ApplicationContext to load the configuration in XML file ”example.xml” and its classpath ”applicationcontext/example.xml”. The file at this classpath need to be existed. This command would let the application instantiate the Spring container and read the configurations in the XML file. This rule is written in RSL and is shown in List 4.2.

Listing 4.1: Example of ClassPathXmlApplicationContext usage.

```
1 ApplicationContext context = new ClassPathXmlApplicationContext("
    ↪ applicationcontext/example.xml");
```

At line 1, ”Rule xmlPath-check” is used to define this rule as ”xmlPath-check”. At line 2, ”for (class c in project p)” is used to iterate all the Java class in this project and store them in the information pool. Variable c is assigned to be the identifier of class and p is assigned to be the identifier of project. Users can use these two variables to indicate class and project without declaring them again in the later parts of the RSL script. At line 3, ”where(callExists(...))” is used to refine the classes into only contain classes that has used ApplicationContext implementation of ClassPathXmlApplicationContext in their source code. These information will be stored with its corresponding class. At line 4, ”assert(pathExists(getArg(”ApplicationContext.ClassPathXmlApplicationContext”,

0)))" is used to assert the information we get from commands which are executed before this command. As we can see from this command, it contains a "pathExists(...)" command and a "getArg(...,...)" command insides the "pathExists(...)" command. To execute the assertion correctly and efficiently, the engine follows a bottom-up fashion. First, it processes the most inner command, "getArg(..., 0)". It is used to get the first argument value of command "ClassPathXmlApplicationContext" as a string. Then, it passes this string information to its outer command, "pathExists()". The command "pathExists()" returns the existence of the path in the project. If the file exists in the path, it will return true. If the file does not exist in the classpath, it will return false and store the wrong information into the pool. After "pathExists" pass its result to the assert command. At line 5, "msg()" command will be executed accordingly if and only if the assert returns a false.

Listing 4.2: Pattern of Spring ClassPathXmlApplicationContext API usage

```
1 Rule xmlPath-check {
2     for (class c in project p) {
3         where (callExists("ApplicationContext.ClassPathXmlApplicationContext
              ↪ ")) {
4             assert (pathExists(getArg("ApplicationContext.
                  ↪ ClassPathXmlApplicationContext", 0))) {
5               msg("The path %s referenced by ApplicationContext.
                    ↪ ClassPathXMLApplicationContext() does not exist", getArg
                    ↪ ("ApplicationContext.ClassPathXmlApplicationContext", 0))
6             }
7         }
8     }
9 }
```

4.2 Method Existence

For XML-based configuration file, users can specify it with different bean definitions. Inside these bean definitions, users can specify methods, which should be executed at specific running stages. Those methods includes initialization method, destruction method, and some other type of methods (List 4.3) [32]. For example, in List 4.3, user can specify init-method (Line 10) and destroy-method (Line 12) as attribute in XML configuration file for bean to perform certain actions upon initialization and destruction [26].

Listing 4.3: Example of Method Attribute Usage [32].

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="
       ↪ http://www.springframework.org/schema/context"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans.xsd
7     http://www.springframework.org/schema/context
8     http://www.springframework.org/schema/context/spring-context.xsd">
9
10    <bean id = "..." class = "..." init-method = "..."/>
11
12    <bean id = "..." class = "..." destroy-method = "..."/>
13
14 </beans>
```

However, there are constraints when specifying these methods in XML defined beans. The

method name is listed as the attribute value for bean definitions [26]. In order to be called upon time, method with exact name should exist in the corresponding class. For example, in the following List 4.4, there is a bean defined with a init-method ”initIt” and a destory-method ”cleanUp”. The init-method is usually used to initialize some values. The destory-method is usually used to destroy some information that is set before. In order to let this configuration run correctly, corresponding class needs to have these two methods defined (example shown in List 4.5) . In this example, the start() method is called before the message property is set, and the finish() method is called after the context.close().

Listing 4.4: Example of init-method and destroy-method [26].

```
1 <bean id="exampleMethodBean" class="examples.MethodsMatch"
2        init-method="start" destroy-method="finish">
3        ...
4 </bean>
```

Listing 4.5: Example of init-method and destroy-method in corresponding Java class .

```
1 package examples;
2 
3 public class MehodsMatch
4 {
5     ...
6     public void start() throws Exception {
7       ...
8     }
9     public void finish() throws Exception {
10      ...
11    }
```

```
12
13 }
```

To construct the above rule, we have the following RSL script. At line 1, ”Rule method-exist” is used to define the rule as ”method-exist”. At line 2, ”for (class c in project p)” is used to iterate all the Java class (as c) in project p. The information got from this command is stored in the information pool. At line 3, ”for (file xml in p)”, similar to last command, the tool iterates all the XML files (as xml) in project p. The XML file information is stored in the information pool. As we can see here, it does not need to declared the p again with putting project ahead of it. We have already declare this p as the identifier of project at line 3. At line 4, the tool iterates all the element named as <bean> in all the XML files. At line 5, the tool refines the elements and Java classes with element's class attribute and the name of Java class. The engine removes the redundant Java classes and <bean> elements. At line 6, the tool iterates all the attribute value as String s in all the refined <bean> elements, if and only if there is ”*method” presents. We includes the wild card ”*” feature in the tool to improve the flexibility of the tool. In this case, we can use one ”*method” to represent all the attribute names that follow this pattern. For example, using ”*method” will enable the tool to search for both init-method and destory-method. At line 7, the tool asserts the existence of the method name (get from getting the attribute from <bean> elements) in its corresponding Java class. As we can see, the ”exists()()” command has two parenthesis. The arguments in first parenthesis are used to iterate the method in class c. The arguments/expression in second parenthesis are used to do the assertion. It asserts whether there is a method in c is named as s. At line 8, ”msg()” command will be executed accordingly if and only if the assert() command return a false.

Listing 4.6: Pattern of Method Existence

```
1  Rule method-exist {
2    for (file xml in project p){
3      for (element <bean> in xml) {
4        for (class c in p) {
5            where (getAttr(<bean>, "class") == getFQN(c)) {
6                for (String s in getAttr(<bean>, "*method")) {
7                    assert(exists(method m in c)(getName(m) == s)) {
8                        msg("The referenced method name %s in XML is not
                             ↪ defined", s)
9                    }
10               }
11           }
12       }
13     }
14   }
15 }
```

4.3 <property> Tags and Fields Correspondence

In Spring, there are two ways to do dependency injection (DI). The ApplicationContext supports both constructor-based and setter-based DI for the beans it manages. Both of them can fulfill the DI process [17]. In this section, the setter-based DI and its rule are discussed.

Setter-based DI is accomplished by the container calling setter methods on your beans after

invoking a no-argument constructor or a no-argument static factory method to instantiate your bean [17]. By convention, fields must be declared for the setter methods. If the Java source code fail to have this setter method defined, the ApplicationContext throws an exception as a result of a missing or invalid property in Java file [10]. Here is an standard example (XML configuration is shown in List 4.7) of setter-based DI usage in Spring [17]. Its corresponding Java file is shown in List 4.8. As we can see from the example, to fulfill the setter-based DI, setter methods and their fields are usually needed. Name attributes of element <property> include "mathNotes", "studentId", and "studentName". Fields and setter methods of them are presented in the Java file. Noted, developers sometimes would like to have the field name defined differently with their setter method. Because of this possibility, we decided to check fields and setter method separately.

Listing 4.7: XML configuration of setter-based DI in Spring.

```
1 <bean id="examplePropertyBean" class="examples.ExampleProperty">
2     <property name="mathNotes">
3         <ref bean="mathObjectValue"/>
4     </property>
5     <property name="studentId" value="1"/>
6     <property name="studentName" value="Joe">
7 </bean>
8
9 <bean id="mathObjectValue" class="examples.Math"/>
```

Listing 4.8: Java file of setter-based DI in Spring.

```
1 public class ExampleProperty {
```

```
2
3     private Math mathNotes;
4     private String studentName;
5     private int studentId;
6
7     public void setMathNotes(Math mathNotes) {
8         this.mathNotes = mathNotes;
9     }
10
11    public void setStudentName(String studentName) {
12        this.studentName = studentName;
13    }
14
15    public void setStudentId(int studentId) {
16        this.studentId = studentId;
17    }
18 }
```

To construct the above rule of fields, we have the following RSL script (List 4.9). At line 1, ”Rule field-property-map”, the rule is defined as ”field-property-map”. At line 2, the tool iterates all the Java classes (as c) in project p and store the class information at the information pool. At line 3, the tool iterates all the XML files (as xml) in project p and store the XML file information at the information pool. At line 4, the tool iterates all the elements named as <bean> in XML files. At line 5, the tool refines all the element and corresponding Java class according to the where statement. The tool removes the redundant Java classes and <bean> elements, which can not find a match with any others. At line

6, the tool finds all the <property> elements in all the refined beans. At line 7, the tool asserts the existence of the field name in the corresponding class. The field name has been used as "name" attribute in the <property> element . At line 8, "msg()" command will be executed accordingly if and only if the assert() command return a false.

Listing 4.9: Pattern of <property> name attribute and field name check

```
1  Rule field-property-map {
2    for (file xml in project p){
3      for (element <bean> in xml) {
4         for (class c in p) {
5          where (getAttr(<bean>, "class") == getFQN(c)) {
6            for (element <property> in <bean>) {
7              assert(exists(field f in c)(getAttr(<property>, "name") ==
                   ↪ getName(f))) {
8                msg("There is no field of %s corresponding to the property %s
                     ↪ of bean %s", getFQN(c), getAttr(<property>, "name"),
                     ↪ getAttr(<bean>, "id"))
9              }
10           }
11         }
12       }
13     }
14   }
15 }
```

To construct above rule of setter method existence, we have the following RSL script (List 4.10). At line 1, the rule is defined as "property-setter-map". At line 2, the tool iterates all

the Java classes (as c) in project p. All the class information are stored the class information in the information pool. At line 3, the tool iterates all the XML files (as xml) in project p and store the XML file information in the information pool. At line 4, the tool iterates all the element named as <bean> in XML files. At line 5, the tool refines the element and Java classes according to the where statement. The tool removes the redundant Java classes and <bean> elements. At line 6, the tool finds all the <property> elements in all the refined beans. At line 7, the tool assets the existence of the setter method in the corresponding Java class. Arguments in first parenthesis of ”exists()()” command are used to iterate the method in class c. There are two expressions to be asserted with, which are specified in the second parenthesis of ”exists()()” command. First expression asserts that the first three characters of this method name is ”set”. This expression makes sure the iterated method is a setter method. In second expression, it first gets the substring of method name (starting at forth character, excluding the ”set”). Then it asserts whether there is a match between the attribute ”name” in the <property> element with the this substring. At line 8, ”msg()” command will be executed accordingly if and only if the assert() command return a false.

Listing 4.10: Pattern of <property> name attribute and setter method existence check

```
1 Rule property-setter-map {
2   for (class c in project p) {
3     for (file xml in p) {
4       for (element <bean> in xml) {
5         where (getAttr(<bean>, "class") == getFQN(c)) {
6           for (element <property> in <bean>) {
7             assert(exists (method m in c) (substring(getName(m), 0, 3)=="set"
                 ↪  AND upperCase(substring(getName(m),3)) == upperCase(
                 ↪ getAttr(<property>, "name")))){
```

```
8               msg("There is no field of %s corresponding to the property %s
                    ↪ of bean %s", getFQN(c), getAttr(<property>, "name"),
                    ↪ getAttr(<bean>, "id"))
9             }
10          }
11        }
12      }
13    }
14  }
15 }
```

4.4 Usage of <context:annotation-config> in XML file

Element tag <context:annotation-config> is used to activate applied annotations in already registered beans in application context [25]. It imports the annotations, including @Autowired, @Qualifier, @PostConstruct, @PreDestroy, and @Resource. When <context:annotation-config> is used, at least one of above mentioned annotations should exist [24]. The usage of <context:annotation-config> is as followed (shown in List 4.11).

Listing 4.11: Usage of <context:annotation-config> in XML configuration.

