

**EXTENDING STATIC ANALYSIS WITH
APPLICATION-SPECIFIC RULES BY ANALYZING
RUNTIME EXECUTION TRACES**



A Thesis

by

Ersin Ersoy

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Science

Özyeğin University
July 2016

Copyright © 2016 by Ersin Ersoy

**EXTENDING STATIC ANALYSIS WITH
APPLICATION-SPECIFIC RULES BY ANALYZING
RUNTIME EXECUTION TRACES**

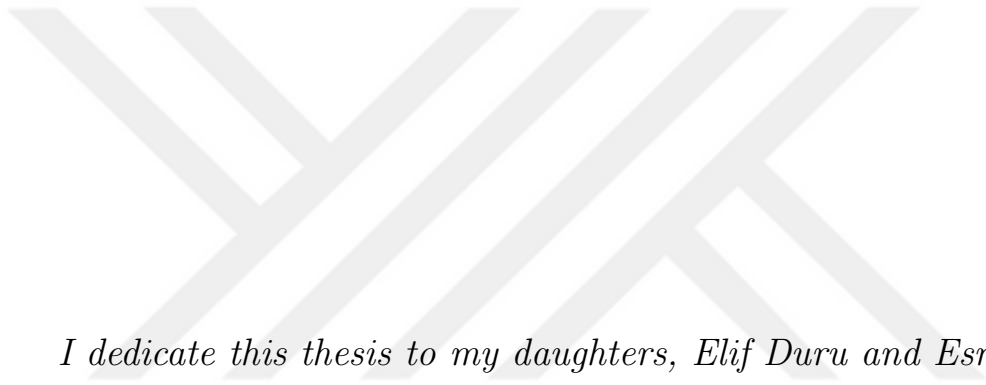
Approved by:

Asst. Prof. Hasan Sözer (Advisor)
Department of Computer Science
Özyeğin University

Asst. Prof. Mehmet Aktaş
Department of Computer Engineering
Yıldız Technical University

Asst. Prof. Barış Aktemur
Department of Computer Science
Özyeğin University

Date Approved: 2016



*I dedicate this thesis to my daughters, Elif Duru and Esmâ Derin.
You provided the inspiration necessary for me to complete this process.*

ABSTRACT

Static code analysis tools can generate alerts regarding only generic issues such as uninitialized variables. They cannot detect violations of application-specific rules. Tools can be extended with specialized checkers that implement the verification of these rules. However, such rules are usually not documented explicitly. Moreover, the implementation of specialized checkers is a manual process that requires expertise. In this thesis, we present a novel approach to provide these extensions automatically. In our approach, application-specific programming rules are automatically extracted from execution traces collected at runtime. These traces are analyzed offline to identify programming rules, of which violation lead to errors. Then, specialized checkers for these rules are introduced as extensions to a static analysis tool so that their violations can be checked throughout the source code. We evaluated our approach with two industrial case studies from the telecommunications domain. We were able to detect real faults with checkers that were automatically generated based on the analysis of execution logs.

ÖZETÇE

Statik kod analiz araçları genel geçer hata tipleri için uyarı oluşturabilmektedir. İlk değerleri atanmamış olan değişkenlere ilişkin hata uyarıları örnek olarak verilebilir. Uygulamaya özel kuralların ihlalini ise mevcut araçlar algılayamamaktadırlar. Bu araçlar, özelleştirilmiş kontrol kuralları ile genişletilebilir ve bu şekilde uygulamaya özel kuralları kontrol edebilirler. Ancak bu kurallar genellikle açık bir şekilde dokümente edilmiş değildir. Üstelik özelleştirilmiş kontrol kuralları manuel olarak hazırlanmaktadır ve bu uzmanlık gerektiren bir iştir. Bu tezde, statik kod analiz kurallarının otomatik olarak genişletilmesi için yeni bir yaklaşım sunulmuştur. Bu yaklaşımda, uygulamaya özel programlama kuralları, uygulama çalışırken oluşan kayıtlardan otomatik olarak elde edilmektedir. Bu kayıtlar çevrimdışı olarak analiz edilip hataya neden olan programlama kuralı ihlali bulunmaktadır. Sonrasında, belirlenen hataya uygun olan özelleştirilmiş kontrol kuralı kullanılarak statik kod analiz aracı genişletilmektedir ve uygulamanın tamamı genişletilen araç ile analiz edilmektedir. Bu yaklaşım telekomünikasyon alanındaki iki vaka analizi ile değerlendirilmiştir. Uygulamaların çalışması esnasında toplanan kayıtlarının analizi sonucunda oluşturulan özelleştirilmiş kontrol kuralları ile gerçek hataların bulunabildiği görülmüştür.

ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor, Dr. Hasan Sözer for all his help and guidance he has provided over the past two years. I would also like to thank Dr. Tankut Barış Aktemur and Dr. Mehmet Aktaş for accepting to be part of the evaluation committee for this thesis.

Secondly, I deeply thank my wife, Gönül Ersoy, my mother, Neşe Ersoy and my father, Ahmet Ersoy for their unconditional trust, timely encouragement, and endless patience. It was their love that raised me up again when I got weary.

I would also like to thank software developers at Turkcell Technology for sharing their code base and supporting the case studies reported in this thesis.

This work is supported by The Scientific and Research Council of Turkey (TUBITAK), grant number 113E548.

TABLE OF CONTENTS

DEDICATION	iii
ABSTRACT	iv
ÖZETÇE	v
ACKNOWLEDGMENTS	vi
LIST OF TABLES	ix
LIST OF FIGURES	x
I INTRODUCTION	1
II BACKGROUND	3
2.1 Static Code Analysis	3
2.2 Sonarqube	8
2.3 PMD	10
III RELATED WORK	13
IV OVERALL APPROACH	15
4.1 Analysis of Execution Logs	17
4.2 Root Cause Analysis	18
4.3 Generation of Specialized Checkers	20
4.4 Extension of Static Code Analysis Tool	21
V EVALUATION	23
5.1 Subject systems	23
5.2 Case Studies	24
VI RESULTS AND DISCUSSION	31
6.1 Results	31
6.2 Discussions	32
6.3 Threats to Validity	33

VII CONCLUSIONS AND FUTURE WORK	34
APPENDIX A — ABSTRACT SYNTAX TREE EXAMPLES . . .	35
APPENDIX B — SONAR DASHBOARD SCREENSHOTS	40
APPENDIX C — REAL FAULT EXAMPLES	45
REFERENCES	46



LIST OF TABLES

1	A sample set of rules that are included in the <i>Basic Rules</i> ruleset of PMD.	10
2	A sample list of errors detected for the SFA system.	33



LIST OF FIGURES

1	The control flow graph model of the code snippet shown in Listing 2.1.	5
2	A call graph that is derived based on Listing 2.2.	7
3	An example AST and the corresponding source code snippet in <i>PMD Rule Designer</i>	12
4	The overall approach.	15
5	A snippet from a log file regarding a <code>NullPointerException</code> error. . . .	25
6	A snippet from the output of <i>Log Parser</i> regarding the instances of the <code>NullPointerException</code> error recorded as shown in Figure 5.	25
7	A snippet from a log file regarding a <code>LazyInitializationException</code> error.	27
8	A snippet from the output of <i>Log Parser</i> regarding the instances of the <code>LazyInitializationException</code> error recorded as shown in Figure 7.	27
9	A snippet from a log file regarding a <code>NullPointerException</code> error on CMS.	29
10	A snippet from the output of <i>Log Parser</i> regarding the instances of the <code>NullPointerException</code> error recorded as shown in Figure 9.	29
11	Generated AST for sample code A.1	36
12	Generated AST for sample code A.2	37
13	Generated AST for sample code A.3	39
14	Sonar dashboard for TSFA module	41
15	Sonar dashboard for TSFAWEB module	42
16	Sonar dashboard for CMS	43
17	Sonar Rules Page	44

CHAPTER I

INTRODUCTION

Static analysis tools [1] can detect the violation of programming rules by checking (violation of) patterns throughout control flow and data flow paths in the program. The detected violations are reported in the form of a list of alerts. Although static analysis tools have been successfully utilized in the industry [2, 3, 4], they have limitations as well. A major limitation is that these tools can generate alerts regarding only generic issues such as uninitialized variables. They fall short to detect the violation of application-specific rules [5]. For example, it might be implicitly assumed for a system that a particular method (e.g., *open*) is always called before a call to another method (e.g., *connect*). It might also be necessary to check some of the arguments and/or return values before/after certain method calls. Such rules are application-specific and they are not considered by static analysis tools by default. Static analysis tools can not also detect many generic faults that can lead to null pointer exceptions for instance. In many cases, it is very hard or undecidable to show whether an execution path is feasible or infeasible without the runtime context information [6].

Some of the static analysis tools can be extended or modified in a modular way. These extensions or modifications can fulfill two different purposes *i)* to suppress some of the generated alerts [7], *ii)* to generate more alerts by introducing additional rules [5]. Hence, one can extend static analysis tools with specialized checkers to detect the violation of application-specific rules as well. However, the implementation of specialized checkers is a manual process that requires expertise. In fact,

state-of-the-art static analysis tools provide special extension mechanisms for defining new rules, which can be then checked by these tools. Yet, such rules have to be defined manually and they are usually not documented explicitly or formally.

In this thesis, we present a novel approach to extend static analysis tools, in which specialized checkers are generated automatically. Our approach employs offline analysis of execution traces collected at runtime. These traces comprise a set of encountered errors. The source code is analyzed to identify the root causes of the detected errors. Then, rules are inferred to prevent these root causes. Specialized checkers are automatically generated for these rules and they are introduced as extensions to a static analysis tool. The extended tool can detect the violation of the inferred rules throughout the source code.

We evaluated our approach with two industrial case studies from the telecommunications domain. The execution logs of previous versions of two large scale legacy systems were analyzed to generate specialized checkers for certain types of errors. The static analysis tool that is employed in the company is extended with these checkers. The extended tool was able to detect real faults. We observed that some of these faults were fixed in later versions of the source code.