```
1 <?xml version = "1.0" encoding = "UTF-8"?>
2 <beans xmlns = "http://www.springframework.org/schema/beans"
3    ......
4    <context:annotation-config/>
5    ......
```

```
6 </beans>
```

To construct the above rule, we have the following rule script written in RSL. At line 1, it defines the rule as "context-annotation-config-annotation". At line 2, the tool iterates all the XML files (as xml) in project p. At line 3, the tool refines the XML files by checking the existence of tag <context:annotation-config>. It is done by using "exists()()" command. As we can see, the second parenthesis is empty. It means that there is no constraints applied to the existing checking. If there is a <context:annotation-config> tag in XML file, then the tool will refine it. At line 4, it iterates all the Java classes in project p. At line 5, it asserts whether there is any related annotations presented in Java files. As we can see from the command, "getAnnotated(..., "*")", the second argument is defined as "*". The "*" means that the annotation specified as first argument could be at anywhere inside the Java class. It could be annotated at method, or at field, or at class. The use of join is storing all the fields, methods, which are annotated with related annotations. At line 6, "msg()" command will be executed accordingly if and only if the "assert()" command return a false.

Listing 4.12: Pattern of <context:annotation-config> usage

```
1 Rule context-annotation-config-annotation {
2   for (file xml in project p) {
3     where(exists(element <context:annotation-config> in xml)()) {
4         for (class c in p) {
5             assert (NOT isEmpty(join(getAnnotated(@Autowired, "*"),
                  ↪ getAnnotated(@Qualifier, "*"), getAnnotated(@PostConstruct,
                  ↪  "*"), getAnnotated(@PreDestroy, "*"), getAnnotated(
                  ↪ @Resource, "*")))) {
6                 msg("Although <context:annotation-config> is used in XML,
```

```
        ↪ there is no software entity annotated with @Autowired,
        ↪ @Qualifier, @PostConstruct, @PreDestroy, or @Resource")
                }
            }
        }
    }
}
```

4.5 Usage of @ImportResource Annotation

Similar to last rule mentioned in Section 4.4, users can also import XML configurations into annotation-based configurations. This merging is done by using annotation @ImportResource. Spring implements @ImportResource annotation to load beans from XML configuration file into an Application Context. For example, in the List 4.13 shown below, it imports the configurations from example.xml into Application Context [23]. In this case, users only need to instantiate one Spring container by using AnnotationConfigApplicationContext. It is important to make sure that the configuration details inside the XML file is properly merged. The project needs to have this file located at specified location. Otherwise, there will be runtime errors.

Listing 4.13: Example of importing XML configuration into Annotation configuration

```
@Configuration
@ImportResource(locations = {"classpath*:example.xml"})
public class ImportXMLConfiguration {
}
```

To construct the above rule, we have the following rule script written in RSL. At line 1, it defines the rule as ”import-resource-path”. At line 2, the tool iterates all the Java class files in project p. At line 3, the tool iterates all the paths under ”locations” attribute inside the @ImportResource annotation. The location information is stored as a String with identifier s. At line 4, it checks whether the paths (with identifier s) gathered at line 3 exist or not. At line 5, ”msg()” command will be executed accordingly if and only if the assert() command return a false.

Listing 4.14: Pattern of importing XML configuration into Annotation configuration

```
1 Rule import-resource-path {
2   for (class c in project p) {
3     for(String s in getAnnoAttr(@ImportResource, "locations")) {
4       assert(pathExists(s)) {
5         msg ("The @ImportResource path %s does not exist", s)
6       }
7     }
8   }
9 }
```

4.6 <constructor-arg> Tags and Fields Correspondence

As we've discussed in Section 4.3, it is also acceptable to do the constructor-based dependency injection (DI). For example, in List 4.15, it shows the Java code of a bean class [17]. In this case, the container can use type matching with simple types if you explicitly specify the type of the fields by using the type attribute. In the java source code, it defines the fields with type int and String, respectively. Its corresponding XML configuration is shown

in List 4.16. As we can see in the XML configuration file, there are two child elements <constructor-arg> under the <bean> "exampleBean". Their type attribute match with two fields type declared in the corresponding Java file, respectively.

Listing 4.15: Java code example of constructor-based DI in Spring project

```
package examples;

public class ExampleBean {

    private int studentId;
    private String studentName;

    public ExampleBean(int studentId, String studentName) {
        this.studentId = studentId;
        this.studentName = studentName;
    }
}
```

Listing 4.16: XML configuration example of constructor-based DI in Spring project with type specification

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg type="int" value="1"/>
    <constructor-arg type="java.lang.String" value="Joe"/>
</bean>
```

To construct above rule of constructor-based DI with type attribute, we have the following

RSL script (List 4.17). At line 1, the rule is defined as "constructor-arg-type-field-map". At line 2, the tool iterates all the Java class (as c) in project p and stores the class information at the information pool. At line 3, the tool iterates all the XML files (as xml) in project p and stores the XML file information in the information pool. At line 4, the tool iterates all the <bean> elements in XML file. At line 5, the tool refines the <bean> elements and Java classes. The constrain is the name of the Java file must match the <bean> element attribute. The tool removes the redundant Java classes and <bean> elements. At line 6, the tool iterates all the <constructor-arg> elements in <bean> element. At line 7, the tool checks the existence of the field type (stored as f) in the corresponding class. Whether there is at least one field and its type can much the type attribute of <constructor-arg> element. At line 8, "msg()" command will be executed accordingly if and only if the assert() command return a false.

Listing 4.17: Pattern of <constructor-arg> type attribute and field type check

```
1 Rule constructor-arg-type-field-map {
2     for (class c in project p) {
3      for (file xml in p) {
4         for (element <bean> in xml) { // @bean
5           where (getAttr(<bean>, "class") == getFQN(c)) {
6             for (element <constructor-arg> in <bean>) {
7               assert (exists (field f in c) (getAttr(<constructor-arg>, "type
                  ↪ ") == getType(f))) {
8                 msg ("The type of <constructor-arg> %s does not correspond to
                    ↪ any fields' type in class %s", getAttr(<constructor-arg
                    ↪ >, "type"), getName(c))
```

```
9               }
10            }
11         }
12       }
13     }
14   }
15 }
```

In the same case, the container can also use name matching with fields name. In this way, user explicitly specify the name attributes in <constructor-arg> elements. In the last example, we use constructor type to map. In the following example, we use constructor name mapping. In the Java source code, it defines the fields with name years and ultimateAnswer, respectively. In order to use the constructor name mapping, the corresponding XML configuration is shown in List 4.18. In line 2 and 3, it uses the name attributes instead of type attributes.

Listing 4.18: XML configuration example of constructor-based DI in Spring project with name specification

```
1 <bean id="exampleBean" class="examples.ExampleBean">
2     <constructor-arg name="studentId;" value="1"/>
3     <constructor-arg name="studentName" value="Joe"/>
4 </bean>
```

To construct above rule of constructor-based DI with type attribute, we have the following RSL script (List 4.19). At line 1, the rule is defined as ”constructor-arg-name-field-map”. At line 2, the tool iterates all the Java class (as c) in project p and stores the class information

at the information pool. At line 3, the tool iterates all the XML files (as xml) in project p and stores the XML file information in the information pool. At line 4, the tool iterates all the <bean> elements in XML file. At line 5, the tool refines the <bean> elements and Java classes. The constrain is the name of the Java file must match the <bean> element attribute. The tool removes the redundant Java classes and <bean> elements. At line 6, the tool iterates all the <constructor-arg> elements in <bean> element. At line 7, the tool checks the existence of the field name (stored as f) in the corresponding class. Whether there is at least one field and its name can much the type attribute of <constructor-arg> element. At line 8, ”msg()” command will be executed accordingly if and only if the assert() command return a false.

Listing 4.19: Pattern of <constructor-arg> name attribute and field name check

```
1  Rule constructor-arg-name-field-map {
2    for (class c in project p) {
3      for (file xml in p) {
4        for (element <bean> in xml) {
5          where (getAttr(<bean>, "class") == getFQN(c)) {
6            for (element <constructor-arg> in <bean>) {
7              assert (exists (field f in c) (getAttr(<constructor-arg>, "name")
                   ↪  == getName(f))) {
8                msg ("<constructor-arg> %s does not correspond to any field of
                     ↪ class %s", getAttr(<constructor-arg>, "name"), getName(c)
                     ↪ )
9              }
10           }
```

```
11          }
12        }
13      }
14    }
15  }
```

4.7 Bean Existence

The getBean() method from the BeanFactory interface is a frequently used method. It is used to retrieve the bean with name or type [21]. For example, if we have defined a bean of class Example with name as ”exampleBean”. We have two ways to retrieve this bean. One is retrieve it by using its name ”exampleBean” (as the first line of List 4.20). The other one is retrieve it by using its class type (as the second line of List 4.20). As we know, there are two ways to define the bean, which are annotation-based and XML-based. For type retrieving, there should be a class annotated with @Component, @Service, @Repository, @Controller, and @RestController. Or, there should be a bean defined with this class in XML files. In XML-based configuration, user can use the class attribute to specify the class. For name retrieving, the related method should be annotated with @Bean. Or, there should be a bean in XML files named accordingly. The id attribute of the bean should be defined as the name as the first argument in getBean() method.