The remainder of this thesis is organized as follows. In the following chapter, we provide background information on static code analysis in general and particular tools that are employed in our case studies. We summarize the related studies in Chapter 3. We present the overall approach in Chapter 4. The approach is illustrated and evaluated in Chapter 5, in the context of two industrial case studies. We present and discuss the results in Chapter 6. Finally, in Chapter 7, we provide the conclusions and discuss possible future directions.

CHAPTER II

BACKGROUND

In this chapter, we provide background information on static code analysis. In particular, we introduce two tools, namely Sonar and PMD, which are employed in our case studies.

2.1 Static Code Analysis

Static code analysis (SCA) or static program analysis is used for checking software quality and potential faults without executing the program [8]. It is usually performed by analyzing the source code of the program only. However, it is also possible to perform the analysis on the executable/compiled code [8]. SCA is generally applied in code review phase but it is more and more adopted in implementation phase of the Software Development Life Cycle (SDLC) as well. It can be used easily in implementation phase because SCA functionalities are usually employed in continuous integration environments and Integrated Development Environments (IDE's), such as Eclipse. It is very useful to get warnings regarding potential faults in development time. This creates an opportunity to fix the code quickly and easily.

Common bug types that can be captured by SCA are listed below [9].

- *Improper resource management*: Finding memory leaks, connection left open, file left open, etc.
- *Illegal operations*: Division by zero, array out of bound exceptions, null pointer exceptions, etc.
- *Dead code and data*: Non reachable code or data

- *Incomplete code*: Lack of variable initialization, not valid return statement in function/method, not valid if-then-else statement etc.

In fact, currently SCA tools are not only used to find bugs but also to find code duplication percentage, complexity of the software based on blocks/functions/packages and also to measure size of software using LOC or function point methods. In the beginning, simple SCA tools were developed and used for finding security related problems. These tools mainly used lexical analysis. Their analysis were mainly based on a string search for predefined keyword list. As a result of compiler technology improvements, Abstract Syntax Tree (AST) is started to be used for static analysis. AST converts source code to a kind of tree structure with which it is possible to run some semantic analysis. Currently modern SCA tools use AST [10] as the underlying model. There are several example AST structures provided in Appendix A.

State-of-the-practice SCA tools mainly employ three types of analysis: control flow graph analysis, call graph analysis, and data flow analysis. In the following, we provide brief information on these analysis techniques.

Control flow graph (CFG) basically represents software instructions and the flow of control among these instructions to cover all possible paths end to end for a program [10]. There is a sample code in Listing 2.1. The CFG model of this sample code is shown in Figure 1. Hereby, there are 5 nodes labelled as bb0, bb1, bb2, bb3, and bb4. Each of these nodes represent an instruction taking place in the source code. Directed edges among the nodes represent a possible flow of control among the instructions. When the program is executed, there are only two paths to run instructions in Figure 1. The first path involves bb0, bb1, bb2 and bb4 in the given order. The second possible sequence of instructions include bb0, bb1, bb3 and bb4. This is, of course, a very simple presentation of a CFG model. In order to perform advanced analysis, a SCA tool also maps elements of a CFG to elements of an AST [10].

Listing 2.1: An example code fragment that includes an if-else block.

```
a=getStatus ();  
if (a>1) {  
    result=0; // Fail  
} else {  
    result=1; // Success  
}  
return result ;
```

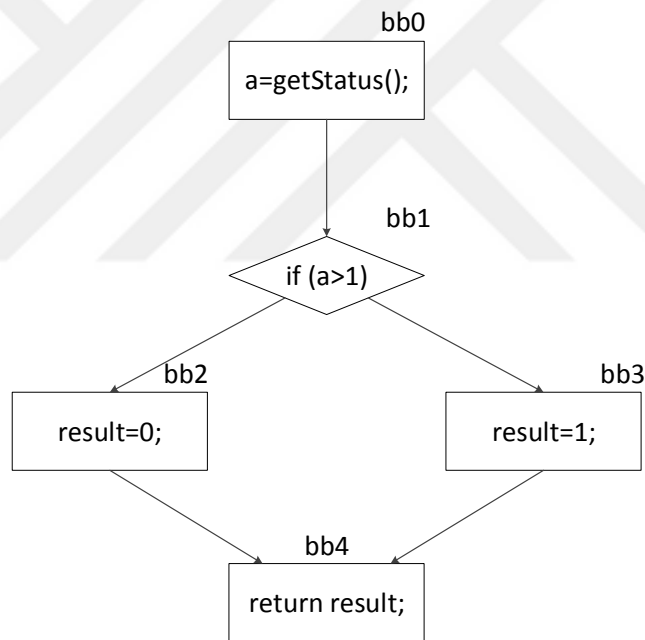


Figure 1: The control flow graph model of the code snippet shown in Listing 2.1.

Call graph represents (call) relationships between methods and functions [10]. There is a sample code in Listing 2.2 and the corresponding call graph is provided in Figure 2. From this call graph, we can see that *sayHello* calls *sayByeBye*, and *sayByeBye* calls itself. We can also see that *saySomething* can call *returnNameFromDB* and *sayHello*.

Listing 2.2: An example code fragment used for deriving a call graph.

```
void sayHello(String name)
{
    System.out.println(" Hello " + name);
    sayByeBye(name);
}
void sayByeBye(String name)
{
    System.out.println(" Bye Bye " + name);
    sayByeBye(name);
}
String returnNameFromDB(int customerId)
{
    return "Ersin";           //Not implemented yet.
}
void saySomething
{
    String customerName= returnNameFromDB(1);
    if(customerName!=null && customerName.length()>0)
    {
        sayHello(customerName);
    }
    else
    {
        System.out.println(" Error !!!");
    }
}
```

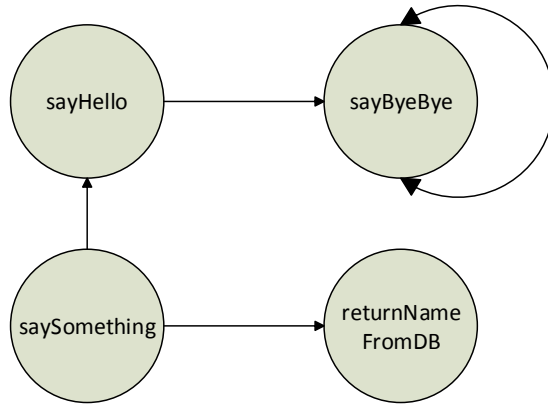



Figure 2: A call graph that is derived based on Listing 2.2.

Data flow analysis is used for detecting *dead code*, *performance issues*, *useless declarations* and *uninitialized variables*. CFG analysis by itself is not enough to perform such analysis. Program statements have to be annotated for recording different values assigned to variables. Each statement is represented with so-called Static Single Assignment (SSA), which assigns each value to single variable. Different value assignments to the same variable are differentiated by using different numeric subscripts on variable names [10]. A sample code fragment and the corresponding SSA form is listed below.

Sample code: **SSA form:**

$sum = a + b;$ $sum_1 = a_1 + b_1;$

$b = b + c;$ $b_2 = b_1 + c_1;$

$sum = sum * b;$ $sum_2 = sum_1 * b_2;$

In the following subsections, we introduce the tools that are used for implementing our approach. These tools were already being used by the company, where we performed our case studies. Therefore, we employed them to ensure compatibility and easy integration with the existing processes.

2.2 *Sonarqube*

*Sonarqube*¹ (also known as *Sonar*) is a common platform on which multiple SCA tools can be deployed. It provides a continuous integration environment, in which all the deployed tools are automatically executed and their results are presented via automatically generated reports. In this work, we used Sonar to execute our custom rules and obtain reports regarding the analysis results.

Sonar provides customization options by means of so-called *dashboards*. Using these dashboards, it is possible to see analysis results at different levels of detail. Sonar also has various plugins that facilitate integration with different SCA tools (Findbugs [11], PMD [12], etc.) and software development environments (Eclipse [13], NetBeans [14], etc.). We developed our custom rules as extensions to PMD. These rules are executed by Sonar using its PMD Sonar plugin.

Sonar has already been employed by several companies, including the one where we performed our case studies. Its main advantage is to provide a common platform [15] for the execution of multiple analysis tools and consolidating their results in its reports. As such, it combines the types of analysis that can be performed only by a combination of SCA tools. The common types of analysis are listed below:

- *Potential bugs* : Analyze potential problems in source code, such as possible null pointer exceptions. There are five severity levels defined by Sonar for such potential issues: blocker, critical, major, minor and info.
- *Coding rules* : Analyze violations of coding rules in source code, such as naming conventions (e.g., Class names should start with uppercase letters). There exist five severity levels for the violations of coding rules as well.
- *Unit Tests* : Report on unit test case count, code coverage of these unit tests. If code coverage is less than 60%, Sonar creates an issue for poorly code coverage.

¹<http://www.sonarqube.org/>

- *Duplications* : Analyze code duplications in all files. It is possible to see the percentage values based on project, class, or in more detail at the block level.
- *Comments* : Report on comment percentage in the form of lines of commented code / total number of lines of code.
- *Architecture and design* : Provide hints for refactoring at the architecture design or detailed design level, e.g., two different classes in different packages generally make use of each other and they should have been placed in the same package.
- *Complexity* : Report on code complexity by using *cyclomatic complexity* as the metric [16].

2.3 PMD

In this section, we will provide brief information about PMD [12], which is the SCA tool that we employed in our approach. In fact, we extended this tool with custom checkers that are automatically generated based on custom rules defined as templates. We will explain the details of this extension mechanism later, in Section 4.3. In this subsection, we will only provide general information about the tool.

PMD is an open-source SCA tool available online². It can analyze source code developed with a variety of languages such as Java, JavaScript, PLSQL, Apache Velocity, XML, and XSL. Its duplicate code checker functionality supports many more, e.g., C, Ruby, Scala, Objective C, Matlab, Python, Go, and Swift.

PMD basically checks for violations of a set of rules in the source code. There are many predefined rules [17] provided by the tool. These rules are categorized in groups such as *security*, *performance*, *design*, etc. Currently there are 29 different groups, each of which is called a *ruleset*. A few example rules included in the *Basic Rules* ruleset is shown in Table 1.

Rule Name	Rule Description
EmptyCatchBlock	Empty Catch Block finds instances where an exception is caught, but no action is performed. Such code blocks swallow an exception which should either be handled or reported.
MisplacedNullCheck	The null check is misplaced. A null value of the variable will lead to a NullPointerException. Either the check is incorrect or useless, i.e., the variable will never be null.
SingularField	This field is used in only one method and the first usage is assigning a value to the field. This probably means that the field can be changed to be a local variable.

Table 1: A sample set of rules that are included in the *Basic Rules* ruleset of PMD.

In addition to such predefined rules categorized in rulesets, PMD allows for extensions with custom rules. There are two ways to define and develop custom rules in

²<https://pmd.github.io/>

PMD: *i*) in the form of plain Java methods, and *ii*) in the form of XPath expressions. We have developed our custom rules using the Java language.

PMD uses JavaCC³ to parse the source code and generate its AST. It provides an API to traverse this AST and define specialized checkers for custom rules. Figure 3 depicts a sample AST and the corresponding source code snippet being viewed with the *PMD Rule Designer* tool. One can follow the source code and the corresponding AST relations via this user interface.

PMD was already being used by the software test department within the company, where we performed our case studies. It was already integrated to the continuous integration tool chain as part of the Sonar environment. So, we implemented our approach around this tool to be able to perform case studies. Otherwise, there is no particular reason for employing the PMD tool in our approach. In principle, our approach could be applied with any SCA tool as long as it supports programmable checker extensions.

³<https://javacc.java.net/>

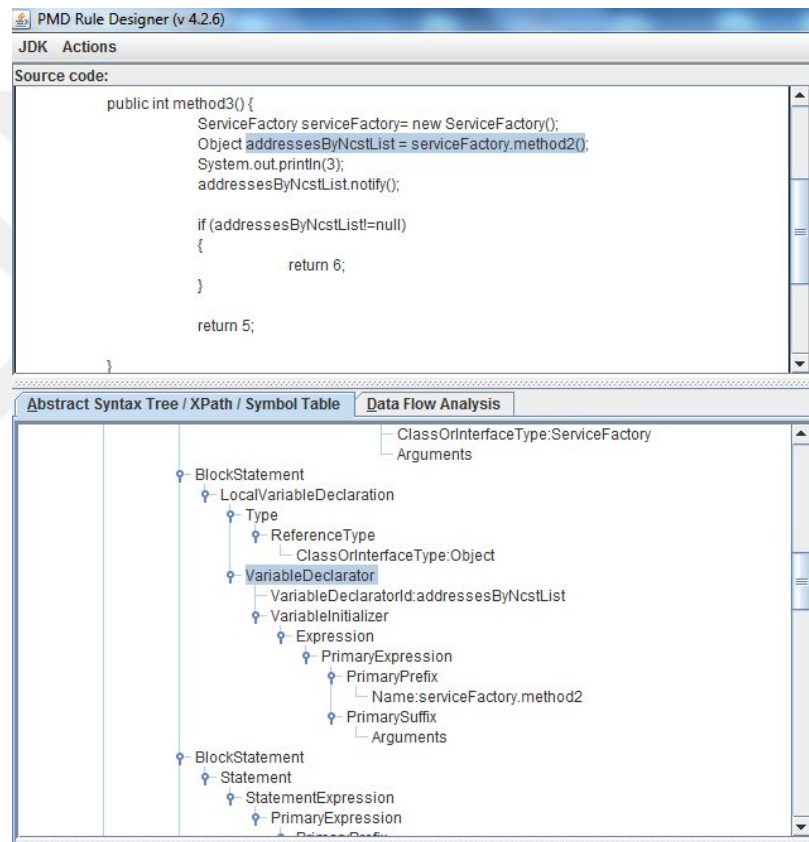


Figure 3: An example AST and the corresponding source code snippet in *PMD Rule Designer*.

CHAPTER III

RELATED WORK

There have been studies for automatically deriving programming rules based on frequently used code patterns [18, 19]. Hereby, pattern recognition, data mining and heuristic algorithms are used for analyzing the program source code and detecting potential rules. Then, the source code is analyzed again to detect inconsistencies with respect to these rules. These studies utilize only (models of) the source code to infer programming rules. They do not make use of runtime execution traces.

Another study [20] makes use of the analysis of previously fixed bugs to derive application-specific programming rules. This is a semi-automated approach, where the programmer defines the rules applied to fix these bugs. Rules are specified in the form of dependence subgraphs [20]. Then, graph matching algorithms are used for locating violations of the defined rules. As such, other instances of the previously fixed bugs are identified in the overall source code. This approach do not also exploit any information collected during runtime execution.

The method proposed by Williams et al. [21] relies on the history of bug fixes as well. In this method, these fixes are manually analyzed to identify common bug types first. Then, programming rules are defined to detect these types of bugs. In their case studies, it was observed that the root cause of many bugs was the lack of checks on the method return values. As a result, a new programming rule was introduced to check the return value of every method. In our work, we utilize runtime execution traces to derive programming rules. This enables us to determine error-prone scenarios more precisely. For instance, rather than enforcing the control of return values for all the methods, we define rules only for methods that contribute to a failure scenario. In

addition, our approach is automated. It does not rely on manual analysis.

In general, studies reported in the literature mainly utilize source code, previously fixed bugs, changes applied to fix these bugs or changes that introduce bugs [22] for deriving programming rules. There exist a few approaches [23, 24, 25] that exploit dynamic analysis and runtime execution traces. DynaMine [25] uses dynamic analysis for validating programming rules that are actually derived by mining the revision history. The program is instrumented to simulate the effect of the violation of rules and as such, provide feedback to the programmer. Previously, console logs were analyzed using machine learning techniques to detect anomalies [23]; however, deriving rules for preventing these anomalies was out of the scope of the study. Daikon [24] derives likely invariants of a program by means of dynamic analysis. However, Daikon focuses on numerical properties of variables as system constraints rather than bug patterns that can represent a wider range of bug types. The output of Daikon has been used for supporting different verification techniques [24]. It was also used for supporting static analysis [26]; however, the goal of this study was to generate test cases based on static analysis, not to extend static analysis.

We have previously introduced an integrated static and dynamic analysis approach for improving both runtime verification and static code analysis tools [7]. In that approach, the goal is to identify alerts, which do not actually cause any failures at runtime. Then, filters are automatically generated for a static code analysis tool to suppress these alerts. Hence, the goal is to reduce false positives and increase precision. In this work, we aim at reducing false negatives by detecting more faults as a result of checking application-specific rules. As such, the goal of the approach proposed in this thesis is to increase recall instead.

CHAPTER IV

OVERALL APPROACH

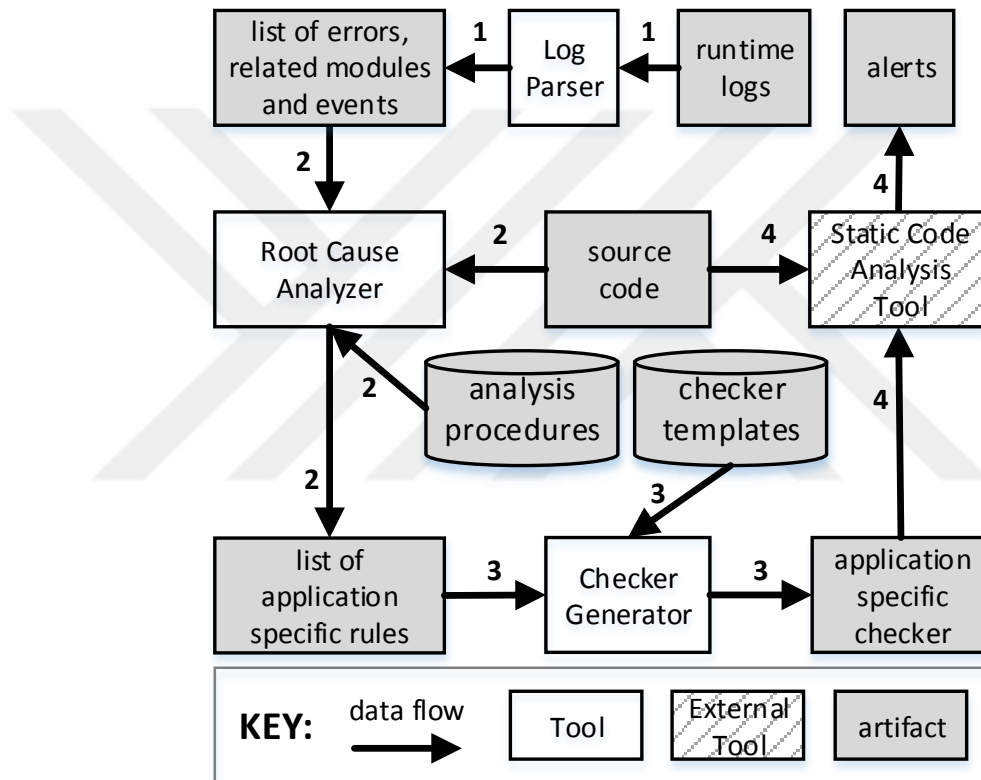


Figure 4: The overall approach.