Listing 4.20: Examples of retrieving bean.

```
1 Object exampleObject1 = context.getBean("exampleBean");
2 Object exampleObject2 = context.getBean(Example.class);
```

To construct the above rule, we have the following RSL script. At line 1, the rule is defined

as "bean-exist". At line 2, the tool iterates all the Java files in project p. At line 3, the tool iterates all the XML files in project p. At line 4, the tool refines the Java files with constraint. The refined Java file should call getBean() method in ApplicationContext interface. At line 5, the tool retrieves the first argument in all getBean() methods. The tool stores them in the information pool. At line 6 and line 11, it differentiate the retrieving method from whether the first argument is ended with ".class" or not. If it is ended with ".class" (at line 6), the tool will assert whether there is a class type annotated with any of the related annotations and its class name can match the getBean() argument. Or, whether there is a XML defined bean has class attribute, which can match the getBean() argument. If it is not ended with ".class" (at line 11), the tool will assert whether there is a bean defined with certain name either by using annotation or XML configuration. The "msg()" will output the according message accordingly if the assert() gives false.

Listing 4.21: Pattern of bean existence check

```
1 Rule bean-exist {
2   for (class c in project p) {
3     for (file xml in p) {
4       where (callExists("getBean")) {
5         for (String s in getArg("getBean", 0)) {
6           where (endsWith(s, ".class")){ // s.endsWith("")
7             assert(exists(class c1 in join(getAnnotated(@Component, "class"),
                 ↪  getAnnotated(@Service, "class"), getAnnotated(@Repository,
                 ↪  "class"), getAnnotated(@Controller, "class"), getAnnotated
                 ↪ (@RestController,"class")))(substring(s, 0, indexOf(s, ".
                 ↪ class")) == getSN(c)) OR exists(element <bean> in xml)(
                 ↪ substring(s, 0, indexOf(s, ".class")) == getSN(getAttr(<
```

```
        ↪ bean>, "class")))) {
8           msg ("The bean class %s mentioned in getBean() is undefined", s
              ↪ )
9         }
10      }
11      where (NOT endsWith(s, ".class")) {
12        assert(exists(String s2 in getAnnoAttr(@Bean, "name"))(s == s2)
            ↪ OR exists (element <bean> in xml)(s == getAttr(<bean>, "id
            ↪ "))) {
13          msg ("The bean %s mentioned in getBean() is undefined", s)
14        }
15      }
16    }
17  }
18 }
19 }
20 }
```

Chapter 5

Evaluation

In this section, we first introduce our data set (Section 5.1) and evaluation metrics (Section 5.2). Next, we present our evaluation on MeEditor's effectiveness of bug detection with ground truth data set (Section 5.3). Finally, it presents our evaluation on MeEditor's effectiveness of bug detection and fixes with real world data set (Section 5.4).

5.1 Data Sets

We construct two major data sets for evaluation on MeEditor. Specifically, we mined Java open-source projects on GitHub [22]. We crawled the website for projects that contain at least one XML file, whose file path has any of the following keywords: ”Spring”, ”security”, ”web”, and ”WEB-INF”. We defined this heuristic based on our experience. Projects with these keywords are more likely to have XML files, which are used to deploy the configuration. After remove the redundant projects, we randomly put the crawly projects into two data sets. The first set (**D1**) contains 180 projects. These 180 projects are evenly distributed for 9 rules. In other words, 20 projects for each rule. The second set (**D2**) contains 500 open-source (real world) projects.

Our experiments with D1 tends to assess the usability of MeEditor, while our experiments with D2 intend to mimic the real application scenarios of MeEditor and asses the tool's

usefulness to developers. Notice that MeEditor is actually applicable to arbitrary project to examine, no matter whether the files have been newly created, recently updated, or changed for a long time. In our experiments with D2, we applied MeEditor to revisions of projects to demonstrate the usefulness of the tool.

5.2 Metrics

We used the following metrics to evaluate the effectiveness of MeEditor.

Precision (P) measures among all reports generated by a technique, how many of them are true positives:

$$P = \frac{\text{\# of correct reports}}{\text{Total \# of generated reports}} \times 100\% \tag{5.1}$$

Precision can be used to evaluate the effectiveness of rule application. For rule application, P measures among all reported bugs, how many of them are real bugs.

Recall (R) measures among all known true positives, how many of them are reported by a technique:

$$R = \frac{\text{\# of correct reports}}{\text{Total \# of true positives}} \times 100\% \tag{5.2}$$

Based on the Spring specification and our manual inspection, we managed to construct a ground truth data set and evaluated the recall of the rule application.

F score (F) combines P and R to measure the accuracy:

$$F = \frac{2 \times P \times R}{P + R} \times 100\% \tag{5.3}$$

5.3 Effectiveness of Bug Detection with Ground Truth Data Set

To evaluate MeEditor's effectiveness of bug detection, we conduct experiments with ground truth data set. The experiments evenly splits D1 (180 projects) into 9 portions and conduct examination to evaluate how bugs detected by MeEditor match the known bugs in our data sets.

Specifically, to build the ground truth data sets, we applied the following mechanisms. For rule **constructor-arg-type-field-map**, **constructor-arg-name-field-map**, **property-setter-map**, **property-field-map**, and **method-exist**, they are the rules which involve in corresponding XML attributes/entities checking in Java files. For each of these 4 rules, we first applied DOM to convert related XML file to a parsing tree for each assigned project. Then, we randomly selected one element, which the rule can be applied to. We applied 3 different ways to introduce bugs into the projects, which are (1) updating the attribute value in XML file; (2) updating the corresponding field/method's name/type in Java file; (3) deleting the corresponding field/method's name/type in Java file. To conduct fair evaluation, we evenly distributed these 20 elements into 3 portions (7, 7, 6) and applied different ways of introducing bugs to each portion. For **context-annotation-config-annotation**, the <context:annotation-config> tries to scan any applicable annotations in all Java files. We applied 2 ways to introduce bugs into the projects, which are (1) deleting all the applicable annotations in all

Java files; (2) updating all the applicable annotations to non-applicable annotation in all Java files. To conduct fair evaluation, we evenly distributed these 20 projects into 2 portions and applied different ways of introducing bugs to each portion. For **xmlPath-check** and **import-resource-path**, they are the rules which involve in checking the existence of XML file in certain path. We applied 3 different ways to introduce bugs into the projects, which are (1) updating the argument to a non-exist classpath of a XML file; (2) deleting the XML file in the argument; (3) moving the XML file to another classpath. To conduct fair evaluaion, we distributed these 20 projects into 3 portions (7, 7, and 6) and applied different ways of introducing bugs to each portion. For **bean-exist**, method getBean() tries to get the bean from both annotation and XML configuration. We applied 5 different ways to introduced bugs into the projects, which are (1) updating the argument to a non-exist bean ”id”; (2) rename the bean ”id” to a new id in XML configuration; (3) renaming the bean ”id” to a new id in annotation; (4) deleting the bean in XML configuration; (5) deleting the bean in annotation. To conduct fair evaluation, we evenly distributed these 20 projects into 5 portions and applied different ways of introducing bugs to each portion.

5.3.1 Bug Detection

Table 5.1 presents our experiment results for the ground truth data set. In the table, **# of Bugs Reported** presents the number of bugs MeEditor detected in the ground truth data set. **Precision**, **Recall**, and **Accuracy** reflect the effectiveness of MeEditor's bug detection capability. For example, MeEditor reported 22 bugs, while 20 of these bugs are true positives for rule **field-property-map**. Based on rule **field-property-map**, MeEditor detected 22 bugs in the ground truth data set, although there are only 20 known bugs in the set. The intersection between bug reports and our ground truth is 20 bugs. Thus, MeEditor achieved 91% precision, 100% recall, and 95% accuracy for rule **field-property-map**.

For each rule, given 20 known bugs, MeEditor reported 17-22 bugs, and 16-20 of them are true bugs. Among all the rules, MeEditor achieved 94% precision, 94% recall, and 94% accuracy for bug detection. Two reasons can explain why MeEditor could not achieve 100% accuracy. First, for each rule, although we extracted all the rules from Spring specification, it does not cover all the programming possibilities. Developers do not follow the programming conventions but they can still achieve the same configured systems. Second, the tool overlooks some inheritance evidence. As a result, MeEditor can not give 100% accuracy.

Table 5.1: Rule Examination in 180 Projects.

Rule	# of Bugs Reported	# of Known Bugs	# of Correctly Reported Bugs	Precision (%)	Recall (%)	Accuracy (%)
xmlPath-check	20	20	20	100	100	100
method-exist	18	20	18	100	90	95
field-property-map	22	20	20	91	100	95
property-setter-map	20	20	20	100	100	100
context-annotation-config-annotation	20	20	20	100	100	100
import-resource-path	17	20	17	100	85	92
constructor-arg-type-field-map	21	20	20	95	100	97
constructor-arg-name-field-map	21	20	20	95	100	97
bean-exist	22	20	16	72	80	76
Total	181	180	171	94	94	94

5.4 Effectiveness and Usefulness of Bug Detection and Fix in Real World Projects

In this section, the effectiveness of bug detection with MeEditor will be discussed in real word projects. We conduct two experiments. The first experiment applied MeEditor to the latest version of 500 projects. The bug reports were validated manually according to the Spring specification. It also provided us rule usage information. For each rule, we collected the projects which use the rule. The second experiment applied MeEditor to the history

versions of those projects which used the rule.

5.4.1 Bug Detection for Latest Version

The effectiveness of bug detection results are shown in Table 5.2. For each rule, the table has 4 columns. In the table, # of **Pattern Use** shows how many projects use this pattern. # of **Bugs Reported** shows how many projects are reported as erroneous project. # of **Correctly Reported Bugs** shows how many of the erroneous projects are truly incorrect. **Precision** reflect the effectiveness of MeEditor's bug detection capability.

Table 5.2: Rule Examination in 500 Projects.

Rule	# of Pattern Use	# of Bugs Reported	# of Correctly Reported Bugs	Precision (%)
xmlPath-check	18	2	2	100
method-exist	16	4	3	75
field-property-map	68	15	10	67
property-setter-map	68	21	21	100
context-annotation-config-annotation	101	14	14	100
import-resource-path	0	0	0	N/A
constructor-arg-type-field-map	69	1	1	100
constructor-arg-name-field-map	69	3	0	0
bean-exist	34	20	12	60
Total	443	80	63	79

For rule **xmlPath-check**, 18 out of 500 projects use this rule. MeEditor reports bugs in 2 out of these 18 projects. By checking the bug reports generated by MeEditor manually, these two are recognized as true erroneous projects. According to path arguments listed in ClassPathXmlApplicationContext command, there are no such file located in the path for both projects. MeEditor gives 100% precision on this rule.

For rule **method-exist**, 16 out of 500 projects are involved in this rule. MeEditor reports

bugs in 4 out of these 16 projects. By checking the bug reports generated by MeEditor manually, 3 out of these 4 are recognized as true erroneous projects. These three projects does not define the method in corresponding class nor the super class of corresponding classes. One false positive is introduced by inherits the method from imported super class. MeEditor gives 75% precision on this rule.