The overall approach is depicted in Figure 4, which involves 4 steps. First, *Log Parser* takes runtime logs as input, parses these logs, and generates the list of errors recorded together with the related modules and events (1). Then, this list is provided to *Root Cause Analyzer*, which analyzes the source code to identify the cause of the error by utilizing a set of predefined analysis procedures (2). For instance, if a null pointer reference error is detected at runtime, the corresponding analysis procedure

analyzes the source code to locate the corresponding object and its last definition before the error. Let's assume that such an object was defined as the return value of a method call. Then, a rule is inferred, imposing that the return value of that particular method must be checked before use. The list of such rules are provided to *Checker Generator*, which uses a library of predefined templates to generate a specialized checker for each rule (3). The generated checkers are included as extensions to a *Static Code Analysis Tool*, which applies them to the source code and reports alerts in case violations are flagged (4).

As can be seen in Figure 4, the overall approach is automated; however, it relies on a set of predefined analysis procedures and checker templates for performing root cause analysis and checker generation, respectively. One analysis procedure should be defined for each error type and one checker template should be defined for each rule type. There can be many different types of application-specific rules that can be checked with static analysis. These rules and the considered error types are open-ended in principle and they can be extended when needed. Currently, we consider the following types of errors and programming rules that are parametrized with respect to the involved method and argument names.

- *java.lang.NullPointerException*: The return value of a method must be checked for null reference after it is called, e.g., $r = m(x); \text{if}(r \neq \text{null}) \{ \dots \}$ or $\text{if}(r == \text{null}) \{ \dots \}$
- *org.hibernate.LazyInitializationException*: The JPA Entity¹ should be initialized at a transactional level (when persistence context is alive) before being used at a non-transactional level, e.g., object a is a JPA Entity with *LAZY* fetch type and it is an aggregate within object b . Then, a must be fetched from the database when b is being initialized, for a possible access after the persistent

¹A JPA (Java Persistence API) entity is a POJO (Plain Old Java Object) class, which has the ability to represent objects in a database. They can be reached within a persistent context.

context is lost.

In the following, we explain the steps of the approach in more detail by explaining the handling of null pointer exceptions as the running example. Then, in Chapter 5, we illustrate the application of the approach in the context of two industrial case studies. We developed the necessary tools and applied our approach for software systems written in Java. In principle, the approach can be instantiated for different programming languages and environments. Our design and implementation choices were driven by the needs and the context of the industrial case.

4.1 Analysis of Execution Logs

The first step of our approach involves the analysis of execution logs. We have applied our approach to legacy systems. Hence, we did not have the chance to instrument the code and collect execution traces according to a predefined format. We had to utilize the existing log files. Therefore, *Log Parser* is implemented as a dedicated parser for these files. However, it can be replaced with any parser to be able to process log files in other formats as well. So, our approach is agnostic to the log file structure as long as the necessary information can be derived. The recorded events should include at least the following information: *i)* Sequence of events and in particular, encountered errors; *ii)* The types of encountered errors; *iii)* The location of the encountered errors in the source code, i.e., package, class, method name, line number. This information is usually recorded in any system in some format. Even standard Java exception reports include such information together with a detailed stack trace. Hence, existing instrumentation and logging tools can be employed to obtain the necessary information.

Log Parser is parametric with respect to the focused error types and modules of the system. We can filter out some error types or modules that are deemed irrelevant or uncritical.

4.2 *Root Cause Analysis*

Once *Log Parser* retrieves the relevant error records together with their context information, it provides them to *Root Cause Analyzer*. This tool performs two main tasks: *i)* finding the root cause of the error, *ii)* determining whether this root cause is application-specific or not. We are not interested in generic errors. Hence, it is important to be sure that the root cause of the error is application-specific. For instance, consider the code snippet in Listing 4.1. When executed, it causes a *java.lang.NullPointerException*; however, *Root Cause Analyzer* ignores this error because, the cause of the error is an uninitialized object at Line 1. This is a generic error. The object has a null value because it is simply left uninitialized.

Listing 4.1: An example code snippet for a generic error that is ignored by *Root Cause Analyzer*.

```
static Report aReport;

public static void print() {
    System.out.println(aReport);
}
```

Listing 4.2: An example code snippet for an application-specific error that is considered by *Root Cause Analyzer*.

```
static Report aReport
    = getServiceReport();

public static void print() {
    System.out.println(aReport);
}
```

If the null value is obtained from a specific method in the application, then such an error is deemed relevant (See Listing 4.2). That means, the return value of the corresponding method (e.g., *getServiceReport* in Listing 4.2) must be always checked before use. This is a type of rule that is determined by *Root Cause Analyzer*.

Algorithm 1 Root cause analysis procedure applied for errors that lead to null pointer exceptions.

```

1:  $u \leftarrow$  use of object that causes the exception
2:  $dcp \leftarrow$  definition clear path for  $u$ 
3:  $d \leftarrow$  definition of object in  $dcp$ 
4: if  $\exists$  method  $m$  as part of  $d$  then
5:    $p \leftarrow$  package of  $m$ 
6:   if  $p \in$  application packages then
7:      $reportRule(\text{RETURNVALCHECK}, m)$ 
8:   end if
9: end if

```

Root Cause Analyzer employs a set of predefined analysis procedures that are coupled with error types. For example, the analysis procedure applied for errors that cause null pointer exceptions is listed in Algorithm 1. Hereby, the use of the object that caused a null pointer exception is located as the first step. Second, the definition clear path is derived, which is the path from definition of the object to the use of the object, on which the definition is not killed by another definition of the same object. If the definition on this path is performed with a method call, the procedure checks where the method is defined. If the method is defined within one of the packages of the application, then a rule is reported for checking the return value of this method.

Root Cause Analyzer provides the type of rule to be applied and the parameters of the rule (e.g., name of the method, of which return value must be checked) to *Checker Generator* so that a specialized checker can be created. In the following subsection, we discuss the checker generation process.

4.3 Generation of Specialized Checkers

Most of the static code analysis tools are extensible and they provide application programming interfaces (API) for implementing custom checkers. *Checker Generator* generates specialized checkers by utilizing/extending PMD as the static code analysis tool.

As mentioned in Chapter 2, there are two ways to define custom rules and develop specialized checkers for these rules in PMD: *i)* with Java, and *ii)* with XPath expressions. We developed specialized checkers with Java, conforming to the Visitor design pattern [27]. Each checker is basically defined as an extension of an abstract class, namely, *AbstractJavaRule*. The *visit* method that is inherited from this class must be overwritten to implement the custom check. This method takes two arguments: *i)* *node* of type *ASTMethodDeclaration* and *ii)* *data* of type *Object*. The return value is of type *Object*. This visitor method is called by PMD for each method in the source code. In each call, the visited method is provided as the first argument. There are also other visitor methods defined for other types of nodes in the AST such as expressions and declarations.

Checker generation is performed based on parametrized templates. We defined a template for each rule type. Each template extends the *AbstractJavaRule* class and overwrites the necessary visitor methods. A checker is generated by instantiating the corresponding template by assigning concrete values to its parameters. For instance, consider a specialized checker that enforces the handling of possible null references returned from a method in the application. The corresponding pseudo code that is implemented with PMD is listed in Algorithm 2. Hereby, all variable declarations are obtained as a set (V at Line 1). For each of these declarations (v), the node ID (vid) is obtained (Line 3). The name of the method call (m) is also obtained, assuming that the declaration involves a method call (Line 4). If there indeed exists such a method call and if the name of the method matches the expected name

(i.e., *METHOD*), then an additional check is performed (*isNullCheckPerformed* at Line 6). This check traverses the AST starting from the node with id *vid* and searches for control statements that compare the corresponding variable (*v*) with respect to null (i.e., *if(v != null) {...}* or *if(v == null) {...}*). If there is no such a control statement before the use of the variable, then a violation of the rule is registered at Line 8.

Algorithm 2 visit method of a specialized checker for a custom rule, i.e., *handle possible null pointer after calling the method*.

```

1:  $V = \text{getChildrenOfType}(\text{ASTVariableDeclarator});$ 
2: for all  $v \in V$  do
3:    $\text{vid} = v.\text{getID}();$ 
4:    $m = v.\text{getMethodCall}();$ 
5:   if  $m! = \text{null} \ \& \ m.\text{name} == \text{METHOD}$  then
6:      $\text{isChecked} = \text{isNullCheckPerformed}(\text{vid})$ 
7:     if  $!\text{isChecked}$  then
8:        $\text{addViolation}(\text{vid})$ 
9:     end if
10:  end if
11: end for

```

Checker Generator generates specialized checkers by instantiating the corresponding template with the parameters (e.g., *METHOD*) provided by *Root Cause Analyzer*. Hence, multiple checkers can be generated based on the same rule type.

4.4 *Extension of Static Code Analysis Tool*

PMD is extended with the custom checkers generated by *Checker Generator* and it is executed by Sonar version 4.0 (currently known as SonarQube™, available as version 5.6).

The extension is performed in two steps: *i*) adding a jar file that includes the custom checker, and *ii*) extending the XML configuration file for rule definition. The jar file basically contains an instantiation of a checker code template. The resulting concrete source code is compiled into a jar file and included in a specific folder (i.e.,

extensions/rules/pmd) under the Sonar installation. The rule regarding the introduced checker must also be defined in the XML configuration file of PMD as a new entry. Then, the rule can be activated on the Sonar dashboard. A sample XML code snippet for the introduction of a new rule is shown in Listing 4.3. Hereby, the *name*, *message* and *description* are displayed to the user as part of the listed alerts, when violations are detected. The *class* property points at the corresponding checker code. In addition, a *priority*, and a compatible version number (*since*) have to be provided for each rule. Such a new *rule* entry is added inside the *ruleset* entry for each generated specialized checker.