For rule **field-property-map** and rule **property-setter-map**, both of rules use <property> tag to indicate the usage. So the # of Pattern Use are the same for these two rules. It gives that 68 out of 500 projects are involved in these two rules. For **field-property-map**, MeEditor reports bugs in 15 out of these 68 projects. By checking the bug reports generated by MeEditor manually, 10 out of these 15 are recognized as true erroneous projects. The fields can not be found in its corresponding class nor the super classes of its corresponding class. The other 5 false positives are introduced by same type. The fields are used in setter method with correct name. The correct name means that it has the named field as the bean definition in XML file. However, they are not consistent with the convention, which the name of field should be part of the setter method. For example, a field is named as ”example” in Java source code. By convention, its setter method should be ”...setExample(...)”. It is difficult to keep track of the naming for MeEditor. MeEditor gives 67% precision on this rule. For rule **property-setter-map**, MeEditor reports bugs in 21 out of these 68 projects. By manually checking the bug reports generated by MeEditor, all of them are identified as true bugs. Since the setter-based DI is depending on the correctness of setter method declaration, all of them does not have the setter method defined in their class or their super class. MeEditor gives 100% precision on this rule.

For rule **context-annotation-config-annotation**, 101 out of 500 projects are involved in this

rule. MeEditor reports bugs in 14 out of these 101 projects. By checking the bug reports generated by MeEditor manually, all of then are recognized as true erroneous projects. No related annotations are used in the Java files. MeEditor gives 100% precision on this rule. The high usage of this rule suggest that it is common for users to import annotations to XML configuration. This could be phenomenon of transitioning from XML-based configuration to annotation-based configuration.

For rule **import-resource-path**, there are no projects using this rule. The results suggest if the developers is using annotation as their major metadata format, they are not likely to mix the use with XML metadata.

For rule **constructor-arg-type-field-map** and rule **constructor-arg-name-field-map**, both of these rule use <constructor-arg> tag. So the # of Pattern Use is the same for these two rules. MeEditor gives 69 out of 500 projects are involved in these two rules. For rule **constructor-arg-type-field-map**, MeEditor reports bugs in 1 out of these 69 projects. By checking the bug reports generated by MeEditor manually, it is recognized as true erroneous project. There are no fields declared with such type in both itself class and its super classes. MeEditor gives 100% precision for this rule. For rule **constructor-arg-name-field-map**, MeEditor reports bugs in 3 out of these 69 projects. By manually checking the bug reports generated by MeEditor, none of them is identified as true bug. The false positives are all introduced by using not conventional way to defined the constructor arguments. MeEditor gives 0% precision on this rule.

For rule **bean-exist**, 34 out of 500 projects are involved in this rule. MeEditor reports bugs in 20 out of these 34 projects. By checking the bug reports generated by MeEditor manually,

12 out of 20 are recognized as true erroneous projects. There are two types of false positive. 3 out of these 8 false positives use other Interface, including AbstractApplicationContext, and ConfigurableApplicationContext. It is difficult for the tool to differentiate the above mentioned two interface with ApplicationContext. 5 out of 8 false positives use variable identifiers to store the bean information and use the variable alias as the argument. It is difficult for the tool to keep track of variables alias. MeEditor gives 60% precision on this rule.

Overall, the total # of Pattern Use is 443. MeEditor reports 80 erroneous projects for all rules. 63 out of these 80 projects are true erroneous projects according to manual inspection. MeEditor gives 79% precision on all the rule checking. As we can see from table 5.2, large portion of projects, which use the rules, are correct based on the rule checking. We use the data set to evaluate whether MeEditor can (1) detect any real bug in latest version of projects (this section), and (2) suggest fixes that correspond to developers' actual fixes in reality (next section 5.4.2).

5.4.2 Fixed Bugs in History Commits

For each used projects (mentioned in last section), we first found all the files that related to each rule. Then, the entire project was downloaded at each commit, where the related files make changes by the developers. MeEditor check the correctness of all history commits of the entire project. By checking the bug reports generated by MeEditor manually, we identified 23 erroneous commits, which the developers were later fixed according to the version history. This observation indicates that developers did introduce errors that MeEditor can detect. MeEditor can help developers reveal their mistakes when they modify the Java files

or XML files.

Table 5.3 presents the 23 erroneous project-rule pairs that developers later fixed. Specifically, column **Bug Index** shows the index we assigned to each project-rule pair. **Project Name** shows the name of project. **Violated Rule** shows the rule MeEditor used to identify the bug. **Root Cause** shows explains how developers introduced the bug. **Fixing Strategy** describes how developers resolved the bugs in a later version. **Vdiff(fix, bug)** describes the version difference between the bug-fixing commit and bug-introducing commit (both inclusive).

As we can see from Table 5.3, there are 18 projects that has errors in their version histories. 4 out of 18 projects violate more than one rules in their version histories. Rest of them only violate one rule in their version histories. Specifically, those 18 projects violate 4 rules in total, including **context-annotation-config-annotation**, **field-property-map**, **setter-property-map**, and **method-exist**. For **context-annotation-config-annotation**, 10 projects violate this rule in their version histories. The fixing strategies are the same for these 10 projects. The developers add @Autowired annotation to fields to fix the errors. It indicates that developers usually use <context:annotation-config> to import @Autowired annotation. For **field-property-map**, 5 projects violate this rule in their version histories. The fixing strategies are deleting beans, deleting ambiguous file, updating <property> name attributes, adding fields, and deleting <property> elements. For **setter-property-map**, 6 projects violate this rule in their version histories. The fix strategies are deleting beans, deleting ambiguous file, updating <property> name attributes, and updating class attribute. Interestingly, 3 projects introduce and fix field-property-map and setter-property-map errors together. This observation indicates that these two rules are usually highly related to each other. It is suggested to checking the other rule when one rule is violated. For **method-exist**, 1 project violates this

rule. The fixing strategy is deleting the init-method attribute. Explanatory examples of code change snippet of the erroneous project-rule pairs will be discussed at the end of this section.

Additionally, for 3 out of the 23 reports developers applied fixes in the immediate next commit. In average, developers need 9 commits to apply fix to the project. Interestingly, for each erroneous project-rule pair, all the developers only make mistake once. In other words, developers won't make same mistake after they fix the bugs. These observations imply that if developers had used MeEditor to examine their updated project before committing program changes, they should have avoided checking in the erroneous program changes, or even have fixed the introduced bugs earlier.

At the end of this second, a few violation examples will be shown. One example will be discussed for each rule. All violations are listed in Appendix B.

First example is context-annotation-config-annotation violation in project **calltag**. In **calltag** [14], it violated context-annotation-config-annotation rule in its early commit and fixed it later. Developer introduced XML file of applicationContext.xml. In this XML file, there is <context:annotation-config> element. However, the author did not introduce any related annotations, including @Autowired, @Qulifier, @PostConstruct, @PreDestory, or @Resource. The root cause of this violation is that developer commented the @Autowired annotation in earlier version (line 2). It leaves no related annotation in the project for <context:annotation-config> to import. After 9 git commits, the author made modifications to address this misuse. Developer added a new field to one of the Java file along with @Autowired annotation (line 5). List 5.1 shows the modification that the author made.

Listing 5.1: Changes of project calltag with context-annotation-config-annotation error

```
1 @@ -25,18 +31,31 @@ public class IndexController extends
    ↪ ParameterizableViewController {
2  // @Autowired
3     private TwitterListener twitterListener;
4
5 + @Autowired
6 + private Configuration twitterConf;
7            ...............
8               ...........
```

Second example is field-property-map violation in project **unikkit** [43]. In **unikkit**, it violated field-property-map rule in its early history version and fixed it later. When editing the XML file (shown in List 5.2), there is a bean of class "EventLazyModel", which has "eventManager" name attribute in <property> element. However, it did not contain "eventManager" field in its initial corresponding Java file. After 4 commits, the author of **unikkit** made modifications to address this misuse. Developer added a corresponding field "eventManager" to match the <property> name attribute of corresponding bean definition.

Listing 5.2: XML file of project unikkit with field-property-map error.

```
1 + <bean id="eventModel"
2 + class="org.mamce.unikkit.lazy.model.EventLazyModel" >
3 +  <property name="eventManager" ref="eventManager" />
4 + </bean>
```

Listing 5.3: Changes of project unikkit with field-property-map error.

```
+import org.primefaces.model.ScheduleEvent;

 public class EventLazyModel extends LazyScheduleModel {

+ private EventManager eventManager;
+
        .......
        .......
```

Third example is setter-property-map violation in project **featurepub** [20]. In **featurepub**, it violated setter-property-map rule in its early history version and fixed it later. It did not declare the setter method of "encoder" in class "Publisher". After 1 commits, the author of **featurepub** made modification to address this misuse. Developer updated the corresponding class attribute to "BasicPublisher" (shown in List 5.4). In Class "BasicPublisher", it has setter method of "encoder".

Listing 5.4: Changes of project featurepub with setter-property-map error.

```
-  <bean id="publisher" class="es.uva.idelab.featurepub.Publisher" >
+  <bean id="publisher" class="es.uva.idelab.featurepub.BasicPublisher" >
```

The last example is method-exist violation in project **pocket-review** [36]. In **pocket-review**, it violated method-exist rule in its early history version and fixed it later. In XML file spring-dao.xml, it has init-method attribute "init". According to the method-exist rule, in its corresponding Class "DProductDaoBean", there should be a "init" method. However, in Class "DProductDaoBean", it did not contain such method. After 2 commits, the author updated the bean to address this misuse (shown in List 5.5). The bean definition is updated

by deleting the init-method attribute.

Listing 5.5: Changes of project pocket-review with method-exist error.

```
--- a/RnR-GAE-Service/src/main/resources/spring-dao.xml
+++ b/RnR-GAE-Service/src/main/resources/spring-dao.xml
@@ -12,39 +12,39 @@

  <!-- ============== DaoBeans for entities in package com.wadpam.rnr.domain
     ↪  ============ -->
     <bean id="dProductDao"
- class="com.wadpam.rnr.dao.DProductDaoBean" init-method="init">
+ class="com.wadpam.rnr.dao.DProductDaoBean">
     </bean>
    ...............
    ...........
```

Table 5.3: The 23 erroneous rule use fixed by developers.