Listing 4.3: A sample XML code snippet for the introduction of a new rule in PMD.

```
<ruleset name="Sfa Custom Rules"
...
<rule name="PossibleNullPointerAtMethodCallRule-
    JavaForMethodTemplateDAOFind"
    since="0.1"
    message="Handle possible nullpointer after calling
        templateDao.find method"
    class="com.turkcell.custom.pmd.rules.
        PossibleNullPointerAtMethodCallRuleForMethodTemplateDAOFind"
    <description>
        Handle possible nullpointers after calling
            templateDao.find method
    </description>
    <priority>3</priority>
</rule>
...
</ruleset>
```


CHAPTER V

EVALUATION

We have performed two case studies to detect faults in a Sales Force Automation (SFA) system and Campaign management system (CMS). These faults lead to `LazyInitializationException` errors and `NullPointerException` errors, which are caused by application specific method calls. CMS does not employ JPA. Hence, `LazyInitializationException` errors are not applicable for this case study. As such, we focused on only `NullPointerException` errors for the CMS case.

In this chapter, we illustrate the application of our approach on SFA and CMS. Then, we will discuss the results in the next chapter. We can not share real code snippets due to confidentiality; however, we will present representative (slightly modified) examples that remain relevant for illustration and evaluation.

5.1 Subject systems

In the following, we briefly introduce the subject systems that used in our case studies.

5.1.1 Sales Force Automation (SFA) system

The code base of this system is maintained by Turkcell¹, which is the largest mobile operator in Turkey. The system comprises more than 200 KLOC. It is operational since 2013, serving 2000 users. This system is used by corporate sales group to manage and monitor sales, opportunity, meeting activities. It can be used via both desktop and mobile platforms. We analyzed two modules of the SFA system, namely TSFA and TSFAWEB. TSFAWEB is the user interface module. TSFA is the core module of SFA. All services are deployed in TSFA to be accessible by the other modules.

¹<http://www.turkcell.com.tr>

5.1.2 Campaign management system(CMS)

The code base of this system is maintained by Turkcell as well. The system comprises more than 250 KLOC. It is used for handling inbound and outbound campaign definitions and executions. There are lots of integration points for reach different types customers and communication channels such as SMS, e-mail, IVR etc. CMS processes millions of requests every day via these integration points. The system is operational since 2009, serving all Turkcell group companies (Turkcell, Superonline², Lifecell³, KKTCcell⁴, etc.).

5.2 Case Studies

We have utilized *Log Parser* to focus our search in the log files. In particular, we have discarded some error types and some packages that are not critical. First, we downloaded all the log files regarding a previous version of the system from different application servers⁵.

5.2.1 NullPointerException Error In SFA

A sample log content is shown in Figure 5. This snippet shows information collected regarding a NullPointerException error. *Log Parser* consolidates information regarding the recorded errors in multiple log files possibly generated at different application servers. For instance, Figure 6 shows a snippet from the output of *Log Parser* regarding the instances of the NullPointerException error as shown in Figure 5. Hereby, we can see the location in the source code (e.g., line number and file name), where the error first occurred.

Then, the source code is analyzed by *Root Cause Analyzer* based on the output of *Log Parser*. The corresponding source code snippet is listed in Listing 5.1. The

²<http://www.superonline.net/>

³<http://www.lifecell.com.ua/ru/>

⁴<http://www.kktcell.com/>

⁵Several instances of the system are running on different application servers.

```

2014-12-23 19:02:41,808 ERROR util.LogUtil (LogUtil.java:56) <> - thread id:38;; class:com.turkcell.crm.sfa.service.rest.OpportunityServiceResource;;method:getOpptyNotes;;
class:sun.reflect.GeneratedMethodAccessor9943;;method:invoke;;NullPointerException ;;java.lang.NullPointerException
at com.turkcell.crm.sfa.service.rest.OpportunityServiceResource.getOpptyNotes(OpportunityServiceResource.java:293)
at sun.reflect.GeneratedMethodAccessor9943.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at org.springframework.web.method.support.InvocableHandlerMethod.invoke(InvocableHandlerMethod.java:219)
at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:132)
at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:104)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandleMethod(RequestMappingHandlerAdapter.java:745)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:686)
at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:80)
at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:925)
at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:856)
at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:936)
at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:827)

```

Figure 5: A snippet from a log file regarding a `NullPointerException` error.

```

*****ERROR type:0*****
firstErrorline      :2014-12-23 19:02:41,808 ERROR util.LogUtil (LogUtil.java:56) <> - thread id:38;;
class:com.turkcell.crm.sfa.service.rest.OpportunityServiceResource;;method:getOpptyNotes;;
class:sun.reflect.GeneratedMethodAccessor9943;;method:invoke;;NullPointerException ;;
firstClassStackTrace:at com.turkcell.crm.sfa.service.rest.OpportunityServiceResource.getOpptyNotes(OpportunityServiceResource.java:293)
firstTStackTrace:at com.turkcell.crm.sfa.service.rest.OpportunityServiceResource.getOpptyNotes(OpportunityServiceResource.java:293)
*****
***** FULL ERROR DETAIL: *****
*****TOPLAM TUM DOSYALARDA 1 kere olmus *****
*****AYNI KODUN ATTIGI FARKLI CESIT HATA SAYISI 1 *****
----- TYPE : 1 -----
java.lang.NullPointerException

```

Figure 6: A snippet from the output of *Log Parser* regarding the instances of the `NullPointerException` error recorded as shown in Figure 5.

error was in Line 3, since the object *oppty* was null. Then, *Root Cause Analyzer* located the point in the source code, where this object was last defined. That is Line 1 in Listing 5.1. It turned out that the definition was coming from a method call, i.e., *templateDao.find(Oppty.class, opptyNo)*; This method performs an object-relational mapping [28]. It creates and returns an object by utilizing information from a relational database. If the required information cannot be found in the database, a null value is returned.

Listing 5.1: The code snippet corresponding to the logged error.

```
1 Opty opty = templateDao.find(Opty.class, optyNo);
2 ...
3 if (opty.getCoptycategory().equals(...))
4 {
5     ...
6 }
```

Then, an application-specific rule is inferred as: the return value of the method *find* must be checked for null references before use. A specialized checker is automatically generated based on this rule. It checks the whole code base and searches for initialized objects using the return value of the method *find* without a null reference check. As the last step, Sonar is extended with the specialized checker, which can be (de)activated on the Sonar dashboard.

```

2015-02-06 16:01:59,659 ERROR util.LogUtil (LogUtil.java:80) <> - thread id:62;;
class:com.turkcell.crm.sfa.web.bean.user.PositionListBackingBean;;method:fetchNewRecordDataForUpdate;; entity_type: Pozisyon;;
LazyInitializationException: failed to lazily initialize a collection of role: com.turkcell.crm.sfa.entity.user.Position.substitutePositions, no session or session was closed
org.hibernate.LazyInitializationException: failed to lazily initialize a collection of role: com.turkcell.crm.sfa.entity.user.Position.substitutePositions, no session or session was
closed
at org.hibernate.collection.AbstractPersistentCollection.throwLazyInitializationException(AbstractPersistentCollection.java:380)
at org.hibernate.collection.AbstractPersistentCollection.throwLazyInitializationExceptionIfNotConnected(AbstractPersistentCollection.java:372)
at org.hibernate.collection.AbstractPersistentCollection.readSize(AbstractPersistentCollection.java:119)
at org.hibernate.collection.PersistentBag.isEmpty(PersistentBag.java:255)
at org.apache.commons.collections.CollectionUtils.isEmpty(CollectionUtils.java:978)
at org.apache.commons.collections.CollectionUtils.isNotEmpty(CollectionUtils.java:991)
at com.turkcell.crm.sfa.web.bean.user.PositionListBackingBean.fetchNewRecordDataForUpdate(PositionListBackingBean.java:1231)
at sun.reflect.GeneratedMethodAccessor43082.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at com.sun.el.parser.AstValue.invoke(AstValue.java:234)
at com.sun.el.MethodExpressionImpl.invoke(MethodExpressionImpl.java:297)

```

Figure 7: A snippet from a log file regarding a `LazyInitializationException` error.

```

*****ERROR type:1*****
firstErrorline      :2015-02-06 16:01:59,659 ERROR util.LogUtil (LogUtil.java:80) <> - thread id:62;;
firstClassStackTrace:
class:com.turkcell.crm.sfa.web.bean.user.PositionListBackingBean;;method:fetchNewRecordDataForUpdate;; firstClassStackTrace:at
org.hibernate.collection.AbstractPersistentCollection.throwLazyInitializationException(AbstractPersistentCollection.java:380)
firstTSFAClassStackTrace:at com.turkcell.crm.sfa.web.bean.user.PositionListBackingBean.fetchNewRecordDataForUpdate(PositionListBackingBean.java:1231)
*****
***** FULL ERROR DETAIL: *****
*****TOPLAM TUM DOSYALARDA 1 kere olmus *****
*****AYNI KODUN ATTIGI FARKLI CESIT HATA SAYISI 1 *****
-----
----- TYPE : 1 -----
entity_type: Pozisyon;;LazyInitializationException: failed to lazily initialize a collection of role: com.turkcell.crm.sfa.entity.user.Position.substitutePositions, no session or
session was closedorg.hibernate.LazyInitializationException: failed to lazily initialize a collection of role: com.turkcell.crm.sfa.entity.user.Position.substitutePositions, no session
or session was closed

```

Figure 8: A snippet from the output of *Log Parser* regarding the instances of the `LazyInitializationException` error recorded as shown in Figure 7.

5.2.2 `LazyInitializationException` Error In SFA

Another sample log content is shown in Figure 7. This snippet shows information collected regarding a `LazyInitializationException` error. Figure 8 shows a snippet from the output of *Log Parser* regarding the instances of the `LazyInitializationException` error as shown in Figure 7.

Then, the source code is analyzed by *Root Cause Analyzer* based on the output of *Log Parser*. The corresponding source code snippet is listed in Listing 5.2. The error was in Line 7, since `getSubstitutePositions` method of the object *selectedPosition* was not initialized. Then, *Root Cause Analyzer* located the point in the source code, where this object was last defined. That is Line 4 in Listing 5.2. It turned out that the definition was coming from the definition of *selectedPosition*.