Bug Index	Project Name	Violated Rule	Root Cause	Fixing Strategy	Vdiff(fix, bug)
1	calltag	context-annotation-config-annotation	No related annotations	Add @Autowired	9
2	calltag	field-property-map	No required fields	Delete beans	26
3	calltag	setter-property-map	No required setter methods	Delete beans	26
4	contract	context-annotation-config-annotation	No related annotations	Add @Autowired	12
5	LIBRARY	context-annotation-config-annotation	No related annotations	Add @Autowired	7
6	mvc-component-tester	context-annotation-config-annotation	No related annotations	Add @Autowired	2
7	ndx-jersey2	context-annotation-config-annotation	No related annotations	Add @Autowired	1
8	NewBlog	context-annotation-config-annotation	No related annotations	Add @Autowired	6
9	scaffolding	context-annotation-config-annotation	No related annotations	Add @Autowired	2
10	tmm__phonebook	context-annotation-config-annotation	No related annotations	Add @Autowired	1
11	tutorial	context-annotation-config-annotation	No related annotations	Add @Autowired	15
12	WSSpring	context-annotation-config-annotation	No related annotations	Add @Autowired	3
13	mentor-project-2013	field-property-map	Directory ambiguity	Delete ambiguous directory	2
14	mentor-project-2013	setter-property-map	Directory ambiguity	Delete ambiguous directory	2
15	soundService	field-property-map	Mismatch field names	Update the <property> name attribute	8
16	soundService	setter-property-map	Mismatch setter methods	Update the <property> name attribute	8
17	unikkit	field-property-map	No required field	Add related field	4
18	wl-oauth	field-property-map	Mismatch field	Update <property> name attribute	31
19	wl-solr-search	field-property-map	No required fields	Delete <property> elements	6
20	pocket-review	method-exist	No required Method	Delete init-method attribute	2
21	pocket-review	setter-property-map	No required setter method	Delete bean	32
22	featurepub	setter-property-map	No required setter methods	Update class attribute	1
23	spatialsearch	setter-property-map	No required setter methods	Update class attribute	6

Chapter 6

Threats To Validity

Threats to External Validity. In this thesis, the detected/recognized bugs are limited to our experiment data sets. The results/observations may not generalize well to close-source projects. To improve the generalization, we would like to include diverse projects into our evaluation in the future. If it is possible, it would be even better to include close-source projects. With those projects included, our findings will be more representative.

Threats to Construct Validity. In our evaluation for MeEditor's effectiveness of bug detection, we used mechanisms mentioned in Section 5.3 to generate bugs. These bug introduction mechanisms may not be representative to the real-world scenarios. The real-world bugs introduced by developers during software maintenance may be different. As a result, our empirical measurements can be biased. To improve the construction of data sets, we will build data sets with real-world bugs in projects to evaluate MeEditor's capability of bug detection.

Threats to Internal Validity. Due to the human bias and our domain knowledge limitation, our manual inspection for the bug reports by MeEditor may be subjective. To mitigate the problem, we had two developers to examine the bug reports generated by MeEditor for agreement. The two developers agreed with each other in most scenarios. When they disagreed upon certain rules mainly because of their domain knowledge limitation, they shared relevant documentation or examples with each other for discussion until coming to an agree-

ment. Among the 500 latest projects, there are 63 bug reports. We sent emails to the owner developers to check whether any of these 63 bugs is true. As we gather more comments from developers, we will further improve the quality of bug detection.

Chapter 7

Related Works

7.1 Domain Specific Language – PBSE

Domain-Specific Language (DSL) is a computer language which is developed to give better solutions when building applications in specific domains [3]. Examples of DSL includes, MATLAB for matrix related programming [6], SQL for relational databases queries [8], Mathemetica for symbolic mathematics programming [29] and so on. In contrast, there is another type of computer language, General-Specific Language (GSL), which is broadly applicable across domains [3].

7.1.1 PBSE

Before introduce the RSL, it is crucial to introduce the PBSE. As a DSL, PBSE is a reusable metadata format, which can be applied to multiple languages. PBSE gives us lots of insights into our RSL definition. PBSE is built for simplifying the processes of configuration [59]. It is created as a new metadata format to substitute the annotations and XML files. An automated translation tool has been built, which can annotate the Java source code according to the PBSE specification.

PBSE is a concise language. The comparison of PBSE and other metadata formats is conducted in Tilevich and Song [59]. With the increase of the source code lines, the other two styles of metadata, annotations and XML, are increasing. In other words, the complexity of the of writing the metadata in annotation and XML is increasing with the increasing of the source code complexity. However, the number of lines in PBSE metadata is not increasing with the increasing of source code. In other words, the complexity of PBSE metadata is not influenced by the complexity of the source code. Even though metadata format of annotations have already decrease the number of lines compared with XML metadata format in large degrees, it is still larger than PBSE matedata format. Thus, it can be concluded that PBSE is a relatively concise metadata formats.

7.2 Existing Researches

To check the metadata usage, some approaches has been proposed [48, 49, 50, 51, 52, 53, 54]. Existing research provides support for checking XML [48, 49, 53]. For example, XQuery is a widely used query and functional programming language. XQuery can offer helps in querying and transforming collections of structured or unstructured data in XML documents [49]. Similarly, CDuce [48] and XDuce [54] are separately developed DSLs for XML processing. To validate Java metadata usage, Eichberg et al. [51] designed a DSL for users to define constraints. To check user-specified constraints, the researchers automatically converted Java bytecode to XML documents, and converted constraints to XQuery path expressions. Similarly, Darwin [50] and Noguera and Duchien [57] individually defined DSLs for users to specify and then validate the constraints on annotation usage. In this way, developers need to learn multiple DSLs to check the usage of both annotations and XML files.

Chapter 8

Conclusion

Metadata are difficult to create and maintain in enterprise applications. In this thesis, we developed RSL – a domain specific language for representing rules in Spring and built MeEditor – a engine/tool to run the script in RSL. By constructing the rule in RSL, MeEditor can identify rule violation to report bugs. Unlike prior works, MeEditor and RSL are self-contained, which they do not rely on other implementation.

9 rules are extracted from Spring specification and are written in RSL. To evaluate the effectiveness and correctness of MeEditor, 180 plus 500 open-source projects are mined from Github. First, MeEditor run RSL scripts to the ground truth data set. According to the experiments, it detects bug with 94% precision, 94% recall, and 94% accuracy. Second, MeEditor run the RSL script to real world projects (total 500 projects). For the latest version of these 500 projects, MeEditor gives 79% precision according to our manual inspection. Then, MeEditor is applied to the version histories of those projects, which use the rule and is identified as correct project for latest version. MeEditor identifies 23 bugs, which later fixed by developers. In average, it takes developers 9 commits to fix the bug. By using MeEditor before committing the changes would save the time of debugging. This demonstrates the usefulness of MeEditor and the necessity of static analysis tools for metadata.

There is still significant space for future improvements in static analysis for metadata usage.

Our current experiments only focus on 9 rules in Spring specification. However, it is still interesting to know the bug evolution of other rules. As the future work, we will explore more rules in Spring or other web frameworks and apply MeEditor to more diverse projects. Meanwhile, it is important to improve the accuracy of MeEditor. Besides, we also envision MeEditor to be used in Integrated Development Environment (IDE) for web applications.

Bibliography

[1] Introduction to web application deployment descriptors. URL https://docs.oracle.com/cd/E19226-01/820-7627/bncbj/index.html.

[2] A xml deployment descriptors. URL https://docs.oracle.com/cd/A91202_01/901_doc/java.901/a90188/xml.htm.

[3] Domain-specific language. URL https://en.wikipedia.org/wiki/Domain-specific_language.

[4] Intro to inversion of control and dependency injection with spring. URL https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring#:~:text=Dependency%20injection%20is%20a%20pattern,than%20by%20the%20objects%20themselves.

[5] Project library. URL https://github.com/dawe73/LIBRARY.

[6] Matlab. URL https://en.wikipedia.org/wiki/MATLAB.

[7] Project newblog. URL https://github.com/PengMQ/NewBlog.

[8] Sql. URL https://en.wikipedia.org/wiki/SQL.

[9] Project spatialsearch. URL https://github.com/virtualandy/SpatialSearch.

[10] The ioc container. URL https://docs.spring.io/spring-framework/docs/5.0.0.M5/spring-framework-reference/html/beans.html#beans-factory-metadata.

[11] Project wsspring. URL https://github.com/gojeanil/WSSpring.

[12] Xml schema tutorial, . URL https://www.w3schools.com/xml/schema_intro.asp.

[13] Xml syntax, . URL https://www.w3schools.com/xml/xml_syntax.asp.

[14] Project calltag. URL https://github.com/bekzod/calltag.

[15] Securing rest urls with spring, . URL https://stackoverflow.com/questions/13836451/securing-rest-urls-with-spring.

[16] Project contract, . URL https://github.com/zm121987/contract.

[17] Core technologies. URL https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-metadata.

[18] What are all the programming languages that use dependency injection? URL https://www.quora.com/What-are-all-the-programming-languages-that-use-dependency-injection.

[19] Package org.w3c.dom. URL https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/package-summary.html.

[20] Project featurepub. URL https://github.com/azuledu/featurepub.

[21] Understanding getbean() in spring. URL https://www.baeldung.com/spring-getbean.

[22] Github. URL https://github.com/.

[23] Spring @importresource annotation example, . URL https://www.javaguides.net/2018/09/spring-importresource-annotation-example.html.

[24] Spring - annotation based configuration, . URL https://www.tutorialspoint.com/spring/spring_annotation_based_configuration.htm.

[25] Spring mvc: <context:annotation-config> vs <context:component-scan>, . URL https://howtodoinjava.com/spring-mvc/spring-mvc-difference-between-contextannotation-config-vs-contextcomponent-scan/.

[26] Spring init-method and destroy-method example. URL https://mkyong.com/spring/spring-init-method-and-destroy-method-example/.

[27] Javacc, the most popular parser generator for use with java applications. URL https://javacc.github.io/javacc/.

[28] Jdt. URL https://wiki.eclipse.org/JDT.

[29] Wolfram mathematica. URL https://en.wikipedia.org/wiki/Wolfram_Mathematica.

[30] Project mentor-project-2013. URL https://github.com/kevinthorley/mentor-project-2013.

[31] Configuration metadata, . URL https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-configuration-metadata.html.

[32] Bean definition, . URL https://www.tutorialspoint.com/spring/spring_bean_definition.htm.

[33] Project mvc-component-tester. URL https://github.com/russellpwirtz/mvc-component-tester.

[34] Project ndx-jersey2. URL https://github.com/syedshahul/ndx-jersey2.

[35] Package org.springframework.beans. URL https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/package-summary.html.

[36] Project pocket-review. URL https://github.com/ounghuy/pocket-review.

[37] Project scaffolding. URL https://github.com/ln1058/scaffolding.

[38] Project soundservice. URL https://github.com/wooice/soundService.

[39] Spring security jdk based proxy issue while using @secured annotation on controller method, . URL https://stackoverflow.com/questions/35860442/spring-security-jdk-based-proxy-issue-while-using-secured-annotation-on-control.

[40] Spring - java based configuration, . URL https://www.tutorialspoint.com/spring/spring_java_based_configuration.htm.

[41] Project tmm_phonebook. URL https://github.com/vkuchyn/tmm_phonebook.

[42] Project tutorial. URL https://github.com/kws/tutorial.

[43] Project unikkit. URL https://github.com/unikkit/unikkit.

[44] Project wl-oauth, . URL https://github.com/ox-it/wl-oauth.

[45] Project wl-solr-search, . URL https://github.com/ox-it/wl-solr-search.

[46] Introduction to xml. URL https://www.w3schools.com/xml/xml_whatis.asp.

[47] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 794–807, 2020.

[48] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an xml-centric general-purpose language. *ACM SIGPLAN Notices*, 38(9):51–63, 2003.

[49] Don Chamberlin. Xquery: An xml query language. *IBM systems journal*, 41(4):597–615, 2002.

[50] Ian Darwin. Annabot: A static verifier for java annotation usage. *Advances in Software Engineering*, 2010, 2009.

[51] Michael Eichberg, Thorsten Schäfer, and Mira Mezini. Using annotations to check structural properties of classes. In *International Conference on Fundamental Approaches to Software Engineering*, pages 237–252. Springer, 2005.

[52] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 13–23. IEEE, 2003.

[53] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for xml. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 67–80, 2001.

[54] Haruo Hosoya and Benjamin C Pierce. Xduce: A statically typed xml processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

[55] Keyul. 10 most popular web frameworks in 2020, November 2019. URL https://medium.com/front-end-weekly/10-most-popular-web-frameworks-in-2020-167b9103e08a.

[56] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango Argoty. Secure coding practices in java: Challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, pages 372–383, 2018.

[57] Carlos Noguera and Laurence Duchien. Annotation framework validation using do-

main models. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 48–62. Springer, 2008.

[58] Andrew Binstock Simon Maple. Jvm ecosystem report 2018 – about your platform and application, October 2018. URL https://snyk.io/blog/jvm-ecosystem-report-2018-platform-application/.

[59] Eli Tilevich and Myoungkyu Song. Reusable enterprise metadata with pattern-based structural expressions. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 25–36, 2010.

Appendices

Appendix A

RSL Core Syntax

$$Program\ startpoint := Start$$

$$Start := (BodyDeclaration)*$$

$$BodyDeclaration := Stmt$$

$Statement\ Stmt := ForStatement$

$|WhereStatement$

$|AssertStatement$ // Assert the truth of the argument

$|MsgStatement$ // Deliver the error message of the violated pattern

$|Block$ // a block that used to clearly separate each statement

$ForStatement := \textbf{for}\ '('Exp\ \textbf{in}\ Exp')'\ Stmt$

$WhereStatement := \textbf{where}\ '('Exp')'Stmt$

$AssertStatement := \textbf{assert}\ '('Exp\ ) *\ ')'Stmt$

$MsgStatement := \textbf{msg}\ '('StringLiteral\ ','\ (Exp)*')'$

$$Block :='\{'(BlockStatement)*'\}'$$

$$BlockStatement :=Stmt$$

$$Expression\ Exp :=LogicalExpression$$

$$LogicalExpression :=RelationalExpression$$
$$(BinaryOperator\ RelationalExpression)*$$

$$RelationalExpression :=PrimaryExpression$$
$$(RelationalOperator\ PrimaryExpression)*$$

$$AssignmentOperator := =$$

$$RelationalOperator := < \mid > \mid <= \mid >= \mid == \mid != $$

$$BinaryOperator :=(\textbf{AND} \mid \textbf{OR})$$

$$PrimaryExpression :=PrimaryPrefix\ (PrimarySuffix)*$$

$$PrimaryPrefix :=Literal$$
$$|Name$$
$$|FileExpression$$
$$|MethodExpression$$

$|FieldExpression$

$|ParameterExpression$

$|JavaClassExpression$

$|ProjectExpression$

$|ElementExpression$

$|AnnotationExpression$

$|UppercaseExpression$

$|GetIdExpression$

$|GetAttrExpression$

$|GetNameExpression$

$|GetFieldExpression$

$|GetAnnotatedExpression$

$|GetElementExpression$

$|GetMethodExpression$

$|GetArgExpression$

$|PathExistExpression$

$|GetTypeExpression$

$|CallExpression$

$|Arguments$

$$
\begin{aligned}
PrimarySuffix &:= '.' \; Identifier \\
Literal &:= < INTEGER_LITERAL > \\
&\quad | < FLOATING_POINT_LITERAL > \\
&\quad | < CHARACTOR_LITERAL > \\
&\quad | BooleanLiteral \\
&\quad | NullLiteral \\
&\quad | StringLiteral \\
&\quad | Name
\end{aligned}
$$

$$
\begin{aligned}
JavaClassExpression &:= \textbf{class} \; Name \\
ProjectExpression &:= \textbf{project} \; Name \\
MethodExpression &:= \textbf{method} \; Name \\
FieldExpression &:= \textbf{field} \; Name \\
ParameterExpression &:= \textbf{parameter} \; Name \\
FileExpression &:= \textbf{file} \; Name \\
ElementExpression &:= \textbf{element} Name \\
UppercaseExpression &:= \textbf{upperCase} \; '(' Exp ')' \\
GetIdExpression &:= \textbf{getId} \; '(' Exp ')' \\
GetAttrExpression &:= \textbf{getAttr} \; '(' XMLName \; ',' \; StringLiteral \; ')'
\end{aligned}
$$

$$GetNameExpression := \textbf{getName}\ '('Exp')'$$

$$GetFieldExpression := \textbf{getField}\ '('JavaName\ ','\ StringLiteral\ ')'$$

$$GetAnnotatedExpression := \textbf{getAnnotated}\ '('JavaName\ ','\ StringLiteral\ ')'$$

$$GetElementExpression := \textbf{getElement}\ '('Name\ ','\ XMLName\ ')'$$

$$GetMethodExpression := \textbf{getAnnotated}\ '('JavaName\ ','\ StringLiteral\ ')'$$

$$PathExistExpression := \textbf{pathExist}\ '('Exp')'$$

$$GetArgExpression := \textbf{getArg}\ '('JavaName\ ','\ Integer\ ')'$$

$$GetTypeExpression := \textbf{getType}\ '('Exp')'$$

$$CallExpression := \textbf{call}\ '('JavaName')'$$

$$Arguments := \textbf{join}\ '('[ArgumentList]')'$$

$$Name := JavaName | XMLName.\ \text{//differentiate the name category}$$

$$JavaName := Identifier('.'\ Identifier)*$$

$$XMLName := '<'\ Identifier\ '>'\ ('.'\ '<'\ Identifier\ '>')*$$

$$BooleanLiteral := \textbf{true}|\textbf{false}$$

$$NullLiteral := \textbf{null}$$

$$StringLiteral := <STRING_LITERAL>$$

$$ArgumentList := Exp\ (','\ Exp)*$$

$$Identifier := <IDENTIFIER>$$

Appendix B

Fix Bugs in History Commits

Project calltag [14]

In **calltag**, author introduced three types of bugs in the history versions and later fixed all of them. The first type of bug is involved with the rule of context-annotation-config-annotation. The author introduced the applicationContext.xml with <context:annotation-config> element. However, the author did not introduce any corresponding annotation. After 9 git commits, the author make modifications to address this misuse. List B.1 shows the modification that the author made.

Listing B.1: Changes of project calltag with context-annotation-config-annotation error

```
1 @@ -25,18 +31,31 @@ public class IndexController extends
    ↪ ParameterizableViewController {
2  // @Autowired
3      private TwitterListener twitterListener;
4
5 + @Autowired
6 + private Configuration twitterConf;
7 +
8 +
9      @Override
```

```
10      protected ModelAndView handleRequestInternal(HttpServletRequest hsr,
            ↪ HttpServletResponse hsr1) throws Exception {
11  -
12  +// twitterListener = new TwitterListener();
13              ...............
14                  ...........
```

The second type of bug is involved with the rule of field-property-map, the developer failed to declare the fields, which are used in the <property> tags. After 26 commits, the project calltag is correct on this pattern. The fix is deleting corresponding bean elements that contains the misused <property> tags (shown in B.2).

Listing B.2: Changes of project calltag with field-property-map and setter-property-map errors

```
1       --- a/src/main/webapp/WEB-INF/applicationContext.xml
2   +++ b/src/main/webapp/WEB-INF/applicationContext.xml
3   @@ -19,42 +19,13 @@
4           <property name=captionpos"captionpossuffixcaptionpos" value=
                ↪ captionpos"captionpos.captionposjspcaptionpos" />
5       </bean>
6
7   + <bean class=captionpos"captionposorgcaptionpos.captionposspringframework
        ↪ captionpos.captionposwebcaptionpos.captionposservletcaptionpos.
        ↪ captionposmvccaptionpos.captionposannotationcaptionpos.
        ↪ captionposDefaultAnnotationHandlerMappingcaptionpos" />
8   + <bean class=captionpos"captionposorgcaptionpos.captionposspringframework
```

```
    ↪ captionpos.captionposwebcaptionpos.captionposservletcaptionpos.
    ↪ captionposmvccaptionpos.captionposannotationcaptionpos.
    ↪ captionposAnnotationMethodHandlerAdaptercaptionpos" />
9  +
10
11      <!-- Session Management -->
12      <bean name=captionpos"captionpossessionInterceptorcaptionpos" class=
        ↪ captionpos"captionposcomcaptionpos.captionposcalltagcaptionpos.
        ↪ captionposauthcaptionpos.captionposSessionInterceptorcaptionpos"
        ↪ />
13 -
14 - <bean id=captionpos"captionposurlMappingcaptionpos" class=captionpos"
    ↪ captionposorgcaptionpos.captionposspringframeworkcaptionpos.
    ↪ captionposwebcaptionpos.captionposservletcaptionpos.captionposhandler
    ↪ captionpos.captionposSimpleUrlHandlerMappingcaptionpos">
15 - <property name=captionpos"captionposmappingscaptionpos">
16 - <props>
17 - <prop key=captionpos"captionposindexcaptionpos.captionposhtmcaptionpos">
    ↪ indexController</prop>
18 - <prop key=captionpos"captionposmaincaptionpos.captionposhtmcaptionpos">
    ↪ mainController</prop>
19 - <prop key=captionpos"captionpostwilliocaptionpos.captionposhtmcaptionpos">
    ↪ twillioController</prop>
20 - </props>
21 - </property>
```

```
22 - <property name=captionpos"captionposinterceptorscaptionpos">
23 - <list>
24 - <ref bean=captionpos"captionpossessionInterceptorcaptionpos"/>
25 - </list>
26 - </property>
27 - </bean>
28
29 -
30 - <bean name=captionpos"captionposindexControllercaptionpos"
31 - class=captionpos"captionposcomcaptionpos.captionposcalltagcaptionpos.
      ↪ captionposcontrollercaptionpos.captionposIndexControllercaptionpos">
32 - <property name=captionpos"captionposviewNamecaptionpos" value=captionpos"
      ↪ captionposindexcaptionpos" />
33 - </bean>
34 -
35 -
36 - <bean name=captionpos"captionposmainControllercaptionpos"
37 - class=captionpos"captionposcomcaptionpos.captionposcalltagcaptionpos.
      ↪ captionposcontrollercaptionpos.captionposMainControllercaptionpos">
38 - <property name=captionpos"captionposviewNamecaptionpos" value=captionpos"
      ↪ captionposmaincaptionpos" />
39 - </bean>
40 -
41 - <bean name=captionpos"captionpostwillioControllercaptionpos"
42 - class=captionpos"captionposcomcaptionpos.captionposcalltagcaptionpos.
```

```
     ↪ captionposcontrollercaptionpos.captionposTwillioControllercaptionpos"
     ↪ >
43 - <property name=captionpos"captionposviewNamecaptionpos" value=captionpos"
     ↪ captionpostwilliocaptionpos" />
44 - </bean>
45 -
```

The last type of bug is involved with the rule of setter-property-map. Developer failed to declared the setter method correspondingly, which are used in the <property> tags. After 26 commits, the project calltag is correct on this pattern. The fix is deleting corresponding bean elements that contains the misused <property> tags (shown in B.2).