This object is not initialized correctly. There is a JPA⁶ entity named *Position* which has *getSubstitutePositions* method that is used in the application to get substitute positions of a position. If the *Position* object is initialized by using the *positionService.getPositionByIdWithSubstitutePositions* method, *getSubstitutePositions* method runs properly. However, if the *Position* object is initialized by another *positionService* method like *getPositionById*, it causes `org.hibernate.LazyInitializationException`.

Listing 5.2: The code snippet corresponding to the logged error.

```
1 Position selectedPosition;  
2 Position solUpperPosition;  
3 ...  
4 selectedPosition = positionService.getPositionById(positionId);  
5 ...  
6 solUpperPosition = positionService.getPositionById(  
7     selectedPosition.getSubstitutePositions().get(0)...  
8 );  
9 }
```

Then, an application-specific rule is inferred as: if the *getSubstitutePositions* method of *Position* object is used, then the *Position* object should be initialized properly using the *getPositionByIdWithSubstitutePositions* method. A specialized checker is automatically generated based on this rule. It checks the whole code base and searches for the usage of the *getSubstitutePositions* method. As the last step, Sonar is extended with the specialized checker.

5.2.3 NullPointerException Error In CMS

The third sample log content is regarding a `NullPointerException` error as shown in Figure 9. This time the log files come from the CMS system. Figure 10 shows a

⁶<http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html>

```

2016-06-15 02:40:33,490 INFO [STDOUT] 06-15@02:40 33 ERROR ( Logger.java:94) - null : NULL : USER->X : Exception Detail : null
java.lang.NullPointerException
    at com.turkcelltech.comet.commonutil.services.impl.OBCampaignDefinitionServiceImpl.getCampaignDefinition(OBCampaignDefinitionServiceImpl.java:157)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:310)
    at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:198)
    at $Proxy171.getCampaignDefinition(Unknown Source)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

```

Figure 9: A snippet from a log file regarding a `NullPointerException` error on CMS.

```

*****ERROR type:0*****
firstErrorline      :
firstClassStackTrace:
com.turkcelltech.comet.commonutil.services.impl.OBCampaignDefinitionServiceImpl.getCampaignDefinition(OBCampaignDefinitionServiceImpl.java:157)
firstTSFAClassStackTrace:at
com.turkcelltech.comet.commonutil.services.impl.OBCampaignDefinitionServiceImpl.getCampaignDefinition(OBCampaignDefinitionServiceImpl.java:157)
*****
*****FULL ERROR DETAIL: *****
*****TOPLAM TUM DOSYALARDA 4 kere olmus *****
*****AYNI KODUN ATTIGI FARKLI CESIT HATA SAYISI 2 *****
-----

```

Figure 10: A snippet from the output of *Log Parser* regarding the instances of the `NullPointerException` error recorded as shown in Figure 9.

snippet from the output of *Log Parser* regarding the instances of the `NullPointerException` error as shown in Figure 9. Hereby, we can see that this error was observed 4 times at different application servers. Currently, we do not utilize this information; however, it can be used for prioritization in the future.

The source code of CMS is analyzed by *Root Cause Analyzer* based on the output of *Log Parser*. The corresponding source code snippet is listed in Listing 5.3. The error was in Line 2, since the object `schedulerList` was null. Then, *Root Cause Analyzer* located the point in the source code, where this object was last defined. That is Line 1 in Listing 5.3. It turned out that the definition was coming from a method call, i.e., `schedulerService.getSchedulerList(schedulerSearch)`; This method performs a database call to get scheduler information. It creates and returns an object by utilizing information from a relational database. If there is no scheduler record in the database, a null value is returned.

Listing 5.3: The code snippet corresponding to the logged error on CMS

```
1 ...
2 List<Scheduler> schedulerList =
3     schedulerService.getSchedulerList(schedulerSearch);
4
5 if (schedulerList.size() > 0)
6 {
7     ...
8 }
```

Then, an application-specific rule is inferred as: the return value of the method *getSchedulerList* must be checked for null references before use. A specialized checker is automatically generated based on this rule. It checks the whole code base and searches for initialized objects using the return value of the method *getSchedulerList* without a null reference check. As the last step, Sonar is extended with the specialized checker.

CHAPTER VI

RESULTS AND DISCUSSION

We performed our evaluation in two different ways for the two subject systems. For the first subject system, SFA, we did not modify the source code. We aimed at detecting real faults manifested in the system by checkers automatically generated based on existing log files.

For the second subject system, CMS, we also employed a controlled experiment approach in addition to our case study. We injected a set of faults of the same type to test the effectiveness of our approach in detecting instances of a detected fault type in other parts of the source code. So, we injected 5 faults to 5 different classes of the CMS randomly. Then, we triggered one of the faults at runtime to capture the log of the failure caused by that fault. We applied our approach to generate checkers regarding the recorded failure. Then, we checked if all the injected faults can be detected in the source code. In the following, we share and discuss the obtained results. Then, we discuss threats to validity.

6.1 Results

After the extension of the static code analysis tool and its execution on the SFA system, 26 additional alerts were generated regarding the first rule concerning `NullPointerException` errors caused by application-specific methods (See Figure 14). Examples of these faults exist in Appendix C. Regarding the `LazyInitializationException` error, there was one additional alert (See Figure 15). All these alerts were true positives and the corresponding code locations really required to be fixed. In fact, we saw that 3 of these locations caused runtime errors afterwards and they were fixed in a later version of the source code. These errors were caused by the bug type listed in Listing 5.1 and

Listing 5.2. If our approach were applied and all the reported alerts were addressed, these errors would not occur at all. As a result, 27 real faults were detected with specialized checkers and 3 of them were actually activated during operational time. This result shows the importance and high potential of information collected at runtime as a source for improving recall in static analysis.

For the CMS case, we used the checker generated based on a reported failure regarding one of the injected fault instances. We observed that all the injected 5 instances were detected by the generated checker (See Figure 16).

6.2 *Discussions*

Our approach is used for extending static code analysis tool to catch application specific bugs automatically. Although we generated checker templates for a set of errors, there is a possibility that we cannot create a checker template for all types application specific errors. There is a table in Table 2 that includes a list of sample errors encountered in the SFA system logs for a period of time. The error type number 2 and 5 are errors for which we generated checker templates. For the error type number 4, it is possible to generate a checker template as well. However, it is not possible to generate checker templates for the other error types in the list. For example, an instance of error type number 1 occurred when the system tried to insert a record to a table that violates integrity constraints. Since our checker templates use static code analysis, it is not possible to capture errors related to runtime data. As another example, an instance of the error type number 6 occurs when the remote host is not available. This is not an application specific problem but an infrastructure problem. So, it is not possible to detect other instances of such an error with static analysis.

No	Exception Description	Custom Checker Possible?
1	ORA-00001:unique constraint	N
2	java.lang.NullPointerException	Y
3	Not all named parameters have been set	N
4	InvocationTargetException	Y
5	LazyInitializationException	Y
6	UnknownHostException:	NA

Table 2: A sample list of errors detected for the SFA system.

6.3 Threats to Validity

There are some validity threats to our evaluation. First, we have developed checker templates only for two different error types. Also, we used two subject systems from the same application domain. Therefore, we plan to develop more checker templates for different error types and perform more case studies in different application domains. Second, we have applied our approach only for software systems written in Java. In principle, the approach can be instantiated for different programming languages and environments. For the SFA case, we detected 26 additional faults of one type and only one for the other type. We can infer that this approach is more useful for certain exceptions for some systems, and not for others. Our controlled experiment is also limited to 5 instances of one fault type. We used dedicated log parser to analyse the log files. However, it can be replaced with any parser to be able to process log files in other formats as well.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

In this thesis, we proposed an approach for improving the recall of static code analysis. In our approach, we extract application-specific programming rules by analyzing execution traces collected at runtime. We automatically generate specialized checkers for these rules as part of a static code analysis tool. Then, the static code analysis tool can detect more faults with high precision since it checks for instances of a fault that previously caused an error at runtime.

We conducted two industrial case studies from the telecommunications domain. We utilized existing execution logs of a legacy system to extend static code analysis. We were able to detect real faults, which had to be fixed later on. Hence, information collected at runtime can be effectively exploited for improving recall in static analysis.

In the future, we plan to extend our approach to cover different types of errors and rules. We also plan to conduct more case studies.

APPENDIX A

ABSTRACT SYNTAX TREE EXAMPLES

There are three AST examples which are very similar but there are some small differences. All three examples are about calling methodX method in same class and method. First Example is base example so there is no *local variable declaration* and *this* usage. In second example, there is *local variable declaration* as extra. And third example, there is only *this* usage as extra. In Figure 12 and 13, there are also rectangle and arrow to indicate the difference.