Between the introduced and fixed history version of calltag project, the commit subjects include the sentences of ”work in progress”, ”made mistake in last commit using Boolean object converting it to primitive bool”, ”testing”. These sentences indicate that the author of calltag was fixing the bugs along developing.

Project contract [16]

Project **contract** was introduced context-annotation-config-annotation related errors in its history version. The author of Project **contract** introduced contract-action.xml, contract-biz.xml, contract-mapper.xml, contract-service.xml to the project at the initial commit. For each of these four XML file, they all included the < context:annotation-config /> elements. However, there are no corresponding annotations in the Java source code. After 12 git commits, the author correctly introduced the annotations, as shown in list B.3. The errors were fixed by adding @Autowired annotation on the fields.

Commits subjects includes ”initial commit”, ”Unit Test”, ”dsad”, ”dddd”. The author did not provide valuable commit subjects. It is difficult to understand why he introduced the error or how he solved the errors.

Listing B.3: Changes of project contract with context-annotation-config-annotation error

```
1  +public class ContractServiceImpl implements ContractService{
2  + private static Logger logger = Logger.getLogger(ContractServiceImpl.class
      ↪ );
3  + @Autowired
4  + private ContractMapper contractMapper;
5  + @Autowired
6  + private SequenceMapper sequenceMapper;
7  + @Autowired
8  + private UserMapper userMapper;
9  + @Override
10 + public boolean addContract(Contract contract) throws SicException
11 + {
12            ................
13            ................
```

Project LIBRARY [5]

Project **LIBRARY** was introduced context-annotation-config-annotation related errors in its history version. The author of project LIBRARY introduced <context:annotation-config> element in servlet-context.xml. However, the project was missing qualified annotations for this element in XML configuration. After 7 commits, the project LIBRARY is correct on

this pattern. The errors were fixed by adding @Autowired annotation on the fields. Commit subjects includes ”Config Spring”, ”Implementace databáze”.

Listing B.4: Changes of project LIBRARY with context-annotation-config-annotation error

```
1 @Controller
2  public class CategoryController {
3 - private List<Category> categories = new ArrayList<Category>();
4 - //pouze pro tes v DB je  generovno  take smazat!!!
5 - private static int ID = 1;
6 + @Autowired
7 + private IEntityService entityService;
8 
9 + private List<Category> categories = new ArrayList<Category>();
```

Project mvc-component-tester [33]

Project **mvc-component-tester** was introduced context-annotation-config-annotation related errors in its history version. The author introduced <context:annotation-config> element in XML file. After 2 commits, the project mvc-component-tester is correct on this pattern. The errors were fixed by adding @Autowired annotation on the fields. Commit subjects include ”Initial import”, ”Zookeeper/Curator updates”.

Listing B.5: Changes of project mvc-component-tester with context-annotation-config-annotation error

```
1 @Controller
2 -public class ComponentController {
3 +public class ComponentController implements InitializingBean {
```

```
4  +
5  + private static final Logger LOGGER = LoggerFactory.getLogger(
     ↪ ComponentController.class);
6
7      @Autowired
8      JmsTemplate jmsTemplate;
9
10 + @Autowired
11 + CuratorFramework curatorFramework;
```

Project ndx-jersey2 [34]

Project **ndx-jersey2** was introduced context-annotation-config-annotation related errors in its history version. The author introduced <context:annotation-config> element in XML file. After 1 commits, the project ndx-jersey2 is correct on this pattern. The errors were fixed by adding @Autowired annotation on the fields. Commit subjects include ”jersey2-modules”, ”WIP Jersey2 Spring integeration”.

Listing B.6: Changes of project ndx-jersey2 with context-annotation-config-annotation error

```
1 - //private Manager manager;
2 + private Manager manager;
3
4 - /* @Autowired
5 + @Autowired
6     public void setManager(Manager manager) {
7       this.manager = manager;
```

```
8 - }*/
9 + }
```

Project NewBlog [7]

Project **NewBlog** was introduced context-annotation-config-annotation related errors in its history version. The author introduced <context:annotation-config> element in XML file. After 6 commits, the project NewBlog is correct on this pattern. The errors were fixed by adding @Autowired annotation on the fields. Commit subjects include "init project", "create email template successfully".

Listing B.7: Changes of project NewBlog with context-annotation-config-annotation error

```
1 - public Email(String toAddress, String fromAddress, String emailSubject,
      ↪ String emailContent) {
2 - this.toAddress = toAddress;
3 + private FreeMarkerConfigurer freeMarkerConfig;
4 +
5 + @Required
6 + @Autowired
7 + public void setFreeMarkerConfig(FreeMarkerConfigurer freeMarkerConfig) {
8 + this.freeMarkerConfig = freeMarkerConfig;
9 +
10 + }
```

Project scaffolding [37]

Project **scaffolding** was introduced context-annotation-config-annotation related errors in its history version. The author introduced <context:annotation-config> element in XML file. After 2 commits, the project scaffolding is correct on this pattern. The errors were fixed by adding @Autowired annotation on the fields. Commit subjects include ”add”, ”add right”.

Listing B.8: Changes of project scaffolding with context-annotation-config-annotation error

```
1     public class HomeController {
2 
3   private static final Logger logger = LoggerFactory.getLogger(
        ↪ HomeController.class);
4 -
5 + @Autowired
6 + private UserService userService;
7 +// @Resource
8 +// public void setUserService(UserService userService) {
9 +//  this.userService = userService;
10 +// }
```

Project tmm_phonebook [41]

Project **tmm_phonebook** was introduced context-annotation-config-annotation related errors in its history version. The author introduced <context:annotation-config> element in XML file. After 1 commits, the project tmm_phonebook is correct on this pattern. The errors were fixed by adding @Autowired annotation on the fields. Commit subjects include ”Configured maven, spring, hibernate. Created Entity”, ”Created DAO service. Covered

with test create user. Get By Id methods".

Listing B.9: Changes of project tmm_phonebook

```
1     +@Repository
2 +public class UserActionDao implements UserAction {
3 +
4 + @Autowired
5 + protected SessionFactory sessionFactory;
6 +
7 + @Autowired
8 + private IdGeneratorService idGeneratorService;
9 +
10 + @Transactional
11 + public String createUser(String login, String phone) {
12 + UserEntity toSave = new UserEntity(login, phone);
```

Project tutorial [42]

Project **tutorial** was introduced context-annotation-config-annotation related errors in its history version. The author introduced <context:annotation-config> element in XML file. After 15 commits, the project tutorial tutorial is correct on this pattern. The errors were fixed by adding @Autowired annotation on the fields. Commit subjects include "Created tutorial", "Rename packages so each tutorial has its own package", "Clone tutorial 01 into 02", "Added an example that uses views", "Cloned 02 into 03", "Clone tutorial 3 to start 4", "Added some samples of how to pass parameters to the controller", "Started 05".

Listing B.10: Changes of project tutorial with context-annotation-config-annotation error

```
@Controller
public class SampleController {
+ private static final Comparator<Person> LAST_FIRST_NAME_COMP = new
    ↪ Comparator<Person>() {
+  @Override
+  public int compare(Person o1, Person o2) {
+   int rv = o1.getLastName().compareToIgnoreCase(o2.getLastName());
+   if (rv == 0)
+    rv = o1.getFirstName().compareToIgnoreCase(o2.getFirstName());
+   return rv;
+  }
+ };
+
+ @Autowired
+ private PersonService personService;
```

Project WSSpring [11]

Project **WSSpring** was introduced context-annotation-config-annotation related errors in its history version. The author introduced <context:annotation-config > element in XML file. After 3 commits, the project WSSpring is correct on this pattern. The errors were fixed by adding @Autowired annotation on the fields. Commit subjects include ”initial version”, ”src files check in”.

Listing B.11: Changes of project WSSpring with context-annotation-config-annotation error

```
    +@Repository
```

```
2 +@Transactional
3 +public class EmployeeDAO {
4 +
5 +
6 + @Autowired
7 + private SessionFactory sessionFactory;
8 +
9 + public Employee getById(int id)
10 + {
11 + return (Employee) sessionFactory.getCurrentSession().get(Employee.class,
    ↪ id);
12 + }
```

Project mentor-project-2013 [30]

Project **mentor-project-2013** was introduced field-property-map and setter-property-map related errors in its history version. The author introduced ambiguity when creating directories. The project contained a org.lahendrix.socialbarter directory and a org/lahendrix/-socialbarter. These two directory will generate same fully qualified name for its file inside. Fully qualified names are always used by XML configuration when signing class to the beans. After 2 commits, the author is correct on the pattern. The error was introduced by history commit with subject of "Updated pom and other config info". It is solved by deleting "/src/main/java/org.lahendrix.socialbarter/Greeter.java" since it will cause ambiguity.

Listing B.12: Changes of project mentor-project-2013 with field-property-map and setter-property-map error

```
1 deleted file mode 100644
2 index 887da5c..0000000
3 --- a/src/main/java/org.lahendrix.socialbarter/Greeter.java
```

Project soundService [38]

Project **soundService** had two types of error in its history versions. First type of error was field-property-map related errors in its history version. The author introduced errors when modifying XML file. Since author made changes in multiple versions to this XML file, the initial and final versions of XML file and its corresponding Java file are presented here. As we can see in the initial XML file, it contained ”SoundSocialDAO” and ”SoundActivityDAO” as property name. However, its corresponding Java file did not contain these two fields. After 8 commits, the author is correct on this pattern. The ”SoundSocialDAO” and ”SoundActivityDAO” were deleted. The second type of error was setter-property-map related errors. It is also solved by correcting the name attribute value.