Listing A.1: Sample code for AST in Figure 11

```
public class Dummy
{
    public static void dummy() throws Exception
    {
        result = methodX(2, 2);
    }
}
```

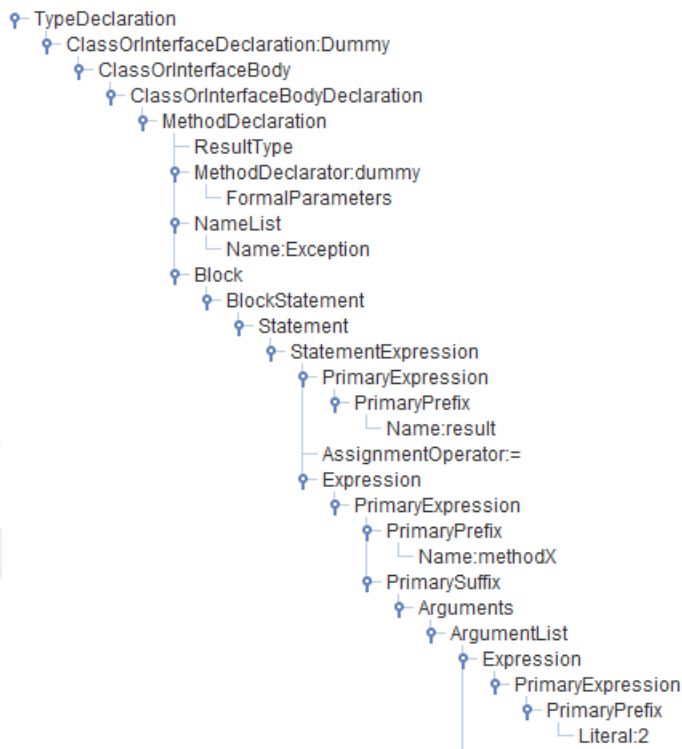


Figure 11: Generated AST for sample code A.1

Listing A.2: Sample code for AST in Figure 12

```
public class Dummy
{
    public static void dummy() throws Exception
    {
        int result = methodX(2, 2);
    }
}
```

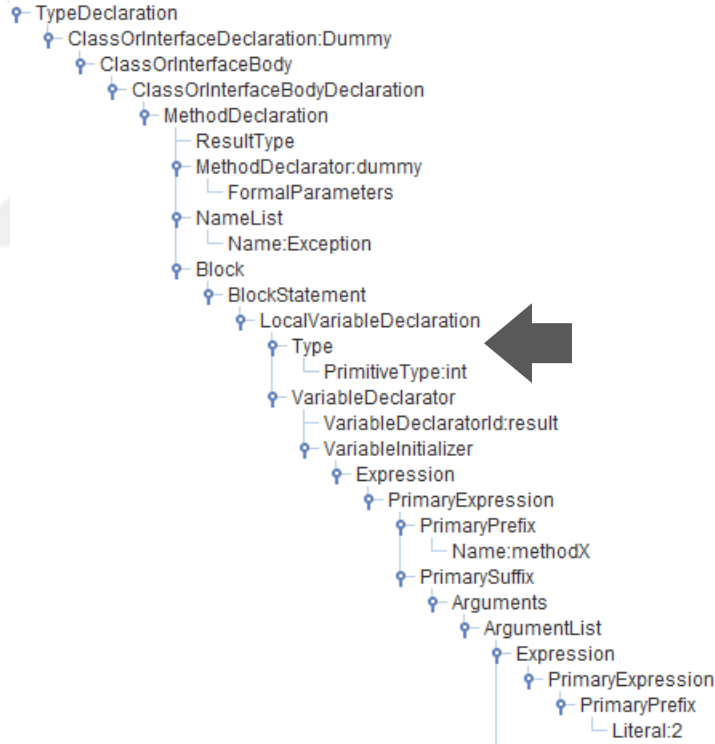



Figure 12: Generated AST for sample code A.2

Listing A.3: Sample code for AST in Figure 13

```
public class Dummy
{
    public static void dummy() throws Exception
    {
        result = this.methodX(2, 2);
    }
}
```