Listing B.13: Changes of project soundService with field-property-map error.

```
1 ---------initial--------------
2 <bean id="soundSocialService" class="com.sound.service.sound.impl.
    ↪ SoundSocialService">
3     <property name="userDAO" ref="userDAO" />
4     <property name="SoundSocialDAO" ref="SoundSocialDAO" />
5     <property name="SoundActivityDAO" ref="SoundActivityDAO" />
6 </bean>
7 -------corresponding java file (initial)-----------
8 public class SoundSocialService implements com.sound.service.sound.itf.
```

```
    ↪ SoundSocialService{
 9
10   @Autowired
11   UserDAO userDAO;
12
13   @Autowired
14   SoundDAO soundDAO;
15
16   @Autowired
17   SoundLikeDAO soundLikeDAO;
18
19   @Autowired
20   SoundRepostDAO soundRepostDAO;
21 ---------final--------------------
22 <bean id="soundSocialService" class="com.sound.service.sound.impl.
    ↪ SoundSocialService">
23     <property name="userDAO" ref="userDAO" />
24     <property name="soundDAO" ref="soundDAO" />
25     <property name="soundLikeDAO" ref="soundLikeDAO" />
26     <property name="soundRecordDAO" ref="soundRecordDAO" />
27 </bean>
28
29 -------corresponding java file (final) -----------
30 public class SoundSocialService implements com.sound.service.sound.itf.
    ↪ SoundSocialService{
```

```
31
32  @Autowired
33  UserDAO userDAO;
34
35  @Autowired
36  SoundDAO soundDAO;
37
38  @Autowired
39  SoundLikeDAO soundLikeDAO;
40
41  @Autowired
42  SoundRecordDAO soundRecordDAO;
```

Project unikkit [43]

Project **unikkit** was introduced field-property-map related errors in its history version. The author introduced errors when modifying XML file. In XML file (shown in List B.15), the bean of class ”EventLazyModel” has ”eventManager” name attribute in property tag. However, in its initial corresponding Java file did not contain this field. After 4 commits, the author is correct on this pattern. The field ”eventManager” was introduced. Commits subjects includes ”Initial version. Added event and quote bean xml” and ”Updated event lazy implementation”.

Listing B.14: Changes of project unikkit with field-property-map error.

```
1 +import org.primefaces.model.ScheduleEvent;
```

```
2
3  public class EventLazyModel extends LazyScheduleModel {
4
5 + private EventManager eventManager;
6 +
7 + /**
8 + * @return the eventManager
9 + */
10 + public EventManager getEventManager() {
11 +  return eventManager;
12 + }
```

Listing B.15: XML file of project unikkit with field-property-map error.

```
1    <bean id="eventModel"
2         class="org.mamce.unikkit.lazy.model.EventLazyModel" >
3      <property name="eventManager" ref="eventManager" />
4    </bean>
```

Project wl-oauth [44]

Project **wl-oauth** [44] was introduced field-property-map related errors in its history version. The error was introduced at the XML configuration of bean with id "uk.ac.ox.oucs.oauth.service. OAuthAdminService". As we can see from the lists below, it mis-typed the "oAuthService" as "OAuthService". After 31 commits, author is correct on this pattern. It finally made the name attribute value of <property> tag consistent with field name in Java file.

Listing B.16: Changes of project wl-oauth with field-property-map error.

```
--- a/impl/src/main/java/uk/ac/ox/oucs/oauth/service/OAuthAdminServiceImpl.
    ↪ java
+++ b/impl/src/main/java/uk/ac/ox/oucs/oauth/service/OAuthAdminServiceImpl.
    ↪ java
@@ -11,7 +11,7 @@ import java.util.Collection;
  * @author Colin Hebert
  */
 public class OAuthAdminServiceImpl implements OAuthAdminService {
- private OAuthService oAuthService;
+ private OAuthService oauthService;
     private ConsumerDao consumerDao;
     private AccessorDao accessorDao;
```

```
--- a/pack/src/main/webapp/WEB-INF/components.xml
+++ b/pack/src/main/webapp/WEB-INF/components.xml
@@ -20,7 +20,7 @@
     </bean>

     <bean id="uk.ac.ox.oucs.oauth.service.OAuthAdminService" class="uk.ac.
         ↪ ox.oucs.oauth.service.OAuthAdminServiceImpl">
- <property name="OAuthService" ref="uk.ac.ox.oucs.oauth.service.
    ↪ OAuthService"/>
+ <property name="oauthService" ref="uk.ac.ox.oucs.oauth.service.
    ↪ OAuthService"/>
         <property name="consumerDao" ref="uk.ac.ox.oucs.oauth.dao.
```

```
            ↪ ConsumerDao"/>
10      <property name="accessorDao" ref="uk.ac.ox.oucs.oauth.dao.
            ↪ AccessorDao"/>
11    </bean>
```

Project wl-solr-search [45]

Project **wl-solr-search** was introduced field-property-map related errors in its history version. The error was introduced from inconsistent <property> name attribute in XML file with its corresponding field in Java file. After 6 commits, the author is correct on this pattern. The wrong <property> name attribute ”indexingExecutor” and ”sessionManager” were deleted.

Listing B.17: Changes of project wl-solr-search with field-property-map error.

```
1    --- a/pack/src/main/webapp/WEB-INF/solr.xml
2 +++ b/pack/src/main/webapp/WEB-INF/solr.xml
3 @@ -24,8 +24,7 @@
4         <property name="searchToolRequired" value="${search.tool.required
            ↪ }"/>
5         <property name="ignoreUserSites" value="${search.usersites.ignored
            ↪ }"/>
6         <property name="contentProducerFactory" ref="uk.ac.ox.oucs.search.
            ↪ solr.ContentProducerFactory"/>
7 - <property name="indexingExecutor" ref="solrIndexingExecutor"/>
8 - <property name="sessionManager" ref="org.sakaiproject.tool.api.
    ↪ SessionManager"/>
9 + <property name="indexQueueing" ref="uk.ac.ox.oucs.search.solr.queueing.
```

```
  ↪ IndexQueueing"/>
10    </bean>
```

Project pocket-review [36]

Project **pocket-review** had two type of errors in its history versions. The first type of errors was method-exist related error. The XML configuration used no exist method as its init-method attribute value. After 2 commits, the author is correct on this pattern. The bean definition is updated by deleting the init-method attribute.

Listing B.18: Changes of project pocket-review with method-exist error.

```
1  --- a/RnR-GAE-Service/src/main/resources/spring-dao.xml
2  +++ b/RnR-GAE-Service/src/main/resources/spring-dao.xml
3  @@ -12,39 +12,39 @@
4
5    <!-- ============== DaoBeans for entities in package com.wadpam.rnr.domain
     ↪  ============ -->
6      <bean id="dProductDao"
7  - class="com.wadpam.rnr.dao.DProductDaoBean" init-method="init">
8  + class="com.wadpam.rnr.dao.DProductDaoBean">
9      </bean>
10
11     <bean id="dRatingDao"
12 -  class="com.wadpam.rnr.dao.DRatingDaoBean" init-method="init">
13 +  class="com.wadpam.rnr.dao.DRatingDaoBean">
```

```
14     </bean>
15
16     <bean id="dLikeDao"
17 - class="com.wadpam.rnr.dao.DLikeDaoBean" init-method="init">
18 + class="com.wadpam.rnr.dao.DLikeDaoBean">
19     </bean>
20
21     <bean id="dThumbsDao"
22 - class="com.wadpam.rnr.dao.DThumbsDaoBean" init-method="init">
23 + class="com.wadpam.rnr.dao.DThumbsDaoBean">
24     </bean>
25
26     <bean id="dCommentDao"
27 - class="com.wadpam.rnr.dao.DCommentDaoBean" init-method="init">
28 + class="com.wadpam.rnr.dao.DCommentDaoBean">
29     </bean>
30
31     <bean id="dFavoritesDao"
32 - class="com.wadpam.rnr.dao.DFavoritesDaoBean" init-method="init">
33 + class="com.wadpam.rnr.dao.DFavoritesDaoBean">
34     </bean>
35
36     <bean id="dAppDao"
37 - class="com.wadpam.rnr.dao.DAppDaoBean" init-method="init">
38 + class="com.wadpam.rnr.dao.DAppDaoBean">
```

```
39     </bean>
40
41     <bean id="dAppSettingsDao"
42 - class="com.wadpam.rnr.dao.DAppSettingsDaoBean" init-method="init">
43 + class="com.wadpam.rnr.dao.DAppSettingsDaoBean">
44     </bean>
45
46     <bean id="dAppAdminDao"
47 - class="com.wadpam.rnr.dao.DAppAdminDaoBean" init-method="init">
48 + class="com.wadpam.rnr.dao.DAppAdminDaoBean">
49     </bean>
50
51  </beans>
```

The second type of error was setter-property-map related error. The class ”GaePreAuthenticatedProcessingFilter” was missing the setter method for ”authenticationManager”. After 32 commits, this project is correct on this rule. The author updated the project with deleting this bean definition.

Listing B.19: Changes of project pocket-review with setter-property-map error.

```
1 - <bean id="gaePreAuthenticationProcessingFilter" class="com.wadpam.rnr.
    ↪ security.GaePreAuthenticatedProcessingFilter">
2 - <property name="authenticationManager" ref="
    ↪ backofficeAuthenticationManager" />
3 - </bean>
```

Project featurepub [20]

Project **featurepub** was involved in setter-property-map error. It did not declare the setter method of ”encoder” in class ”Publisher”. After 1 commits, project featurepub is correct on this rule. The project updated the corresponding Class to ”BasicPublisher” for bean with id ”publisher”. In BasicPublisher, it does has setter method of ”encoder”.

Listing B.20: Changes of project featurepub with setter-property-map error.

```
1 -  <bean id="publisher" class="es.uva.idelab.featurepub.Publisher" >
2 +  <bean id="publisher" class="es.uva.idelab.featurepub.BasicPublisher" >
```

Project SpatialSearch [9]

Project **SpatialSearch** was involved in setter-property-map error. It did not declare the setter method of ”sessionFactory” in class ”HibernateDAOImpl”. After 6 commits, the project is correct on this rule. The project updated the corresponding Class to ”AbstractHibernateDAO” for bean with id ”publisher”. In AbstractHibernateDAO, it does has setter method of ”sessionFactory”.

Listing B.21: Changes of project spatialsearch with setter-property-map error.

```
1 - <bean id="state2010DAO" class="org.geo.spatialsearch.dao.
    ↪ AbstractHibernateDAO">
2 + <bean id="state2010DAO" class="org.geo.spatialsearch.dao.HibernateDAOImpl
    ↪ ">
```

```
1    Exists
2        Field
3         Identifier: f
```

```
4        NoTagName
5         Identifier: c
6        Relational
7         GetAttr
8          XMLName
9           Identifier: property
10         StringLiteral: "name"
11        Equal
12        GetName
13         NoTagName
14          Identifier: f
```