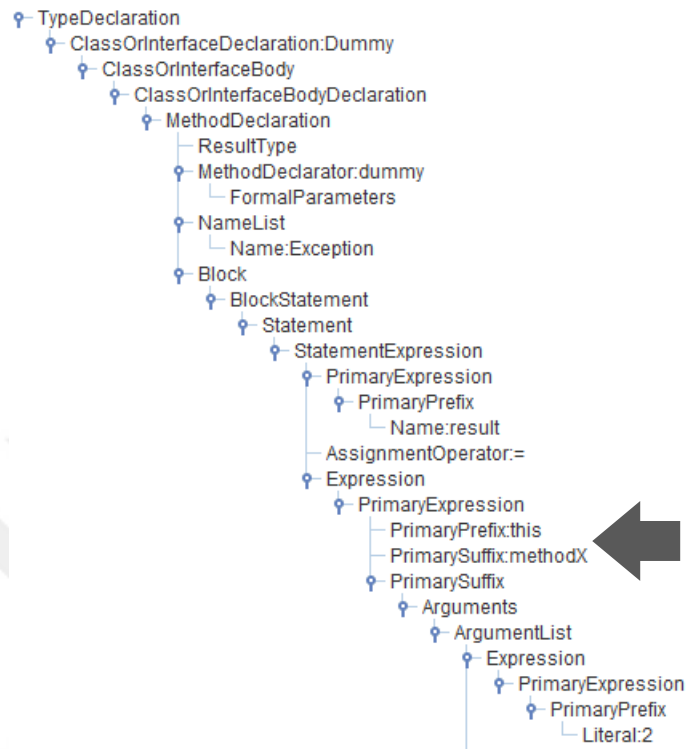


Figure 13: Generated AST for sample code A.3

APPENDIX B

SONAR DASHBOARD SCREENSHOTS



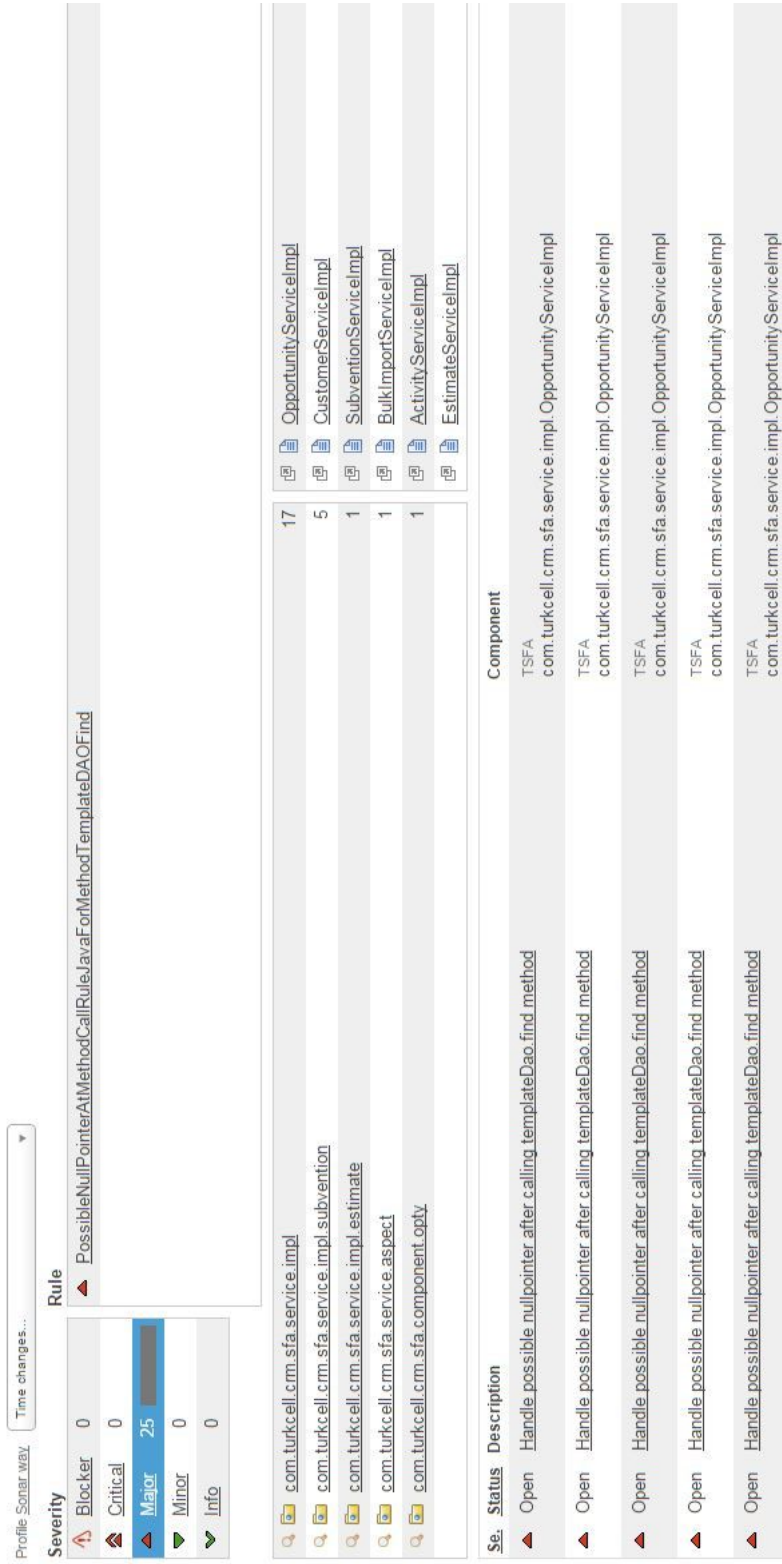


Figure 14: Sonar dashboard for TSFA module

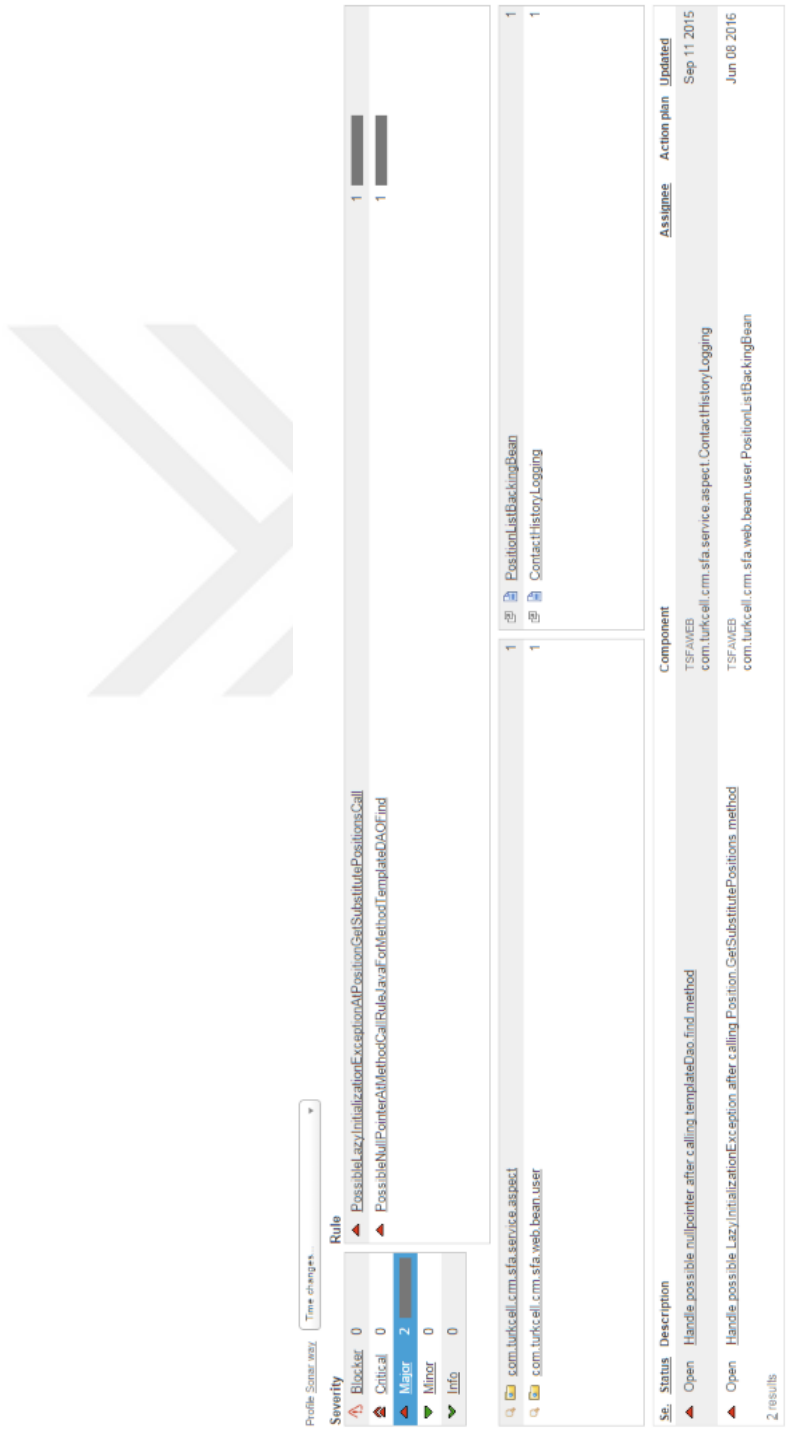


Figure 15: Sonar dashboard for TSFAWEB module

Thingol >

Dashboard
Hotspots
Issues
Time Machine
TOOLS
Components
Issues Dnldown
Design
Libraries
Clouds
Compare
sonarqube

Profile: Sonar.WAZ Time change:

Severity
 Blocker 0
 Critical 0
 Major 5
 Minor 0
 Info 0

Rule
 PossibleNullPointerInMethodCallRuleForMethodSchedulerServiceGetScheduledList

5

com.turkcelltech.comet.commonutils.services.impl

5

- ODCampaignDefinitionServiceImpl 1
- SampleServiceImpl 1
- SmsProcessorServiceImpl 1
- SubscriptionServiceImpl 1
- WspProcessorServiceImpl 1

Se.	Status	Description	Component	Assignee	Action plan	Updated
▲	Open	Handle possible nullpointer after calling schedulerService.getScheduledList method	Thngol com.turkcelltech.comet.commonutils.services.impl.ODCampaignDefinitionServiceImpl			Jun 15 2016
▲	Open	Handle possible nullpointer after calling schedulerService.getScheduledList method	Thngol com.turkcelltech.comet.commonutils.services.impl.SampleServiceImpl			14:52
▲	Open	Handle possible nullpointer after calling schedulerService.getScheduledList method	Thngol com.turkcelltech.comet.commonutils.services.impl.SmsProcessorServiceImpl			14:57
▲	Open	Handle possible nullpointer after calling schedulerService.getScheduledList method	Thngol com.turkcelltech.comet.commonutils.services.impl.SubscriptionServiceImpl			14:57
▲	Open	Handle possible nullpointer after calling schedulerService.getScheduledList method	Thngol com.turkcelltech.comet.commonutils.services.impl.WspProcessorServiceImpl			14:57

2 RESULTS

Figure 16: Sonar dashboard for CMS

[Dashboards](#) [Projects](#) [Measures](#) [Issues](#) [Quality Profiles](#) [Log in](#)

[Quality Profiles](#) > [Java](#) > [Sonarway](#) >

[Coding rules](#) | [Alerts](#) | [Projects](#) | [Permalinks](#) | [Profile inheritance](#) | [Channels](#)

[Download](#)

Active Severity

Active	Severity	Name [expand / collapse]
<input checked="" type="checkbox"/>	Major	PossibleLazyInitializationExceptionAPIPositionGetSubstitutePositionsCall
<input checked="" type="checkbox"/>	Major	PossibleNullPointerAPIMethodCallRuleForMethodSchedulerServiceGetScheduledList
<input checked="" type="checkbox"/>	Major	PossibleNullPointerAPIMethodCallRuleJavaForMethodTemplateDAOFind

3 results

Sort by:

Figure 17: Sonar Rules Page

APPENDIX C

REAL FAULT EXAMPLES

Listing C.1: Fault example 1

```
1 ...
2 TcstPosition customerTeamRegular = templateDao.find(TcstPosition.
   class, customerTeamRegularId);
3 ...
4 cstDto.setNposition(customerTeamRegular.getNposition());
5 ...
```

Listing C.2: Fault example 2

```
1 ...
2 TcstPosition t = templateDao.find(TcstPosition.class, ncstpostn);
3 ...
4 if (t.getCtypecstpostn()==...)
5 {
6     ...
7 }
```

Listing C.3: Fault example 3

```
1 ...
2 BulkImport bulk = templateDao.find(BulkImport.class, bulkKey);
3 ...
4 bulk.setNprocessedreccount(bulk.getNprocessedreccount() + 1);
5 ...
```

References

- [1] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?,” in *Proceedings of the 35th International Conference on Software Engineering*, pp. 672–681, 2013.
- [2] U. Yuksel and H. Sozer, “Automated classification of static code analysis alerts: A case study,” in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, (Eindhoven, The Netherlands), pp. 532–535, 2013.
- [3] R. Krishnan, M. Nadworny, and N. Bharill, “Static analysis tools for security checking in code at motorola,” *ACM SIG Ada Letters*, vol. 28, no. 1, pp. 76–82, 2008.
- [4] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, “On the value of static analysis for fault detection in software,” *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006.
- [5] B. Sun, G. Shu, A. Podgurski, and B. Robinson, “Extending static analysis by mining project-specific rules,” in *Proceedings of the 34th International Conference on Software Engineering*, (Zurich, Switzerland), pp. 1054–1063, 2012.
- [6] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh, “Using static analysis to find bugs,” *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.
- [7] H. Sozer, “Integrated static code analysis and runtime verification,” *Software: Practice and Experience*, vol. 45, no. 10, pp. 1359–1373, 2015.
- [8] B. Chess and G. McGraw, “Static analysis for security,” *IEEE Computer Society*, vol. 2, no. 6, pp. 76–79, 2004.
- [9] P. Hellstrm, “Tools for static code analysis: A survey,” Master’s thesis, Linkping University, The address of the publisher, 2 2009. An optional note.
- [10] B. Chess and J. West, *Secure Programming with Static Analysis*. Boston, MA: Pearson Education, Inc., 2007.
- [11] “FindBugs official website,” 2016. [online] <http://findbugs.sourceforge.net>.
- [12] “PMD official website,” 2016. [online] <https://pmd.github.io/>.
- [13] “Eclipse official website,” 2016. [online] <http://eclipse.org/>.
- [14] “NetBeans official website,” 2016. [online] <http://netbeans.org/>.
- [15] G. A. Campbell and P. P. Papapetrou, *Sonarqube in Action*. Shelter Island, NY: Manning, 2014.

- [16] N. Fenton and S. Pfleeger, *Software metrics: a rigorous and practical approach*. International Thomson Computer Press, 1996.
- [17] “PMD official rulesets,” 2016. [online] <http://pmd.sourceforge.net/pmd-4.3.0/rules/index.html>.
- [18] R. Chang, A. Podgurski, and J. Yang, “Discovering neglected conditions in software by mining dependence graphs,” *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 579–596, 2008.
- [19] R. Chang and A. Podgurski, “Discovering programming rules and violations by mining interprocedural dependences,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 24, pp. 51–66, 2011.
- [20] B. Sun, X. Chen, R. Changand, and A. Podgurski, “Automated support for propagating bug fixes,” in *Proceedings of the 19th International Symposium on Software Reliability Engineering*, (Seattle, WA, USA), pp. 187–196, 2008.
- [21] C. Williams and J. Holingsworth, “Automatic mining of source code repositories to improve bug finding techniques,” *IEEE Transactions on Software Engineering*, vol. 31, pp. 466–480, 2005.
- [22] S. Kim, T. Zimmermann, K. Pan, and E. Whitehead, “Automatic identification of bug-introducing changes,” in *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*, (Washington, DC, USA), pp. 81–90, 2006.
- [23] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, “Detecting large-scale system problems by mining console logs,” in *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, pp. 117–132, 2009.
- [24] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao, “The Daikon system for dynamic detection of likely invariants,” *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, 2007.
- [25] B. Livshits and T. Zimmerman, “Dynamine: Finding common error patterns by mining software revision histories,” *SIGSOFT Software Engineering Notes*, vol. 30, pp. 296–305, 2005.
- [26] C. Csallner and Y. Smaragdakis, “DSD-Crasher: A hybrid analysis tool for bug finding,” in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pp. 245–254, 2006.
- [27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [28] D. Barry and T. Stanienda, “Solving the java object storage problem,” *IEEE Computer*, vol. 31, no. 11, pp. 33–40, 1998.